

---

# CuPy Documentation

*Release 13.1.0*

**Preferred Networks, inc. and Preferred Infrastructure, inc.**

**Apr 30, 2024**



# CONTENTS

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Project Goal . . . . .	2
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Requirements . . . . .	3
2.2	Installing CuPy . . . . .	4
2.3	Uninstalling CuPy . . . . .	6
2.4	Upgrading CuPy . . . . .	7
2.5	Reinstalling CuPy . . . . .	7
2.6	Using CuPy inside Docker . . . . .	7
2.7	FAQ . . . . .	7
<b>3</b>	<b>Using CuPy on AMD GPU (experimental)</b>	<b>11</b>
3.1	Requirements . . . . .	11
3.2	Environment Variables . . . . .	11
3.3	Docker . . . . .	11
3.4	Installing Binary Packages . . . . .	11
3.5	Building CuPy for ROCm From Source . . . . .	12
3.6	Limitations . . . . .	12
<b>4</b>	<b>User Guide</b>	<b>15</b>
4.1	Basics of CuPy . . . . .	15
4.2	User-Defined Kernels . . . . .	18
4.3	Accessing CUDA Functionalities . . . . .	28
4.4	Fast Fourier Transform with CuPy . . . . .	30
4.5	Memory Management . . . . .	36
4.6	Performance Best Practices . . . . .	39
4.7	Interoperability . . . . .	41
4.8	Differences between CuPy and NumPy . . . . .	50
4.9	API Compatibility Policy . . . . .	53
<b>5</b>	<b>API Reference</b>	<b>57</b>
5.1	The N-dimensional array (ndarray) . . . . .	57
5.2	Universal functions (cupy.ufunc) . . . . .	73
5.3	Routines (NumPy) . . . . .	98
5.4	Routines (SciPy) . . . . .	318
5.5	CuPy-specific functions . . . . .	736
5.6	Low-level CUDA support . . . . .	748
5.7	Custom kernels . . . . .	813
5.8	Distributed . . . . .	836

5.9	Environment variables . . . . .	851
5.10	Comparison Table . . . . .	854
5.11	Python Array API Support . . . . .	895
<b>6</b>	<b>Contribution Guide</b>	<b>911</b>
6.1	Classification of Contributions . . . . .	911
6.2	Development Cycle . . . . .	911
6.3	Issues and Pull Requests . . . . .	913
6.4	Coding Guidelines . . . . .	914
6.5	Unit Testing . . . . .	915
6.6	Documentation . . . . .	917
6.7	Tips for Developers . . . . .	918
<b>7</b>	<b>Upgrade Guide</b>	<b>919</b>
7.1	CuPy v13 . . . . .	919
7.2	CuPy v12 . . . . .	921
7.3	CuPy v11 . . . . .	922
7.4	CuPy v10 . . . . .	923
7.5	CuPy v9 . . . . .	926
7.6	CuPy v8 . . . . .	927
7.7	CuPy v7 . . . . .	928
7.8	CuPy v6 . . . . .	928
7.9	CuPy v5 . . . . .	928
7.10	CuPy v4 . . . . .	929
7.11	CuPy v2 . . . . .	930
7.12	Compatibility Matrix . . . . .	931
<b>8</b>	<b>License</b>	<b>933</b>
8.1	NumPy . . . . .	933
8.2	SciPy . . . . .	934
8.3	cuSignal . . . . .	934
	<b>Python Module Index</b>	<b>937</b>
	<b>Index</b>	<b>939</b>

## OVERVIEW

CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. CuPy acts as a drop-in replacement to run existing NumPy/SciPy code on **NVIDIA CUDA** or **AMD ROCm** platforms.

CuPy provides a `ndarray`, sparse matrices, and the associated routines for GPU devices, all having the same API as NumPy and SciPy:

- **N-dimensional array** (`ndarray`): *cupy.ndarray*
  - Data types (dtypes): boolean (`bool_`), integer (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`), float (`float16`, `float32`, `float64`), and complex (`complex64`, `complex128`)
  - Supports the semantics identical to `numpy.ndarray`, including basic / advanced indexing and broadcasting
- **Sparse matrices**: *cupyx.scipy.sparse*
  - 2-D sparse matrix: `csr_matrix`, `coo_matrix`, `csc_matrix`, and `dia_matrix`
- **NumPy Routines**
  - *Module-level Functions* (`cupy.*`)
  - *Linear Algebra Functions* (`cupy.linalg.*`)
  - *Fast Fourier Transform* (`cupy.fft.*`)
  - *Random Number Generator* (`cupy.random.*`)
- **SciPy Routines**
  - *Discrete Fourier Transforms* (`cupyx.scipy.fft.*` and `cupyx.scipy.fftpack.*`)
  - *Advanced Linear Algebra* (`cupyx.scipy.linalg.*`)
  - *Multidimensional Image Processing* (`cupyx.scipy.ndimage.*`)
  - *Sparse Matrices* (`cupyx.scipy.sparse.*`)
  - *Sparse Linear Algebra* (`cupyx.scipy.sparse.linalg.*`)
  - *Special Functions* (`cupyx.scipy.special.*`)
  - *Signal Processing* (`cupyx.scipy.signal.*`)
  - *Statistical Functions* (`cupyx.scipy.stats.*`)

Routines are backed by CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuSOLVER, cuRAND), Thrust, CUB, and cuTENSOR to provide the best performance.

It is also possible to easily implement *custom CUDA kernels* that work with `ndarray` using:

- **Kernel Templates**: Quickly define element-wise and reduction operation as a single CUDA kernel
- **Raw Kernel**: Import existing CUDA C/C++ code

- **Just-in-time Transpiler (JIT):** Generate CUDA kernel from Python source code
- **Kernel Fusion:** Fuse multiple CuPy operations into a single CUDA kernel

CuPy can run in multi-GPU or cluster environments. The distributed communication package (*cupyx.distributed*) provides collective and peer-to-peer primitives for `ndarray`, backed by NCCL.

For users who need more fine-grain control for performance, accessing *low-level CUDA features* are available:

- **Stream and Event:** CUDA stream and per-thread default stream are supported by all APIs
- **Memory Pool:** Customizable memory allocator with a built-in memory pool
- **Profiler:** Supports profiling code using CUDA Profiler and NVTX
- **Host API Binding:** Directly call CUDA libraries, such as NCCL, cuDNN, cuTENSOR, and cuSPARSELt APIs from Python

CuPy implements standard APIs for data exchange and interoperability, such as *DLPack*, *CUDA Array Interface*, `__array_ufunc__` (NEP 13), `__array_function__` (NEP 18), and *Array API Standard*. Thanks to these protocols, CuPy easily *integrates* with NumPy, PyTorch, TensorFlow, MPI4Py, and any other libraries supporting the standard.

Under AMD ROCm environment, CuPy automatically translates all CUDA API calls to ROCm HIP (hipBLAS, hipFFT, hipSPARSE, hipRAND, hipCUB, hipThrust, RCCL, etc.), allowing code written using CuPy to run on both NVIDIA and AMD GPU without any modification.

## 1.1 Project Goal

The goal of the CuPy project is to provide Python users GPU acceleration capabilities, without the in-depth knowledge of underlying GPU technologies. The CuPy team focuses on providing:

- A complete NumPy and SciPy API coverage to become a full drop-in replacement, as well as advanced CUDA features to maximize the performance.
- Mature and quality library as a fundamental package for all projects needing acceleration, from a lab environment to a large-scale cluster.

## INSTALLATION

### 2.1 Requirements

- **NVIDIA CUDA GPU** with the Compute Capability 3.0 or larger.
- **CUDA Toolkit**: v11.2 / v11.3 / v11.4 / v11.5 / v11.6 / v11.7 / v11.8 / v12.0 / v12.1 / v12.2 / v12.3 / v12.4
  - If you have multiple versions of CUDA Toolkit installed, CuPy will automatically choose one of the CUDA installations. See *Working with Custom CUDA Installation* for details.
  - This requirement is optional if you install CuPy from `conda-forge`. However, you still need to have a compatible driver installed for your GPU. See *Installing CuPy from Conda-Forge* for details.
- **Python**: v3.9 / v3.10 / v3.11 / v3.12

---

**Note:** Currently, CuPy is tested against **Ubuntu** 20.04 LTS / 22.04 LTS (x86\_64), **CentOS** 7 / 8 (x86\_64) and Windows Server 2016 (x86\_64).

---

#### 2.1.1 Python Dependencies

NumPy/SciPy-compatible API in CuPy v13 is based on NumPy 1.26 and SciPy 1.11, and has been tested against the following versions:

- **NumPy**: v1.22 / v1.23 / v1.24 / v1.25 / v1.26
- **SciPy** (*optional*): v1.7 / v1.8 / v1.9 / v1.10 / v1.11
  - Required only when coping sparse matrices from GPU to CPU (see *Sparse matrices (cupyx.scipy.sparse)*.)
- **Optuna** (*optional*): v3.x
  - Required only when using *Automatic Kernel Parameters Optimizations (cupyx.optimizing)*.

---

**Note:** SciPy and Optuna are optional dependencies and will not be installed automatically.

---

---

**Note:** Before installing CuPy, we recommend you to upgrade `setuptools` and `pip`:

```
$ python -m pip install -U setuptools pip
```

---

## 2.1.2 Additional CUDA Libraries

Part of the CUDA features in CuPy will be activated only when the corresponding libraries are installed.

- **cuTENSOR**: v2.0
  - The library to accelerate tensor operations. See [Environment variables](#) for the details.
- **NCCL**: v2.16 / v2.17
  - The library to perform collective multi-GPU / multi-node computations.
- **cuDNN**: v8.8
  - The library to accelerate deep neural network computations.
- **cuSPARSELt**: v0.2.0
  - The library to accelerate sparse matrix-matrix multiplication.

## 2.2 Installing CuPy

### 2.2.1 Installing CuPy from PyPI

Wheels (precompiled binary packages) are available for Linux and Windows. Package names are different depending on your CUDA Toolkit version.

CUDA	Command
<b>v11.2 ~ 11.8</b> (x86_64 / aarch64)	<code>pip install cupy-cuda11x</code>
<b>v12.x</b> (x86_64 / aarch64)	<code>pip install cupy-cuda12x</code>

---

**Note:** To enable features provided by additional CUDA libraries (cuTENSOR / NCCL / cuDNN), you need to install them manually. If you installed CuPy via wheels, you can use the installer command below to setup these libraries in case you don't have a previous installation:

```
$ python -m cupyx.tools.install_library --cuda 11.x --library cutensor
```

---

---

**Note:** Append `--pre -U -f https://pip.cupy.dev/pre` options to install pre-releases (e.g., `pip install cupy-cuda11x --pre -U -f https://pip.cupy.dev/pre`).

---

When using wheels, please be careful not to install multiple CuPy packages at the same time. Any of these packages and cupy package (source installation) conflict with each other. Please make sure that only one CuPy package (cupy or cupy-cudaXX where XX is a CUDA version) is installed:

```
$ pip freeze | grep cupy
```

---



## 2.2.2 Installing CuPy from Conda-Forge

Conda is a cross-language, cross-platform package management solution widely used in scientific computing and other fields. The above `pip install` instruction is compatible with conda environments. Alternatively, for both Linux (x86\_64, ppc64le, aarch64-sbsa) and Windows once the CUDA driver is correctly set up, you can also install CuPy from the conda-forge channel:

```
$ conda install -c conda-forge cupy
```

and conda will install a pre-built CuPy binary package for you, along with the CUDA runtime libraries (`cuda-toolkit` for CUDA 11 and below, or `cuda-XXXXX` for CUDA 12 and above). It is not necessary to install CUDA Toolkit in advance.

If you aim at minimizing the installation footprint, you can install the `cupy-core` package:

```
$ conda install -c conda-forge cupy-core
```

which only depends on `numpy`. None of the CUDA libraries will be installed this way, and it is your responsibility to install the needed dependencies yourself, either from conda-forge or elsewhere. This is equivalent of the `cupy-cudaXX` wheel installation.

Conda has a built-in mechanism to determine and install the latest version of `cuda-toolkit` or any other CUDA components supported by your driver. However, if for any reason you need to force-install a particular CUDA version (say 11.8), you can do:

```
$ conda install -c conda-forge cupy cuda-version=11.8
```

**Note:** `cuDNN`, `cuTENSOR`, and `NCCL` are available on conda-forge as optional dependencies. The following command can install them all at once:

```
$ conda install -c conda-forge cupy cudnn cutensor nccl
```

Each of them can also be installed separately as needed.

**Note:** If you encounter any problem with CuPy installed from conda-forge, please feel free to report to [cupy-feedstock](#), and we will help investigate if it is just a packaging issue in conda-forge's recipe or a real issue in CuPy.

**Note:** If you did not install CUDA Toolkit by yourself, for CUDA 11 and below the `nvcc` compiler might not be available, as the `cuda-toolkit` package from conda-forge does not include the `nvcc` compiler toolchain. If you would like to use it from a local CUDA installation, you need to make sure the version of CUDA Toolkit matches that of `cuda-toolkit` to avoid surprises. For CUDA 12 and above, `nvcc` can be installed on a per-conda environment basis via

```
$ conda install -c conda-forge cuda-nvcc
```

### 2.2.3 Installing CuPy from Source

Use of wheel packages is recommended whenever possible. However, if wheels cannot meet your requirements (e.g., you are running non-Linux environment or want to use a version of CUDA / cuDNN / NCCL not supported by wheels), you can also build CuPy from source.

---

**Note:** CuPy source build requires g++-6 or later. For Ubuntu 18.04, run `apt-get install g++`. For Ubuntu 16.04, CentOS 6 or 7, follow the instructions [here](#).

---

---

**Note:** When installing CuPy from source, features provided by additional CUDA libraries will be disabled if these libraries are not available at the build time. See [Installing cuDNN and NCCL](#) for the instructions.

---

---

**Note:** If you upgrade or downgrade the version of CUDA Toolkit, cuDNN, NCCL or cuTENSOR, you may need to reinstall CuPy. See [Reinstalling CuPy](#) for details.

---

You can install the latest stable release version of the [CuPy source package](#) via `pip`.

```
$ pip install cupy
```

If you want to install the latest development version of CuPy from a cloned Git repository:

```
$ git clone --recursive https://github.com/cupy/cupy.git
$ cd cupy
$ pip install .
```

---

**Note:** Cython 0.29.22 or later is required to build CuPy from source. It will be automatically installed during the build process if not available.

---

## 2.3 Uninstalling CuPy

Use `pip` to uninstall CuPy:

```
$ pip uninstall cupy
```

---

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where XX is a CUDA version number).

---

---

**Note:** If CuPy is installed via `conda`, please do `conda uninstall cupy` instead.

---

## 2.4 Upgrading CuPy

Just use `pip install` with `-U` option:

```
$ pip install -U cupy
```

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where `XX` is a CUDA version number).

## 2.5 Reinstalling CuPy

To reinstall CuPy, please uninstall CuPy and then install it. When reinstalling CuPy, we recommend using `--no-cache-dir` option as `pip` caches the previously built binaries:

```
$ pip uninstall cupy
$ pip install cupy --no-cache-dir
```

**Note:** If you are using a wheel, `cupy` shall be replaced with `cupy-cudaXX` (where `XX` is a CUDA version number).

## 2.6 Using CuPy inside Docker

We are providing the [official Docker images](#). Use [NVIDIA Container Toolkit](#) to run CuPy image with GPU. You can login to the environment with `bash`, and run the Python interpreter:

```
$ docker run --gpus all -it cupy/cupy /bin/bash
```

Or run the interpreter directly:

```
$ docker run --gpus all -it cupy/cupy /usr/bin/python3
```

## 2.7 FAQ

### 2.7.1 `pip` fails to install CuPy

Please make sure that you are using the latest `setuptools` and `pip`:

```
$ pip install -U setuptools pip
```

Use `-vvvv` option with `pip` command. This will display all logs of installation:

```
$ pip install cupy -vvvv
```

If you are using `sudo` to install CuPy, note that `sudo` command does not propagate environment variables. If you need to pass environment variable (e.g., `CUDA_PATH`), you need to specify them inside `sudo` like this:

```
$ sudo CUDA_PATH=/opt/nvidia/cuda pip install cupy
```

If you are using certain versions of conda, it may fail to build CuPy with error `g++: error: unrecognized command line option '-R'`. This is due to a bug in conda (see [conda/conda#6030](#) for details). If you encounter this problem, please upgrade your conda.

## 2.7.2 Installing cuDNN and NCCL

We recommend installing cuDNN and NCCL using binary packages (i.e., using `apt` or `yum`) provided by NVIDIA.

If you want to install tar-gz version of cuDNN and NCCL, we recommend installing it under the `CUDA_PATH` directory. For example, if you are using Ubuntu, copy `*.h` files to `include` directory and `*.so*` files to `lib64` directory:

```
$ cp /path/to/cudnn.h $CUDA_PATH/include
$ cp /path/to/libcudnn.so* $CUDA_PATH/lib64
```

The destination directories depend on your environment.

If you want to use cuDNN or NCCL installed in another directory, please use `CFLAGS`, `LDFLAGS` and `LD_LIBRARY_PATH` environment variables before installing CuPy:

```
$ export CFLAGS=-I/path/to/cudnn/include
$ export LDFLAGS=-L/path/to/cudnn/lib
$ export LD_LIBRARY_PATH=/path/to/cudnn/lib:$LD_LIBRARY_PATH
```

## 2.7.3 Working with Custom CUDA Installation

If you have installed CUDA on the non-default directory or multiple CUDA versions on the same host, you may need to manually specify the CUDA installation directory to be used by CuPy.

CuPy uses the first CUDA installation directory found by the following order.

1. `CUDA_PATH` environment variable.
2. The parent directory of `nvcc` command. CuPy looks for `nvcc` command from `PATH` environment variable.
3. `/usr/local/cuda`

For example, you can build CuPy using non-default CUDA directory by `CUDA_PATH` environment variable:

```
$ CUDA_PATH=/opt/nvidia/cuda pip install cupy
```

---

**Note:** CUDA installation discovery is also performed at runtime using the rule above. Depending on your system configuration, you may also need to set `LD_LIBRARY_PATH` environment variable to `$CUDA_PATH/lib64` at runtime.

---

## 2.7.4 CuPy always raises `cupy.cuda.compiler.CompileException`

If CuPy raises a `CompileException` for almost everything, it is possible that CuPy cannot detect CUDA installed on your system correctly. The followings are error messages commonly observed in such cases.

- `nvrtc: error: failed to load builtins`
- `catastrophic error: cannot open source file "cuda_fp16.h"`
- `error: cannot overload functions distinguished by return type alone`
- `error: identifier "__half_raw" is undefined`

Please try setting `LD_LIBRARY_PATH` and `CUDA_PATH` environment variable. For example, if you have CUDA installed at `/usr/local/cuda-9.2`:

```
$ export CUDA_PATH=/usr/local/cuda-9.2
$ export LD_LIBRARY_PATH=$CUDA_PATH/lib64:$LD_LIBRARY_PATH
```

Also see [Working with Custom CUDA Installation](#).

## 2.7.5 Build fails on Ubuntu 16.04, CentOS 6 or 7

In order to build CuPy from source on systems with legacy GCC (g++-5 or earlier), you need to manually set up g++-6 or later and configure NVCC environment variable.

On Ubuntu 16.04:

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install g++-6
$ export NVCC="nvcc --compiler-bindir gcc-6"
```

On CentOS 6 / 7:

```
$ sudo yum install centos-release-scl
$ sudo yum install devtoolset-7-gcc-c++
$ source /opt/rh/devtoolset-7/enable
$ export NVCC="nvcc --compiler-bindir gcc"
```



## USING CUPY ON AMD GPU (EXPERIMENTAL)

CuPy has an experimental support for AMD GPU (ROCm).

### 3.1 Requirements

- AMD GPU supported by ROCm
- **ROCm: v4.3 / v5.0**
  - See the [ROCm Installation Guide](#) for details.

The following ROCm libraries are required:

```
$ sudo apt install hipblas hipspase rocspase rocrand rocthrust rocsolver rocfft hipcub.  
↪rocprim rccl
```

### 3.2 Environment Variables

When building or running CuPy for ROCm, the following environment variables are effective.

- ROCM\_HOME: directory containing the ROCm software (e.g., /opt/rocm).

### 3.3 Docker

You can try running CuPy for ROCm using Docker.

```
$ docker run -it --device=/dev/kfd --device=/dev/dri --group-add video cupy/cupy-rocm
```

### 3.4 Installing Binary Packages

Wheels (precompiled binary packages) are available for Linux (x86\_64). Package names are different depending on your ROCm version.

ROCm	Command
v4.3	\$ pip install cupy-rocm-4-3
v5.0	\$ pip install cupy-rocm-5-0

## 3.5 Building CuPy for ROCm From Source

To build CuPy from source, set the `CUPY_INSTALL_USE_HIP`, `ROCM_HOME`, and `HCC_AMDGPU_TARGET` environment variables. (`HCC_AMDGPU_TARGET` is the ISA name supported by your GPU. Run `rocminfo` and use the value displayed in `Name:` line (e.g., `gfx900`). You can specify a comma-separated list of ISAs if you have multiple GPUs of different architectures.)

```
$ export CUPY_INSTALL_USE_HIP=1
$ export ROCM_HOME=/opt/rocm
$ export HCC_AMDGPU_TARGET=gfx906
$ pip install cupy
```

---

**Note:** If you don't specify the `HCC_AMDGPU_TARGET` environment variable, CuPy will be built for the GPU architectures available on the build host. This behavior is specific to ROCm builds; when building CuPy for NVIDIA CUDA, the build result is not affected by the host configuration.

---

## 3.6 Limitations

The following features are not available due to the limitation of ROCm or because that they are specific to CUDA:

- CUDA Array Interface
- cuTENSOR
- Handling extremely large arrays whose size is around 32-bit boundary (HIP is known to fail with sizes  $2^{**32}-1024$ )
- Atomic addition in FP16 (`cupy.ndarray.scatter_add` and `cupyx.scatter_add`)
- Multi-GPU FFT and FFT callback
- Some random number generation algorithms
- Several options in RawKernel/RawModule APIs: Jitify, dynamic parallelism
- Per-thread default stream

The following features are not yet supported:

- Sparse matrices (`cupyx.scipy.sparse`)
- cuDNN (hipDNN)
- Hermitian/symmetric eigenvalue solver (`cupy.linalg.eigh`)
- Polynomial roots (uses Hermitian/symmetric eigenvalue solver)
- Splines in `cupyx.scipy.interpolate` (`make_interp_spline`, `spline` modes of `RegularGridInterpolator/interp`), as they depend on sparse matrices.

The following features may not work in edge cases (e.g., some combinations of dtype):

---

**Note:** We are investigating the root causes of the issues. They are not necessarily CuPy's issues, but ROCm may have some potential bugs.

---

- `cupy.ndarray.__getitem__` (#4653)



- `cupy.ix_` (#4654)
- Some polynomial routines (#4758, #4759)
- `cupy.broadcast` (#4662)
- `cupy.convolve` (#4668)
- `cupy.correlate` (#4781)
- Some random sampling routines (`cupy.random`, #4770)
- `cupy.linalg.einsum`
- `cupyx.scipy.ndimage` and `cupyx.scipy.signal` (#4878, #4879, #4880)



## USER GUIDE

This user guide provides an overview of CuPy and explains its important features; details are found in *CuPy API Reference*.

### 4.1 Basics of CuPy

In this section, you will learn about the following things:

- Basics of `cupy.ndarray`
- The concept of *current device*
- host-device and device-device array transfer

#### 4.1.1 Basics of `cupy.ndarray`

CuPy is a GPU array backend that implements a subset of NumPy interface. In the following code, `cp` is an abbreviation of `cupy`, following the standard convention of abbreviating `numpy` as `np`:

```
>>> import numpy as np
>>> import cupy as cp
```

The `cupy.ndarray` class is at the core of CuPy and is a replacement class for NumPy's `numpy.ndarray`.

```
>>> x_gpu = cp.array([1, 2, 3])
```

`x_gpu` above is an instance of `cupy.ndarray`. As one can see, CuPy's syntax here is identical to that of NumPy. The main difference between `cupy.ndarray` and `numpy.ndarray` is that the CuPy arrays are allocated on the *current device*, which we will talk about later.

Most of the array manipulations are also done in the way similar to NumPy. Take the Euclidean norm (a.k.a L2 norm), for example. NumPy has `numpy.linalg.norm()` function that calculates it on CPU.

```
>>> x_cpu = np.array([1, 2, 3])
>>> l2_cpu = np.linalg.norm(x_cpu)
```

Using CuPy, we can perform the same calculations on GPU in a similar way:

```
>>> x_gpu = cp.array([1, 2, 3])
>>> l2_gpu = cp.linalg.norm(x_gpu)
```

CuPy implements many functions on `cupy.ndarray` objects. See the [reference](#) for the supported subset of NumPy API. Knowledge of NumPy will help you utilize most of the CuPy features. We, therefore, recommend you familiarize yourself with the [NumPy documentation](#).

## 4.1.2 Current Device

CuPy has a concept of a *current device*, which is the default GPU device on which the allocation, manipulation, calculation, etc., of arrays take place. Suppose ID of the current device is 0. In such a case, the following code would create an array `x_on_gpu0` on GPU 0.

```
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

To switch to another GPU device, use the [Device](#) context manager:

```
>>> with cp.cuda.Device(1):  
...     x_on_gpu1 = cp.array([1, 2, 3, 4, 5])  
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

All CuPy operations (except for multi-GPU features and device-to-device copy) are performed on the currently active device.

In general, CuPy functions expect that the array is on the same device as the current one. Passing an array stored on a non-current device may work depending on the hardware configuration but is generally discouraged as it may not be performant.

---

**Note:** If the array’s device and the current device mismatch, CuPy functions try to establish [peer-to-peer memory access](#) (P2P) between them so that the current device can directly read the array from another device. Note that P2P is available only when the topology permits it. If P2P is unavailable, such an attempt will fail with `ValueError`.

---

`cupy.ndarray.device` attribute indicates the device on which the array is allocated.

```
>>> with cp.cuda.Device(1):  
...     x = cp.array([1, 2, 3, 4, 5])  
>>> x.device  
<CUDA Device 1>
```

---

**Note:** When only one device is available, explicit device switching is not needed.

---

## 4.1.3 Current Stream

Associated with the concept of current devices are *current streams*, which help avoid explicitly passing streams in every single operation so as to keep the APIs pythonic and user-friendly. In CuPy, all CUDA operations such as data transfer (see the [Data Transfer](#) section) and kernel launches are enqueued onto the current stream, and the queued tasks on the same stream will be executed in serial (but *asynchronously* with respect to the host).

The default current stream in CuPy is CUDA’s null stream (i.e., stream 0). It is also known as the *legacy* default stream, which is unique per device. However, it is possible to change the current stream using the `cupy.cuda.Stream` API, please see [Accessing CUDA Functionalities](#) for example. The current stream in CuPy can be retrieved using `cupy.cuda.get_current_stream()`.

It is worth noting that CuPy’s current stream is managed on a *per thread, per device* basis, meaning that on different Python threads or different devices the current stream (if not the null stream) can be different.

## 4.1.4 Data Transfer

### Move arrays to a device

`cupy.asarray()` can be used to move a `numpy.ndarray`, a list, or any object that can be passed to `numpy.array()` to the current device:

```
>>> x_cpu = np.array([1, 2, 3])
>>> x_gpu = cp.asarray(x_cpu) # move the data to the current device.
```

`cupy.asarray()` can accept `cupy.ndarray`, which means we can transfer the array between devices with this function.

```
>>> with cp.cuda.Device(0):
...     x_gpu_0 = cp.ndarray([1, 2, 3]) # create an array in GPU 0
>>> with cp.cuda.Device(1):
...     x_gpu_1 = cp.asarray(x_gpu_0) # move the array to GPU 1
```

**Note:** `cupy.asarray()` does not copy the input array if possible. So, if you put an array of the current device, it returns the input object itself.

If we do copy the array in this situation, you can use `cupy.array()` with `copy=True`. Actually `cupy.asarray()` is equivalent to `cupy.array(arr, dtype, copy=False)`.

### Move array from a device to the host

Moving a device array to the host can be done by `cupy.asnumpy()` as follows:

```
>>> x_gpu = cp.array([1, 2, 3]) # create an array in the current device
>>> x_cpu = cp.asnumpy(x_gpu) # move the array to the host.
```

We can also use `cupy.ndarray.get()`:

```
>>> x_cpu = x_gpu.get()
```

## 4.1.5 Memory management

Check [Memory Management](#) for a detailed description of how memory is managed in CuPy using memory pools.

## 4.1.6 How to write CPU/GPU agnostic code

CuPy's compatibility with NumPy makes it possible to write CPU/GPU agnostic code. For this purpose, CuPy implements the `cupy.get_array_module()` function that returns a reference to `cupy` if any of its arguments resides on a GPU and `numpy` otherwise. Here is an example of a CPU/GPU agnostic function that computes  $\log 1p$ :

```
>>> # Stable implementation of log(1 + exp(x))
>>> def softplus(x):
...     xp = cp.get_array_module(x) # 'xp' is a standard usage in the community
...     print("Using:", xp.__name__)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

When you need to manipulate CPU and GPU arrays, an explicit data transfer may be required to move them to the same location – either CPU or GPU. For this purpose, CuPy implements two sister methods called `cupy.asnumpy()` and `cupy.asarray()`. Here is an example that demonstrates the use of both methods:

```
>>> x_cpu = np.array([1, 2, 3])
>>> y_cpu = np.array([4, 5, 6])
>>> x_cpu + y_cpu
array([5, 7, 9])
>>> x_gpu = cp.asarray(x_cpu)
>>> x_gpu + y_cpu
Traceback (most recent call last):
...
TypeError: Unsupported type <class 'numpy.ndarray'>
>>> cp.asnumpy(x_gpu) + y_cpu
array([5, 7, 9])
>>> cp.asnumpy(x_gpu) + cp.asnumpy(y_cpu)
array([5, 7, 9])
>>> x_gpu + cp.asarray(y_cpu)
array([5, 7, 9])
>>> cp.asarray(x_gpu) + cp.asarray(y_cpu)
array([5, 7, 9])
```

The `cupy.asnumpy()` method returns a NumPy array (array on the host), whereas `cupy.asarray()` method returns a CuPy array (array on the current device). Both methods can accept arbitrary input, meaning that they can be applied to any data that is located on either the host or device and can be converted to an array.

## 4.2 User-Defined Kernels

CuPy provides easy ways to define three types of CUDA kernels: elementwise kernels, reduction kernels and raw kernels. In this documentation, we describe how to define and call each kernels.

### 4.2.1 Basics of elementwise kernels

An elementwise kernel can be defined by the `ElementwiseKernel` class. The instance of this class defines a CUDA kernel which can be invoked by the `__call__` method of this instance.

A definition of an elementwise kernel consists of four parts: an input argument list, an output argument list, a loop body code, and the kernel name. For example, a kernel that computes a squared difference  $f(x, y) = (x - y)^2$  is defined as follows:

```
>>> squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

The argument lists consist of comma-separated argument definitions. Each argument definition consists of a *type specifier* and an *argument name*. Names of NumPy data types can be used as type specifiers.

---

**Note:** `n`, `i`, and names starting with an underscore `_` are reserved for the internal use.

---

The above kernel can be called on either scalars or arrays with broadcasting:

```
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cp.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

Output arguments can be explicitly specified (next to the input arguments):

```
>>> z = cp.empty((2, 5), dtype=np.float32)
>>> squared_diff(x, y, z)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
```

## 4.2.2 Type-generic kernels

If a type specifier is one character, then it is treated as a **type placeholder**. It can be used to define a type-generic kernels. For example, the above `squared_diff` kernel can be made type-generic as follows:

```
>>> squared_diff_generic = cp.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic')
```

Type placeholders of a same character in the kernel definition indicate the same type. The actual type of these placeholders is determined by the actual argument type. The `ElementwiseKernel` class first checks the output arguments and then the input arguments to determine the actual type. If no output arguments are given on the kernel invocation, then only the input arguments are used to determine the type.

The type placeholder can be used in the loop body code:

```
>>> squared_diff_generic = cp.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     """
...         T diff = x - y;
...         z = diff * diff;
...     """,
...     'squared_diff_generic')
```

More than one type placeholder can be used in a kernel definition. For example, the above kernel can be further made generic over multiple arguments:

```
>>> squared_diff_super_generic = cp.ElementwiseKernel(
...     'X x, Y y',
...     'Z z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_super_generic')
```

Note that this kernel requires the output argument explicitly specified, because the type `Z` cannot be automatically determined from the input arguments.

### 4.2.3 Raw argument specifiers

The `ElementwiseKernel` class does the indexing with broadcasting automatically, which is useful to define most elementwise computations. On the other hand, we sometimes want to write a kernel with manual indexing for some arguments. We can tell the `ElementwiseKernel` class to use manual indexing by adding the `raw` keyword preceding the type specifier.

We can use the special variable `i` and method `_ind.size()` for the manual indexing. `i` indicates the index within the loop. `_ind.size()` indicates total number of elements to apply the elementwise operation. Note that it represents the size **after** broadcast operation.

For example, a kernel that adds two vectors with reversing one of them can be written as follows:

```
>>> add_reverse = cp.ElementwiseKernel(
...     'T x, raw T y', 'T z',
...     'z = x + y[_ind.size() - i - 1]',
...     'add_reverse')
```

(Note that this is an artificial example and you can write such operation just by `z = x + y[::-1]` without defining a new kernel). A raw argument can be used like an array. The indexing operator `y[_ind.size() - i - 1]` involves an indexing computation on `y`, so `y` can be arbitrarily shaped and strode.

Note that raw arguments are not involved in the broadcasting. If you want to mark all arguments as raw, you must specify the `size` argument on invocation, which defines the value of `_ind.size()`.

### 4.2.4 Texture memory

Texture objects (*TextureObject*) can be passed to *ElementwiseKernel* with their type marked by a unique type placeholder distinct from any other types used in the same kernel, as its actual datatype is determined when populating the texture memory. The texture coordinates can be computed in the kernel by the per-thread loop index `i`.

### 4.2.5 Reduction kernels

Reduction kernels can be defined by the *ReductionKernel* class. We can use it by defining four parts of the kernel code:

1. Identity value: This value is used for the initial value of reduction.
2. Mapping expression: It is used for the pre-processing of each element to be reduced.
3. Reduction expression: It is an operator to reduce the multiple mapped values. The special variables `a` and `b` are used for its operands.
4. Post mapping expression: It is used to transform the resulting reduced values. The special variable `a` is used as its input. Output should be written to the output parameter.

`ReductionKernel` class automatically inserts other code fragments that are required for an efficient and flexible reduction implementation.

For example, L2 norm along specified axes can be written as follows:



```
>>> l2norm_kernel = cp.ReductionKernel(
...     'T x', # input params
...     'T y', # output params
...     'x * x', # map
...     'a + b', # reduce
...     'y = sqrt(a)', # post-reduction map
...     '0', # identity value
...     'l2norm' # kernel name
... )
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

**Note:** `raw` specifier is restricted for usages that the axes to be reduced are put at the head of the shape. It means, if you want to use `raw` specifier for at least one argument, the `axis` argument must be `0` or a contiguous increasing sequence of integers starting from `0`, like `(0, 1)`, `(0, 1, 2)`, etc.

**Note:** Texture memory is not yet supported in `ReductionKernel`.

## 4.2.6 Raw kernels

Raw kernels can be defined by the `RawKernel` class. By using raw kernels, you can define kernels from raw CUDA source.

`RawKernel` object allows you to call the kernel with CUDA's `cuLaunchKernel` interface. In other words, you have control over grid size, block size, shared memory size and stream.

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y)) # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Raw kernels operating on complex-valued arrays can be created as well:

```
>>> complex_kernel = cp.RawKernel(r'''
... #include <cupy/complex.cuh>
```

(continues on next page)

(continued from previous page)

```

... extern "C" __global__
... void my_func(const complex<float>* x1, const complex<float>* x2,
...             complex<float>* y, float a) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + a * x2[tid];
... }
... ''' , 'my_func')
>>> x1 = cupy.arange(25, dtype=cupy.complex64).reshape(5, 5)
>>> x2 = 1j*cupy.arange(25, dtype=cupy.complex64).reshape(5, 5)
>>> y = cupy.zeros((5, 5), dtype=cupy.complex64)
>>> complex_kernel((5,), (5,), (x1, x2, y, cupy.float32(2.0))) # grid, block and
↳arguments
>>> y
array([[ 0. +0.j,  1. +2.j,  2. +4.j,  3. +6.j,  4. +8.j],
       [ 5. +10.j,  6. +12.j,  7. +14.j,  8. +16.j,  9. +18.j],
       [10. +20.j, 11. +22.j, 12. +24.j, 13. +26.j, 14. +28.j],
       [15. +30.j, 16. +32.j, 17. +34.j, 18. +36.j, 19. +38.j],
       [20. +40.j, 21. +42.j, 22. +44.j, 23. +46.j, 24. +48.j]],
      dtype=complex64)

```

Note that while we encourage the usage of `complex<T>` types for complex numbers (available by including `<cupy/complex.cuh>` as shown above), for CUDA codes already written using functions from `cuComplex.h` there is no need to make the conversion yourself: just set the option `translate_cucomplex=True` when creating a [RawKernel](#) instance.

The CUDA kernel attributes can be retrieved by either accessing the `attributes` dictionary, or by accessing the [RawKernel](#) object's attributes directly; the latter can also be used to set certain attributes:

```

>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''' , 'my_add')
>>> add_kernel.attributes
{'max_threads_per_block': 1024, 'shared_size_bytes': 0, 'const_size_bytes': 0, 'local_
↳size_bytes': 0, 'num_regs': 10, 'ptx_version': 70, 'binary_version': 70, 'cache_mode_ca
↳': 0, 'max_dynamic_shared_size_bytes': 49152, 'preferred_shared_memory_carveout': -1}
>>> add_kernel.max_dynamic_shared_size_bytes
49152
>>> add_kernel.max_dynamic_shared_size_bytes = 50000 # set a new value for the
↳attribute
>>> add_kernel.max_dynamic_shared_size_bytes
50000

```

Dynamical parallelism is supported by [RawKernel](#). You just need to provide the linking flag (such as `-dc`) to [RawKernel](#)'s `options` argument. The static CUDA device runtime library (`cudadevrt`) is automatically discovered by CuPy. For further detail, see [CUDA Toolkit's documentation](#).

Accessing texture (surface) memory in [RawKernel](#) is supported via CUDA Runtime's Texture (Surface) Object API, see the documentation for [TextureObject](#) ([SurfaceObject](#)) as well as CUDA C Programming Guide. For using the Texture Reference API, which is marked as deprecated as of CUDA Toolkit 10.1, see the introduction to [RawModule](#) below.

If your kernel relies on the C++ std library headers such as `<type_traits>`, it is likely you will encounter compilation errors. In this case, try enabling CuPy's [Jitify](#) support by setting `jitify=True` when creating the [RawKernel](#) instance. It provides basic C++ std support to remedy common errors.

---

**Note:** The kernel does not have return values. You need to pass both input arrays and output arrays as arguments.

---



---

**Note:** When using `printf()` in your CUDA kernel, you may need to synchronize the stream to see the output. You can use `cupy.cuda.Stream.null.synchronize()` if you are using the default stream.

---



---

**Note:** In all of the examples above, we declare the kernels in an `extern "C"` block, indicating that the C linkage is used. This is to ensure the kernel names are not mangled so that they can be retrived by name.

---

## 4.2.7 Kernel arguments

Python primitive types and NumPy scalars are passed to the kernel by value. Array arguments (pointer arguments) have to be passed as CuPy ndarrays. No validation is performed by CuPy for arguments passed to the kernel, including types and number of arguments.

Especially note that when passing a CuPy [ndarray](#), its `dtype` should match with the type of the argument declared in the function signature of the CUDA source code (unless you are casting arrays intentionally).

As an example, `cupy.float32` and `cupy.uint64` arrays must be passed to the argument typed as `float*` and `unsigned long long*`, respectively. CuPy does not directly support arrays of non-primitive types such as `float3`, but nothing prevents you from casting a `float*` or `void*` to a `float3*` in a kernel.

Python primitive types, `int`, `float`, `complex` and `bool` map to `long long`, `double`, `cuDoubleComplex` and `bool`, respectively.

NumPy scalars (`numpy.generic`) and NumPy arrays (`numpy.ndarray`) **of size one** are passed to the kernel by value. This means that you can pass by value any base NumPy types such as `numpy.int8` or `numpy.float64`, provided the kernel arguments match in size. You can refer to this table to match CuPy/NumPy dtype and CUDA types:

CuPy/NumPy type	Corresponding kernel types	itemsize (bytes)
<code>bool</code>	<code>bool</code>	1
<code>int8</code>	<code>char</code> , <code>signed char</code>	1
<code>int16</code>	<code>short</code> , <code>signed short</code>	2
<code>int32</code>	<code>int</code> , <code>signed int</code>	4
<code>int64</code>	<code>long long</code> , <code>signed long long</code>	8
<code>uint8</code>	<code>unsigned char</code>	1
<code>uint16</code>	<code>unsigned short</code>	2
<code>uint32</code>	<code>unsigned int</code>	4
<code>uint64</code>	<code>unsigned long long</code>	8
<code>float16</code>	<code>half</code>	2
<code>float32</code>	<code>float</code>	4
<code>float64</code>	<code>double</code>	8
<code>complex64</code>	<code>float2</code> , <code>cuFloatComplex</code> , <code>complex&lt;float&gt;</code>	8
<code>complex128</code>	<code>double2</code> , <code>cuDoubleComplex</code> , <code>complex&lt;double&gt;</code>	16

The CUDA standard guarantees that the size of fundamental types on the host and device always match. The `itemsize` of `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, `long`, `signed long` and `unsigned long` are however platform

dependent. To pass any CUDA vector builtins such as `float3` or any other user defined structure as kernel arguments (provided it matches the device-side kernel parameter type), see [Custom user types](#) below.

## 4.2.8 Custom user types

It is possible to use custom types (composite types such as structures and structures of structures) as kernel arguments by defining a custom NumPy dtype. When doing this, it is your responsibility to match host and device structure memory layout. The CUDA standard guarantees that the size of fundamental types on the host and device always match. It may however impose device alignment requirements on composite types. This means that for composite types the struct member offsets may be different from what you might expect.

When a kernel argument is passed by value, the CUDA driver will copy exactly `sizeof(param_type)` bytes starting from the beginning of the NumPy object data pointer, where `param_type` is the parameter type in your kernel. You have to match `param_type`'s memory layout (ex: size, alignment and struct padding/packing) by defining a corresponding NumPy dtype.

For builtin CUDA vector types such as `int2` and `double4` and other packed structures with named members you can directly define such NumPy dtypes as the following:

```
>>> import numpy as np
>>> names = ['x', 'y', 'z']
>>> types = [np.float32]*3
>>> float3 = np.dtype({'names': names, 'formats': types})
>>> arg = np.random.rand(3).astype(np.float32).view(float3)
>>> print(arg)
[(0.9940819, 0.62873816, 0.8953669)]
>>> arg['x'] = 42.0
>>> print(arg)
[(42., 0.62873816, 0.8953669)]
```

Here `arg` can be used directly as a kernel argument. When there is no need to name fields you may prefer this syntax to define packed structures such as vectors or matrices:

```
>>> import numpy as np
>>> float5x5 = np.dtype({'names': ['dummy'], 'formats': [(np.float32,(5,5))]}))
>>> arg = np.random.rand(25).astype(np.float32).view(float5x5)
>>> print(arg.itemsize)
100
```

Here `arg` represents a 100-byte scalar (i.e. a NumPy array of size 1) that can be passed by value to any kernel. Kernel parameters are passed by value in a dedicated 4kB memory bank which has its own cache with broadcast. Upper bound for total kernel parameters size is thus 4kB (see [this link](#)). It may be important to note that this dedicated memory bank is not shared with the device `__constant__` memory space.

For now, CuPy offers no helper routines to create user defined composite types. Such composite types can however be built recursively using NumPy dtype *offsets* and *itemsize* capabilities, see [cupy/examples/custum\\_struct](#) for examples of advanced usage.

**Warning:** You cannot directly pass static arrays as kernel arguments with the type `arg[N]` syntax where `N` is a compile time constant. The signature of `__global__ void kernel(float arg[5])` is seen as `__global__ void kernel(float* arg)` by the compiler. If you want to pass five floats to the kernel by value you need to define a custom structure `struct float5 { float val[5]; }` and modify the kernel signature to `__global__ void kernel(float5 arg)`.

## 4.2.9 Raw modules

For dealing a large raw CUDA source or loading an existing CUDA binary, the `RawModule` class can be more handy. It can be initialized either by a CUDA source code, or by a path to the CUDA binary. It accepts most of the arguments as in `RawKernel`. The needed kernels can then be retrieved by calling the `get_function()` method, which returns a `RawKernel` instance that can be invoked as discussed above.

```
>>> loaded_from_source = r'''
... extern "C"{
...
... __global__ void test_sum(const float* x1, const float* x2, float* y, \
...                          unsigned int N)
... {
...     unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     if (tid < N)
...     {
...         y[tid] = x1[tid] + x2[tid];
...     }
... }
...
... __global__ void test_multiply(const float* x1, const float* x2, float* y, \
...                               unsigned int N)
... {
...     unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     if (tid < N)
...     {
...         y[tid] = x1[tid] * x2[tid];
...     }
... }
... }'''
>>> module = cp.RawModule(code=loaded_from_source)
>>> ker_sum = module.get_function('test_sum')
>>> ker_times = module.get_function('test_multiply')
>>> N = 10
>>> x1 = cp.arange(N**2, dtype=cp.float32).reshape(N, N)
>>> x2 = cp.ones((N, N), dtype=cp.float32)
>>> y = cp.zeros((N, N), dtype=cp.float32)
>>> ker_sum((N,), (N,), (x1, x2, y, N**2)) # y = x1 + x2
>>> assert cp.allclose(y, x1 + x2)
>>> ker_times((N,), (N,), (x1, x2, y, N**2)) # y = x1 * x2
>>> assert cp.allclose(y, x1 * x2)
```

The instruction above for using complex numbers in `RawKernel` also applies to `RawModule`.

For CUDA kernels that need to access global symbols, such as constant memory, the `get_global()` method can be used, see its documentation for further detail.

Note that the deprecated API `cupy.RawModule.get_texref()` has been removed since CuPy vX.X due to the removal of texture reference support from CUDA.

To support C++ template kernels, `RawModule` additionally provide a `name_expressions` argument. A list of template specializations should be provided, so that the corresponding kernels can be generated and retrieved by type:

```
>>> code = r'''
... template<typename T>
... __global__ void fx3(T* arr, int N) {
...     unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
...     if (tid < N) {
...         arr[tid] = arr[tid] * 3;
...     }
... }
... '''
>>>
>>> name_exp = ['fx3<float>', 'fx3<double>']
>>> mod = cp.RawModule(code=code, options=('-std=c++11',),
...     name_expressions=name_exp)
>>> ker_float = mod.get_function(name_exp[0]) # compilation happens here
>>> N=10
>>> a = cp.arange(N, dtype=cp.float32)
>>> ker_float((1,), (N,), (a, N))
>>> a
array([ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27.], dtype=float32)
>>> ker_double = mod.get_function(name_exp[1])
>>> a = cp.arange(N, dtype=cp.float64)
>>> ker_double((1,), (N,), (a, N))
>>> a
array([ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27.])
```

---

**Note:** The name expressions used to both initialize a [RawModule](#) instance and retrieve the kernels are the original (*unmangled*) kernel names with all template parameters unambiguously specified. The name mangling and demangling are handled under the hood so that users do not need to worry about it.

---

## 4.2.10 Kernel fusion

`cupy.fuse()` is a decorator that fuses functions. This decorator can be used to define an elementwise or reduction kernel more easily than [ElementwiseKernel](#) or [ReductionKernel](#).

By using this decorator, we can define the `squared_diff` kernel as follows:

```
>>> @cp.fuse()
... def squared_diff(x, y):
...     return (x - y) * (x - y)
```

The above kernel can be called on either scalars, NumPy arrays or CuPy arrays likes the original function.

```
>>> x_cp = cp.arange(10)
>>> y_cp = cp.arange(10)[::-1]
>>> squared_diff(x_cp, y_cp)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
>>> x_np = np.arange(10)
>>> y_np = np.arange(10)[::-1]
>>> squared_diff(x_np, y_np)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
```

At the first function call, the fused function analyzes the original function based on the abstracted information of arguments (e.g. their dtypes and ndims) and creates and caches an actual CUDA kernel. From the second function call with the same input types, the fused function calls the previously cached kernel, so it is highly recommended to reuse the same decorated functions instead of decorating local functions that are defined multiple times.

`cupy.fuse()` also supports simple reduction kernel.

```
>>> @cp.fuse()
... def sum_of_products(x, y):
...     return cp.sum(x * y, axis = -1)
```

You can specify the kernel name by using the `kernel_name` keyword argument as follows:

```
>>> @cp.fuse(kernel_name='squared_diff')
... def squared_diff(x, y):
...     return (x - y) * (x - y)
```

**Note:** Currently, `cupy.fuse()` can fuse only simple elementwise and reduction operations. Most other routines (e.g. `cupy.matmul()`, `cupy.reshape()`) are not supported.

## 4.2.11 JIT kernel definition

The `cupyx.jit.rawkernel` decorator can create raw CUDA kernels from Python functions.

In this section, a Python function wrapped with the decorator is called a *target function*.

A target function consists of elementary scalar operations, and users have to manage how to parallelize them. CuPy's array operations which automatically parallelize operations (e.g., `add()`, `sum()`) are not supported. If a custom kernel based on such array functions is desired, please refer to the *Kernel fusion* section.

### Basic Usage

Here is a short example for how to write a `cupyx.jit.rawkernel` to copy the values from `x` to `y` using a grid-stride loop:

```
>>> from cupyx import jit
>>>
>>> @jit.rawkernel()
... def elementwise_copy(x, y, size):
...     tid = jit.blockIdx.x * jit.blockDim.x + jit.threadIdx.x
...     ntid = jit.gridDim.x * jit.blockDim.x
...     for i in range(tid, size, ntid):
...         y[i] = x[i]

>>> size = cupy.uint32(2 ** 22)
>>> x = cupy.random.normal(size=(size,), dtype=cupy.float32)
>>> y = cupy.empty((size,), dtype=cupy.float32)

>>> elementwise_copy((128,), (1024,), (x, y, size)) # RawKernel style
>>> assert (x == y).all()
```

(continues on next page)

(continued from previous page)

```
>>> elementwise_copy[128, 1024](x, y, size) # Numba style
>>> assert (x == y).all()
```

Both styles to launch the kernel, as shown above, are supported. The first two entries are the grid and block sizes, respectively. `grid` (`RawKernel` style `(128,)` or Numba style `[128]`) is the sizes of the grid, i.e., the numbers of blocks in each dimension; `block` (`(1024,)` or `[1024]`) is the dimensions of each thread block, please refer to [cupyx.jit.\\_interface.\\_JitRawKernel](#) for details. Launching a CUDA kernel on a GPU with pre-determined grid/block sizes requires basic understanding in the [CUDA Programming Model](#).

The compilation will be deferred until the first function call. CuPy's JIT compiler infers the types of arguments at the call time, and will cache the compiled kernels for speeding up any subsequent calls.

See [Custom kernels](#) for a full list of API.

## Basic Design

CuPy's JIT compiler generates CUDA code via Python AST. We decided not to use Python bytecode to analyze the target function to avoid performance degradation. The CUDA source code generated from the Python bytecode will not effectively optimized by CUDA compiler, because for-loops and other control statements of the target function are fully transformed to jump instruction when converting the target function to bytecode.

## Typing rule

The types of local variables are inferred at the first assignment in the function. The first assignment must be done at the top-level of the function; in other words, it must *not* be in `if/else` bodies or `for`-loops.

## Limitations

JIT does not work inside Python's interactive interpreter (REPL) as the compiler needs to get the source code of the target function.

## 4.3 Accessing CUDA Functionalities

### 4.3.1 Streams and Events

In this section we discuss basic usages for CUDA streams and events. For the API reference please see [Streams and events](#). For their roles in the CUDA programming model, please refer to [CUDA Programming Guide](#).

CuPy provides high-level Python APIs [Stream](#) and [Event](#) for creating streams and events, respectively. Data copies and kernel launches are enqueued onto the [Current Stream](#), which can be queried via [get\\_current\\_stream\(\)](#) and changed either by setting up a context manager:

```
>>> import numpy as np
>>>
>>> a_np = np.arange(10)
>>> s = cp.cuda.Stream()
>>> with s:
...     a_cp = cp.asarray(a_np) # H2D transfer on stream s
...     b_cp = cp.sum(a_cp)     # kernel launched on stream s
...     assert s == cp.cuda.get_current_stream()
```

(continues on next page)



(continued from previous page)

```
...
>>> # fall back to the previous stream in use (here the default stream)
>>> # when going out of the scope of s
```

or by using the `use()` method:

```
>>> s = cp.cuda.Stream()
>>> s.use() # any subsequent operations are done on steam s
<Stream ... (device ...)>
>>> b_np = cp.asnumpy(b_cp)
>>> assert s == cp.cuda.get_current_stream()
>>> cp.cuda.Stream.null.use() # fall back to the default (null) stream
<Stream 0 (device -1)>
>>> assert cp.cuda.Stream.null == cp.cuda.get_current_stream()
```

Events can be created either manually or through the `record()` method. `Event` objects can be used for timing GPU activities (via `get_elapsed_time()`) or setting up inter-stream dependencies:

```
>>> e1 = cp.cuda.Event()
>>> e1.record()
>>> a_cp = b_cp * a_cp + 8
>>> e2 = cp.cuda.get_current_stream().record()
>>>
>>> # set up a stream order
>>> s2 = cp.cuda.Stream()
>>> s2.wait_event(e2)
>>> with s2:
...     # the a_cp is guaranteed updated when this copy (on s2) starts
...     a_np = cp.asnumpy(a_cp)
>>>
>>> # timing
>>> e2.synchronize()
>>> t = cp.cuda.get_elapsed_time(e1, e2) # only include the compute time, not the copy_
↳time
```

Just like the `Device` objects, `Stream` and `Event` objects can also be used for synchronization.

---

**Note:** In CuPy, the `Stream` objects are managed on the per thread, per device basis.

---



---

**Note:** On NVIDIA GPUs, there are two stream singleton objects `null` and `ptds`, referred to as the *legacy* default stream and the *per-thread* default stream, respectively. CuPy uses the former as default when no user-defined stream is in use. To change this behavior, set the environment variable `CUPY_CUDA_PER_THREAD_DEFAULT_STREAM` to 1, see *Environment variables*. This is not applicable to AMD GPUs.

---

To interoperate with streams created in other Python libraries, CuPy provides the `ExternalStream` API to wrap an existing stream pointer (given as a Python `int`). See *Interoperability* for details.

### 4.3.2 CUDA Driver and Runtime API

Under construction. Please see [Runtime API](#) for the API reference.

## 4.4 Fast Fourier Transform with CuPy

CuPy covers the full Fast Fourier Transform (FFT) functionalities provided in NumPy ([`cupy.fft`](#)) and a subset in SciPy ([`cupyx.scipy.fft`](#)). In addition to those high-level APIs that can be used as is, CuPy provides additional features to

1. access advanced routines that [cuFFT](#) offers for NVIDIA GPUs,
2. control better the performance and behavior of the FFT routines.

Some of these features are *experimental* (subject to change, deprecation, or removal, see [API Compatibility Policy](#)) or may be absent in [hipFFT/rocFFT](#) targeting AMD GPUs.

### 4.4.1 SciPy FFT backend

Since SciPy v1.4, a backend mechanism is provided so that users can register different FFT backends and use SciPy's API to perform the actual transform with the target backend, such as CuPy's [`cupyx.scipy.fft`](#) module. For a one-time only usage, a context manager [`scipy.fft.set\_backend\(\)`](#) can be used:

```
import cupy as cp
import cupyx.scipy.fft as cufft
import scipy.fft

a = cp.random.random(100).astype(cp.complex64)
with scipy.fft.set_backend(cufft):
    b = scipy.fft.fft(a) # equivalent to cufft.fft(a)
```

However, such usage can be tedious. Alternatively, users can register a backend through [`scipy.fft.register\_backend\(\)`](#) or [`scipy.fft.set\_global\_backend\(\)`](#) to avoid using context managers:

```
import cupy as cp
import cupyx.scipy.fft as cufft
import scipy.fft
scipy.fft.set_global_backend(cufft)

a = cp.random.random(100).astype(cp.complex64)
b = scipy.fft.fft(a) # equivalent to cufft.fft(a)
```

---

**Note:** Please refer to [SciPy FFT documentation](#) for further information.

---

---

**Note:** To use the backend together with an explicit `plan` argument requires SciPy version 1.5.0 or higher. See below for how to create FFT plans.

---

## 4.4.2 User-managed FFT plans

For performance reasons, users may wish to create, reuse, and manage the FFT plans themselves. CuPy provides a high-level *experimental* API `get_fft_plan()` for this need. Users specify the transform to be performed as they would with most of the high-level FFT APIs, and a plan will be generated based on the input.

```
import cupy as cp
from cupyx.scipy.fft import get_fft_plan

a = cp.random.random((4, 64, 64)).astype(cp.complex64)
plan = get_fft_plan(a, axes=(1, 2), value_type='C2C') # for batched, C2C, 2D transform
```

The returned plan can be used either explicitly as an argument with the `cupyx.scipy.fft` APIs:

```
import cupyx.scipy.fft

# the rest of the arguments must match those used when generating the plan
out = cupyx.scipy.fft.fft2(a, axes=(1, 2), plan=plan)
```

or as a context manager for the `cupy.fft` APIs:

```
with plan:
    # the arguments must match those used when generating the plan
    out = cp.fft.fft2(a, axes=(1, 2))
```

## 4.4.3 FFT plan cache

However, there are occasions when users may *not* want to manage the FFT plans by themselves. Moreover, plans could also be reused internally in CuPy's routines, to which user-managed plans would not be applicable. Therefore, starting CuPy v8 we provide a built-in plan cache, enabled by default. The plan cache is done on a *per device, per thread* basis, and can be retrieved by the `get_plan_cache()` API.

```
>>> import cupy as cp
>>>
>>> cache = cp.fft.config.get_plan_cache()
>>> cache.show_info()
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 0 / 16 (counts)
current / max memsize: 0 / (unlimited) (bytes)
hits / misses: 0 / 0 (counts)

cached plans (most recently used first):

>>> # perform a transform, which would generate a plan and cache it
>>> a = cp.random.random((4, 64, 64))
>>> out = cp.fft.fftn(a, axes=(1, 2))
>>> cache.show_info() # hit = 0
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 1 / 16 (counts)
current / max memsize: 262144 / (unlimited) (bytes)
hits / misses: 0 / 1 (counts)
```

(continues on next page)

(continued from previous page)

```

cached plans (most recently used first):
key: ((64, 64), (64, 64), 1, 4096, (64, 64), 1, 4096, 105, 4, 'C', 2, None), plan type:↳
↳PlanNd, memory usage: 262144

>>> # perform the same transform again, the plan is looked up from cache and reused
>>> out = cp.fft.fftn(a, axes=(1, 2))
>>> cache.show_info() # hit = 1
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 1 / 16 (counts)
current / max memsize: 262144 / (unlimited) (bytes)
hits / misses: 1 / 1 (counts)

cached plans (most recently used first):
key: ((64, 64), (64, 64), 1, 4096, (64, 64), 1, 4096, 105, 4, 'C', 2, None), plan type:↳
↳PlanNd, memory usage: 262144

>>> # clear the cache
>>> cache.clear()
>>> cp.fft.config.show_plan_cache_info() # = cache.show_info(), for all devices
===== cuFFT plan cache info (all devices) =====
----- cuFFT plan cache (device 0) -----
cache enabled? True
current / max size   : 0 / 16 (counts)
current / max memsize: 0 / (unlimited) (bytes)
hits / misses: 0 / 0 (counts)

cached plans (most recently used first):

```

The returned PlanCache object has other methods for finer control, such as setting the cache size (either by counts or by memory usage). If the size is set to 0, the cache is disabled. Please refer to its documentation for more detail.

**Note:** As shown above each FFT plan has an associated working area allocated. If an out-of-memory error happens, one may want to inspect, clear, or limit the plan cache.

**Note:** The plans returned by `get_fft_plan()` are not cached.

#### 4.4.4 FFT callbacks

cuFFT provides FFT callbacks for merging pre- and/or post- processing kernels with the FFT routines so as to reduce the access to global memory. This capability is supported *experimentally* by CuPy. Users need to supply custom load and/or store kernels as strings, and set up a context manager via `set_cufft_callbacks()`. Note that the load (store) kernel pointer has to be named as `d_loadCallbackPtr` (`d_storeCallbackPtr`).

```

import cupy as cp

# a load callback that overwrites the input array to 1

```

(continues on next page)

(continued from previous page)

```
code = r'''
__device__ cufftComplex CB_ConvertInputC(
    void *dataIn,
    size_t offset,
    void *callerInfo,
    void *sharedPtr)
{
    cufftComplex x;
    x.x = 1.;
    x.y = 0.;
    return x;
}
__device__ cufftCallbackLoadC d_loadCallbackPtr = CB_ConvertInputC;
'''

a = cp.random.random((64, 128, 128)).astype(cp.complex64)

# this fftn call uses callback
with cp.fft.config.set_cufft_callbacks(cb_load=code):
    b = cp.fft.fftn(a, axes=(1,2))

# this does not use
c = cp.fft.fftn(cp.ones(shape=a.shape, dtype=cp.complex64), axes=(1,2))

# result agrees
assert cp.allclose(b, c)

# "static" plans are also cached, but are distinct from their no-callback counterparts
cp.fft.config.get_plan_cache().show_info()
```

**Note:** Internally, this feature requires recompiling a Python module *for each distinct pair* of load and store kernels. Therefore, the first invocation will be very slow, and this cost is amortized if the callbacks can be reused in the subsequent calculations. The compiled modules are cached on disk, with a default position `$HOME/.cupy/callback_cache` that can be changed by the environment variable `CUPY_CACHE_DIR`.

## 4.4.5 Multi-GPU FFT

CuPy currently provides two kinds of *experimental* support for multi-GPU FFT.

**Warning:** Using multiple GPUs to perform FFT is not guaranteed to be more performant. The rule of thumb is if the transform fits in 1 GPU, you should avoid using multiple.

The first kind of support is with the high-level `fftn()` and `ifftn()` APIs, which requires the input array to reside on one of the participating GPUs. The multi-GPU calculation is done under the hood, and by the end of the calculation the result again resides on the device where it started. Currently only 1D complex-to-complex (C2C) transform is supported; complex-to-real (C2R) or real-to-complex (R2C) transforms (such as `rfftn()` and friends) are not. The transform can be either batched (batch size > 1) or not (batch size = 1).

```
import cupy as cp

cp.fft.config.use_multi_gpus = True
cp.fft.config.set_cufft_gpus([0, 1]) # use GPU 0 & 1

shape = (64, 64) # batch size = 64
dtype = cp.complex64
a = cp.random.random(shape).astype(dtype) # reside on GPU 0

b = cp.fft.fft(a) # computed on GPU 0 & 1, reside on GPU 0
```

If you need to perform 2D/3D transforms (ex: `fftn()`) instead of 1D (ex: `fft()`), it would likely still work, but in this particular use case it loops over the transformed axes under the hood (which is exactly what is done in NumPy too), which could lead to suboptimal performance.

The second kind of usage is to use the low-level, *private* CuPy APIs. You need to construct a `Plan1d` object and use it as if you are programming in C/C++ with `cuFFT`. Using this approach, your input array can reside on the host as a `numpy.ndarray` so that its size can be much larger than what a single GPU can accommodate, which is one of the main reasons to run multi-GPU FFT.

```
import numpy as np
import cupy as cp

# no need to touch cp.fft.config, as we are using low-level API

shape = (64, 64)
dtype = np.complex64
a = np.random.random(shape).astype(dtype) # reside on CPU

if len(shape) == 1:
    batch = 1
    nx = shape[0]
elif len(shape) == 2:
    batch = shape[0]
    nx = shape[1]

# compute via cuFFT
cufft_type = cp.cuda.cufft.CUFFT_C2C # single-precision c2c
plan = cp.cuda.cufft.Plan1d(nx, cufft_type, batch, devices=[0,1])
out_cp = np.empty_like(a) # output on CPU
plan.fft(a, out_cp, cufft.CUFFT_FORWARD)

out_np = numpy.fft.fft(a) # use NumPy's fft
# np.fft.fft always returns np.complex128
if dtype is numpy.complex64:
    out_np = out_np.astype(dtype)

# check result
assert np.allclose(out_cp, out_np, rtol=1e-4, atol=1e-7)
```

For this use case, please consult the `cuFFT` documentation on multi-GPU transform for further detail.

---

**Note:** The multi-GPU plans are cached if auto-generated via the high-level APIs, but not if manually generated via

the low-level APIs.

#### 4.4.6 Half-precision FFT

cuFFT provides `cufftXtMakePlanMany` and `cufftXtExec` routines to support a wide range of FFT needs, including 64-bit indexing and half-precision FFT. CuPy provides an *experimental* support for this capability via the new (though *private*) `XtPlanNd` API. For half-precision FFT, on supported hardware it can be twice as fast than its single-precision counterpart. NumPy does not yet provide the necessary infrastructure for half-precision complex numbers (i.e., `numpy.complex32`), though, so the steps for this feature is currently a bit more involved than common cases.

```
import cupy as cp
import numpy as np

shape = (1024, 256, 256) # input array shape
idtype = odtype = edtype = 'E' # = numpy.complex32 in the future

# store the input/output arrays as fp16 arrays twice as long, as complex32 is not yet
# available
a = cp.random.random((shape[0], shape[1], 2*shape[2])).astype(cp.float16)
out = cp.empty_like(a)

# FFT with cuFFT
plan = cp.cuda.cufft.XtPlanNd(shape[1:],
                              shape[1:], 1, shape[1]*shape[2], idtype,
                              shape[1:], 1, shape[1]*shape[2], odtype,
                              shape[0], edtype,
                              order='C', last_axis=-1, last_size=None)

plan.fft(a, out, cp.cuda.cufft.CUFFT_FORWARD)

# FFT with NumPy
a_np = cp.asnumpy(a).astype(np.float32) # upcast
a_np = a_np.view(np.complex64)
out_np = np.fft.fftn(a_np, axes=(-2,-1))
out_np = np.ascontiguousarray(out_np).astype(np.complex64) # downcast
out_np = out_np.view(np.float32)
out_np = out_np.astype(np.float16)

# don't worry about accuracy for now, as we probably lost a lot during casting
print('ok' if cp.mean(cp.abs(out - cp.asarray(out_np))) < 0.1 else 'not ok')
```

The 64-bit indexing support for all high-level FFT APIs is planned for a future CuPy release.

## 4.5 Memory Management

CuPy uses *memory pool* for memory allocations by default. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

There are two different memory pools in CuPy:

- Device memory pool (GPU device memory), which is used for GPU memory allocations.
- Pinned memory pool (non-swappable CPU memory), which is used during CPU-to-GPU data transfer.

**Attention:** When you monitor the memory usage (e.g., using `nvidia-smi` for GPU memory or `ps` for CPU memory), you may notice that memory not being freed even after the array instance become out of scope. This is an expected behavior, as the default memory pool “caches” the allocated memory blocks.

See *Low-level CUDA support* for the details of memory management APIs.

For using pinned memory more conveniently, we also provide a few high-level APIs in the `cupyx` namespace, including `cupyx.empty_pinned()`, `cupyx.empty_like_pinned()`, `cupyx.zeros_pinned()`, and `cupyx.zeros_like_pinned()`. They return NumPy arrays backed by pinned memory. If CuPy’s pinned memory pool is in use, the pinned memory is allocated from the pool.

---

**Note:** CuPy v8 and above provides a *FFT plan cache* that could use a portion of device memory if FFT and related functions are used. The memory taken can be released by shrinking or disabling the cache.

---

### 4.5.1 Memory Pool Operations

The memory pool instance provides statistics about memory allocation. To access the default memory pool instance, use `cupy.get_default_memory_pool()` and `cupy.get_default_pinned_memory_pool()`. You can also free all unused memory blocks hold in the memory pool. See the example code below for details:

```
import cupy
import numpy

mempool = cupy.get_default_memory_pool()
pinned_mempool = cupy.get_default_pinned_memory_pool()

# Create an array on CPU.
# NumPy allocates 400 bytes in CPU (not managed by CuPy memory pool).
a_cpu = numpy.ndarray(100, dtype=numpy.float32)
print(a_cpu.nbytes)                # 400

# You can access statistics of these memory pools.
print(mempool.used_bytes())         # 0
print(mempool.total_bytes())        # 0
print(pinned_mempool.n_free_blocks()) # 0

# Transfer the array from CPU to GPU.
# This allocates 400 bytes from the device memory pool, and another 400
# bytes from the pinned memory pool. The allocated pinned memory will be
# released just after the transfer is complete. Note that the actual
```

(continues on next page)



(continued from previous page)

```

# allocation size may be rounded to larger value than the requested size
# for performance.
a = cupy.array(a_cpu)
print(a.nbytes)                # 400
print(mempool.used_bytes())    # 512
print(mempool.total_bytes())   # 512
print(pinned_mempool.n_free_blocks()) # 1

# When the array goes out of scope, the allocated device memory is released
# and kept in the pool for future reuse.
a = None # (or `del a`)
print(mempool.used_bytes())    # 0
print(mempool.total_bytes())   # 512
print(pinned_mempool.n_free_blocks()) # 1

# You can clear the memory pool by calling `free_all_blocks`.
mempool.free_all_blocks()
pinned_mempool.free_all_blocks()
print(mempool.used_bytes())    # 0
print(mempool.total_bytes())   # 0
print(pinned_mempool.n_free_blocks()) # 0

```

See `cupy.cuda.MemoryPool` and `cupy.cuda.PinnedMemoryPool` for details.

## 4.5.2 Limiting GPU Memory Usage

You can hard-limit the amount of GPU memory that can be allocated by using `CUPY_GPU_MEMORY_LIMIT` environment variable (see *Environment variables* for details).

```

# Set the hard-limit to 1 GiB:
# $ export CUPY_GPU_MEMORY_LIMIT="1073741824"

# You can also specify the limit in fraction of the total amount of memory
# on the GPU. If you have a GPU with 2 GiB memory, the following is
# equivalent to the above configuration.
# $ export CUPY_GPU_MEMORY_LIMIT="50%"

import cupy
print(cupy.get_default_memory_pool().get_limit()) # 1073741824

```

You can also set the limit (or override the value specified via the environment variable) using `cupy.cuda.MemoryPool.set_limit()`. In this way, you can use a different limit for each GPU device.

```

import cupy

mempool = cupy.get_default_memory_pool()

with cupy.cuda.Device(0):
    mempool.set_limit(size=1024**3) # 1 GiB

with cupy.cuda.Device(1):
    mempool.set_limit(size=2*1024**3) # 2 GiB

```

**Note:** CUDA allocates some GPU memory outside of the memory pool (such as CUDA context, library handles, etc.). Depending on the usage, such memory may take one to few hundred MiB. That will not be counted in the limit.

---

### 4.5.3 Changing Memory Pool

You can use your own memory allocator instead of the default memory pool by passing the memory allocation function to `cupy.cuda.set_allocator()` / `cupy.cuda.set_pinned_memory_allocator()`. The memory allocator function should take 1 argument (the requested size in bytes) and return `cupy.cuda.MemoryPointer` / `cupy.cuda.PinnedMemoryPointer`.

CuPy provides two such allocators for using managed memory and stream ordered memory on GPU, see `cupy.cuda.malloc_managed()` and `cupy.cuda.malloc_async()`, respectively, for details. To enable a memory pool backed by managed memory, you can construct a new `MemoryPool` instance with its allocator set to `malloc_managed()` as follows

```
import cupy

# Use managed memory
cupy.cuda.set_allocator(cupy.cuda.MemoryPool(cupy.cuda.malloc_managed).malloc)
```

Note that if you pass `malloc_managed()` directly to `set_allocator()` without constructing a `MemoryPool` instance, when the memory is freed it will be released back to the system immediately, which may or may not be desired.

Stream Ordered Memory Allocator is a new feature added since CUDA 11.2. CuPy provides an *experimental* interface to it. Similar to CuPy's memory pool, Stream Ordered Memory Allocator also allocates/deallocates memory *asynchronously* from/to a memory pool in a stream-ordered fashion. The key difference is that it is a built-in feature implemented in the CUDA driver by NVIDIA, so other CUDA applications in the same process can easily allocate memory from the same pool.

To enable a memory pool that manages stream ordered memory, you can construct a new `MemoryAsyncPool` instance:

```
import cupy

# Use asynchronous stream ordered memory
cupy.cuda.set_allocator(cupy.cuda.MemoryAsyncPool().malloc)

# Create a custom stream
s = cupy.cuda.Stream()

# This would allocate memory asynchronously on stream s
with s:
    a = cupy.empty((100,), dtype=cupy.float64)
```

Note that in this case we do not use the `MemoryPool` class. The `MemoryAsyncPool` takes a different input argument from that of `MemoryPool` to indicate which pool to use. Please refer to `MemoryAsyncPool`'s documentation for further detail.

Note that if you pass `malloc_async()` directly to `set_allocator()` without constructing a `MemoryAsyncPool` instance, the device's *current* memory pool will be used.

When using stream ordered memory, it is important that you maintain a correct stream semantics yourselves using, for example, the `Stream` and `Event` APIs (see *Streams and Events* for details); CuPy does not attempt to act smartly for you. Upon deallocation, the memory is freed asynchronously either on the stream it was allocated (first attempt), or

on any current CuPy stream (second attempt). It is permitted that the stream on which the memory was allocated gets destroyed before all memory allocated on it is freed.

In addition, applications/libraries internally use `cudaMalloc` (CUDA's default, synchronous allocator) could have unexpected interplay with Stream Ordered Memory Allocator. Specifically, memory freed to the memory pool might not be immediately visible to `cudaMalloc`, leading to potential out-of-memory errors. In this case, you can either call `free_all_blocks()` or just manually perform a (event/stream/device) synchronization, and retry.

Currently the `MemoryAsyncPool` interface is *experimental*. In particular, while its API is largely identical to that of `MemoryPool`, several of the pool's methods require a sufficiently new driver (and of course, a supported hardware, CUDA version, and platform) due to CUDA's limitation.

You can even disable the default memory pool by the code below. Be sure to do this before any other CuPy operations.

```
import cupy

# Disable memory pool for device memory (GPU)
cupy.cuda.set_allocator(None)

# Disable memory pool for pinned memory (CPU).
cupy.cuda.set_pinned_memory_allocator(None)
```

## 4.6 Performance Best Practices

Here we gather a few tricks and advices for improving CuPy's performance.

### 4.6.1 Benchmarking

It is utterly important to first identify the performance bottleneck before making any attempt to optimize your code. To help set up a baseline benchmark, CuPy provides a useful utility `cupyx.profiler.benchmark()` for timing the elapsed time of a Python function on both CPU and GPU:

```
>>> from cupyx.profiler import benchmark
>>>
>>> def my_func(a):
...     return cp.sqrt(cp.sum(a**2, axis=-1))
...
>>> a = cp.random.random((256, 1024))
>>> print(benchmark(my_func, (a,), n_repeat=20))
my_func          :    CPU:    44.407 us   +/- 2.428 (min:    42.516 / max:    53.098) us
↳ GPU-0:    181.565 us   +/- 1.853 (min:   180.288 / max:   188.608) us
```

Because GPU executions run asynchronously with respect to CPU executions, a common pitfall in GPU programming is to mistakenly measure the elapsed time using CPU timing utilities (such as `time.perf_counter()` from the Python Standard Library or the `%timeit` magic from IPython), which have no knowledge in the GPU runtime. `cupyx.profiler.benchmark()` addresses this by setting up CUDA events on the *Current Stream* right before and after the function to be measured and synchronizing over the end event (see *Streams and Events* for detail). Below we sketch what is done internally in `cupyx.profiler.benchmark()`:

```
>>> import time
>>> start_gpu = cp.cuda.Event()
>>> end_gpu = cp.cuda.Event()
```

(continues on next page)

(continued from previous page)

```
>>>
>>> start_gpu.record()
>>> start_cpu = time.perf_counter()
>>> out = my_func(a)
>>> end_cpu = time.perf_counter()
>>> end_gpu.record()
>>> end_gpu.synchronize()
>>> t_gpu = cp.cuda.get_elapsed_time(start_gpu, end_gpu)
>>> t_cpu = end_cpu - start_cpu
```

Additionally, `cupyx.profiler.benchmark()` runs a few warm-up runs to reduce timing fluctuation and exclude the overhead in first invocations.

## One-Time Overheads

Be aware of these overheads when benchmarking CuPy code.

## Context Initialization

It may take several seconds when calling a CuPy function for the first time in a process. This is because the CUDA driver creates a CUDA context during the first CUDA API call in CUDA applications.

## Kernel Compilation

CuPy uses on-the-fly kernel synthesis. When a kernel call is required, it compiles a kernel code optimized for the dimensions and dtypes of the given arguments, sends them to the GPU device, and executes the kernel.

CuPy caches the kernel code sent to GPU device within the process, which reduces the kernel compilation time on further calls.

The compiled code is also cached in the directory `${HOME}/.cupy/kernel_cache` (the path can be overwritten by setting the `CUPY_CACHE_DIR` environment variable). This allows reusing the compiled kernel binary across the process.

### 4.6.2 In-depth profiling

Under construction. To mark with NVTX/roctx ranges, you can use the `cupyx.profiler.time_range()` API. To start/stop the profiler, you can use the `cupyx.profiler.profile()` API.

### 4.6.3 Use CUB/cuTENSOR backends for reduction and other routines

For reduction operations (such as `sum()`, `prod()`, `amin()`, `amax()`, `argmin()`, `argmax()`) and many more routines built upon them, CuPy ships with our own implementations so that things just work out of the box. However, there are dedicated efforts to further accelerate these routines, such as `CUB` and `cuTENSOR`.

In order to support more performant backends wherever applicable, starting v8 CuPy introduces an environment variable `CUPY_ACCELERATORS` to allow users to specify the desired backends (and in what order they are tried). For example, consider summing over a 256-cubic array:

```
>>> from cupyx.profiler import benchmark
>>> a = cp.random.random((256, 256, 256), dtype=cp.float32)
>>> print(benchmark(a.sum, (), n_repeat=100))
sum          :    CPU:   12.101 us   +/- 0.694 (min:   11.081 / max:   17.649) us
↳ GPU-0: 10174.898 us   +/-180.551 (min:10084.576 / max:10595.936) us
```

We can see that it takes about 10 ms to run (on this GPU). However, if we launch the Python session using `CUPY_ACCELERATORS=cub python`, we get a ~100x speedup for free (only ~0.1 ms):

```
>>> print(benchmark(a.sum, (), n_repeat=100))
sum          :    CPU:   20.569 us   +/- 5.418 (min:   13.400 / max:   28.439) us
↳ GPU-0:   114.740 us   +/- 4.130 (min:  108.832 / max:  122.752) us
```

CUB is a backend shipped together with CuPy. It also accelerates other routines, such as inclusive scans (ex: [cumsum\(\)](#)), histograms, sparse matrix-vector multiplications (not applicable in CUDA 11), and [ReductionKernel](#). cuTENSOR offers optimized performance for binary elementwise ufuncs, reduction and tensor contraction. If cuTENSOR is installed, setting `CUPY_ACCELERATORS=cub,cutensor`, for example, would try CUB first and fall back to cuTENSOR if CUB does not provide the needed support. In the case that both backends are not applicable, it falls back to CuPy's default implementation.

Note that while in general the accelerated reductions are faster, there could be exceptions depending on the data layout. In particular, the CUB reduction only supports reduction over contiguous axes. In any case, we recommend to perform some benchmarks to determine whether CUB/cuTENSOR offers better performance or not.

---

**Note:** CuPy v11 and above uses CUB by default. To turn it off, you need to explicitly specify the environment variable `CUPY_ACCELERATORS=""`.

---

## 4.6.4 Overlapping work using streams

Under construction.

## 4.6.5 Use JIT compiler

Under construction. For now please refer to [JIT kernel definition](#) for a quick introduction.

## 4.6.6 Prefer float32 over float64

Under construction.

## 4.7 Interoperability

CuPy can be used in conjunction with other libraries.

### 4.7.1 NumPy

`cupy.ndarray` implements `__array_ufunc__` interface (see [NEP 13 — A Mechanism for Overriding Ufuncs](#) for details). This enables NumPy ufuncs to be directly operated on CuPy arrays. `__array_ufunc__` feature requires NumPy 1.13 or later.

```
import cupy
import numpy

arr = cupy.random.randn(1, 2, 3, 4).astype(cupy.float32)
result = numpy.sum(arr)
print(type(result)) # => <class 'cupy._core.core.ndarray'>
```

`cupy.ndarray` also implements `__array_function__` interface (see [NEP 18 — A dispatch mechanism for NumPy's high level array functions](#) for details). This enables code using NumPy to be directly operated on CuPy arrays. `__array_function__` feature requires NumPy 1.16 or later; As of NumPy 1.17, `__array_function__` is enabled by default.

### 4.7.2 Numba

Numba is a Python JIT compiler with NumPy support.

`cupy.ndarray` implements `__cuda_array_interface__`, which is the CUDA array interchange interface compatible with Numba v0.39.0 or later (see [CUDA Array Interface](#) for details). It means you can pass CuPy arrays to kernels JITted with Numba. The following is a simple example code borrowed from [numba/numba#2860](#):

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

print(out) # => [0 0 0 0 0 0 0 0 0 0]

add[1, 32](a, b, out)

print(out) # => [ 0  3  6  9 12 15 18 21 24 27]
```

In addition, `cupy.asarray()` supports zero-copy conversion from Numba CUDA array to CuPy array.

```
import numpy
import numba
import cupy

x = numpy.arange(10) # type: numpy.ndarray
```

(continues on next page)

(continued from previous page)

```
x_numba = numba.cuda.to_device(x) # type: numba.cuda.cudadrv.devicearray.DeviceNDArray
x_cupy = cupy.asarray(x_numba) # type: cupy.ndarray
```

**Warning:** `__cuda_array_interface__` specifies that the object lifetime must be managed by the user, so it is an undefined behavior if the exported object is destroyed while still in use by the consumer library.

**Note:** CuPy uses two environment variables controlling the exchange behavior: `CUPY_CUDA_ARRAY_INTERFACE_SYNC` and `CUPY_CUDA_ARRAY_INTERFACE_EXPORT_VERSION`.

### 4.7.3 mpi4py

MPI for Python (`mpi4py`) is a Python wrapper for the Message Passing Interface (MPI) libraries.

MPI is the most widely used standard for high-performance inter-process communications. Recently several MPI vendors, including MPICH, Open MPI and MVAPICH, have extended their support beyond the MPI-3.1 standard to enable “CUDA-awareness”; that is, passing CUDA device pointers directly to MPI calls to avoid explicit data movement between the host and the device.

With the `__cuda_array_interface__` (as mentioned above) and DLPack data exchange protocols (see [DLPack](#) below) implemented in CuPy, `mpi4py` now provides (experimental) support for passing CuPy arrays to MPI calls, provided that `mpi4py` is built against a CUDA-aware MPI implementation. The following is a simple example code borrowed from [mpi4py Tutorial](#):

```
# To run this script with N MPI processes, do
# mpiexec -n N python this_script.py

import cupy
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()

# Allreduce
sendbuf = cupy.arange(10, dtype='i')
recvbuf = cupy.empty_like(sendbuf)
comm.Allreduce(sendbuf, recvbuf)
assert cupy.allclose(recvbuf, sendbuf*size)
```

This new feature is added since `mpi4py` 3.1.0. See the [mpi4py website](#) for more information.

#### 4.7.4 PyTorch

PyTorch is a machine learning framework that provides high-performance, differentiable tensor operations.

PyTorch also supports `__cuda_array_interface__`, so zero-copy data exchange between CuPy and PyTorch can be achieved at no cost. The only caveat is PyTorch by default creates CPU tensors, which do not have the `__cuda_array_interface__` property defined, and users need to ensure the tensor is already on GPU before exchanging.

```
>>> import cupy as cp
>>> import torch
>>>
>>> # convert a torch tensor to a cupy array
>>> a = torch.rand((4, 4), device='cuda')
>>> b = cp.asarray(a)
>>> b *= b
>>> b
array([[0.8215962 , 0.82399917, 0.65607935, 0.30354425],
       [0.422695  , 0.8367199 , 0.00208597, 0.18545236],
       [0.00226746, 0.46201342, 0.6833052 , 0.47549972],
       [0.5208748 , 0.6059282 , 0.1909013 , 0.5148635 ]], dtype=float32)
>>> a
tensor([[0.8216, 0.8240, 0.6561, 0.3035],
        [0.4227, 0.8367, 0.0021, 0.1855],
        [0.0023, 0.4620, 0.6833, 0.4755],
        [0.5209, 0.6059, 0.1909, 0.5149]], device='cuda:0')
>>> # check the underlying memory pointer is the same
>>> assert a.__cuda_array_interface__['data'][0] == b.__cuda_array_interface__['data'][0]
>>>
>>> # convert a cupy array to a torch tensor
>>> a = cp.arange(10)
>>> b = torch.as_tensor(a, device='cuda')
>>> b += 3
>>> b
tensor([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12], device='cuda:0')
>>> a
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> assert a.__cuda_array_interface__['data'][0] == b.__cuda_array_interface__['data'][0]
```

PyTorch also supports zero-copy data exchange through DLPack (see *DLPack* below):

```
import cupy
import torch

# Create a PyTorch tensor.
tx1 = torch.randn(1, 2, 3, 4).cuda()

# Convert it into a CuPy array.
cx = cupy.from_dlpack(tx1)

# Convert it back to a PyTorch tensor.
tx2 = torch.from_dlpack(cx)
```

`pytorch-pfn-extras` library provides additional integration features with PyTorch, including memory pool sharing and stream sharing:



```

>>> import cupy
>>> import torch
>>> import pytorch_pfn_extras as ppe
>>>
>>> # Perform CuPy memory allocation using the PyTorch memory pool.
>>> ppe.cuda.use_torch_mempool_in_cupy()
>>> torch.cuda.memory_allocated()
0
>>> arr = cupy.arange(10)
>>> torch.cuda.memory_allocated()
512
>>>
>>> # Change the default stream in PyTorch and CuPy:
>>> stream = torch.cuda.Stream()
>>> with ppe.cuda.stream(stream):
...     ...

```

## Using custom kernels in PyTorch

With the DLPack protocol, it becomes very simple to implement functions in PyTorch using CuPy user-defined kernels. Below is the example of a PyTorch autograd function that computes the forward and backward pass of the logarithm using `cupy.RawKernel`s.

```

import cupy
import torch

cupy_custom_kernel_fwd = cupy.RawKernel(
    r"""
extern "C" __global__
void cupy_custom_kernel_fwd(const float* x, float* y, int size) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < size)
        y[tid] = log(x[tid]);
}
""",
    "cupy_custom_kernel_fwd",
)

cupy_custom_kernel_bwd = cupy.RawKernel(
    r"""
extern "C" __global__
void cupy_custom_kernel_bwd(const float* x, float* gy, float* gx, int size) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < size)
        gx[tid] = gy[tid] / x[tid];
}
""",
    "cupy_custom_kernel_bwd",
)

```

(continues on next page)

(continued from previous page)

```

class CuPyLog(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.input = x
        # Enforce contiguous arrays to simplify RawKernel indexing.
        cupy_x = cupy.ascontiguousarray(cupy.from_dlpack(x.detach()))
        cupy_y = cupy.empty(cupy_x.shape, dtype=cupy_x.dtype)
        x_size = cupy_x.size
        bs = 128
        cupy_custom_kernel_fwd(
            (bs,), ((x_size + bs - 1) // bs,), (cupy_x, cupy_y, x_size)
        )
        # the ownership of the device memory backing cupy_y is implicitly
        # transferred to torch_y, so this operation is safe even after
        # going out of scope of this function.
        torch_y = torch.from_dlpack(cupy_y)
        return torch_y

    @staticmethod
    def backward(ctx, grad_y):
        # Enforce contiguous arrays to simplify RawKernel indexing.
        cupy_input = cupy.from_dlpack(ctx.input.detach()).ravel()
        cupy_grad_y = cupy.from_dlpack(grad_y.detach()).ravel()
        cupy_grad_x = cupy.zeros(cupy_grad_y.shape, dtype=cupy_grad_y.dtype)
        gy_size = cupy_grad_y.size
        bs = 128
        cupy_custom_kernel_bwd(
            (bs,),
            ((gy_size + bs - 1) // bs,),
            (cupy_input, cupy_grad_y, cupy_grad_x, gy_size),
        )
        # the ownership of the device memory backing cupy_grad_x is implicitly
        # transferred to torch_y, so this operation is safe even after
        # going out of scope of this function.
        torch_grad_x = torch.from_dlpack(cupy_grad_x)
        return torch_grad_x

```

**Note:** Directly feeding a `torch.Tensor` to `cupy.from_dlpack()` is only supported in the (new) DLPack data exchange protocol added in CuPy v10+ and PyTorch 1.10+. For earlier versions, you will need to wrap the Tensor with `torch.utils.dlpack.to_dlpack()` as shown in the above examples.

### 4.7.5 RMM

RMM (RAPIDS Memory Manager) provides highly configurable memory allocators.

RMM provides an interface to allow CuPy to allocate memory from the RMM memory pool instead of from CuPy's own pool. It can be set up as simple as:

```
import cupy
import rmm
cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)
```

Sometimes, a more performant allocator may be desirable. RMM provides an option to switch the allocator:

```
import cupy
import rmm
rmm.reinitialize(pool_allocator=True) # can also set init pool size etc here
cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)
```

For more information on CuPy's memory management, see *Memory Management*.

### 4.7.6 DLPack

DLPack is a specification of tensor structure to share tensors among frameworks.

CuPy supports importing from and exporting to DLPack data structure (*cupy.from\_dlpack()* and *cupy.ndarray.toDlpack()*).

Here is a simple example:

```
import cupy

# Create a CuPy array.
cx1 = cupy.random.randn(1, 2, 3, 4).astype(cupy.float32)

# Convert it into a DLPack tensor.
dx = cx1.toDlpack()

# Convert it back to a CuPy array.
cx2 = cupy.from_dlpack(dx)
```

TensorFlow also supports DLPack, so zero-copy data exchange between CuPy and TensorFlow through DLPack is possible:

```
>>> import tensorflow as tf
>>> import cupy as cp
>>>
>>> # convert a TF tensor to a cupy array
>>> with tf.device('/GPU:0'):
...     a = tf.random.uniform((10,))
...
>>> a
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.9672388 , 0.57568085, 0.53163004, 0.6536236 , 0.20479882,
       0.84908986, 0.5852566 , 0.30355775, 0.1733712 , 0.9177849 ],
      dtype=float32)>
```

(continues on next page)

(continued from previous page)

```

>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> cap = tf.experimental.dlpack.to_dlpack(a)
>>> b = cp.from_dlpack(cap)
>>> b *= 3
>>> b
array([1.4949363 , 0.60699713, 1.3276931 , 1.5781245 , 1.1914308 ,
        2.3180873 , 1.9560868 , 1.3932796 , 1.9299742 , 2.5352407 ],
      dtype=float32)
>>> a
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([1.4949363 , 0.60699713, 1.3276931 , 1.5781245 , 1.1914308 ,
        2.3180873 , 1.9560868 , 1.3932796 , 1.9299742 , 2.5352407 ],
      dtype=float32)>
>>>
>>> # convert a cupy array to a TF tensor
>>> a = cp.arange(10)
>>> cap = a.toDlpack()
>>> b = tf.experimental.dlpack.from_dlpack(cap)
>>> b.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])>
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

```

Be aware that in TensorFlow all tensors are immutable, so in the latter case any changes in `b` cannot be reflected in the CuPy array `a`.

Note that as of DLPack v0.5 for correctness the above approach (implicitly) requires users to ensure that such conversion (both importing and exporting a CuPy array) must happen on the same CUDA/HIP stream. If in doubt, the current CuPy stream in use can be fetched by, for example, calling `cupy.cuda.get_current_stream()`. Please consult the other framework's documentation for how to access and control the streams.

## DLPack data exchange protocol

To obviate user-managed streams and DLPack tensor objects, the [DLPack data exchange protocol](#) provides a mechanism to shift the responsibility from users to libraries. Any compliant objects (such as `cupy.ndarray`) must implement a pair of methods `__dlpack__` and `__dlpack_device__`. The function `cupy.from_dlpack()` accepts such object and returns a `cupy.ndarray` that is safely accessible on CuPy's current stream. Likewise, `cupy.ndarray` can be exported via any compliant library's `from_dlpack()` function.

---

**Note:** CuPy uses `CUPY_DLPACK_EXPORT_VERSION` to control how to handle tensors backed by CUDA managed memory.

---

### 4.7.7 Device Memory Pointers

#### Import

CuPy provides *UnownedMemory* API that allows interoperating with GPU device memory allocated in other libraries.

```
# Create a memory chunk from raw pointer and its size.
mem = cupy.cuda.UnownedMemory(140359025819648, 1024, owner=None)

# Wrap it as a MemoryPointer.
memptr = cupy.cuda.MemoryPointer(mem, offset=0)

# Create an ndarray view backed by the memory pointer.
arr = cupy.ndarray((16, 16), dtype=cupy.float32, memptr=memptr)
assert arr.nbytes <= arr.data.mem.size
```

Be aware that you are responsible for specifying a correct shape, dtype, strides, and order such that it fits in the chunk when creating an *ndarray* view.

The *UnownedMemory* API does not manage the lifetime of the memory allocation. You must ensure that the pointer is alive while in use by CuPy. In case the pointer lifetime is managed by a Python object, you can pass it to the *owner* argument of the *UnownedMemory* to keep the reference to the object.

#### Export

You can pass memory pointers allocated in CuPy to other libraries.

```
arr = cupy.arange(10)
print(arr.data.ptr, arr.nbytes) # => (140359025819648, 80)
```

The memory allocated by CuPy will be freed when the *ndarray* (*arr*) gets destructed. You must keep *ndarray* instance alive while the pointer is in use by other libraries.

### 4.7.8 CUDA Stream Pointers

#### Import

CuPy provides *ExternalStream* API that allows interoperating with CUDA streams created in other libraries.

```
import torch

# Create a stream on PyTorch.
s = torch.cuda.Stream()

# Switch the current stream in PyTorch.
with torch.cuda.stream(s):
    # Switch the current stream in CuPy, using the pointer of the stream created in
    # PyTorch.
    with cupy.cuda.ExternalStream(s.cuda_stream):
        # This block runs on the same CUDA stream.
        torch.arange(10, device='cuda')
        cupy.arange(10)
```

The `ExternalStream` API does not manage the lifetime of the stream. You must ensure that the stream pointer is alive while in use by CuPy.

You also need to make sure that the `ExternalStream` object is used on the device where the stream was created. CuPy can validate that for you if you pass `device_id` argument when creating `ExternalStream`.

## Export

You can pass streams created in CuPy to other libraries.

```
s = cupy.cuda.Stream()
print(s.ptr, s.device_id)  # => (93997451352336, 0)
```

The CUDA stream will be destroyed when the `Stream` (`s`) gets destructed. You must keep the `Stream` instance alive while the pointer is in use by other libraries.

## 4.8 Differences between CuPy and NumPy

The interface of CuPy is designed to obey that of NumPy. However, there are some differences.

### 4.8.1 Cast behavior from float to integer

Some casting behaviors from float to integer are not defined in C++ specification. The casting from a negative float to unsigned integer and infinity to integer is one of such examples. The behavior of NumPy depends on your CPU architecture. This is the result on an Intel CPU:

```
>>> np.array([-1], dtype=np.float32).astype(np.uint32)
array([4294967295], dtype=uint32)
>>> cupy.array([-1], dtype=np.float32).astype(np.uint32)
array([0], dtype=uint32)
```

```
>>> np.array([float('inf')], dtype=np.float32).astype(np.int32)
array([-2147483648], dtype=int32)
>>> cupy.array([float('inf')], dtype=np.float32).astype(np.int32)
array([2147483647], dtype=int32)
```

### 4.8.2 Random methods support dtype argument

NumPy's random value generator does not support a `dtype` argument and instead always returns a `float64` value. We support the option in CuPy because `cuRAND`, which is used in CuPy, supports both `float32` and `float64`.

```
>>> np.random.randn(dtype=np.float32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: randn() got an unexpected keyword argument 'dtype'
>>> cupy.random.randn(dtype=np.float32)
array(0.10689262300729752, dtype=float32)
```

### 4.8.3 Out-of-bounds indices

CuPy handles out-of-bounds indices differently by default from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.

```
>>> x = np.array([0, 1, 2])
>>> x[[1, 3]] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 1 with size 3
>>> x = cupy.array([0, 1, 2])
>>> x[[1, 3]] = 10
>>> x
array([10, 10,  2])
```

### 4.8.4 Duplicate values in indices

CuPy's `__setitem__` behaves differently from NumPy when integer arrays reference the same location multiple times. In that case, the value that is actually stored is undefined. Here is an example of CuPy.

```
>>> a = cupy.zeros((2,))
>>> i = cupy.arange(10000) % 2
>>> v = cupy.arange(10000).astype(np.float32)
>>> a[i] = v
>>> a
array([ 9150.,  9151.])
```

NumPy stores the value corresponding to the last element among elements referencing duplicate locations.

```
>>> a_cpu = np.zeros((2,))
>>> i_cpu = np.arange(10000) % 2
>>> v_cpu = np.arange(10000).astype(np.float32)
>>> a_cpu[i_cpu] = v_cpu
>>> a_cpu
array([9998., 9999.])
```

### 4.8.5 Zero-dimensional array

#### Reduction methods

NumPy's reduction functions (e.g. `numpy.sum()`) return scalar values (e.g. `numpy.float32`). However CuPy counterparts return zero-dimensional `cupy.ndarray`s. That is because CuPy scalar values (e.g. `cupy.float32`) are aliases of NumPy scalar values and are allocated in CPU memory. If these types were returned, it would be required to synchronize between GPU and CPU. If you want to use scalar values, cast the returned arrays explicitly.

```
>>> type(np.sum(np.arange(3))) == np.int64
True
>>> type(cupy.sum(cupy.arange(3))) == cupy.ndarray
True
```

## Type promotion

CuPy automatically promotes dtypes of `cupy.ndarray`s in a function with two or more operands, the result dtype is determined by the dtypes of the inputs. This is different from NumPy's rule on type promotion, when operands contain zero-dimensional arrays. Zero-dimensional `numpy.ndarray`s are treated as if they were scalar values if they appear in operands of NumPy's function, This may affect the dtype of its output, depending on the values of the “scalar” inputs.

```
>>> (np.array(3, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float32')
>>> (np.array(3000000, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float64')
>>> (cupy.array(3, dtype=np.int32) * cupy.array([1., 2.], dtype=np.float32)).dtype
dtype('float64')
```

### 4.8.6 Matrix type (`numpy.matrix`)

SciPy returns `numpy.matrix` (a subclass of `numpy.ndarray`) when dense matrices are computed from sparse matrices (e.g., `coo_matrix + ndarray`). However, CuPy returns `cupy.ndarray` for such operations.

There is no plan to provide `numpy.matrix` equivalent in CuPy. This is because the use of `numpy.matrix` is no longer recommended since NumPy 1.15.

### 4.8.7 Data types

Data type of CuPy arrays cannot be non-numeric like strings or objects. See [Overview](#) for details.

### 4.8.8 Universal Functions only work with CuPy array or scalar

Unlike NumPy, Universal Functions in CuPy only work with CuPy array or scalar. They do not accept other objects (e.g., lists or `numpy.ndarray`).

```
>>> np.power([np.arange(5)], 2)
array([[ 0,  1,  4,  9, 16]])
```

```
>>> cupy.power([cupy.arange(5)], 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Unsupported type <class 'list'>
```

### 4.8.9 Random seed arrays are hashed to scalars

Like Numpy, CuPy's `RandomState` objects accept seeds either as numbers or as full numpy arrays.

```
>>> seed = np.array([1, 2, 3, 4, 5])
>>> rs = cupy.random.RandomState(seed=seed)
```

However, unlike Numpy, array seeds will be hashed down to a single number and so may not communicate as much entropy to the underlying random number generator.



### 4.8.10 NaN (not-a-number) handling

By default CuPy's reduction functions (e.g., `cupy.sum()`) handle NaNs in complex numbers differently from NumPy's counterparts:

```
>>> a = [0.5 + 3.7j, complex(0.7, np.nan), complex(np.nan, -3.9), complex(np.nan, np.
↳ nan)]
>>>
>>> a_np = np.asarray(a)
>>> print(a_np.max(), a_np.min())
(0.7+nanj) (0.7+nanj)
>>>
>>> a_cp = cp.asarray(a_np)
>>> print(a_cp.max(), a_cp.min())
(nan-3.9j) (nan-3.9j)
```

The reason is that internally the reduction is performed in a strided fashion, thus it does not ensure a proper comparison order and cannot follow NumPy's rule to always propagate the first-encountered NaN. Note that this difference does not apply when CUB is enabled (which is the default for CuPy v11 or later.)

### 4.8.11 Contiguity / Strides

To provide the best performance, the contiguity of a resulting ndarray is not guaranteed to match with that of NumPy's output.

```
>>> a = np.array([[1, 2], [3, 4]], order='F')
>>> print((a + a).flags.f_contiguous)
True
```

```
>>> a = cp.array([[1, 2], [3, 4]], order='F')
>>> print((a + a).flags.f_contiguous)
False
```

## 4.9 API Compatibility Policy

This document expresses the design policy on compatibilities of CuPy APIs. Development team should obey this policy on deciding to add, extend, and change APIs and their behaviors.

This document is written for both users and developers. Users can decide the level of dependencies on CuPy's implementations in their codes based on this document. Developers should read through this document before creating pull requests that contain changes on the interface. Note that this document may contain ambiguities on the level of supported compatibilities.

### 4.9.1 Versioning and Backward Compatibilities

The updates of CuPy are classified into three levels: major, minor, and revision. These types have distinct levels of backward compatibilities.

- **Major update** contains disruptive changes that break the backward compatibility.
- **Minor update** contains additions and extensions to the APIs that keep the backward compatibility supported.
- **Revision update** contains improvements on the API implementations without changing any API specifications.

Note that we do not support full backward compatibility, which is almost infeasible for Python-based APIs, since there is no way to completely hide the implementation details.

### 4.9.2 Processes to Break Backward Compatibilities

#### Deprecation, Dropping, and Its Preparation

Any APIs may be *deprecated* at some minor updates. In such a case, the deprecation note is added to the API documentation, and the API implementation is changed to fire a deprecation warning (if possible). There should be another way to reimplement the same functionality previously written using the deprecated APIs.

Any APIs may be marked as *to be dropped in the future*. In such a case, the dropping is stated in the documentation with the major version number on which the API is planned to be dropped, and the API implementation is changed to fire a future warning (if possible).

The actual dropping should be done through the following steps:

- Make the API deprecated. At this point, users should not use the deprecated API in their new application codes.
- After that, mark the API as *to be dropped in the future*. It must be done in the minor update different from that of the deprecation.
- At the major version announced in the above update, drop the API.

Consequently, it takes at least two minor versions to drop any APIs after the first deprecation.

#### API Changes and Its Preparation

Any APIs may be marked as *to be changed in the future* for changes without backward compatibility. In such a case, the change is stated in the documentation with the version number on which the API is planned to be changed, and the API implementation is changed to fire the future warning on the certain usages.

The actual change should be done in the following steps:

- Announce that the API will be changed in the future. At this point, the actual version of change need not be accurate.
- After the announcement, mark the API as *to be changed in the future* with version number of planned changes. At this point, users should not use the marked API in their new application codes.
- At the major update announced in the above update, change the API.

### 4.9.3 Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

#### Documented Interface

CuPy has an official API documentation. Many applications can be written based on the documented features. We support backward compatibilities of documented features. In other words, codes only based on the documented features run correctly with minor-/revision- updated versions.

Developers are encouraged to use apparent names for objects of implementation details. For example, attributes outside of the documented APIs should have one or more underscores at the prefix of their names.

#### Undocumented behaviors

Behaviors of CuPy implementation not stated in the documentation are undefined. Undocumented behaviors are not guaranteed to be stable between different minor/revision versions.

Minor update may contain changes to undocumented behaviors. For example, suppose an API X is added at the minor update. In the previous version, attempts to use X cause `AttributeError`. This behavior is not stated in the documentation, so this is undefined. Thus, adding the API X in minor version is permissible.

Revision update may also contain changes to undefined behaviors. Typical example is a bug fix. Another example is an improvement on implementation, which may change the internal object structures not shown in the documentation. As a consequence, **even revision updates do not support compatibility of pickling, unless the full layout of pickled objects is clearly documented.**

#### Documentation Error

Compatibility is basically determined based on the documentation, though it sometimes contains errors. It may make the APIs confusing to assume the documentation always stronger than the implementations. We therefore may fix the documentation errors in any updates that may break the compatibility in regard to the documentation.

---

**Note:** Developers MUST NOT fix the documentation and implementation of the same functionality at the same time in revision updates as “bug fix”. Such a change completely breaks the backward compatibility. If you want to fix the bugs in both sides, first fix the documentation to fit it into the implementation, and start the API changing procedure described above.

---

#### Object Attributes and Properties

Object attributes and properties are sometimes replaced by each other at minor updates. It does not break the user codes, except for the codes depending on how the attributes and properties are implemented.

## Functions and Methods

Methods may be replaced by callable attributes keeping the compatibility of parameters and return values in minor updates. It does not break the user codes, except for the codes depending on how the methods and callable attributes are implemented.

## Exceptions and Warnings

The specifications of raising exceptions are considered as a part of standard backward compatibilities. No exception is raised in the future versions with correct usages that the documentation allows, unless the API changing process is completed.

On the other hand, warnings may be added at any minor updates for any APIs. It means minor updates do not keep backward compatibility of warnings.

### 4.9.4 Installation Compatibility

The installation process is another concern of compatibilities. We support environmental compatibilities in the following ways.

- Any changes of dependent libraries that force modifications on the existing environments must be done in major updates. Such changes include following cases:
  - dropping supported versions of dependent libraries (e.g. dropping cuDNN v2)
  - adding new mandatory dependencies (e.g. adding h5py to setup\_requires)
- Supporting optional packages/libraries may be done in minor updates (e.g. supporting h5py in optional features).

---

**Note:** The installation compatibility does not guarantee that all the features of CuPy correctly run on supported environments. It may contain bugs that only occurs in certain environments. Such bugs should be fixed in some updates.

---

## API REFERENCE

- `genindex`
  - `modindex`
- 

### 5.1 The N-dimensional array (`ndarray`)

`cupy.ndarray` is the CuPy counterpart of NumPy `numpy.ndarray`. It provides an intuitive interface for a fixed-size multidimensional array which resides in a CUDA device.

For the basic concept of `ndarrays`, please refer to the [NumPy documentation](#).

---

<code>cupy.ndarray</code> ( <i>self</i> , <i>shape</i> [, <i>dtype</i> , <i>memptr</i> , ...])	Multi-dimensional array on a CUDA device.
--	---

---

#### 5.1.1 `cupy.ndarray`

**class** `cupy.ndarray`(*self*, *shape*, *dtype*=`float`, *memptr*=`None`, *strides*=`None`, *order*=`'C'`)

Multi-dimensional array on a CUDA device.

This class implements a subset of methods of `numpy.ndarray`. The difference is that this class allocates the array content on the current GPU device.

##### Parameters

- **shape** (*tuple of ints*) – Length of axes.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **memptr** (`cupy.cuda.MemoryPointer`) – Pointer to the array content head.
- **strides** (*tuple of ints or None*) – Strides of data in memory.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

##### Variables

- **base** (*None or cupy.ndarray*) – Base array from which this array is created as a view.
- **data** (`cupy.cuda.MemoryPointer`) – Pointer to the array content head.
- **~ndarray.dtype** (`numpy.dtype`) – Dtype object of element type.

**See also:**

[Data type objects \(dtype\)](#)

- `~ndarray.size` (`int`) – Number of elements this array holds.

This is equivalent to product over the shape tuple.

**See also:**

`numpy.ndarray.size`

## Methods

### `__getitem__()`

`x.__getitem__(y) <==> x[y]`

Supports both basic and advanced indexing.

---

**Note:** Currently, it does not support `slices` that consists of more than one boolean arrays

---

---

**Note:** CuPy handles out-of-bounds indices differently from NumPy. NumPy handles them by raising an error, but CuPy wraps around them.

---

## Example

```
>>> a = cupy.arange(3)
>>> a[[1, 3]]
array([1, 0])
```

### `__setitem__()`

`x.__setitem__(slices, y) <==> x[slices] = y`

Supports both basic and advanced indexing.

---

**Note:** Currently, it does not support `slices` that consists of more than one boolean arrays

---

---

**Note:** CuPy handles out-of-bounds indices differently from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.

---

```
>>> import cupy
>>> x = cupy.arange(3)
>>> x[[1, 3]] = 10
>>> x
array([10, 10, 2])
```

---

**Note:** The behavior differs from NumPy when integer arrays in `slices` reference the same location multiple times. In that case, the value that is actually stored is undefined.

---

```
>>> import cupy
>>> a = cupy.zeros((2,))
>>> i = cupy.arange(10000) % 2
>>> v = cupy.arange(10000).astype(cupy.float64)
>>> a[i] = v
>>> a
array([9150., 9151.])
```

On the other hand, NumPy stores the value corresponding to the last index among the indices referencing duplicate locations.

```
>>> import numpy
>>> a_cpu = numpy.zeros((2,))
>>> i_cpu = numpy.arange(10000) % 2
>>> v_cpu = numpy.arange(10000).astype(numpy.float64)
>>> a_cpu[i_cpu] = v_cpu
>>> a_cpu
array([9998., 9999.])
```

**\_\_len\_\_()**

Return len(self).

**\_\_iter\_\_()**

Implement iter(self).

**\_\_copy\_\_(self)**

**all**(self, axis=None, out=None, keepdims=False) → ndarray

**any**(self, axis=None, out=None, keepdims=False) → ndarray

**argmax**(self, axis=None, out=None, dtype=None, keepdims=False) → ndarray

Returns the indices of the maximum along a given axis.

---

**Note:** dtype and keepdim arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** axis argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

**See also:**

[cupy.argmax\(\)](#) for full documentation, [numpy.ndarray.argmax\(\)](#)

**argmin**(self, axis=None, out=None, dtype=None, keepdims=False) → ndarray

Returns the indices of the minimum along a given axis.

---

**Note:** dtype and keepdim arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** axis argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

See also:

[`cupy.argmax\(\)`](#) for full documentation, [`numpy.ndarray.argmax\(\)`](#)

**argpartition**(*self*, *kth*, *axis=-1*) → *ndarray*

Returns the indices that would partially sort an array.

**Parameters**

- **kth** (*int* or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (*int* or *None*) – Axis along which to sort. Default is -1, which means sort along the last axis. If None is supplied, the array is flattened before sorting.

**Returns**

Array of the same type and shape as a.

**Return type**

[`cupy.ndarray`](#)

See also:

[`cupy.argpartition\(\)`](#) for full documentation, [`numpy.ndarray.argpartition\(\)`](#)

**argsort**(*self*, *axis=-1*) → *ndarray*

Returns the indices that would sort an array with stable sorting

**Parameters**

**axis** (*int* or *None*) – Axis along which to sort. Default is -1, which means sort along the last axis. If None is supplied, the array is flattened before sorting.

**Returns**

Array of indices that sort the array.

**Return type**

[`cupy.ndarray`](#)

See also:

[`cupy.argsort\(\)`](#) for full documentation, [`numpy.ndarray.argsort\(\)`](#)

**astype**(*self*, *dtype*, *order='K'*, *casting=None*, *subok=None*, *copy=True*) → *ndarray*

Casts the array to given data type.

**Parameters**

- **dtype** – Type specifier.
- **order** (*{'C', 'F', 'A', 'K'}*) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.
- **copy** (*bool*) – If it is False and no cast happens, then this method returns the array itself. Otherwise, a copy is returned.

**Returns**

If copy is False and no cast is required, then the array itself is returned. Otherwise, it returns a (possibly casted) copy of the array.

---

**Note:** This method currently does not support casting, and subok arguments.

---



See also:

`numpy.ndarray.astype()`

**choose**(*self*, *choices*, *out=None*, *mode='raise'*)

**clip**(*self*, *min=None*, *max=None*, *out=None*) → *ndarray*

Returns an array with values limited to [min, max].

See also:

`cupy.clip()` for full documentation, `numpy.ndarray.clip()`

**compress**(*self*, *condition*, *axis=None*, *out=None*) → *ndarray*

Returns selected slices of this array along given axis.

**Warning:** This function may synchronize the device.

See also:

`cupy.compress()` for full documentation, `numpy.ndarray.compress()`

**conj**(*self*) → *ndarray*

**conjugate**(*self*) → *ndarray*

**copy**(*self*, *order='C'*) → *ndarray*

Returns a copy of the array.

This method makes a copy of a given array in the current device. Even when a given array is located in another device, you can copy it to the current device.

#### Parameters

**order** ({'C', 'F', 'A', 'K'}) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.

See also:

`cupy.copy()` for full documentation, `numpy.ndarray.copy()`

**cumprod**(*self*, *axis=None*, *dtype=None*, *out=None*) → *ndarray*

Returns the cumulative product of an array along a given axis.

See also:

`cupy.cumprod()` for full documentation, `numpy.ndarray.cumprod()`

**cumsum**(*self*, *axis=None*, *dtype=None*, *out=None*) → *ndarray*

Returns the cumulative sum of an array along a given axis.

See also:

`cupy.cumsum()` for full documentation, `numpy.ndarray.cumsum()`

**diagonal**(*self*, *offset=0*, *axis1=0*, *axis2=1*) → *ndarray*

Returns a view of the specified diagonals.

See also:

`cupy.diagonal()` for full documentation, `numpy.ndarray.diagonal()`

**dot**(*self*, *ndarray b*, *ndarray out=None*)

Returns the dot product with given array.

**See also:**

[`cupy.dot\(\)`](#) for full documentation, [`numpy.ndarray.dot\(\)`](#)

**dump**(*self*, *file*)

Dumps a pickle of the array to a file.

Dumped file can be read back to [`cupy.ndarray`](#) by [`cupy.load\(\)`](#).

**dumps**(*self*) → *bytes*

Dumps a pickle of the array to a string.

**fill**(*self*, *value*)

Fills the array with a scalar value.

**Parameters**

**value** – A scalar value to fill the array content.

**See also:**

[`numpy.ndarray.fill\(\)`](#)

**flatten**(*self*, *order='C'*) → *ndarray*

Returns a copy of the array flatten into one dimension.

**Parameters**

**order** (*{'C', 'F', 'A', 'K'}*) – ‘C’ means to flatten in row-major (C-style) order. ‘F’ means to flatten in column-major (Fortran- style) order. ‘A’ means to flatten in column-major order if *self* is Fortran *contiguous* in memory, row-major order otherwise. ‘K’ means to flatten *self* in the order the elements occur in memory. The default is ‘C’.

**Returns**

A copy of the array with one dimension.

**Return type**

[`cupy.ndarray`](#)

**See also:**

[`numpy.ndarray.flatten\(\)`](#)

**get**(*self*, *stream=None*, *order='C'*, *out=None*, *blocking=True*)

Returns a copy of the array on host memory.

**Parameters**

- **stream** ([`cupy.cuda.Stream`](#)) – CUDA stream object. If given, the stream is used to perform the copy. Otherwise, the current stream is used.
- **order** (*{'C', 'F', 'A'}*) – The desired memory layout of the host array. When *order* is ‘A’, it uses ‘F’ if the array is fortran-contiguous and ‘C’ otherwise. The *order* will be ignored if *out* is specified.
- **out** ([`numpy.ndarray`](#)) – Output array. In order to enable asynchronous copy, the underlying memory should be a pinned memory.
- **blocking** (*bool*) – If set to *False*, the copy runs asynchronously on the given (if given) or current stream, and users are responsible for ensuring the stream order. Default is *True*, so the copy is synchronous (with respect to the host).

**Returns**

Copy of the array on host memory.

**Return type**

`numpy.ndarray`

**item**(*self*)

Converts the array with one element to a Python scalar

**Returns**

The element of the array.

**Return type**

`int` or `float` or `complex`

**See also:**

`numpy.ndarray.item()`

**max**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the maximum along a given axis.

**See also:**

`cupy.amax()` for full documentation, `numpy.ndarray.max()`

**mean**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the mean along a given axis.

**See also:**

`cupy.mean()` for full documentation, `numpy.ndarray.mean()`

**min**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the minimum along a given axis.

**See also:**

`cupy.amin()` for full documentation, `numpy.ndarray.min()`

**nonzero**(*self*) → *tuple*

Return the indices of the elements that are non-zero.

Returned Array is containing the indices of the non-zero elements in that dimension.

**Returns**

Indices of elements that are non-zero.

**Return type**

*tuple* of arrays

**Warning:** This function may synchronize the device.

**See also:**

`numpy.nonzero()`

**partition**(*self*, *kth*, *int axis=-1*)

Partitions an array.

**Parameters**

- **kth** (*int* or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (*int*) – Axis along which to sort. Default is -1, which means sort along the last axis.

See also:

[`cupy.partition\(\)`](#) for full documentation, [`numpy.ndarray.partition\(\)`](#)

**prod**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*) → *ndarray*

Returns the product along a given axis.

See also:

[`cupy.prod\(\)`](#) for full documentation, [`numpy.ndarray.prod\(\)`](#)

**ptp**(*self*, *axis=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns (maximum - minimum) along a given axis.

See also:

[`cupy.ptp\(\)`](#) for full documentation, [`numpy.ndarray.ptp\(\)`](#)

**put**(*self*, *indices*, *values*, *mode='wrap'*)

Replaces specified elements of an array with given values.

See also:

[`cupy.put\(\)`](#) for full documentation, [`numpy.ndarray.put\(\)`](#)

**ravel**(*self*, *order='C'*) → *ndarray*

Returns an array flattened into one dimension.

See also:

[`cupy.ravel\(\)`](#) for full documentation, [`numpy.ndarray.ravel\(\)`](#)

**reduced\_view**(*self*, *dtype=None*) → *ndarray*

Returns a view of the array with minimum number of dimensions.

#### Parameters

**dtype** – (Deprecated) Data type specifier. If it is given, then the memory sequence is reinterpreted as the new type.

#### Returns

A view of the array with reduced dimensions.

#### Return type

[`cupy.ndarray`](#)

**repeat**(*self*, *repeats*, *axis=None*)

Returns an array with repeated arrays along an axis.

See also:

[`cupy.repeat\(\)`](#) for full documentation, [`numpy.ndarray.repeat\(\)`](#)

**reshape**(*self*, *\*shape*, *order='C'*)

Returns an array of a different shape and the same content.

See also:

[`cupy.reshape\(\)`](#) for full documentation, [`numpy.ndarray.reshape\(\)`](#)

**round**(*self*, *decimals*=0, *out*=None) → *ndarray*

Returns an array with values rounded to the given number of decimals.

**See also:**

[`cupy.around\(\)`](#) for full documentation, [`numpy.ndarray.round\(\)`](#)

**scatter\_add**(*self*, *slices*, *value*)

Adds given values to specified elements of an array.

**See also:**

[`cupyx.scatter\_add\(\)`](#) for full documentation.

**scatter\_max**(*self*, *slices*, *value*)

Stores a maximum value of elements specified by indices to an array.

**See also:**

[`cupyx.scatter\_max\(\)`](#) for full documentation.

**scatter\_min**(*self*, *slices*, *value*)

Stores a minimum value of elements specified by indices to an array.

**See also:**

[`cupyx.scatter\_min\(\)`](#) for full documentation.

**searchsorted**(*self*, *v*, *side*='left', *sorter*=None)

Finds indices where elements of *v* should be inserted to maintain order.

For full documentation, see [`cupy.searchsorted\(\)`](#)

Returns:

**See also:**

[`numpy.searchsorted\(\)`](#)

**set**(*self*, *arr*, *stream*=None)

Copies an array on the host memory to [`cupy.ndarray`](#).

#### Parameters

- **arr** ([`numpy.ndarray`](#)) – The source array on the host memory.
- **stream** ([`cupy.cuda.Stream`](#)) – CUDA stream object. If given, the stream is used to perform the copy. Otherwise, the current stream is used.

**sort**(*self*, *int axis*=-1)

Sort an array, in-place with a stable sorting algorithm.

#### Parameters

**axis** (*int*) – Axis along which to sort. Default is -1, which means sort along the last axis.

---

**Note:** For its implementation reason, `ndarray.sort` currently supports only arrays with their own data, and does not support `kind` and `order` parameters that `numpy.ndarray.sort` does support.

---

**See also:**

[`cupy.sort\(\)`](#) for full documentation, [`numpy.ndarray.sort\(\)`](#)

**squeeze**(*self*, *axis=None*) → *ndarray*

Returns a view with size-one axes removed.

**See also:**

[`cupy.squeeze\(\)`](#) for full documentation, [`numpy.ndarray.squeeze\(\)`](#)

**std**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*) → *ndarray*

Returns the standard deviation along a given axis.

**See also:**

[`cupy.std\(\)`](#) for full documentation, [`numpy.ndarray.std\(\)`](#)

**sum**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) → *ndarray*

Returns the sum along a given axis.

**See also:**

[`cupy.sum\(\)`](#) for full documentation, [`numpy.ndarray.sum\(\)`](#)

**swapaxes**(*self*, *Py\_ssize\_t axis1*, *Py\_ssize\_t axis2*) → *ndarray*

Returns a view of the array with two axes swapped.

**See also:**

[`cupy.swapaxes\(\)`](#) for full documentation, [`numpy.ndarray.swapaxes\(\)`](#)

**take**(*self*, *indices*, *axis=None*, *out=None*) → *ndarray*

Returns an array of elements at given indices along the axis.

**See also:**

[`cupy.take\(\)`](#) for full documentation, [`numpy.ndarray.take\(\)`](#)

**toDlpack**(*self*)

Zero-copy conversion to a DLPack tensor.

DLPack is a open in memory tensor structure proposed in this repository: [dmlc/dlpack](#).

This function returns a PyCapsule object which contains a pointer to a DLPack tensor converted from the own ndarray. This function does not copy the own data to the output DLPack tensor but it shares the pointer which is pointing to the same memory region for the data.

**Returns**

Output DLPack tensor which is encapsulated in a PyCapsule object.

**Return type**

dltensor (PyCapsule)

**See also:**

[`fromDlpack\(\)`](#) is a method for zero-copy conversion from a DLPack tensor (which is encapsulated in a PyCapsule object) to a [\*ndarray\*](#)

**Warning:** As of the DLPack v0.3 specification, it is (implicitly) assumed that the user is responsible to ensure the Producer and the Consumer are operating on the same stream. This requirement might be relaxed/changed in a future DLPack version.

---

**Example**

```

>>> import cupy
>>> array1 = cupy.array([0, 1, 2], dtype=cupy.float32)
>>> dltensor = array1.toDlpack()
>>> array2 = cupy.fromDlpack(dltensor)
>>> cupy.testing.assert_array_equal(array1, array2)

```

**tobytes**(*self*, *order*='C') → bytes

Turns the array into a Python bytes object.

**tofile**(*self*, *fid*, *sep*=", *format*='%s')

Writes the array to a file.

**See also:**

[numpy.ndarray.tofile\(\)](#)

**tolist**(*self*)

Converts the array to a (possibly nested) Python list.

**Returns**

The possibly nested Python list of array elements.

**Return type**

list

**See also:**

[numpy.ndarray.tolist\(\)](#)

**trace**(*self*, *offset*=0, *axis1*=0, *axis2*=1, *dtype*=None, *out*=None) → ndarray

Returns the sum along diagonals of the array.

**See also:**

[cupy.trace\(\)](#) for full documentation, [numpy.ndarray.trace\(\)](#)

**transpose**(*self*, \**axes*)

Returns a view of the array with axes permuted.

**See also:**

[cupy.transpose\(\)](#) for full documentation, [numpy.ndarray.reshape\(\)](#)

**var**(*self*, *axis*=None, *dtype*=None, *out*=None, *ddof*=0, *keepdims*=False) → ndarray

Returns the variance along a given axis.

**See also:**

[cupy.var\(\)](#) for full documentation, [numpy.ndarray.var\(\)](#)

**view**(*self*, *dtype*=None, *type*=None)

Returns a view of the array.

**Parameters**

**dtype** – If this is different from the data type of the array, the returned view reinterpret the memory sequence as an array of this type.

**Returns**

A view of the array. A reference to the original array is stored at the [base](#) attribute.

**Return type***cupy.ndarray***See also:***numpy.ndarray.view()***\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self&lt;value.

**\_\_le\_\_**(*value*, /)

Return self&lt;=value.

**\_\_gt\_\_**(*value*, /)

Return self&gt;value.

**\_\_ge\_\_**(*value*, /)

Return self&gt;=value.

**\_\_bool\_\_**()

True if self else False

**Attributes****T**

Shape-reversed view of the array.

If ndim &lt; 2, then this is just a reference to the array itself.

**base****cstruct**

C representation of the array.

This property is used for sending an array to CUDA kernels. The type of returned C structure is different for different dtypes and ndims. The definition of C type is written in `cupy/carray.cuh`.

**data****device**

CUDA device on which this array resides.

**dtype****flags**

Object containing memory-layout information.

It only contains `c_contiguous`, `f_contiguous`, and `owndata` attributes. All of these are read-only. Accessing by indexes is also supported.

**See also:***numpy.ndarray.flags*



**flat****imag****itemsize**

Size of each element in bytes.

**See also:**`numpy.ndarray.itemsize`**nbytes**

Total size of all elements in bytes.

It does not count skips between elements.

**See also:**`numpy.ndarray.nbytes`**ndim**

Number of dimensions.

`a.ndim` is equivalent to `len(a.shape)`.**See also:**`numpy.ndarray.ndim`**real****shape**

Lengths of axes.

Setter of this property involves reshaping without copy. If the array cannot be reshaped without copy, it raises an exception.

**size****strides**

Strides of axes in bytes.

**See also:**`numpy.ndarray.strides`

## 5.1.2 Conversion to/from NumPy arrays

`cupy.ndarray` and `numpy.ndarray` are not implicitly convertible to each other. That means, NumPy functions cannot take `cupy.ndarrays` as inputs, and vice versa.

- To convert `numpy.ndarray` to `cupy.ndarray`, use `cupy.array()` or `cupy.asarray()`.
- To convert `cupy.ndarray` to `numpy.ndarray`, use `cupy.asnumpy()` or `cupy.ndarray.get()`.

Note that converting between `cupy.ndarray` and `numpy.ndarray` incurs data transfer between the host (CPU) device and the GPU device, which is costly in terms of performance.

<code>cupy.array(obj[, dtype, copy, order, subok, ...])</code>	Creates an array on the current device.
<code>cupy.asarray(a[, dtype, order, blocking])</code>	Converts an object to array.
<code>cupy.asnumpy(a[, stream, order, out, blocking])</code>	Returns an array on the host memory from an arbitrary source array.

## cupy.array

`cupy.array(obj, dtype=None, copy=True, order='K', subok=False, ndmin=0, *, blocking=False)`

Creates an array on the current device.

This function currently does not support the `subok` option.

### Parameters

- **obj** – `cupy.ndarray` object or any other object that can be passed to `numpy.array()`.
- **dtype** – Data type specifier.
- **copy** (*bool*) – If `False`, this function returns `obj` if possible. Otherwise this function always returns a new array.
- **order** (`{'C', 'F', 'A', 'K'}`) – Row-major (C-style) or column-major (Fortran-style) order. When `order` is `'A'`, it uses `'F'` if `a` is column-major and uses `'C'` otherwise. And when `order` is `'K'`, it keeps strides as closely as possible. If `obj` is `numpy.ndarray`, the function returns `'C'` or `'F'` order array.
- **subok** (*bool*) – If `True`, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).
- **ndmin** (*int*) – Minimum number of dimensions. Ones are inserted to the head of the shape if needed.
- **blocking** (*bool*) – Default is `False`, meaning if a H2D copy is needed it would run asynchronously on the current stream, and users are responsible for ensuring the stream order. For example, writing to the source `obj` without proper ordering while copying would result in a race condition. If set to `True`, the copy is synchronous (with respect to the host).

### Returns

An array on the current device.

### Return type

`cupy.ndarray`

---

**Note:** This method currently does not support `subok` argument.

---

---

**Note:** If `obj` is an `numpy.ndarray` instance that contains big-endian data, this function automatically swaps its byte order to little-endian, which is the NVIDIA and AMD GPU architecture's native use.

---

### See also:

`numpy.array()`

## cupy.asarray

`cupy.asarray(a, dtype=None, order=None, *, blocking=False)`

Converts an object to array.

This is equivalent to `array(a, dtype, copy=False, order=order)`.

### Parameters

- **a** – The source object.
- **dtype** – Data type specifier. It is inferred from the input by default.
- **order** (`{'C', 'F', 'A', 'K'}`) – Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'K'. **order** is ignored for objects that are not `cupy.ndarray`, but have the `__cuda_array_interface__` attribute.
- **blocking** (`bool`) – Default is `False`, meaning if a H2D copy is needed it would run asynchronously on the current stream, and users are responsible for ensuring the stream order. For example, writing to the source `a` without proper ordering while copying would result in a race condition. If set to `True`, the copy is synchronous (with respect to the host).

### Returns

An array on the current device. If `a` is already on the device, no copy is performed.

### Return type

`cupy.ndarray`

---

**Note:** If `a` is an `numpy.ndarray` instance that contains big-endian data, this function automatically swaps its byte order to little-endian, which is the NVIDIA and AMD GPU architecture's native use.

---

See also:

`numpy.asarray()`

## cupy.asnumpy

`cupy.asnumpy(a, stream=None, order='C', out=None, *, blocking=True)`

Returns an array on the host memory from an arbitrary source array.

### Parameters

- **a** – Arbitrary object that can be converted to `numpy.ndarray`.
- **stream** (`cupy.cuda.Stream`) – CUDA stream object. If given, the stream is used to perform the copy. Otherwise, the current stream is used. Note that if `a` is not a `cupy.ndarray` object, then this argument has no effect.
- **order** (`{'C', 'F', 'A'}`) – The desired memory layout of the host array. When **order** is 'A', it uses 'F' if the array is fortran-contiguous and 'C' otherwise. The **order** will be ignored if **out** is specified.
- **out** (`numpy.ndarray`) – The output array to be written to. It must have compatible shape and dtype with those of `a`'s.
- **blocking** (`bool`) – If set to `False`, the copy runs asynchronously on the given (if given) or current stream, and users are responsible for ensuring the stream order. Default is `True`, so the copy is synchronous (with respect to the host).

**Returns**

Converted array on the host memory.

**Return type**

`numpy.ndarray`

### 5.1.3 Code compatibility features

`cupy.ndarray` is designed to be interchangeable with `numpy.ndarray` in terms of code compatibility as much as possible. But occasionally, you will need to know whether the arrays you're handling are `cupy.ndarray` or `numpy.ndarray`. One example is when invoking module-level functions such as `cupy.sum()` or `numpy.sum()`. In such situations, `cupy.get_array_module()` can be used.

---

<code>cupy.get_array_module(*args)</code>	Returns the array module for arguments.
---	---

---

#### `cupy.get_array_module`

`cupy.get_array_module(*args)`

Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a `cupy.ndarray` object, the `cupy` module is returned.

**Parameters**

**args** – Values to determine whether NumPy or CuPy should be used.

**Returns**

`cupy` or `numpy` is returned based on the types of the arguments.

**Return type**

module

---

**Example**

A NumPy/CuPy generic function can be written as follows

```
>>> def softplus(x):
...     xp = cupy.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

---

<code>cupyx.scipy.get_array_module(*args)</code>	Returns the array module for arguments.
--	---

---

#### `cupyx.scipy.get_array_module`

`cupyx.scipy.get_array_module(*args)`

Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a `cupy.ndarray` object, the `cupyx.scipy` module is returned.

**Parameters**

**args** – Values to determine whether NumPy or CuPy should be used.

**Returns**

`cupyx.scipy` or `scipy` is returned based on the types of the arguments.

**Return type**

module

## 5.2 Universal functions (`cupy.ufunc`)

---

**Hint:** [NumPy API Reference: Universal functions \(numpy.ufunc\)](#)

---

CuPy provides universal functions (a.k.a. ufuncs) to support various elementwise operations. CuPy's ufunc supports following features of NumPy's one:

- Broadcasting
- Output type determination
- Casting rules

### 5.2.1 ufunc

---

<code>ufunc(name, nin, nout, _Ops ops[, preamble, ...])</code>	Universal function.
--	---------------------

---

#### `cupy.ufunc`

```
class cupy.ufunc(name, nin, nout, _Ops ops, preamble=u'', loop_prep=u'', doc=u'', default_casting=None, _Ops
                 out_ops=None, *, cutensor_op=None, scatter_op=None)
```

Universal function.

**Variables**

- `~ufunc.name` (*str*) – The name of the universal function.
- `~ufunc.nin` (*int*) – Number of input arguments.
- `~ufunc.nout` (*int*) – Number of output arguments.
- `~ufunc.nargs` (*int*) – Number of all arguments.

**Methods**`__call__()`

Applies the universal function to arguments elementwise.

**Parameters**

- **args** – Input arguments. Each of them can be a `cupy.ndarray` object or a scalar. The output arguments can be omitted or be specified by the `out` argument.
- **out** (`cupy.ndarray`) – Output array. It outputs to new arrays default.
- **dtype** – Data type specifier.

**Returns**

Output array or a tuple of output arrays.

**accumulate**(*self*, *array*, *axis=0*, *dtype=None*, *out=None*)

Accumulate array applying ufunc.

**See also:**

`numpy.ufunc.accumulate()`

**at**(*self*, *a*, *indices*, *b=None*)

Apply in place operation on the operand *a* for elements specified by *indices*.

**See also:**

`numpy.ufunc.at()`

**outer**(*self*, *A*, *B*, *\*\*kwargs*)

Apply the ufunc operation to all pairs of elements in *A* and *B*.

**See also:**

`numpy.ufunc.outer()`

**reduce**(*self*, *array*, *axis=0*, *dtype=None*, *out=None*, *keepdims=False*)

Reduce array applying ufunc.

**See also:**

`numpy.ufunc.reduce()`

**reduceat**(*self*, *array*, *indices*, *axis=0*, *dtype=None*, *out=None*)

Reduce array applying ufunc with indices.

**See also:**

`numpy.ufunc.reduceat()`

**\_\_eq\_\_**(*value*, /)

Return *self*==*value*.

**\_\_ne\_\_**(*value*, /)

Return *self*!=*value*.

**\_\_lt\_\_**(*value*, /)

Return *self*<*value*.

**\_\_le\_\_**(*value*, /)

Return *self*<=*value*.

**\_\_gt\_\_**(*value*, /)

Return *self*>*value*.

**\_\_ge\_\_**(*value*, /)

Return *self*>=*value*.

## Attributes

**name**

**nargs**

**nin**

**nout**

**types**

A list of type signatures.

Each type signature is represented by type character codes of inputs and outputs separated by '->'.

## Methods

These methods are only available for selected ufuncs.

- `ufunc.reduce`: `add()`, `multiply()`
- `ufunc.accumulate`: `add()`, `multiply()`
- `ufunc.reduceat`: `add()`
- `ufunc.outer`: All ufuncs
- `ufunc.at`: `add()`, `subtract()`, `maximum()`, `minimum()`, `bitwise_and()`, `bitwise_or()`, `bitwise_xor()`

**Hint:** In case you need support for other ufuncs, submit a feature request along with your use-case in [the tracker issue](#).

## 5.2.2 Available ufuncs

### Math operations

<code>add(x1, x2, /[, out, casting, dtype])</code>	Adds two arrays elementwise.
<code>subtract(x1, x2, /[, out, casting, dtype])</code>	Subtracts arguments elementwise.
<code>multiply(x1, x2, /[, out, casting, dtype])</code>	Multiplies two arrays elementwise.
<code>matmul</code>	<code>matmul(x1, x2, /, out=None, **kwargs)</code>
<code>divide</code>	<code>true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>logaddexp(x1, x2, /[, out, casting, dtype])</code>	Computes $\log(\exp(x1) + \exp(x2))$ elementwise.
<code>logaddexp2(x1, x2, /[, out, casting, dtype])</code>	Computes $\log_2(\exp_2(x1) + \exp_2(x2))$ elementwise.
<code>true_divide(x1, x2, /[, out, casting, dtype])</code>	Elementwise true division (i.e.
<code>floor_divide(x1, x2, /[, out, casting, dtype])</code>	Elementwise floor division (i.e.
<code>negative(x, /[, out, casting, dtype])</code>	Takes numerical negative elementwise.
<code>positive(x, /[, out, casting, dtype])</code>	Takes numerical positive elementwise.
<code>power(x1, x2, /[, out, casting, dtype])</code>	Computes $x1 ** x2$ elementwise.
<code>float_power(x1, x2, /[, out, casting, dtype])</code>	First array elements raised to powers from second array, element-wise.

continues on next page

Table 1 – continued from previous page

<i>remainder</i>	<code>mod(x1, x2, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>mod(x1, x2, /[, out, casting, dtype])</code>	Computes the remainder of Python division element-wise.
<code>fmod(x1, x2, /[, out, casting, dtype])</code>	Computes the remainder of C division elementwise.
<code>divmod(x1, x2[, out1, out2], / [[, out, ...]])</code>	
<code>absolute(x, /[, out, casting, dtype])</code>	Elementwise absolute value function.
<code>fabs(x, /[, out, casting, dtype])</code>	Calculates absolute values element-wise.
<code>rint(x, /[, out, casting, dtype])</code>	Rounds each element of an array to the nearest integer.
<code>sign(x, /[, out, casting, dtype])</code>	Elementwise sign function.
<code>heaviside(x1, x2, /[, out, casting, dtype])</code>	Compute the Heaviside step function.
<i>conj</i>	<code>conjugate(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>conjugate(x, /[, out, casting, dtype])</code>	Returns the complex conjugate, element-wise.
<code>exp(x, /[, out, casting, dtype])</code>	Elementwise exponential function.
<code>exp2(x, /[, out, casting, dtype])</code>	Elementwise exponentiation with base 2.
<code>log(x, /[, out, casting, dtype])</code>	Elementwise natural logarithm function.
<code>log2(x, /[, out, casting, dtype])</code>	Elementwise binary logarithm function.
<code>log10(x, /[, out, casting, dtype])</code>	Elementwise common logarithm function.
<code>expm1(x, /[, out, casting, dtype])</code>	Computes $\exp(x) - 1$ elementwise.
<code>log1p(x, /[, out, casting, dtype])</code>	Computes $\log(1 + x)$ elementwise.
<code>sqrt(x, /[, out, casting, dtype])</code>	Elementwise square root function.
<code>square(x, /[, out, casting, dtype])</code>	Elementwise square function.
<code>cbrt(x, /[, out, casting, dtype])</code>	Elementwise cube root function.
<code>reciprocal(x, /[, out, casting, dtype])</code>	Computes $1 / x$ elementwise.
<code>gcd(x1, x2, /[, out, casting, dtype])</code>	Computes gcd of $x1$ and $x2$ elementwise.
<code>lcm(x1, x2, /[, out, casting, dtype])</code>	Computes lcm of $x1$ and $x2$ elementwise.

## cupy.add

`cupy.add(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Adds two arrays elementwise.

**See also:**

`numpy.add`

## cupy.subtract

`cupy.subtract(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Subtracts arguments elementwise.

**See also:**

`numpy.subtract`



## cupy.multiply

`cupy.multiply(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Multiplies two arrays elementwise.

**See also:**

`numpy.multiply`

## cupy.matmul

`cupy.matmul = <cupy._core._gufuncs._GUFunc object>`

`matmul(x1, x2, /, out=None, **kwargs)`

Matrix product of two arrays.

Returns the matrix product of two arrays and is the implementation of the `@` operator introduced in Python 3.5 following PEP465.

The main difference against `cupy.dot` are the handling of arrays with more than 2 dimensions. For more information see `numpy.matmul()`.

### Parameters

- **x1** (`cupy.ndarray`) – The left argument.
- **x2** (`cupy.ndarray`) – The right argument.
- **out** (`cupy.ndarray`, *optional*) – Output array.
- **\*\*kwargs** – ufunc keyword arguments.

### Returns

Output array.

### Return type

`cupy.ndarray`

**See also:**

`numpy.matmul()`

## cupy.divide

`cupy.divide()`

`true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise true division (i.e. division as floating values).

**See also:**

`numpy.true_divide`

**cupy.logaddexp**

`cupy.logaddexp(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $\log(\exp(x1) + \exp(x2))$  elementwise.

**See also:**

`numpy.logaddexp`

**cupy.logaddexp2**

`cupy.logaddexp2(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $\log_2(\exp_2(x1) + \exp_2(x2))$  elementwise.

**See also:**

`numpy.logaddexp2`

**cupy.true\_divide**

`cupy.true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise true division (i.e. division as floating values).

**See also:**

`numpy.true_divide`

**cupy.floor\_divide**

`cupy.floor_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise floor division (i.e. integer quotient).

**See also:**

`numpy.floor_divide`

**cupy.negative**

`cupy.negative(x, /, out=None, *, casting='same_kind', dtype=None)`

Takes numerical negative elementwise.

**See also:**

`numpy.negative`

### **cupy.positive**

**cupy.positive**(*x*, /, *out*=None, \*, *casting*='same\_kind', *dtype*=None)

Takes numerical positive elementwise.

**See also:**

`numpy.positive`

### **cupy.power**

**cupy.power**(*x1*, *x2*, /, *out*=None, \*, *casting*='same\_kind', *dtype*=None)

Computes  $x1 ** x2$  elementwise.

**See also:**

`numpy.power`

### **cupy.float\_power**

**cupy.float\_power**(*x1*, *x2*, /, *out*=None, \*, *casting*='same\_kind', *dtype*=None)

First array elements raised to powers from second array, element-wise.

**See also:**

`numpy.float_power`

### **cupy.remainder**

**cupy.remainder**()

`mod(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the remainder of Python division elementwise.

**See also:**

`numpy.remainder`

### **cupy.mod**

**cupy.mod**(*x1*, *x2*, /, *out*=None, \*, *casting*='same\_kind', *dtype*=None)

Computes the remainder of Python division elementwise.

**See also:**

`numpy.remainder`

## cupy.fmod

`cupy.fmod(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the remainder of C division elementwise.

**See also:**

`numpy.fmod`

## cupy.divmod

`cupy.divmod(x1, x2[, out1, out2], /[, out=(None, None)], *, casting='same_kind', dtype=None)`

## cupy.absolute

`cupy.absolute(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise absolute value function.

**See also:**

`numpy.absolute`

## cupy.fabs

`cupy.fabs(x, /, out=None, *, casting='same_kind', dtype=None)`

**Calculates absolute values element-wise.**

Only real values are handled.

**See also:**

`numpy.fabs`

## cupy rint

`cupy.rint(x, /, out=None, *, casting='same_kind', dtype=None)`

Rounds each element of an array to the nearest integer.

**See also:**

`numpy.rint`

## cupy.sign

`cupy.sign(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise sign function.

It returns -1, 0, or 1 depending on the sign of the input.

**See also:**

`numpy.sign`

### cupy.heaviside

`cupy.heaviside(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Compute the Heaviside step function.

**See also:**

`numpy.heaviside`

### cupy.conj

`cupy.conj()`

`conjugate(x, /, out=None, *, casting='same_kind', dtype=None)`

Returns the complex conjugate, element-wise.

**See also:**

`numpy.conjugate`

### cupy.conjugate

`cupy.conjugate(x, /, out=None, *, casting='same_kind', dtype=None)`

Returns the complex conjugate, element-wise.

**See also:**

`numpy.conjugate`

### cupy.exp

`cupy.exp(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise exponential function.

**See also:**

`numpy.exp`

### cupy.exp2

`cupy.exp2(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise exponentiation with base 2.

**See also:**

`numpy.exp2`

### cupy.log

`cupy.log(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise natural logarithm function.

**See also:**

`numpy.log`

### cupy.log2

`cupy.log2(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise binary logarithm function.

**See also:**

`numpy.log2`

### cupy.log10

`cupy.log10(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise common logarithm function.

**See also:**

`numpy.log10`

### cupy.expm1

`cupy.expm1(x, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $\exp(x) - 1$  elementwise.

**See also:**

`numpy.expm1`

### cupy.log1p

`cupy.log1p(x, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $\log(1 + x)$  elementwise.

**See also:**

`numpy.log1p`

### **cupy.sqrt**

`cupy.sqrt(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise square root function.

**See also:**

`numpy.sqrt`

### **cupy.square**

`cupy.square(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise square function.

**See also:**

`numpy.square`

### **cupy.cbrt**

`cupy.cbrt(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise cube root function.

**See also:**

`numpy.cbrt`

### **cupy.reciprocal**

`cupy.reciprocal(x, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $1 / x$  elementwise.

**See also:**

`numpy.reciprocal`

### **cupy.gcd**

`cupy.gcd(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes gcd of `x1` and `x2` elementwise.

**See also:**

`numpy.gcd`

## cupy.lcm

`cupy.lcm(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes lcm of x1 and x2 elementwise.

**See also:**

`numpy.lcm`

## Trigonometric functions

<code>sin(x, /[, out, casting, dtype])</code>	Elementwise sine function.
<code>cos(x, /[, out, casting, dtype])</code>	Elementwise cosine function.
<code>tan(x, /[, out, casting, dtype])</code>	Elementwise tangent function.
<code>arcsin(x, /[, out, casting, dtype])</code>	Elementwise inverse-sine function (a.k.a.
<code>arccos(x, /[, out, casting, dtype])</code>	Elementwise inverse-cosine function (a.k.a.
<code>arctan(x, /[, out, casting, dtype])</code>	Elementwise inverse-tangent function (a.k.a.
<code>arctan2(x1, x2, /[, out, casting, dtype])</code>	Elementwise inverse-tangent of the ratio of two arrays.
<code>hypot(x1, x2, /[, out, casting, dtype])</code>	Computes the hypoteneous of orthogonal vectors of given length.
<code>sinh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic sine function.
<code>cosh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic cosine function.
<code>tanh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic tangent function.
<code>arcsinh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic sine function.
<code>arccosh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic cosine function.
<code>arctanh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic tangent function.
<code>degrees</code>	<code>rad2deg(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>radians(x, /[, out, casting, dtype])</code>	Converts angles from degrees to radians elementwise.
<code>deg2rad</code>	<code>radians(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>rad2deg(x, /[, out, casting, dtype])</code>	Converts angles from radians to degrees elementwise.

## cupy.sin

`cupy.sin(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise sine function.

**See also:**

`numpy.sin`



### **cupy.cos**

`cupy.cos(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise cosine function.

**See also:**

`numpy.cos`

### **cupy.tan**

`cupy.tan(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise tangent function.

**See also:**

`numpy.tan`

### **cupy.arcsin**

`cupy.arcsin(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse-sine function (a.k.a. arcsine function).

**See also:**

`numpy.arcsin`

### **cupy.arccos**

`cupy.arccos(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse-cosine function (a.k.a. arccosine function).

**See also:**

`numpy.arccos`

### **cupy.arctan**

`cupy.arctan(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse-tangent function (a.k.a. arctangent function).

**See also:**

`numpy.arctan`

### cupy.arctan2

`cupy.arctan2(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse-tangent of the ratio of two arrays.

**See also:**

`numpy.arctan2`

### cupy.hypot

`cupy.hypot(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the hypoteneous of orthogonal vectors of given length.

This is equivalent to `sqrt(x1 **2 + x2 ** 2)`, while this function is more efficient.

**See also:**

`numpy.hypot`

### cupy.sinh

`cupy.sinh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise hyperbolic sine function.

**See also:**

`numpy.sinh`

### cupy.cosh

`cupy.cosh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise hyperbolic cosine function.

**See also:**

`numpy.cosh`

### cupy.tanh

`cupy.tanh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise hyperbolic tangent function.

**See also:**

`numpy.tanh`

### **cupy.arcsinh**

`cupy.arcsinh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse of hyperbolic sine function.

**See also:**

`numpy.arcsinh`

### **cupy.arccosh**

`cupy.arccosh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse of hyperbolic cosine function.

**See also:**

`numpy.arccosh`

### **cupy.arctanh**

`cupy.arctanh(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise inverse of hyperbolic tangent function.

**See also:**

`numpy.arctanh`

### **cupy.degrees**

`cupy.degrees()`

`rad2deg(x, /, out=None, *, casting='same_kind', dtype=None)`

Converts angles from radians to degrees elementwise.

**See also:**

`numpy.rad2deg`, `numpy.degrees`

### **cupy.radians**

`cupy.radians(x, /, out=None, *, casting='same_kind', dtype=None)`

Converts angles from degrees to radians elementwise.

**See also:**

`numpy.deg2rad`, `numpy.radians`

## cupy.deg2rad

`cupy.deg2rad()`

`radians(x, /, out=None, *, casting='same_kind', dtype=None)`

Converts angles from degrees to radians elementwise.

**See also:**

`numpy.deg2rad`, `numpy.radians`

## cupy.rad2deg

`cupy.rad2deg(x, /, out=None, *, casting='same_kind', dtype=None)`

Converts angles from radians to degrees elementwise.

**See also:**

`numpy.rad2deg`, `numpy.degrees`

## Bit-twiddling functions

<code><i>bitwise_and</i>(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise AND of two arrays elementwise.
<code><i>bitwise_or</i>(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise OR of two arrays elementwise.
<code><i>bitwise_xor</i>(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise XOR of two arrays elementwise.
<code><i>invert</i>(x, /[, out, casting, dtype])</code>	Computes the bitwise NOT of an array elementwise.
<code><i>left_shift</i>(x1, x2, /[, out, casting, dtype])</code>	Shifts the bits of each integer element to the left.
<code><i>right_shift</i>(x1, x2, /[, out, casting, dtype])</code>	Shifts the bits of each integer element to the right.

## cupy.bitwise\_and

`cupy.bitwise_and(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the bitwise AND of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

`numpy.bitwise_and`

## cupy.bitwise\_or

`cupy.bitwise_or(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the bitwise OR of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

`numpy.bitwise_or`

### cupy.bitwise\_xor

`cupy.bitwise_xor(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the bitwise XOR of two arrays elementwise.

Only integer and boolean arrays are handled.

**See also:**

`numpy.bitwise_xor`

### cupy.invert

`cupy.invert(x, /, out=None, *, casting='same_kind', dtype=None)`

Computes the bitwise NOT of an array elementwise.

Only integer and boolean arrays are handled.

---

**Note:** `cupy.bitwise_not()` is an alias for `cupy.invert()`.

---

**See also:**

`numpy.invert`

### cupy.left\_shift

`cupy.left_shift(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Shifts the bits of each integer element to the left.

Only integer arrays are handled.

**See also:**

`numpy.left_shift`

### cupy.right\_shift

`cupy.right_shift(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Shifts the bits of each integer element to the right.

Only integer arrays are handled

**See also:**

`numpy.right_shift`

## Comparison functions

<code>greater(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 &gt; x2</code> .
<code>greater_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 &gt;= x2</code> .
<code>less(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 &lt; x2</code> .
<code>less_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 &lt;= x2</code> .
<code>not_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 != x2</code> .
<code>equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if <code>x1 == x2</code> .
<code>logical_and(x1, x2, /[, out, casting, dtype])</code>	Computes the logical AND of two arrays.
<code>logical_or(x1, x2, /[, out, casting, dtype])</code>	Computes the logical OR of two arrays.
<code>logical_xor(x1, x2, /[, out, casting, dtype])</code>	Computes the logical XOR of two arrays.
<code>logical_not(x, /[, out, casting, dtype])</code>	Computes the logical NOT of an array.
<code>maximum(x1, x2, /[, out, casting, dtype])</code>	Takes the maximum of two arrays elementwise.
<code>minimum(x1, x2, /[, out, casting, dtype])</code>	Takes the minimum of two arrays elementwise.
<code>fmax(x1, x2, /[, out, casting, dtype])</code>	Takes the maximum of two arrays elementwise.
<code>fmin(x1, x2, /[, out, casting, dtype])</code>	Takes the minimum of two arrays elementwise.

### `cupy.greater`

`cupy.greater(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if `x1 > x2`.

**See also:**

`numpy.greater`

### `cupy.greater_equal`

`cupy.greater_equal(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if `x1 >= x2`.

**See also:**

`numpy.greater_equal`

### `cupy.less`

`cupy.less(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if `x1 < x2`.

**See also:**

`numpy.less`

### **cupy.less\_equal**

`cupy.less_equal(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if  $x1 \leq x2$ .

**See also:**

`numpy.less_equal`

### **cupy.not\_equal**

`cupy.not_equal(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if  $x1 \neq x2$ .

**See also:**

`numpy.equal`

### **cupy.equal**

`cupy.equal(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if  $x1 == x2$ .

**See also:**

`numpy.equal`

### **cupy.logical\_and**

`cupy.logical_and(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the logical AND of two arrays.

**See also:**

`numpy.logical_and`

### **cupy.logical\_or**

`cupy.logical_or(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the logical OR of two arrays.

**See also:**

`numpy.logical_or`

### **cupy.logical\_xor**

`cupy.logical_xor(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the logical XOR of two arrays.

**See also:**

`numpy.logical_xor`

### **cupy.logical\_not**

`cupy.logical_not(x, /, out=None, *, casting='same_kind', dtype=None)`

Computes the logical NOT of an array.

**See also:**

`numpy.logical_not`

### **cupy.maximum**

`cupy.maximum(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Takes the maximum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

`numpy.maximum`

### **cupy.minimum**

`cupy.minimum(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Takes the minimum of two arrays elementwise.

If NaN appears, it returns the NaN.

**See also:**

`numpy.minimum`

### **cupy.fmax**

`cupy.fmax(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Takes the maximum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

`numpy.fmax`



## cupy.fmin

`cupy.fmin(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Takes the minimum of two arrays elementwise.

If NaN appears, it returns the other operand.

**See also:**

`numpy.fmin`

## Floating functions

<code>isfinite(x, /[, out, casting, dtype])</code>	Tests finiteness elementwise.
<code>isinf(x, /[, out, casting, dtype])</code>	Tests if each element is the positive or negative infinity.
<code>isnan(x, /[, out, casting, dtype])</code>	Tests if each element is a NaN.
<code>fabs(x, /[, out, casting, dtype])</code>	Calculates absolute values element-wise.
<code>signbit(x, /[, out, casting, dtype])</code>	Tests elementwise if the sign bit is set (i.e.
<code>copysign(x1, x2, /[, out, casting, dtype])</code>	Returns the first argument with the sign bit of the second elementwise.
<code>nextafter(x1, x2, /[, out, casting, dtype])</code>	Computes the nearest neighbor float values towards the second argument.
<code>modf(x[, out1, out2], / [[, out, casting, dtype])</code>	Extracts the fractional and integral parts of an array elementwise.
<code>ldexp(x1, x2, /[, out, casting, dtype])</code>	Computes $x1 * 2^{x2}$ elementwise.
<code>frexp(x[, out1, out2], / [[, out, casting, ...])</code>	Decomposes each element to mantissa and two's exponent.
<code>fmod(x1, x2, /[, out, casting, dtype])</code>	Computes the remainder of C division elementwise.
<code>floor(x, /[, out, casting, dtype])</code>	Rounds each element of an array to its floor integer.
<code>ceil(x, /[, out, casting, dtype])</code>	Rounds each element of an array to its ceiling integer.
<code>trunc(x, /[, out, casting, dtype])</code>	Rounds each element of an array towards zero.

## cupy.isfinite

`cupy.isfinite(x, /, out=None, *, casting='same_kind', dtype=None)`

Tests finiteness elementwise.

Each element of returned array is `True` only if the corresponding element of the input is finite (i.e. not an infinity nor NaN).

**See also:**

`numpy.isfinite`

### cupy.isinf

`cupy.isinf(x, /, out=None, *, casting='same_kind', dtype=None)`

Tests if each element is the positive or negative infinity.

**See also:**

`numpy.isinf`

### cupy.isnan

`cupy.isnan(x, /, out=None, *, casting='same_kind', dtype=None)`

Tests if each element is a NaN.

**See also:**

`numpy.isnan`

### cupy.signbit

`cupy.signbit(x, /, out=None, *, casting='same_kind', dtype=None)`

Tests elementwise if the sign bit is set (i.e. less than zero).

**See also:**

`numpy.signbit`

### cupy.copysign

`cupy.copysign(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Returns the first argument with the sign bit of the second elementwise.

**See also:**

`numpy.copysign`

### cupy.nextafter

`cupy.nextafter(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes the nearest neighbor float values towards the second argument.

---

**Note:** For values that are close to zero (or denormal numbers), results of `cupy.nextafter()` may be different from those of `numpy.nextafter()`, because CuPy sets `-ftz=true`.

---

**See also:**

`numpy.nextafter`

**cupy.modf**

`cupy.modf(x[, out1, out2], /[, out=(None, None)], *, casting='same_kind', dtype=None)`

Extracts the fractional and integral parts of an array elementwise.

This ufunc returns two arrays.

**See also:**

`numpy.modf`

**cupy.ldexp**

`cupy.ldexp(x1, x2, /, out=None, *, casting='same_kind', dtype=None)`

Computes  $x1 * 2^{x2}$  elementwise.

**See also:**

`numpy.ldexp`

**cupy.frexp**

`cupy.frexp(x[, out1, out2], /[, out=(None, None)], *, casting='same_kind', dtype=None)`

Decomposes each element to mantissa and two's exponent.

This ufunc outputs two arrays of the input dtype and the `int` dtype.

**See also:**

`numpy.frexp`

**cupy.floor**

`cupy.floor(x, /, out=None, *, casting='same_kind', dtype=None)`

Rounds each element of an array to its floor integer.

**See also:**

`numpy.floor`

**cupy.ceil**

`cupy.ceil(x, /, out=None, *, casting='same_kind', dtype=None)`

Rounds each element of an array to its ceiling integer.

**See also:**

`numpy.ceil`

## cupy.trunc

`cupy.trunc(x, /, out=None, *, casting='same_kind', dtype=None)`

Rounds each element of an array towards zero.

See also:

`numpy.trunc`

## 5.2.3 Generalized Universal Functions

In addition to regular ufuncs, CuPy also provides a wrapper class to convert regular cupy functions into Generalized Universal Functions as in NumPy <https://numpy.org/doc/stable/reference/c-api/generalized-ufuncs.html>. This allows to automatically use keyword arguments such as `axes`, `order`, `dtype` without needing to explicitly implement them in the wrapped function.

---

<code>GeneralizedUFunc(func, signature, **kwargs)</code>	Creates a Generalized Universal Function by wrapping a user provided function with the signature.
--	---

---

### cupyx.GeneralizedUFunc

**class** `cupyx.GeneralizedUFunc(func, signature, **kwargs)`

Creates a Generalized Universal Function by wrapping a user provided function with the signature.

`signature` determines if the function consumes or produces core dimensions. The remaining dimensions in given input arrays (`*args`) are considered loop dimensions and are required to broadcast naturally against each other.

#### Parameters

- **func** (*callable*) – Function to call like `func(*args, **kwargs)` on input arrays (`*args`) that returns an array or tuple of arrays. If multiple arguments with non-matching dimensions are supplied, this function is expected to vectorize (broadcast) over axes of positional arguments in the style of NumPy universal functions.
- **signature** (*string*) – Specifies what core dimensions are consumed and produced by `func`. According to the specification of `numpy.gufunc` signature.
- **supports\_batched** (*bool, optional*) – If the wrapped function supports to pass the complete input array with the loop and the core dimensions. Defaults to *False*. Dimensions will be iterated in the *GUFunc* processing code.
- **supports\_out** (*bool, optional*) – If the wrapped function supports out as one of its kwargs. Defaults to *False*.
- **signatures** (*list of tuple of str*) – Contains strings in the form of ‘ii->i’ with i being the char of a dtype. Each element of the list is a tuple with the string and a alternative function to `func` to be executed when the inputs of the function can be casted as described by this function.
- **name** (*str, optional*) – Name for the *GUFunc* object. If not specified, `func`’s name is used.
- **doc** (*str, optional*) – Docstring for the *GUFunc* object. If not specified, `func.__doc__` is used.

## Methods

**\_\_call\_\_**(\*args, \*\*kwargs)

Apply a generalized ufunc.

### Parameters

- **args** – Input arguments. Each of them can be a `cupy.ndarray` object or a scalar. The output arguments can be omitted or be specified by the `out` argument.
- **axes** (*List of tuples of int, optional*) – A list of tuples with indices of axes a generalized ufunc should operate on. For instance, for a signature of ' $(i, j), (j, k) \rightarrow (i, k)$ ' appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be `[(-2, -1), (-2, -1), (-2, -1)]`. For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the output tuples can be omitted.
- **axis** (*int, optional*) – A single axis over which a generalized ufunc should operate. This is a short-cut for ufuncs that operate over a single, shared core dimension, equivalent to passing in axes with entries of `(axis,)` for each single-core-dimension argument and `()` for all others. For instance, for a signature ' $(i), (i) \rightarrow ()$ ', it is equivalent to passing in `axes=[(axis,), (axis,), ()]`.
- **keepdims** (*bool, optional*) – If this is set to True, axes which are reduced over will be left in the result as a dimension with size one, so that the result will broadcast correctly against the inputs. This option can only be used for generalized ufuncs that operate on inputs that all have the same number of core dimensions and with outputs that have no core dimensions, i.e., with signatures like ' $(i), (i) \rightarrow ()$ ' or ' $(m, m) \rightarrow ()$ '. If used, the location of the dimensions in the output can be controlled with `axes` and `axis`.
- **casting** (*str, optional*) – Provides a policy for what kind of casting is permitted. Defaults to `'same_kind'`.
- **dtype** (*dtype, optional*) – Overrides the dtype of the calculation and output arrays. Similar to signature.
- **signature** (*str or tuple of dtype, optional*) – Either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to provide a specific signature for the function to be used if registered in the `signatures` kwarg of the `__init__` method. If the loop specified does not exist for the ufunc, then a `TypeError` is raised. Normally, a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This keyword argument lets you bypass that search and choose a particular loop.
- **order** (*str, optional*) – Specifies the memory layout of the output array. Defaults to `'K'`. `''C''` means the output should be C-contiguous, `'F'` means F-contiguous, `'A'` means F-contiguous if the inputs are F-contiguous and not also not C-contiguous, C-contiguous otherwise, and `'K'` means to match the element ordering of the inputs as closely as possible.
- **out** (`cupy.ndarray`) – Output array. It outputs to new arrays default.

### Returns

Output array or a tuple of output arrays.

**\_\_eq\_\_**(value, /)

Return `self==value`.

```
__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## 5.3 Routines (NumPy)

The following pages describe NumPy-compatible routines. These functions cover a subset of [NumPy routines](#).

### 5.3.1 Array creation routines

---

**Hint:** [NumPy API Reference: Array creation routines](#)

---

#### Ones and zeros

<code><i>empty</i>(shape[, dtype, order])</code>	Returns an array without initializing the elements.
<code><i>empty_like</i>(prototype[, dtype, order, subok, ...])</code>	Returns a new array with same shape and dtype of a given array.
<code><i>eye</i>(N[, M, k, dtype, order])</code>	Returns a 2-D array with ones on the diagonals and zeros elsewhere.
<code><i>identity</i>(n[, dtype])</code>	Returns a 2-D identity array.
<code><i>ones</i>(shape[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with ones.
<code><i>ones_like</i>(a[, dtype, order, subok, shape])</code>	Returns an array of ones with same shape and dtype as a given array.
<code><i>zeros</i>(shape[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with zeros.
<code><i>zeros_like</i>(a[, dtype, order, subok, shape])</code>	Returns an array of zeros with same shape and dtype as a given array.
<code><i>full</i>(shape, fill_value[, dtype, order])</code>	Returns a new array of given shape and dtype, filled with a given value.
<code><i>full_like</i>(a, fill_value[, dtype, order, ...])</code>	Returns a full array with same shape and dtype as a given array.

## cupy.empty

`cupy.empty(shape, dtype=<class 'float'>, order='C')`

Returns an array without initializing the elements.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** (*data-type, optional*) – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

### Returns

A new array with elements not initialized.

### Return type

*cupy.ndarray*

See also:

`numpy.empty()`

## cupy.empty\_like

`cupy.empty_like(prototype, dtype=None, order='K', subok=None, shape=None)`

Returns a new array with same shape and dtype of a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (*cupy.ndarray*) – Base array.
- **dtype** (*data-type, optional*) – Data type specifier. The data type of `a` is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*None*) – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

### Returns

A new array with same shape and dtype of `a` with elements not initialized.

### Return type

*cupy.ndarray*

See also:

`numpy.empty_like()`

## cupy.eye

`cupy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C')`

Returns a 2-D array with ones on the diagonals and zeros elsewhere.

### Parameters

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. **M** == **N** by default.
- **k** (*int*) – Index of the diagonal. Zero indicates the main diagonal, a positive index an upper diagonal, and a negative index a lower diagonal.
- **dtype** (*data-type, optional*) – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

### Returns

A 2-D array with given diagonals filled with ones and zeros elsewhere.

### Return type

*cupy.ndarray*

### See also:

`numpy.eye()`

## cupy.identity

`cupy.identity(n, dtype=<class 'float'>)`

Returns a 2-D identity array.

It is equivalent to `eye(n, n, dtype)`.

### Parameters

- **n** (*int*) – Number of rows and columns.
- **dtype** (*data-type, optional*) – Data type specifier.

### Returns

A 2-D identity array.

### Return type

*cupy.ndarray*

### See also:

`numpy.identity()`



## cupy.ones

`cupy.ones(shape, dtype=<class 'float'>, order='C')`

Returns a new array of given shape and dtype, filled with ones.

This function currently does not support `order` option.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** (*data-type, optional*) – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

### Returns

An array filled with ones.

### Return type

*cupy.ndarray*

### See also:

`numpy.ones()`

## cupy.ones\_like

`cupy.ones_like(a, dtype=None, order='K', subok=None, shape=None)`

Returns an array of ones with same shape and dtype as a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (*cupy.ndarray*) – Base array.
- **dtype** (*data-type, optional*) – Data type specifier. The dtype of `a` is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*None*) – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

### Returns

An array filled with ones.

### Return type

*cupy.ndarray*

### See also:

`numpy.ones_like()`

## cupy.zeros

`cupy.zeros(shape, dtype=<class 'float'>, order='C')`

Returns a new array of given shape and dtype, filled with zeros.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** (*data-type, optional*) – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

### Returns

An array filled with zeros.

### Return type

*cupy.ndarray*

See also:

`numpy.zeros()`

## cupy.zeros\_like

`cupy.zeros_like(a, dtype=None, order='K', subok=None, shape=None)`

Returns an array of zeros with same shape and dtype as a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (*cupy.ndarray*) – Base array.
- **dtype** (*data-type, optional*) – Data type specifier. The dtype of `a` is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*None*) – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

### Returns

An array filled with zeros.

### Return type

*cupy.ndarray*

See also:

`numpy.zeros_like()`

## cupy.full

`cupy.full(shape, fill_value, dtype=None, order='C')`

Returns a new array of given shape and dtype, filled with a given value.

This function currently does not support `order` option.

### Parameters

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **fill\_value** (*Any*) – A scalar value to fill a new array.
- **dtype** (*data-type, optional*) – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

### Returns

An array filled with `fill_value`.

### Return type

*cupy.ndarray*

See also:

`numpy.full()`

## cupy.full\_like

`cupy.full_like(a, fill_value, dtype=None, order='K', subok=None, shape=None)`

Returns a full array with same shape and dtype as a given array.

This function currently does not support `subok` option.

### Parameters

- **a** (*cupy.ndarray*) – Base array.
- **fill\_value** (*Any*) – A scalar value to fill a new array.
- **dtype** (*data-type, optional*) – Data type specifier. The dtype of `a` is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*None*) – Not supported yet, must be `None`.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If `order='K'` and the number of dimensions is unchanged, will try to keep order, otherwise, `order='C'` is implied.

### Returns

An array filled with `fill_value`.

### Return type

*cupy.ndarray*

See also:

`numpy.full_like()`

## From existing data

<code>array(obj[, dtype, copy, order, subok, ...])</code>	Creates an array on the current device.
<code>asarray(a[, dtype, order, blocking])</code>	Converts an object to array.
<code>asanyarray(a[, dtype, order, blocking])</code>	Converts an object to array.
<code>ascontiguousarray(a[, dtype])</code>	Returns a C-contiguous array.
<code>copy(a[, order])</code>	Creates a copy of a given array on the current device.
<code>frombuffer(*args, **kwargs)</code>	Interpret a buffer as a 1-dimensional array.
<code>fromfile(*args, **kwargs)</code>	Reads an array from a file.
<code>fromfunction(*args, **kwargs)</code>	Construct an array by executing a function over each co-ordinate.
<code>fromiter(*args, **kwargs)</code>	Create a new 1-dimensional array from an iterable object.
<code>fromstring(*args, **kwargs)</code>	A new 1-D array initialized from text data in a string.
<code>loadtxt(*args, **kwargs)</code>	Load data from a text file.

## cupy.asanyarray

`cupy.asanyarray(a, dtype=None, order=None, *, blocking=False)`

Converts an object to array.

This is currently equivalent to `cupy.asarray()`, since there is no subclass of `cupy.ndarray` in CuPy. Note that the original `numpy.asanyarray()` returns the input array as is if it is an instance of a subtype of `numpy.ndarray`.

See also:

`cupy.asarray()`, `numpy.asanyarray()`

## cupy.ascontiguousarray

`cupy.ascontiguousarray(a, dtype=None)`

Returns a C-contiguous array.

### Parameters

- **a** (`cupy.ndarray`) – Source array.
- **dtype** – Data type specifier.

### Returns

If no copy is required, it returns a. Otherwise, it returns a copy of a.

### Return type

`cupy.ndarray`

See also:

`numpy.ascontiguousarray()`

## cupy.copy

`cupy.copy(a, order='K')`

Creates a copy of a given array on the current device.

This function allocates the new array on the current device. If the given array is allocated on the different device, then this function tries to copy the contents over the devices.

### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **order** (`{'C', 'F', 'A', 'K'}`) – Row-major (C-style) or column-major (Fortran-style) order. When order is 'A', it uses 'F' if a is column-major and uses 'C' otherwise. And when order is 'K', it keeps strides as closely as possible.

### Returns

The copy of a on the current device.

### Return type

`cupy.ndarray`

### See also:

`numpy.copy()`, `cupy.ndarray.copy()`

## cupy.frombuffer

`cupy.frombuffer(*args, **kwargs)`

Interpret a buffer as a 1-dimensional array.

---

**Note:** Uses NumPy's `frombuffer` and coerces the result to a CuPy array.

---

### See also:

`numpy.frombuffer()`

## cupy.fromfile

`cupy.fromfile(*args, **kwargs)`

Reads an array from a file.

---

**Note:** Uses NumPy's `fromfile` and coerces the result to a CuPy array.

---

---

**Note:** If you let NumPy's `fromfile` read the file in big-endian, CuPy automatically swaps its byte order to little-endian, which is the NVIDIA and AMD GPU architecture's native use.

---

### See also:

`numpy.fromfile()`

## cupy.fromfunction

`cupy.fromfunction(*args, **kwargs)`

Construct an array by executing a function over each coordinate.

---

**Note:** Uses NumPy's `fromfunction` and coerces the result to a CuPy array.

---

**See also:**

`numpy.fromfunction()`

## cupy.fromiter

`cupy.fromiter(*args, **kwargs)`

Create a new 1-dimensional array from an iterable object.

---

**Note:** Uses NumPy's `fromiter` and coerces the result to a CuPy array.

---

**See also:**

`numpy.fromiter()`

## cupy.fromstring

`cupy.fromstring(*args, **kwargs)`

A new 1-D array initialized from text data in a string.

---

**Note:** Uses NumPy's `fromstring` and coerces the result to a CuPy array.

---

**See also:**

`numpy.fromstring()`

## cupy.loadtxt

`cupy.loadtxt(*args, **kwargs)`

Load data from a text file.

---

**Note:** Uses NumPy's `loadtxt` and coerces the result to a CuPy array.

---

**See also:**

`numpy.loadtxt()`

## Numerical ranges

<code>arange(start[, stop, step, dtype])</code>	Returns an array with evenly spaced values within a given interval.
<code>linspace(start, stop[, num, endpoint, ...])</code>	Returns an array with evenly-spaced values within a given interval.
<code>logspace(start, stop[, num, endpoint, base, ...])</code>	Returns an array with evenly-spaced values on a log-scale.
<code>meshgrid(*xi, **kwargs)</code>	Return coordinate matrices from coordinate vectors.
<code>mgrid</code>	Construct a multi-dimensional "meshgrid".
<code>ogrid</code>	Construct a multi-dimensional "meshgrid".

### cupy.arange

`cupy.arange(start, stop=None, step=1, dtype=None)`

Returns an array with evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop). The first three arguments are mapped like the `range` built-in function, i.e. start and step are optional.

#### Parameters

- **start** – Start of the interval.
- **stop** – End of the interval.
- **step** – Step width between each pair of consecutive values.
- **dtype** – Data type specifier. It is inferred from other arguments by default.

#### Returns

The 1-D array of range values.

#### Return type

`cupy.ndarray`

#### See also:

`numpy.arange()`

### cupy.linspace

`cupy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)`

Returns an array with evenly-spaced values within a given interval.

Instead of specifying the step width like `cupy.arange()`, this function requires the total number of elements specified.

#### Parameters

- **start** (*scalar or array\_like*) – Starting value(s) of the sequence.
- **stop** (*scalar or array\_like*) – Ending value(s) of the sequence, unless `endpoint` is set to `False`. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that `stop` is excluded. Note that the step size changes when `endpoint` is `False`.
- **num** – Number of elements.

- **endpoint** (*bool*) – If `True`, the stop value is included as the last element. Otherwise, the stop value is omitted.
- **retstep** (*bool*) – If `True`, this function returns (array, step). Otherwise, it returns only the array.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.
- **axis** (*int*) – The axis in the result to store the samples. Relevant only if start or stop are array-like. By default `0`, the samples will be along a new axis inserted at the beginning. Use `-1` to get an axis at the end.

**Returns**

The 1-D array of ranged values.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.linspace()`

## **cupy.logspace**

`cupy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None, axis=0)`

Returns an array with evenly-spaced values on a log-scale.

Instead of specifying the step width like `cupy.arange()`, this function requires the total number of elements specified.

**Parameters**

- **start** – Start of the interval.
- **stop** – End of the interval.
- **num** – Number of elements.
- **endpoint** (*bool*) – If `True`, the stop value is included as the last element. Otherwise, the stop value is omitted.
- **base** (*float*) – Base of the log space. The step sizes between the elements on a log-scale are the same as base.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.
- **axis** (*int*) – The axis in the result to store the samples. Relevant only if start or stop are array-like. By default `0`, the samples will be along a new axis inserted at the beginning. Use `-1` to get an axis at the end.

**Returns**

The 1-D array of ranged values.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.logspace()`



## cupy.meshgrid

`cupy.meshgrid(*xi, **kwargs)`

Return coordinate matrices from coordinate vectors.

Given one-dimensional coordinate arrays `x1`, `x2`, ..., `xn` this function makes N-D grids.

For one-dimensional arrays `x1`, `x2`, ..., `xn` with lengths `Ni = len(xi)`, this function returns (`N1`, `N2`, `N3`, ..., `Nn`) shaped arrays if `indexing='ij'` or (`N2`, `N1`, `N3`, ..., `Nn`) shaped arrays if `indexing='xy'`.

Unlike NumPy, CuPy currently only supports 1-D arrays as inputs.

### Parameters

- **xi** (*tuple of ndarrays*) – 1-D arrays representing the coordinates of a grid.
- **indexing** (*{'xy', 'ij'}, optional*) – Cartesian ('xy', default) or matrix ('ij') indexing of output.
- **sparse** (*bool, optional*) – If True, a sparse grid is returned in order to conserve memory. Default is False.
- **copy** (*bool, optional*) – If False, a view into the original arrays are returned. Default is True.

### Returns

list of `cupy.ndarray`

See also:

`numpy.meshgrid()`

## cupy.mgrid

`cupy.mgrid = <cupy._creation.ranges.nd_grid object>`

Construct a multi-dimensional “meshgrid”.

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. `5j`), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is **inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

### Parameters

**sparse** (*bool, optional*) – Whether the grid is sparse or not. Default is False.

See also:

`numpy.mgrid` and `numpy.ogrid`

## cupy.ogrid

`cupy.ogrid = <cupy._creation.ranges.nd_grid object>`

Construct a multi-dimensional “meshgrid”.

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. `5j`), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

### Parameters

**sparse** (*bool*, *optional*) – Whether the grid is sparse or not. Default is False.

See also:

`numpy.mgrid` and `numpy.ogrid`

## Building matrices

<code>diag(v[, k])</code>	Returns a diagonal or a diagonal array.
<code>diagflat(v[, k])</code>	Creates a diagonal array from the flattened input.
<code>tri(N[, M, k, dtype])</code>	Creates an array with ones at and below the given diagonal.
<code>tril(m[, k])</code>	Returns a lower triangle of an array.
<code>triu(m[, k])</code>	Returns an upper triangle of an array.
<code>vander(x[, N, increasing])</code>	Returns a Vandermonde matrix.

## cupy.diag

`cupy.diag(v, k=0)`

Returns a diagonal or a diagonal array.

### Parameters

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.

### Returns

If `v` indicates a 1-D array, then it returns a 2-D array with the specified diagonal filled by `v`. If `v` indicates a 2-D array, then it returns the specified diagonal of `v`. In latter case, if `v` is a `cupy.ndarray` object, then its view is returned.

### Return type

`cupy.ndarray`

See also:

`numpy.diag()`

## cupy.diagflat

`cupy.diagflat(v, k=0)`

Creates a diagonal array from the flattened input.

### Parameters

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. See `cupy.diag()` for detail.

### Returns

A 2-D diagonal array with the diagonal copied from **v**.

### Return type

*cupy.ndarray*

See also:

`numpy.diagflat()`

## cupy.tri

`cupy.tri(N, M=None, k=0, dtype=<class 'float'>)`

Creates an array with ones at and below the given diagonal.

### Parameters

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. **M** == **N** by default.
- **k** (*int*) – The sub-diagonal at and below which the array is filled. Zero is the main diagonal, a positive value is above it, and a negative value is below.
- **dtype** – Data type specifier.

### Returns

An array with ones at and below the given diagonal.

### Return type

*cupy.ndarray*

See also:

`numpy.tri()`

## cupy.tril

`cupy.tril(m, k=0)`

Returns a lower triangle of an array.

### Parameters

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal above which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

### Returns

A lower triangle of an array.

**Return type***cupy.ndarray***See also:**`numpy.tril()`**cupy.triu**`cupy.triu(m, k=0)`

Returns an upper triangle of an array.

**Parameters**

- **m** (*array-like*) – Array or array-like object.
- **k** (*int*) – The diagonal below which to zero elements. Zero is the main diagonal, a positive value is above it, and a negative value is below.

**Returns**

An upper triangle of an array.

**Return type***cupy.ndarray***See also:**`numpy.triu()`**cupy.vander**`cupy.vander(x, N=None, increasing=False)`

Returns a Vandermonde matrix.

**Parameters**

- **x** (*array-like*) – 1-D array or array-like object.
- **N** (*int, optional*) – Number of columns in the output.  $N = \text{len}(x)$  by default.
- **increasing** (*bool, optional*) – Order of the powers of the columns. If True, the powers increase from right to left, if False (the default) they are reversed.

**Returns**

A Vandermonde matrix.

**Return type***cupy.ndarray***See also:**`numpy.vander()`

## 5.3.2 Array manipulation routines

---

**Hint:** [NumPy API Reference: Array manipulation routines](#)

---

### Basic operations

<code>copyto(dst, src[, casting, where])</code>	Copies values from one array to another with broadcasting.
<code>shape(a)</code>	Returns the shape of an array

---

### `cupy.copyto`

`cupy.copyto(dst, src, casting='same_kind', where=None)`

Copies values from one array to another with broadcasting.

This function can be called for arrays on different devices. In this case, `casting`, `where`, and broadcasting is not supported, and an exception is raised if these are used.

#### Parameters

- **dst** (`cupy.ndarray`) – Target array.
- **src** (`cupy.ndarray`) – Source array.
- **casting** (`str`) – Casting rule. See `numpy.can_cast()` for detail.
- **where** (`cupy.ndarray` of `bool`) – If specified, this array acts as a mask, and an element is copied only if the corresponding element of `where` is True.

See also:

`numpy.copyto()`

### `cupy.shape`

`cupy.shape(a)`

Returns the shape of an array

#### Parameters

**a** (`array_like`) – Input array

#### Returns

The elements of the shape tuple give the lengths of the corresponding array dimensions.

#### Return type

tuple of ints

## Changing array shape

---

<code>reshape(a, newshape[, order])</code>	Returns an array with new shape and same elements.
<code>ravel(a[, order])</code>	Returns a flattened array.

---

### cupy.reshape

`cupy.reshape(a, newshape, order='C')`

Returns an array with new shape and same elements.

It tries to return a view if possible, otherwise returns a copy.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be reshaped.
- **newshape** (*int or tuple of ints*) – The new shape of the array to return. If it is an integer, then it is treated as a tuple of length one. It should be compatible with `a.size`. One of the elements can be -1, which is automatically replaced with the appropriate value to make the shape compatible with `a.size`.
- **order** (`{'C', 'F', 'A'}`) – Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise.

#### Returns

A reshaped view of `a` if possible, otherwise a copy.

#### Return type

`cupy.ndarray`

See also:

`numpy.reshape()`

### cupy.ravel

`cupy.ravel(a, order='C')`

Returns a flattened array.

It tries to return a view if possible, otherwise returns a copy.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be flattened.
- **order** (`{'C', 'F', 'A', 'K'}`) – The elements of `a` are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements

in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. ‘K’ means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, ‘C’ index order is used.

**Returns**

A flattened view of *a* if possible, otherwise a copy.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.ravel()`

**See also:**

*cupy.ndarray.flat* and *cupy.ndarray.flatten()*

**Transpose-like operations**

<i>moveaxis</i> ( <i>a</i> , <i>source</i> , <i>destination</i> )	Moves axes of an array to new positions.
<i>rollaxis</i> ( <i>a</i> , <i>axis</i> [, <i>start</i> ])	Moves the specified axis backwards to the given place.
<i>swapaxes</i> ( <i>a</i> , <i>axis1</i> , <i>axis2</i> )	Swaps the two axes.
<i>transpose</i> ( <i>a</i> [, <i>axes</i> ])	Permutates the dimensions of an array.

**cupy.moveaxis**

`cupy.moveaxis(a, source, destination)`

Moves axes of an array to new positions.

Other axes remain in their original order.

**Parameters**

- ***a*** (*cupy.ndarray*) – Array whose axes should be reordered.
- ***source*** (*int* or *sequence of int*) – Original positions of the axes to move. These must be unique.
- ***destination*** (*int* or *sequence of int*) – Destination positions for each of the original axes. These must also be unique.

**Returns**

Array with moved axes. This array is a view of the input array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.moveaxis()`

## cupy.rollaxis

`cupy.rollaxis(a, axis, start=0)`

Moves the specified axis backwards to the given place.

### Parameters

- **a** (`cupy.ndarray`) – Array to move the axis.
- **axis** (`int`) – The axis to move.
- **start** (`int`) – The place to which the axis is moved.

### Returns

A view of `a` that the axis is moved to `start`.

### Return type

*cupy.ndarray*

See also:

`numpy.rollaxis()`

## cupy.swapaxes

`cupy.swapaxes(a, axis1, axis2)`

Swaps the two axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to swap the axes.
- **axis1** (`int`) – The first axis to swap.
- **axis2** (`int`) – The second axis to swap.

### Returns

A view of `a` that the two axes are swapped.

### Return type

*cupy.ndarray*

See also:

`numpy.swapaxes()`

## cupy.transpose

`cupy.transpose(a, axes=None)`

Permutes the dimensions of an array.

### Parameters

- **a** (`cupy.ndarray`) – Array to permute the dimensions.
- **axes** (*tuple of ints*) – Permutation of the dimensions. This function reverses the shape by default.

### Returns

A view of `a` that the dimensions are permuted.



**Return type***cupy.ndarray***See also:**`numpy.transpose()`**See also:***cupy.ndarray.T***Changing number of dimensions**

<i>atleast_1d</i> (*args)	Converts arrays to arrays with dimensions $\geq 1$ .
<i>atleast_2d</i> (*args)	Converts arrays to arrays with dimensions $\geq 2$ .
<i>atleast_3d</i> (*args)	Converts arrays to arrays with dimensions $\geq 3$ .
<i>broadcast</i> (*arrays)	Object that performs broadcasting.
<i>broadcast_to</i> (array, shape)	Broadcast an array to a given shape.
<i>broadcast_arrays</i> (*args)	Broadcasts given arrays.
<i>expand_dims</i> (a, axis)	Expands given arrays.
<i>squeeze</i> (a[, axis])	Removes size-one axes from the shape of an array.

**cupy.atleast\_1d**`cupy.atleast_1d(*args)`Converts arrays to arrays with dimensions  $\geq 1$ .**Parameters****args** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects. Only zero-dimensional array is affected.**Returns**

If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

**See also:**`numpy.atleast_1d()`**cupy.atleast\_2d**`cupy.atleast_2d(*args)`Converts arrays to arrays with dimensions  $\geq 2$ .

If an input array has dimensions less than two, then this function inserts new axes at the head of dimensions to make it have two dimensions.

**Parameters****args** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects.**Returns**

If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

`numpy.atleast_2d()`

## `cupy.atleast_3d`

`cupy.atleast_3d(*arys)`

Converts arrays to arrays with dimensions  $\geq 3$ .

If an input array has dimensions less than three, then this function inserts new axes to make it have three dimensions. The place of the new axes are following:

- If its shape is `()`, then the shape of output is `(1, 1, 1)`.
- If its shape is `(N,)`, then the shape of output is `(1, N, 1)`.
- If its shape is `(M, N)`, then the shape of output is `(M, N, 1)`.
- Otherwise, the output is the input array itself.

### Parameters

**arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be `cupy.ndarray` objects.

### Returns

If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

`numpy.atleast_3d()`

## `cupy.broadcast`

`class cupy.broadcast(*arrays)`

Object that performs broadcasting.

CuPy actually uses this class to support broadcasting in various operations. Note that this class does not provide an iterator.

### Parameters

**arrays** (*tuple of arrays*) – Arrays to be broadcasted.

### Variables

- `~broadcast.shape` (*tuple of ints*) – The broadcasted shape.
- `nd` (*int*) – Number of dimensions of the broadcasted shape.
- `~broadcast.size` (*int*) – Total size of the broadcasted shape.
- **values** (*list of arrays*) – The broadcasted arrays.

See also:

`numpy.broadcast`

## Methods

**`__eq__`**(*value*, /)  
Return self==value.

**`__ne__`**(*value*, /)  
Return self!=value.

**`__lt__`**(*value*, /)  
Return self<value.

**`__le__`**(*value*, /)  
Return self<=value.

**`__gt__`**(*value*, /)  
Return self>value.

**`__ge__`**(*value*, /)  
Return self>=value.

## Attributes

**`nd`**

**`shape`**

**`size`**

**`values`**

## `cupy.broadcast_to`

**`cupy.broadcast_to`**(*array*, *shape*)  
Broadcast an array to a given shape.

### Parameters

- **`array`** (`cupy.ndarray`) – Array to broadcast.
- **`shape`** (*tuple of int*) – The shape of the desired array.

### Returns

Broadcasted view.

### Return type

`cupy.ndarray`

### See also:

`numpy.broadcast_to()`

## cupy.broadcast\_arrays

`cupy.broadcast_arrays(*args)`

Broadcasts given arrays.

### Parameters

**args** (*tuple of arrays*) – Arrays to broadcast for each other.

### Returns

A list of broadcasted arrays.

### Return type

*list*

See also:

`numpy.broadcast_arrays()`

## cupy.expand\_dims

`cupy.expand_dims(a, axis)`

Expands given arrays.

### Parameters

- **a** (`cupy.ndarray`) – Array to be expanded.
- **axis** (*int*) – Position where new axis is to be inserted.

### Returns

The number of dimensions is one greater than that of the input array.

### Return type

*cupy.ndarray*

See also:

`numpy.expand_dims()`

## cupy.squeeze

`cupy.squeeze(a, axis=None)`

Removes size-one axes from the shape of an array.

### Parameters

- **a** (`cupy.ndarray`) – Array to be reshaped.
- **axis** (*int or tuple of ints*) – Axes to be removed. This function removes all size-one axes by default. If one of the specified axes is not of size one, an exception is raised.

### Returns

An array without (specified) size-one axes.

### Return type

*cupy.ndarray*

See also:

`numpy.squeeze()`

## Changing kind of array

<code>asarray(a[, dtype, order, blocking])</code>	Converts an object to array.
<code>asanyarray(a[, dtype, order, blocking])</code>	Converts an object to array.
<code>asfarray(a[, dtype])</code>	Converts array elements to float type.
<code>asfortranarray(a[, dtype])</code>	Return an array laid out in Fortran order in memory.
<code>ascontiguousarray(a[, dtype])</code>	Returns a C-contiguous array.
<code>asarray_chkfinite(a[, dtype, order])</code>	Converts the given input to an array, and raises an error if the input contains NaNs or Infs.
<code>require(a[, dtype, requirements])</code>	Return an array which satisfies the requirements.

### cupy.asfarray

`cupy.asfarray(a, dtype=<class 'numpy.float64'>)`

Converts array elements to float type.

#### Parameters

- **a** (`cupy.ndarray`) – Source array.
- **dtype** – str or dtype object, optional

#### Returns

The input array *a* as a float ndarray.

#### Return type

`cupy.ndarray`

#### See also:

`numpy.asfarray()`

### cupy.asfortranarray

`cupy.asfortranarray(a, dtype=None)`

Return an array laid out in Fortran order in memory.

#### Parameters

- **a** (`ndarray`) – The input array.
- **dtype** (*str or dtype object, optional*) – By default, the data-type is inferred from the input data.

#### Returns

The input *a* in Fortran, or column-major, order.

#### Return type

`ndarray`

#### See also:

`numpy.asfortranarray()`

## cupy.asarray\_chkfinite

`cupy.asarray_chkfinite(a, dtype=None, order=None)`

Converts the given input to an array, and raises an error if the input contains NaNs or Infs.

### Parameters

- **a** – array like.
- **dtype** – data type, optional
- **order** – {'C', 'F', 'A', 'K'}, optional

### Returns

An array on the current device.

### Return type

*cupy.ndarray*

---

**Note:** This function performs device synchronization.

---

### See also:

`numpy.asarray_chkfinite()`

## cupy.require

`cupy.require(a, dtype=None, requirements=None)`

Return an array which satisfies the requirements.

### Parameters

- **a** (*ndarray*) – The input array.
- **dtype** (*str or dtype object, optional*) – The required data-type. If None preserve the current dtype.
- **requirements** (*str or list of str*) – The requirements can be any of the following
  - 'F\_CONTIGUOUS' ('F', 'FORTRAN') - ensure a Fortran-contiguous array.
  - 'C\_CONTIGUOUS' ('C', 'CONTIGUOUS') - ensure a C-contiguous array.
  - 'OWNDATA' ('O') - ensure an array that owns its own data.

### Returns

The input array a with specified requirements and type if provided.

### Return type

*ndarray*

### See also:

`numpy.require()`

## Joining arrays

<code>concatenate(tup[, axis, out, dtype, casting])</code>	Joins arrays along an axis.
<code>stack(tup[, axis, out, dtype, casting])</code>	Stacks arrays along a new axis.
<code>vstack(tup, *, dtype, casting)</code>	Stacks arrays vertically.
<code>hstack(tup, *, dtype, casting)</code>	Stacks arrays horizontally.
<code>dstack(tup)</code>	Stacks arrays along the third axis.
<code>column_stack(tup)</code>	Stacks 1-D and 2-D arrays as columns into a 2-D array.
<code>row_stack(tup, *, dtype, casting)</code>	Stacks arrays vertically.

### cupy.concatenate

`cupy.concatenate(tup, axis=0, out=None, *, dtype=None, casting='same_kind')`

Joins arrays along an axis.

#### Parameters

- **tup** (*sequence of arrays*) – Arrays to be joined. All of these should have same dimensionalities except the specified axis.
- **axis** (*int or None*) – The axis to join arrays along. If axis is None, arrays are flattened before use. Default is 0.
- **out** (`cupy.ndarray`) – Output array.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype. Cannot be provided together with out.
- **casting** (*{'no', 'equiv', 'safe', 'same\_kind', 'unsafe'}, optional*) – Controls what kind of data casting may occur. Defaults to 'same\_kind'.

#### Returns

Joined array.

#### Return type

`cupy.ndarray`

See also:

`numpy.concatenate()`

### cupy.stack

`cupy.stack(tup, axis=0, out=None, *, dtype=None, casting='same_kind')`

Stacks arrays along a new axis.

#### Parameters

- **tup** (*sequence of arrays*) – Arrays to be stacked.
- **axis** (*int*) – Axis along which the arrays are stacked.
- **out** (`cupy.ndarray`) – Output array.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype. Cannot be provided together with out.

- **casting** (`{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}`, optional) – Controls what kind of data casting may occur. Defaults to 'same\_kind'.

**Returns**

Stacked array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.stack()`

## **cupy.vstack**

`cupy.vstack(tup, *, dtype=None, casting='same_kind')`

Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

**Parameters**

- **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by `cupy.atleast_2d()` before stacking.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype.
- **casting** (`{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}`, optional) – Controls what kind of data casting may occur. Defaults to 'same\_kind'.

**Returns**

Stacked array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.dstack()`

## **cupy.hstack**

`cupy.hstack(tup, *, dtype=None, casting='same_kind')`

Stacks arrays horizontally.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the first axis. Otherwise, the array is stacked along the second axis.

**Parameters**

- **tup** (*sequence of arrays*) – Arrays to be stacked.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype.
- **casting** (`{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}`, optional) – Controls what kind of data casting may occur. Defaults to 'same\_kind'.

**Returns**

Stacked array.



**Return type***cupy.ndarray***See also:**`numpy.hstack()`**cupy.dstack**`cupy.dstack(tup)`

Stacks arrays along the third axis.

**Parameters****tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by `cupy.atleast_3d()` before stacking.**Returns**

Stacked array.

**Return type***cupy.ndarray***See also:**`numpy.dstack()`**cupy.column\_stack**`cupy.column_stack(tup)`

Stacks 1-D and 2-D arrays as columns into a 2-D array.

A 1-D array is first converted to a 2-D column array. Then, the 2-D arrays are concatenated along the second axis.

**Parameters****tup** (*sequence of arrays*) – 1-D or 2-D arrays to be stacked.**Returns**

A new 2-D array of stacked columns.

**Return type***cupy.ndarray***See also:**`numpy.column_stack()`**cupy.row\_stack**`cupy.row_stack(tup, *, dtype=None, casting='same_kind')`

Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

**Parameters**

- **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by `cupy.atleast_2d()` before stacking.
- **dtype** (*str or dtype*) – If provided, the destination array will have this dtype.
- **casting** (*{'no', 'equiv', 'safe', 'same\_kind', 'unsafe'}, optional*) – Controls what kind of data casting may occur. Defaults to 'same\_kind'.

**Returns**

Stacked array.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.dstack()`

## Splitting arrays

<code>split</code> (ary, indices_or_sections[, axis])	Splits an array into multiple sub arrays along a given axis.
<code>array_split</code> (ary, indices_or_sections[, axis])	Splits an array into multiple sub arrays along a given axis.
<code>dsplit</code> (ary, indices_or_sections)	Splits an array into multiple sub arrays along the third axis.
<code>hsplit</code> (ary, indices_or_sections)	Splits an array into multiple sub arrays horizontally.
<code>vsplit</code> (ary, indices_or_sections)	Splits an array into multiple sub arrays along the first axis.

## `cupy.split`

`cupy.split`(ary, indices\_or\_sections, axis=0)

Splits an array into multiple sub arrays along a given axis.

**Parameters**

- **ary** (`cupy.ndarray`) – Array to split.
- **indices\_or\_sections** (*int or sequence of ints*) – A value indicating how to divide the axis. If it is an integer, then is treated as the number of sections, and the axis is evenly divided. Otherwise, the integers indicate indices to split at. Note that the sequence on the device memory is not allowed.
- **axis** (*int*) – Axis along which the array is split.

**Returns**

A list of sub arrays. Each array is a view of the corresponding input array.

**See also:**

`numpy.split()`

## cupy.array\_split

`cupy.array_split(ary, indices_or_sections, axis=0)`

Splits an array into multiple sub arrays along a given axis.

This function is almost equivalent to `cupy.split()`. The only difference is that this function allows an integer sections that does not evenly divide the axis.

**See also:**

`cupy.split()` for more detail, `numpy.array_split()`

## cupy.dsplit

`cupy.dsplit(ary, indices_or_sections)`

Splits an array into multiple sub arrays along the third axis.

This is equivalent to `split` with `axis=2`.

**See also:**

`cupy.split()` for more detail, `numpy.dsplit()`

## cupy.hsplit

`cupy.hsplit(ary, indices_or_sections)`

Splits an array into multiple sub arrays horizontally.

This is equivalent to `split` with `axis=0` if `ary` has one dimension, and otherwise that with `axis=1`.

**See also:**

`cupy.split()` for more detail, `numpy.hsplit()`

## cupy.vsplit

`cupy.vsplit(ary, indices_or_sections)`

Splits an array into multiple sub arrays along the first axis.

This is equivalent to `split` with `axis=0`.

**See also:**

`cupy.split()` for more detail, `numpy.dsplit()`

## Tiling arrays

<code>tile(A, reps)</code>	Construct an array by repeating A the number of times given by reps.
<code>repeat(a, repeats[, axis])</code>	Repeat arrays along an axis.

## cupy.tile

`cupy.tile(A, reps)`

Construct an array by repeating A the number of times given by reps.

### Parameters

- **A** (`cupy.ndarray`) – Array to transform.
- **reps** (`int` or `tuple`) – The number of repeats.

### Returns

Transformed array with repeats.

### Return type

*cupy.ndarray*

See also:

`numpy.tile()`

## cupy.repeat

`cupy.repeat(a, repeats, axis=None)`

Repeat arrays along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to transform.
- **repeats** (`int`, `list` or `tuple`) – The number of repeats.
- **axis** (`int`) – The axis to repeat.

### Returns

Transformed array with repeats.

### Return type

*cupy.ndarray*

See also:

`numpy.repeat()`

## Adding and removing elements

<code>delete(arr, indices[, axis])</code>	Delete values from an array along the specified axis.
<code>append(arr, values[, axis])</code>	Append values to the end of an array.
<code>resize(a, new_shape)</code>	Return a new array with the specified shape.
<code>unique(ar[, return_index, return_inverse, ...])</code>	Find the unique elements of an array.
<code>trim_zeros(filt[, trim])</code>	Trim the leading and/or trailing zeros from a 1-D array or sequence.

## cupy.delete

`cupy.delete(arr, indices, axis=None)`

Delete values from an array along the specified axis.

### Parameters

- **arr** (`cupy.ndarray`) – Values are deleted from a copy of this array.
- **indices** (*slice, int or array of ints*) – These indices correspond to values that will be deleted from the copy of *arr*. Boolean indices are treated as a mask of elements to remove.
- **axis** (*int or None*) – The axis along which *indices* correspond to values that will be deleted. If *axis* is not given, *arr* will be flattened.

### Returns

A copy of *arr* with values specified by *indices* deleted along *axis*.

### Return type

*cupy.ndarray*

**Warning:** This function may synchronize the device.

### See also:

`numpy.delete()`.

## cupy.append

`cupy.append(arr, values, axis=None)`

Append values to the end of an array.

### Parameters

- **arr** (*array\_like*) – Values are appended to a copy of this array.
- **values** (*array\_like*) – These values are appended to a copy of *arr*. It must be of the correct shape (the same shape as *arr*, excluding *axis*). If *axis* is not specified, values can be any shape and will be flattened before use.
- **axis** (*int or None*) – The axis along which values are appended. If *axis* is not given, both *arr* and *values* are flattened before use.

### Returns

A copy of *arr* with values appended to *axis*. Note that *append* does not occur in-place: a new array is allocated and filled. If *axis* is *None*, *out* is a flattened array.

### Return type

*cupy.ndarray*

### See also:

`numpy.append()`

## cupy.resize

`cupy.resize(a, new_shape)`

Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copies of `a`. Note that this behavior is different from `a.resize(new_shape)` which fills with zeros instead of repeated copies of `a`.

### Parameters

- **a** (*array\_like*) – Array to be resized.
- **new\_shape** (*int or tuple of int*) – Shape of resized array.

### Returns

The new array is formed from the data in the old array, repeated if necessary to fill out the required number of elements. The data are repeated in the order that they are stored in memory.

### Return type

*cupy.ndarray*

See also:

`numpy.resize()`

## cupy.unique

`cupy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *, equal_nan=True)`

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

### Parameters

- **ar** (*array\_like*) – Input array. This will be flattened if it is not already 1-D.
- **return\_index** (*bool, optional*) – If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.
- **return\_inverse** (*bool, optional*) – If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.
- **return\_counts** (*bool, optional*) – If True, also return the number of times each unique item appears in *ar*.
- **axis** (*int or None, optional*) – The axis to operate on. If None, *ar* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. The default is None.
- **equal\_nan** (*bool, optional*) – If True, collapse multiple NaN values in the return array into one.

**Returns**

If there are no optional outputs, it returns the `cupy.ndarray` of the sorted unique values. Otherwise, it returns the tuple which contains the sorted unique values and followings.

- The indices of the first occurrences of the unique values in the original array. Only provided if `return_index` is True.
- The indices to reconstruct the original array from the unique array. Only provided if `return_inverse` is True.
- The number of times each of the unique values comes up in the original array. Only provided if `return_counts` is True.

**Return type**

`cupy.ndarray` or `tuple`

**Notes**

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order.

**Warning:** This function may synchronize the device.

**See also:**

`numpy.unique()`

**cupy.trim\_zeros**

`cupy.trim_zeros(filt, trim='fb')`

Trim the leading and/or trailing zeros from a 1-D array or sequence.

Returns the trimmed array

**Parameters**

- **filt** (`cupy.ndarray`) – Input array
- **trim** (`str`, *optional*) – ‘fb’ default option trims the array from both sides. ‘f’ option trim zeros from front. ‘b’ option trim zeros from back.

**Returns**

trimmed input

**Return type**

`cupy.ndarray`

**See also:**

`numpy.trim_zeros()`

## Rearranging elements

<code>flip(a[, axis])</code>	Reverse the order of elements in an array along the given axis.
<code>fliplr(a)</code>	Flip array in the left/right direction.
<code>flipud(a)</code>	Flip array in the up/down direction.
<code>reshape(a, newshape[, order])</code>	Returns an array with new shape and same elements.
<code>roll(a, shift[, axis])</code>	Roll array elements along a given axis.
<code>rot90(a[, k, axes])</code>	Rotate an array by 90 degrees in the plane specified by axes.

### `cupy.flip`

`cupy.flip(a, axis=None)`

Reverse the order of elements in an array along the given axis.

Note that `flip` function has been introduced since NumPy v1.12. The contents of this document is the same as the original one.

#### Parameters

- **a** (`ndarray`) – Input array.
- **axis** (`int` or `tuple of int` or `None`) – Axis or axes along which to flip over. The default, `axis=None`, will flip over all of the axes of the input array. If axis is negative it counts from the last to the first axis. If axis is a tuple of ints, flipping is performed on all of the axes specified in the tuple.

#### Returns

Output array.

#### Return type

`ndarray`

#### See also:

`numpy.flip()`

### `cupy.fliplr`

`cupy.fliplr(a)`

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

#### Parameters

**a** (`ndarray`) – Input array.

#### Returns

Output array.

#### Return type

`ndarray`



See also:

`numpy.fliplr()`

## `cupy.flipud`

`cupy.flipud(a)`

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

### Parameters

**a** (`ndarray`) – Input array.

### Returns

Output array.

### Return type

`ndarray`

See also:

`numpy.flipud()`

## `cupy.roll`

`cupy.roll(a, shift, axis=None)`

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

### Parameters

- **a** (`ndarray`) – Array to be rolled.
- **shift** (`int` or `tuple of int`) – The number of places by which elements are shifted. If a tuple, then *axis* must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while *axis* is a tuple of ints, then the same value is used for all given axes.
- **axis** (`int` or `tuple of int` or `None`) – The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

### Returns

Output array.

### Return type

`ndarray`

See also:

`numpy.roll()`

## cupy.rot90

`cupy.rot90(a, k=1, axes=(0, 1))`

Rotate an array by 90 degrees in the plane specified by axes.

Note that axes argument has been introduced since NumPy v1.12. The contents of this document is the same as the original one.

### Parameters

- **a** (`ndarray`) – Array of two or more dimensions.
- **k** (`int`) – Number of times the array is rotated by 90 degrees.
- **axes** – (tuple of ints): The array is rotated in the plane defined by the axes. Axes must be different.

### Returns

Output array.

### Return type

`ndarray`

### See also:

`numpy.rot90()`

## 5.3.3 Binary operations

---

**Hint:** [NumPy API Reference: Binary operations](#)

---

### Elementwise bit operations

<code>bitwise_and(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise AND of two arrays elementwise.
<code>bitwise_or(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise OR of two arrays elementwise.
<code>bitwise_xor(x1, x2, /[, out, casting, dtype])</code>	Computes the bitwise XOR of two arrays elementwise.
<code>invert(x, /[, out, casting, dtype])</code>	Computes the bitwise NOT of an array elementwise.
<code>left_shift(x1, x2, /[, out, casting, dtype])</code>	Shifts the bits of each integer element to the left.
<code>right_shift(x1, x2, /[, out, casting, dtype])</code>	Shifts the bits of each integer element to the right.

### Bit packing

<code>packbits(a[, axis, bitorder])</code>	Packs the elements of a binary-valued array into bits in a uint8 array.
<code>unpackbits(a[, axis, bitorder])</code>	Unpacks elements of a uint8 array into a binary-valued output array.

## cupy.packbits

`cupy.packbits(a, axis=None, bitorder='big')`

Packs the elements of a binary-valued array into bits in a uint8 array.

This function currently does not support `axis` option.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`, *optional*) – Not supported yet.
- **bitorder** (`str`, *optional*) – bit order to use when packing the array, allowed values are `'little'` and `'big'`. Defaults to `'big'`.

### Returns

The packed array.

### Return type

`cupy.ndarray`

---

**Note:** When the input array is empty, this function returns a copy of it, i.e., the type of the output array is not necessarily always uint8. This exactly follows the NumPy's behaviour (as of version 1.11), although this is inconsistent to the documentation.

---

### See also:

`numpy.packbits()`

## cupy.unpackbits

`cupy.unpackbits(a, axis=None, bitorder='big')`

Unpacks elements of a uint8 array into a binary-valued output array.

This function currently does not support `axis` option.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **bitorder** (`str`, *optional*) – bit order to use when unpacking the array, allowed values are `'little'` and `'big'`. Defaults to `'big'`.

### Returns

The unpacked array.

### Return type

`cupy.ndarray`

### See also:

`numpy.unpackbits()`

## Output formatting

---

<code>binary_repr(num[, width])</code>	Return the binary representation of the input number as a string.
--	---

---

### cupy.binary\_repr

`cupy.binary_repr(num, width=None)`

Return the binary representation of the input number as a string.

**See also:**

`numpy.binary_repr()`

## 5.3.4 Data type routines

---

**Hint:** [NumPy API Reference: Data type routines](#)

---

---

<code>can_cast(from_, to[, casting])</code>	Returns True if cast between data types can occur according to the casting rule.
<code>min_scalar_type(a)</code>	For scalar <i>a</i> , returns the data type with the smallest size and smallest scalar kind which can hold its value.
<code>result_type(*arrays_and_dtypes)</code>	Returns the type that results from applying the NumPy type promotion rules to the arguments.
<code>common_type(*arrays)</code>	Return a scalar type which is common to the input arrays.

---

### cupy.can\_cast

`cupy.can_cast(from_, to, casting='safe')`

Returns True if cast between data types can occur according to the casting rule. If *from* is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an integer.

**See also:**

`numpy.can_cast()`

### cupy.min\_scalar\_type

`cupy.min_scalar_type(a)`

For scalar *a*, returns the data type with the smallest size and smallest scalar kind which can hold its value. For non-scalar array *a*, returns the vector's dtype unmodified.

**See also:**

`numpy.min_scalar_type()`

## cupy.result\_type

`cupy.result_type(*arrays_and_dtypes)`

Returns the type that results from applying the NumPy type promotion rules to the arguments.

**See also:**

`numpy.result_type()`

## cupy.common\_type

`cupy.common_type(*arrays)`

Return a scalar type which is common to the input arrays.

**See also:**

`numpy.common_type()`

<code>promote_types</code> (alias of <code>numpy.promote_types()</code> )
<code>obj2sctype</code> (alias of <code>numpy.obj2sctype()</code> )

## Creating data types

<code>dtype</code> (alias of <code>numpy.dtype</code> )
<code>format_parser</code> (alias of <code>numpy.format_parser</code> )

## Data type information

<code>finfo</code> (alias of <code>numpy.finfo</code> )
<code>info</code> (alias of <code>numpy.iinfo</code> )
<code>MachAr</code> (alias of <code>numpy.MachAr</code> )

## Data type testing

<code>issctype</code> (alias of <code>numpy.issctype()</code> )
<code>issubdtype</code> (alias of <code>numpy.issubdtype()</code> )
<code>issubscctype</code> (alias of <code>numpy.issubscctype()</code> )
<code>issubclass_</code> (alias of <code>numpy.issubclass_()</code> )
<code>find_common_type</code> (alias of <code>numpy.find_common_type()</code> )

## Miscellaneous

<code>typename</code> (alias of <code>numpy.typename()</code> )
<code>sctype2char</code> (alias of <code>numpy.sctype2char()</code> )
<code>mintypecode</code> (alias of <code>numpy.mintypecode()</code> )

## 5.3.5 Discrete Fourier Transform (`cupy.fft`)

---

**Hint:** NumPy API Reference: Discrete Fourier Transform (`numpy.fft`)

---

### See also:

*Discrete Fourier transforms (`cupyx.scipy.fft`), Fast Fourier Transform with CuPy*

### Standard FFTs

<code>fft(a[, n, axis, norm])</code>	Compute the one-dimensional FFT.
<code>ifft(a[, n, axis, norm])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(a[, s, axes, norm])</code>	Compute the two-dimensional FFT.
<code>ifft2(a[, s, axes, norm])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(a[, s, axes, norm])</code>	Compute the N-dimensional FFT.
<code>ifftn(a[, s, axes, norm])</code>	Compute the N-dimensional inverse FFT.

### `cupy.fft.fft`

`cupy.fft.fft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional FFT.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None* or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".

#### Returns

The transformed array which shape is specified by `n` and type will convert to complex if the input is other.

#### Return type

*cupy.ndarray*

### See also:

`numpy.fft.fft()`

**cupy.fft.ifft**

`cupy.fft.ifft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional inverse FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None* or `int`) – Length of the transformed axis of the output. If *n* is not given, the length of the input along the axis specified by *axis* is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns**

The transformed array which shape is specified by *n* and type will convert to complex if the input is other.

**Return type**

`cupy.ndarray`

See also:

`numpy.fft.ifft()`

**cupy.fft.fft2**

`cupy.fft.fft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional FFT.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If *s* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns**

The transformed array which shape is specified by *s* and type will convert to complex if the input is other.

**Return type**

`cupy.ndarray`

See also:

`numpy.fft.fft2()`

## cupy.fft.ifft2

`cupy.fft.ifft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional inverse FFT.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

### Return type

*cupy.ndarray*

See also:

`numpy.fft.ifft2()`

## cupy.fft.fftn

`cupy.fft.fftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional FFT.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

### Return type

*cupy.ndarray*

See also:

`numpy.fft.fftn()`



## cupy.fft.ifftn

`cupy.fft.ifftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional inverse FFT.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other.

### Return type

*cupy.ndarray*

See also:

`numpy.fft.ifftn()`

## Real FFTs

<code>rfft(a[, n, axis, norm])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(a[, n, axis, norm])</code>	Compute the one-dimensional inverse FFT for real input.
<code>rfft2(a[, s, axes, norm])</code>	Compute the two-dimensional FFT for real input.
<code>irfft2(a[, s, axes, norm])</code>	Compute the two-dimensional inverse FFT for real input.
<code>rfftn(a[, s, axes, norm])</code>	Compute the N-dimensional FFT for real input.
<code>irfftn(a[, s, axes, norm])</code>	Compute the N-dimensional inverse FFT for real input.

## cupy.fft.rfft

`cupy.fft.rfft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None or int*) – Number of points along transformation axis in the input to use. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **n** and type will convert to complex if the input is other. The length of the transformed axis is  $n//2+1$ .

**Return type***cupy.ndarray***See also:**`numpy.fft.rfft()`**cupy.fft.irfft**`cupy.fft.irfft(a, n=None, axis=-1, norm=None)`

Compute the one-dimensional inverse FFT for real input.

**Parameters**

- **a** (*cupy.ndarray*) – Array to be transform.
- **n** (*None* or *int*) – Length of the transformed axis of the output. For *n* output points,  $n//2+1$  input points are necessary. If *n* is not given, it is determined from the length of the input along the axis specified by *axis*.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns**

The transformed array which shape is specified by *n* and type will convert to complex if the input is other. If *n* is not given, the length of the transformed axis is  $2*(m-1)$  where *m* is the length of the transformed axis of the input.

**Return type***cupy.ndarray***See also:**`numpy.fft.irfft()`**cupy.fft.rfft2**`cupy.fft.rfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional FFT for real input.

**Parameters**

- **a** (*cupy.ndarray*) – Array to be transform.
- **s** (*None* or *tuple of ints*) – Shape to use from the input. If *s* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns**

The transformed array which shape is specified by *s* and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]//2+1$ .

**Return type***cupy.ndarray*

See also:

`numpy.fft.rfft2()`

## `cupy.fft.irfft2`

`cupy.fft.irfft2(a, s=None, axes=(-2, -1), norm=None)`

Compute the two-dimensional inverse FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If **s** is not given, they are determined from the lengths of the input along the axes specified by **axes**.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other. If **s** is not given, the length of final transformed axis of output will be  $2^{*(m-1)}$  where  $m$  is the length of the final transformed axis of the input.

### Return type

`cupy.ndarray`

See also:

`numpy.fft.irfft2()`

## `cupy.fft.rfftn`

`cupy.fft.rfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape to use from the input. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]//2+1$ .

### Return type

`cupy.ndarray`

See also:

`numpy.fft.rfftn()`

## cupy.fft.irfftn

`cupy.fft.irfftn(a, s=None, axes=None, norm=None)`

Compute the N-dimensional inverse FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If **s** is not given, they are determined from the lengths of the input along the axes specified by **axes**.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **s** and type will convert to complex if the input is other. If **s** is not given, the length of final transformed axis of output will be  $2 * (m - 1)$  where  $m$  is the length of the final transformed axis of the input.

### Return type

*cupy.ndarray*

See also:

`numpy.fft.irfftn()`

## Hermitian FFTs

---

<code>hfft(a[, n, axis, norm])</code>	Compute the FFT of a signal that has Hermitian symmetry.
<code>ihfft(a[, n, axis, norm])</code>	Compute the FFT of a signal that has Hermitian symmetry.

---

## cupy.fft.hfft

`cupy.fft.hfft(a, n=None, axis=-1, norm=None)`

Compute the FFT of a signal that has Hermitian symmetry.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None or int*) – Length of the transformed axis of the output. For **n** output points,  $n // 2 + 1$  input points are necessary. If **n** is not given, it is determined from the length of the input along the axis specified by **axis**.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

### Returns

The transformed array which shape is specified by **n** and type will convert to complex if the input is other. If **n** is not given, the length of the transformed axis is  $2 * (m - 1)$  where  $m$  is the length of the transformed axis of the input.

**Return type***cupy.ndarray***See also:**`numpy.fft.hfft()`**cupy.fft.ihfft**`cupy.fft.ihfft(a, n=None, axis=-1, norm=None)`

Compute the FFT of a signal that has Hermitian symmetry.

**Parameters**

- **a** (*cupy.ndarray*) – Array to be transform.
- **n** (*None* or *int*) – Number of points along transformation axis in the input to use. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".

**Returns**

The transformed array which shape is specified by **n** and type will convert to complex if the input is other. The length of the transformed axis is  $n//2+1$ .

**Return type***cupy.ndarray***See also:**`numpy.fft.ihfft()`**Helper routines**

<code>fftfreq(n[, d])</code>	Return the FFT sample frequencies.
<code>rfftfreq(n[, d])</code>	Return the FFT sample frequencies for real input.
<code>fftshift(x[, axes])</code>	Shift the zero-frequency component to the center of the spectrum.
<code>ifftshift(x[, axes])</code>	The inverse of <code>fftshift()</code> .

**cupy.fft.fftfreq**`cupy.fft.fftfreq(n, d=1.0)`

Return the FFT sample frequencies.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns**

Array of length **n** containing the sample frequencies.

**Return type***cupy.ndarray***See also:**`numpy.fft.fftfreq()`**cupy.fft.rfftfreq**`cupy.fft.rfftfreq(n, d=1.0)`

Return the FFT sample frequencies for real input.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns**

Array of length  $n/2+1$  containing the sample frequencies.

**Return type***cupy.ndarray***See also:**`numpy.fft.rfftfreq()`**cupy.fft.fftshift**`cupy.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

**Parameters**

- **x** (*cupy.ndarray*) – Input array.
- **axes** (*int or tuple of ints*) – Axes over which to shift. Default is None, which shifts all axes.

**Returns**

The shifted array.

**Return type***cupy.ndarray***See also:**`numpy.fft.fftshift()`

## cupy.fft.ifftshift

`cupy.fft.ifftshift(x, axes=None)`

The inverse of `fftshift()`.

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **axes** (`int` or `tuple` of `ints`) – Axes over which to shift. Default is `None`, which shifts all axes.

### Returns

The shifted array.

### Return type

`cupy.ndarray`

See also:

`numpy.fft.ifftshift()`

## CuPy-specific APIs

See the description below for details.

<code>config.set_cufft_callbacks(...)</code>	A context manager for setting up load and/or store callbacks.
<code>config.set_cufft_gpus(gpus)</code>	Set the GPUs to be used in multi-GPU FFT.
<code>config.get_plan_cache()</code>	Get the per-thread, per-device plan cache, or create one if not found.
<code>config.show_plan_cache_info()</code>	Show all of the plan caches' info on this thread.

## cupy.fft.config.set\_cufft\_callbacks

```
class cupy.fft.config.set_cufft_callbacks(unicode cb_load=u'', unicode cb_store=u'', ndarray
                                         cb_load_aux_arr=None, *, ndarray
                                         cb_store_aux_arr=None)
```

A context manager for setting up load and/or store callbacks.

### Parameters

- **cb\_load** (`str`) – A string contains the device kernel for the load callback. It must define `d_loadCallbackPtr`.
- **cb\_store** (`str`) – A string contains the device kernel for the store callback. It must define `d_storeCallbackPtr`.
- **cb\_load\_aux\_arr** (`cupy.ndarray`, *optional*) – A CuPy array containing data to be used in the load callback.
- **cb\_store\_aux\_arr** (`cupy.ndarray`, *optional*) – A CuPy array containing data to be used in the store callback.

**Note:** Any FFT calls living in this context will have callbacks set up. An example for a load callback is shown below:

```
code = r'''
__device__ cufftComplex CB_ConvertInputC(
    void *dataIn,
    size_t offset,
    void *callerInfo,
    void *sharedPtr) {
    // implementation
}

__device__ cufftCallbackLoadC d_loadCallbackPtr = CB_ConvertInputC;
'''

with cp.fft.config.set_cufft_callbacks(cb_load=code):
    out_arr = cp.fft.fft(in_arr, ...)
```

---

**Note:** Below are the *runtime* requirements for using this feature:

- cython >= 0.29.0
- A host compiler that supports C++11 and above; might need to set up the CXX environment variable.
- nvcc and the full CUDA Toolkit. Note that the cudatoolkit package from Conda-Forge is not enough, as it does not contain static libraries.

---

**Note:** Callbacks only work for transforms over contiguous axes; the behavior for non-contiguous transforms is in general undefined.

---

**Warning:** Using cuFFT callbacks requires compiling and loading a Python module at runtime as well as static linking for each distinct transform and callback, so the first invocation for each combination will be very slow. This is a limitation of cuFFT, so use this feature only when the callback-enabled transform is known more performant and can be reused to amortize the cost.

**Warning:** The generated Python modules are by default cached in `~/ .cupy/callback_cache` for possible reuse (with the same set of load/store callbacks). Due to static linking, however, the file sizes can be excessive! The cache position can be changed via setting `CUPY_CACHE_DIR`.

**See also:**

[cuFFT Callback Routines](#)



## Methods

`__enter__(self)`

`__exit__(self, exc_type, exc_value, traceback)`

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

## `cupy.fft.config.set_cufft_gpus`

`cupy.fft.config.set_cufft_gpus(gpus)`

Set the GPUs to be used in multi-GPU FFT.

### Parameters

**gpus** (*int* or *list of int*) – The number of GPUs or a list of GPUs to be used. For the former case, the first *gpus* GPUs will be used.

**Warning:** This API is currently experimental and may be changed in the future version.

### See also:

[Multiple GPU cuFFT Transforms](#)

## `cupy.fft.config.get_plan_cache`

`cupy.fft.config.get_plan_cache()` → PlanCache

Get the per-thread, per-device plan cache, or create one if not found.

### See also:

[PlanCache](#)

## cupy.fft.config.show\_plan\_cache\_info

`cupy.fft.config.show_plan_cache_info()`

Show all of the plan caches' info on this thread.

**See also:**

`PlanCache`

## Normalization

The default normalization (`norm` is "backward" or `None`) has the direct transforms unscaled and the inverse transforms scaled by  $1/n$ . If the keyword argument `norm` is "forward", it is the exact opposite of "backward": the direct transforms are scaled by  $1/n$  and the inverse transforms are unscaled. Finally, if the keyword argument `norm` is "ortho", both transforms are scaled by  $1/\sqrt{n}$ .

## Code compatibility features

FFT functions of NumPy always return `numpy.ndarray` which type is `numpy.complex128` or `numpy.float64`. CuPy functions do not follow the behavior, they will return `numpy.complex64` or `numpy.float32` if the type of the input is `numpy.float16`, `numpy.float32`, or `numpy.complex64`.

Internally, `cupy.fft` always generates a *cuFFT plan* (see the [cuFFT documentation](#) for detail) corresponding to the desired transform. When possible, an n-dimensional plan will be used, as opposed to applying separate 1D plans for each axis to be transformed. Using n-dimensional planning can provide better performance for multidimensional transforms, but requires more GPU memory than separable 1D planning. The user can disable n-dimensional planning by setting `cupy.fft.config.enable_nd_planning = False`. This ability to adjust the planning type is a deviation from the NumPy API, which does not use precomputed FFT plans.

Moreover, the automatic plan generation can be suppressed by using an existing plan returned by `cupyx.scipy.fftpack.get_fft_plan()` as a context manager. This is again a deviation from NumPy.

Finally, when using the high-level NumPy-like FFT APIs as listed above, internally the cuFFT plans are cached for possible reuse. The plan cache can be retrieved by `get_plan_cache()`, and its current status can be queried by `show_plan_cache_info()`. For finer control of the plan cache, see `PlanCache`.

## Multi-GPU FFT

`cupy.fft` can use multiple GPUs. To enable (disable) this feature, set `cupy.fft.config.use_multi_gpus` to `True` (`False`). Next, to set the number of GPUs or the participating GPU IDs, use the function `cupy.fft.config.set_cufft_gpus()`. All of the limitations listed in the [cuFFT documentation](#) apply here. In particular, using more than one GPU does not guarantee better performance.

## 5.3.6 Functional programming

---

**Hint:** [NumPy API Reference: Functional programming](#)

---

---

**Note:** `cupy.vectorize` applies JIT compiler to the given Python function. See [JIT kernel definition](#) for details.

---

---

<code>apply_along_axis(func1d, axis, arr, *args, ...)</code>	Apply a function to 1-D slices along the given axis.
<code>vectorize(pyfunc[, otypes, doc, excluded, ...])</code>	Generalized function class.
<code>piecewise(x, condlist, funclist)</code>	Evaluate a piecewise-defined function.

---

## cupy.apply\_along\_axis

`cupy.apply_along_axis(func1d, axis, arr, *args, **kwargs)`

Apply a function to 1-D slices along the given axis.

### Parameters

- **func1d** (*function* ( $M,$ )  $\rightarrow$  ( $Nj...$ )) – This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis. It must return a 1-D `cupy.ndarray`.
- **axis** (*integer*) – Axis along which `arr` is sliced.
- **arr** (`cupy.ndarray` ( $Ni...$ ,  $M$ ,  $Nk...$ )) – Input array.
- **args** – Additional arguments for `func1d`.
- **kwargs** – Additional keyword arguments for `func1d`.

### Returns

The output array. The shape of `out` is identical to the shape of `arr`, except along the `axis` dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of `func1d`. So if `func1d` returns a scalar `out` will have one fewer dimensions than `arr`.

### Return type

`cupy.ndarray`

See also:

`numpy.apply_along_axis()`

## cupy.vectorize

`class cupy.vectorize(pyfunc, otypes=None, doc=None, excluded=None, cache=False, signature=None)`

Generalized function class.

See also:

`numpy.vectorize`

### Methods

`__call__(*args)`

Call self as a function.

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`  
Return self<value.

`__le__(value, /)`  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

## `cupy.piecewise`

`cupy.piecewise(x, condlist, funclist)`

Evaluate a piecewise-defined function.

### Parameters

- **x** (`cupy.ndarray`) – input domain
- **condlist** (*list of* `cupy.ndarray`) – Each boolean array/ scalar corresponds to a function in funclist. Length of funclist is equal to that of condlist. If one extra function is given, it is used as the default value when the otherwise condition is met
- **funclist** (*list of* `scalars`) – list of scalar functions.

### Returns

the scalar values in funclist on portions of x defined by condlist.

### Return type

`cupy.ndarray`

<b>Warning:</b> This function currently doesn't support callable functions, args and kw parameters.
---

See also:

`numpy.piecewise()`

## 5.3.7 Indexing routines

---

**Hint:** [NumPy API Reference: Indexing routines](#)

---

## Generating index arrays

<code>c_</code>	
<code>r_</code>	
<code>nonzero(a)</code>	Return the indices of the elements that are non-zero.
<code>where(condition[, x, y])</code>	Return elements, either from x or y, depending on condition.
<code>indices(dimensions[, dtype])</code>	Returns an array representing the indices of a grid.
<code>mask_indices(n, mask_func[, k])</code>	Return the indices to access (n, n) arrays, given a masking function.
<code>tril_indices(n[, k, m])</code>	Returns the indices of the lower triangular matrix.
<code>tril_indices_from(arr[, k])</code>	Returns the indices for the lower-triangle of arr.
<code>triu_indices(n[, k, m])</code>	Returns the indices of the upper triangular matrix.
<code>triu_indices_from(arr[, k])</code>	Returns indices for the upper-triangle of arr.
<code>ix_(*args)</code>	Construct an open mesh from multiple sequences.
<code>ravel_multi_index(multi_index, dims[, mode, ...])</code>	Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.
<code>unravel_index(indices, dims[, order])</code>	Converts array of flat indices into a tuple of coordinate arrays.
<code>diag_indices(n[, ndim])</code>	Return the indices to access the main diagonal of an array.
<code>diag_indices_from(arr)</code>	Return the indices to access the main diagonal of an n-dimensional array.

### cupy.c\_

`cupy.c_ = <cupy._indexing.generate.CClass object>`

### cupy.r\_

`cupy.r_ = <cupy._indexing.generate.RClass object>`

### cupy.nonzero

`cupy.nonzero(a)`

Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of a, containing the indices of the non-zero elements in that dimension.

#### Parameters

**a** (`cupy.ndarray`) – array

#### Returns

Indices of elements that are non-zero.

#### Return type

tuple of arrays

**Warning:** This function may synchronize the device.

See also:

`numpy.nonzero()`

## cupy.where

`cupy.where(condition, x=None, y=None)`

Return elements, either from x or y, depending on condition.

If only condition is given, return `condition.nonzero()`.

### Parameters

- **condition** (`cupy.ndarray`) – When True, take x, otherwise take y.
- **x** (`cupy.ndarray`) – Values from which to choose on True.
- **y** (`cupy.ndarray`) – Values from which to choose on False.

### Returns

Each element of output contains elements of x when condition is True, otherwise elements of y. If only condition is given, return the tuple `condition.nonzero()`, the indices where condition is True.

### Return type

*cupy.ndarray*

**Warning:** This function may synchronize the device if both x and y are omitted.

See also:

`numpy.where()`

## cupy.indices

`cupy.indices(dimensions, dtype=<class 'int'>)`

Returns an array representing the indices of a grid.

Computes an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

### Parameters

- **dimensions** – The shape of the grid.
- **dtype** – Data type specifier. It is int by default.

### Returns

The array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

### Return type

*ndarray*

## Examples

```
>>> grid = cupy.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]          # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]          # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

See also:

`numpy.indices()`

## `cupy.mask_indices`

`cupy.mask_indices(n, mask_func, k=0)`

Return the indices to access (n, n) arrays, given a masking function.

Assume *mask\_func* is a function that, for a square array *a* of size (n, n) with a possible offset argument *k*, when called as `mask_func(a, k)` returns a new array with zeros in certain locations (functions like `triu()` or `tril()` do precisely this). Then this function returns the indices where the non-zero values would be located.

### Parameters

- **n** (*int*) – The returned indices will be valid to access arrays of shape (n, n).
- **mask\_func** (*callable*) – A function whose call signature is similar to that of `triu()`, `tril()`. That is, `mask_func(x, k)` returns a boolean array, shaped like *x*. *k* is an optional argument to the function.
- **k** (*scalar*) – An optional argument which is passed through to *mask\_func*. Functions like `triu()`, `tril()` take a second argument that is interpreted as an offset.

### Returns

The *n* arrays of indices corresponding to the locations where `mask_func(np.ones((n, n)), k)` is True.

### Return type

tuple of arrays

**Warning:** This function may synchronize the device.

See also:

`numpy.mask_indices()`

## cupy.tril\_indices

`cupy.tril_indices(n, k=0, m=None)`

Returns the indices of the lower triangular matrix. Here, the first group of elements contains row coordinates of all indices and the second group of elements contains column coordinates.

### Parameters

- **n** (*int*) – The row dimension of the arrays for which the returned indices will be valid.
- **k** (*int*, *optional*) – Diagonal above which to zero elements.  $k = 0$  (the default) is the main diagonal,  $k < 0$  is below it and  $k > 0$  is above.
- **m** (*int*, *optional*) – The column dimension of the arrays for which the returned arrays will be valid. By default,  $m = n$ .

### Returns

**y** – The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

### Return type

tuple of ndarrays

See also:

`numpy.tril_indices`

## cupy.tril\_indices\_from

`cupy.tril_indices_from(arr, k=0)`

Returns the indices for the lower-triangle of arr.

### Parameters

- **arr** (`cupy.ndarray`) – The indices are valid for square arrays whose dimensions are the same as arr.
- **k** (*int*, *optional*) – Diagonal offset.

See also:

`numpy.tril_indices_from`

## cupy.triu\_indices

`cupy.triu_indices(n, k=0, m=None)`

Returns the indices of the upper triangular matrix. Here, the first group of elements contains row coordinates of all indices and the second group of elements contains column coordinates.

### Parameters

- **n** (*int*) – The size of the arrays for which the returned indices will be valid.
- **k** (*int*, *optional*) – Refers to the diagonal offset. By default,  $k = 0$  i.e. the main diagonal. The positive value of  $k$  denotes the diagonals above the main diagonal, while the negative value includes the diagonals below the main diagonal.
- **m** (*int*, *optional*) – The column dimension of the arrays for which the returned arrays will be valid. By default,  $m = n$ .



**Returns**

**y** – The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

**Return type**

tuple of ndarrays

**See also:**

`numpy.triu_indices`

**cupy.triu\_indices\_from**

`cupy.triu_indices_from(arr, k=0)`

Returns indices for the upper-triangle of arr.

**Parameters**

- **arr** (`cupy.ndarray`) – The indices are valid for square arrays.
- **k** (`int`, *optional*) – Diagonal offset (see ‘triu\_indices’ for details).

**Returns**

**triu\_indices\_from** – Indices for the upper-triangle of *arr*.

**Return type**

tuple of ndarrays

**See also:**

`numpy.triu_indices_from`

**cupy.ix\_**

`cupy.ix_(*args)`

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using *ix\_* one can quickly construct index arrays that will index the cross product. `a[cupy.ix_([1,3],[2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]`.

**Parameters**

**\*args** – 1-D sequences

**Returns**

N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

**Return type**

tuple of ndarrays

## Examples

```
>>> a = cupy.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> ixgrid = cupy.ix_([0,1], [2,4])
>>> ixgrid
(array([[0],
       [1]]), array([[2, 4]]))
```

**Warning:** This function may synchronize the device.

See also:

`numpy.ix_()`

## `cupy.ravel_multi_index`

`cupy.ravel_multi_index(multi_index, dims, mode='wrap', order='C')`

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

### Parameters

- **multi\_index** (*tuple of cupy.ndarray*) – A tuple of integer arrays, one array for each dimension.
- **dims** (*tuple of ints*) – The shape of array into which the indices from `multi_index` apply.
- **mode** (*'raise', 'wrap' or 'clip'*) – Specifies how out-of-bounds indices are handled. Can specify either one mode or a tuple of modes, one mode per index:
  - *'raise'* – raise an error
  - *'wrap'* – wrap around (default)
  - *'clip'* – clip to the rangeIn *'clip'* mode, a negative index which would normally wrap will clip to 0 instead.
- **order** (*'C' or 'F'*) – Determines whether the multi-index should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

### Returns

An array of indices into the flattened version of an array of dimensions `dims`.

### Return type

raveled\_indices (*cupy.ndarray*)

**Warning:** This function may synchronize the device when `mode == 'raise'`.

## Notes

Note that the default *mode* ('wrap') is different than in NumPy. This is done to avoid potential device synchronization.

## Examples

```
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (7,6))
array([22, 41, 37])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (7,6),
...                          order='F')
array([31, 41, 13])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (4,6),
...                          mode='clip')
array([22, 23, 19])
>>> cupy.ravel_multi_index(cupy.asarray([[3,6,6],[4,5,1]]), (4,4),
...                          mode=('clip', 'wrap'))
array([12, 13, 13])
>>> cupy.ravel_multi_index(cupy.asarray((3,1,4,1)), (6,7,8,9))
array(1621)
```

See also:

`numpy.ravel_multi_index()`, `unravel_index()`

## cupy.unravel\_index

`cupy.unravel_index(indices, dims, order='C')`

Converts array of flat indices into a tuple of coordinate arrays.

### Parameters

- **indices** (`cupy.ndarray`) – An integer array whose elements are indices into the flattened version of an array of dimensions `dims`.
- **dims** (*tuple of ints*) – The shape of the array to use for unraveling indices.
- **order** ('C' or 'F') – Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

### Returns

Each array in the tuple has the same shape as the indices array.

### Return type

tuple of ndarrays

## Examples

```
>>> cupy.unravel_index(cupy.array([22, 41, 37]), (7, 6))
(array([3, 6, 6]), array([4, 5, 1]))
>>> cupy.unravel_index(cupy.array([31, 41, 13]), (7, 6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))
```

**Warning:** This function may synchronize the device.

See also:

`numpy.unravel_index()`, `ravel_multi_index()`

## cupy.diag\_indices

`cupy.diag_indices(n, ndim=2)`

Return the indices to access the main diagonal of an array.

Returns a tuple of indices that can be used to access the main diagonal of an array with `ndim >= 2` dimensions and shape `(n, n, ..., n)`.

### Parameters

- **n** (*int*) – The size, along each dimension of the arrays for which the indices are to be returned.
- **ndim** (*int*) – The number of dimensions. default 2.

## Examples

Create a set of indices to access the diagonal of a (4, 4) array:

```
>>> di = cupy.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = cupy.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[di] = 100
>>> a
array([[100,  1,  2,  3],
       [ 4, 100,  6,  7],
       [ 8,  9, 100, 11],
       [12, 13, 14, 100]])
```

Create indices to manipulate a 3-D array:

```
>>> d3 = cupy.diag_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))
```

And use it to set the diagonal of an array of zeros to 1:

```
>>> a = cupy.zeros((2, 2, 2), dtype=int)
>>> a[d3] = 1
>>> a
array([[[1, 0],
        [0, 0]],

       [[0, 0],
        [0, 1]]])
```

See also:

`numpy.diag_indices()`

### `cupy.diag_indices_from`

`cupy.diag_indices_from(arr)`

Return the indices to access the main diagonal of an n-dimensional array. See *diag\_indices* for full details.

#### Parameters

**arr** (`cupy.ndarray`) – At least 2-D.

See also:

`numpy.diag_indices_from()`

### Indexing-like operations

<code>take(a, indices[, axis, out])</code>	Takes elements of an array at specified indices along an axis.	
<code>take_along_axis(a, indices, axis)</code>	Take values from the input array by matching 1d index and data slices.	
<code>choose(a, choices[, out, mode])</code>		
<code>compress(condition, a[, axis, out])</code>	Returns selected slices of an array along given axis.	
<code>diag(v[, k])</code>	Returns a diagonal or a diagonal array.	
<code>diagonal(a[, offset, axis1, axis2])</code>	Returns specified diagonals.	
<code>select(condlist, choicelist[, default])</code>	Return an array drawn from elements in choicelist, depending on conditions.	
<code>lib.stride_tricks.as_strided(x[, shape, strides])</code>	shape,	Create a view into the array with the given shape and strides.

## cupy.take

`cupy.take(a, indices, axis=None, out=None)`

Takes elements of an array at specified indices along an axis.

This is an implementation of “fancy indexing” at single axis.

This function does not support `mode` option.

### Parameters

- **a** (`cupy.ndarray`) – Array to extract elements.
- **indices** (*int or array-like*) – Indices of elements that this function takes.
- **axis** (*int*) – The axis along which to select indices. The flattened input is used by default.
- **out** (`cupy.ndarray`) – Output array. If provided, it should be of appropriate shape and dtype.

### Returns

The result of fancy indexing.

### Return type

*cupy.ndarray*

See also:

`numpy.take()`

## cupy.take\_along\_axis

`cupy.take_along_axis(a, indices, axis)`

Take values from the input array by matching 1d index and data slices.

### Parameters

- **a** (`cupy.ndarray`) – Array to extract elements.
- **indices** (`cupy.ndarray`) – Indices to take along each 1d slice of **a**.
- **axis** (*int*) – The axis to take 1d slices along.

### Returns

The indexed result.

### Return type

*cupy.ndarray*

See also:

`numpy.take_along_axis()`

## cupy.choose

`cupy.choose(a, choices, out=None, mode='raise')`

## cupy.compress

`cupy.compress(condition, a, axis=None, out=None)`

Returns selected slices of an array along given axis.

### Parameters

- **condition** (*1-D array of bools*) – Array that selects which entries to return. If `len(condition)` is less than the size of `a` along the given axis, then output is truncated to the length of the condition array.
- **a** (`cupy.ndarray`) – Array from which to extract a part.
- **axis** (*int*) – Axis along which to take slices. If `None` (default), work on the flattened array.
- **out** (`cupy.ndarray`) – Output array. If provided, it should be of appropriate shape and dtype.

### Returns

A copy of `a` without the slices along `axis` for which `condition` is false.

### Return type

`cupy.ndarray`

**Warning:** This function may synchronize the device.

### See also:

`numpy.compress()`

## cupy.diagonal

`cupy.diagonal(a, offset=0, axis1=0, axis2=1)`

Returns specified diagonals.

This function extracts the diagonals along two specified axes. The other axes are not changed. This function returns a writable view of this array as NumPy 1.10 will do.

### Parameters

- **a** (`cupy.ndarray`) – Array from which the diagonals are taken.
- **offset** (*int*) – Index of the diagonals. Zero indicates the main diagonals, a positive value upper diagonals, and a negative value lower diagonals.
- **axis1** (*int*) – The first axis to take diagonals from.
- **axis2** (*int*) – The second axis to take diagonals from.

### Returns

A view of the diagonals of `a`.

### Return type

`cupy.ndarray`

See also:

`numpy.diagonal()`

## `cupy.select`

`cupy.select(condlist, choicelist, default=0)`

Return an array drawn from elements in *choicelist*, depending on conditions.

### Parameters

- **condlist** (*list of bool arrays*) – The list of conditions which determine from which array in *choicelist* the output elements are taken. When multiple conditions are satisfied, the first one encountered in *condlist* is used.
- **choicelist** (*list of cupy.ndarray*) – The list of arrays from which the output elements are taken. It has to be of the same length as *condlist*.
- **default** (*scalar*) – If provided, will fill element inserted in *output* when all conditions evaluate to False. default value is 0.

### Returns

The output at position *m* is the *m*-th element of the array in *choicelist* where the *m*-th element of the corresponding array in *condlist* is True.

### Return type

*cupy.ndarray*

See also:

`numpy.select()`

## `cupy.lib.stride_tricks.as_strided`

`cupy.lib.stride_tricks.as_strided(x, shape=None, strides=None)`

Create a view into the array with the given shape and strides.

**Warning:** This function has to be used with extreme care, see notes.

### Parameters

- **x** (*ndarray*) – Array to create a new.
- **shape** (*sequence of int, optional*) – The shape of the new array. Defaults to *x*. shape.
- **strides** (*sequence of int, optional*) – The strides of the new array. Defaults to *x*. strides.

### Returns

*view*

### Return type

*ndarray*

See also:

`numpy.lib.stride_tricks.as_strided`



**reshape**

reshape an array.

**Notes**

`as_strided` creates a view into the array given the exact strides and shape. This means it manipulates the internal data structure of `ndarray` and, if done incorrectly, the array elements can point to invalid memory and can corrupt results or crash your program.

**Inserting data into arrays**

<code>place(arr, mask, vals)</code>	Change elements of an array based on conditional and input values.
<code>put(a, ind, v[, mode])</code>	Replaces specified elements of an array with given values.
<code>putmask(a, mask, values)</code>	Changes elements of an array inplace, based on a conditional mask and input values.
<code>fill_diagonal(a, val[, wrap])</code>	Fills the main diagonal of the given array of any dimensionality.

**cupy.place**

`cupy.place(arr, mask, vals)`

Change elements of an array based on conditional and input values.

This function uses the first N elements of *vals*, where N is the number of true values in *mask*.

**Parameters**

- **arr** (`cupy.ndarray`) – Array to put data into.
- **mask** (*array-like*) – Boolean mask array. Must have the same size as *a*.
- **vals** (*array-like*) – Values to put into *a*. Only the first N elements are used, where N is the number of True values in *mask*. If *vals* is smaller than N, it will be repeated, and if elements of *a* are to be masked, this sequence must be non-empty.

**Examples**

```
>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[ 0,  1,  2],
       [44, 55, 44]])
```

**Warning:** This function may synchronize the device.

See also:

`numpy.place()`

## cupy.put

`cupy.put(a, ind, v, mode='wrap')`

Replaces specified elements of an array with given values.

### Parameters

- **a** (`cupy.ndarray`) – Target array.
- **ind** (*array-like*) – Target indices, interpreted as integers.
- **v** (*array-like*) – Values to place in *a* at target indices. If *v* is shorter than *ind* it will be repeated as necessary.
- **mode** (`str`) – How out-of-bounds indices will behave. Its value must be either `'raise'`, `'wrap'` or `'clip'`. Otherwise, `TypeError` is raised.

---

**Note:** Default *mode* is set to `'wrap'` to avoid unintended performance drop. If you need NumPy's behavior, please pass *mode*=`'raise'` manually.

---

See also:

`numpy.put()`

## cupy.putmask

`cupy.putmask(a, mask, values)`

Changes elements of an array inplace, based on a conditional mask and input values.

Sets `a.flat[n] = values[n]` for each *n* where `mask.flat[n]==True`. If *values* is not the same size as *a* and *mask* then it will repeat.

### Parameters

- **a** (`cupy.ndarray`) – Target array.
- **mask** (`cupy.ndarray`) – Boolean mask array. It has to be the same shape as *a*.
- **values** (`cupy.ndarray` or *scalar*) – Values to put into *a* where *mask* is True. If *values* is smaller than *a*, then it will be repeated.

## Examples

```
>>> x = cupy.arange(6).reshape(2, 3)
>>> cupy.putmask(x, x>2, x**2)
>>> x
array([[ 0,  1,  2],
       [ 9, 16, 25]])
```

If *values* is smaller than *a* it is repeated:

```
>>> x = cupy.arange(6)
>>> cupy.putmask(x, x>2, cupy.array([-33, -44]))
>>> x
array([ 0,  1,  2, -44, -33, -44])
```

See also:

`numpy.putmask()`

## `cupy.fill_diagonal`

`cupy.fill_diagonal(a, val, wrap=False)`

Fills the main diagonal of the given array of any dimensionality.

For an array *a* with *a.ndim* > 2, the diagonal is the list of locations with indices *a*[*i*, *i*, ..., *i*] all identical. This function modifies the input array in-place, it does not return a value.

### Parameters

- **a** (`cupy.ndarray`) – The array, at least 2-D.
- **val** (*scalar*) – The value to be written on the diagonal. Its type must be compatible with that of the array *a*.
- **wrap** (*bool*) – If specified, the diagonal is “wrapped” after *N* columns. This affects only tall matrices.

### Examples

```
>>> a = cupy.zeros((3, 3), int)
>>> cupy.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

See also:

`numpy.fill_diagonal()`

## Iterating over arrays

---

*flatiter*(*a*)

Flat iterator object to iterate over arrays.

---

## `cupy.flatiter`

**class** `cupy.flatiter(a)`

Flat iterator object to iterate over arrays.

A flatiter iterator is returned by *x.flat* for any array *x*. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its `next` method.

Iteration is done in row-major, C-style order (the last index varying the fastest).

### Variables

**base** (`cupy.ndarray`) – A reference to the array that is iterated over.

---

**Note:** Restricted support of basic slicing is currently supplied. Advanced indexing is not supported yet.

---

See also:

`numpy.flatiter()`

### Methods

`__getitem__(ind)`

`__setitem__(ind, value)`

`__len__()`

`__next__()`

`__iter__()`

`copy()`

Get a copy of the iterator as a 1-D array.

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### Attributes

**base**

A reference to the array that is iterated over.

## 5.3.8 Input and output

---

**Hint:** [NumPy API Reference: Input and output](#)

---

## NumPy binary files (NPY, NPZ)

<code>load(file[, mmap_mode, allow_pickle])</code>	Loads arrays or pickled objects from <code>.npy</code> , <code>.npz</code> or pickled file.
<code>save(file, arr[, allow_pickle])</code>	Saves an array to a binary file in <code>.npy</code> format.
<code>savez(file, *args, **kwds)</code>	Saves one or more arrays into a file in uncompressed <code>.npz</code> format.
<code>savez_compressed(file, *args, **kwds)</code>	Saves one or more arrays into a file in compressed <code>.npz</code> format.

### cupy.load

`cupy.load(file, mmap_mode=None, allow_pickle=None)`

Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.

This function just calls `numpy.load` and then sends the arrays to the current device. NPZ file is converted to `NpzFile` object, which defers the transfer to the time of accessing the items.

#### Parameters

- **file** (*file-like object or string*) – The file to read.
- **mmap\_mode** (*None, 'r+', 'r', 'w+', 'c'*) – If not `None`, memory-map the file to construct an intermediate `numpy.ndarray` object and transfer it to the current device.
- **allow\_pickle** (*bool*) – Allow loading pickled object arrays stored in `npz` files. Reasons for disallowing pickles include security, as loading pickled data can execute arbitrary code. If pickles are disallowed, loading object arrays will fail. Please be aware that CuPy does not support arrays with dtype of `object`. The default is `False`. This option is available only for NumPy 1.10 or later. In NumPy 1.9, this option cannot be specified (loading pickled objects is always allowed).

#### Returns

CuPy array or `NpzFile` object depending on the type of the file. `NpzFile` object is a dictionary-like object with the context manager protocol (which enables us to use *with* statement on it).

See also:

`numpy.load()`

### cupy.save

`cupy.save(file, arr, allow_pickle=None)`

Saves an array to a binary file in `.npy` format.

#### Parameters

- **file** (*file or str*) – File or filename to save.
- **arr** (*array\_like*) – Array to save. It should be able to feed to `cupy.asnumpy()`.
- **allow\_pickle** (*bool*) – Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3). The default is `True`. This option is available only for

NumPy 1.10 or later. In NumPy 1.9, this option cannot be specified (saving objects using pickles is always allowed).

See also:

`numpy.save()`

## `cupy.savez`

`cupy.savez(file, *args, **kwds)`

Saves one or more arrays into a file in uncompressed `.npz` format.

Arguments without keys are treated as arguments with automatic keys named `arr_0`, `arr_1`, etc. corresponding to the positions in the argument list. The keys of arguments are used as keys in the `.npz` file, which are used for accessing NpzFile object when the file is read by `cupy.load()` function.

### Parameters

- **file** (*file* or *str*) – File or filename to save.
- **\*args** – Arrays with implicit keys.
- **\*\*kwds** – Arrays with explicit keys.

See also:

`numpy.savez()`

## `cupy.savez_compressed`

`cupy.savez_compressed(file, *args, **kwds)`

Saves one or more arrays into a file in compressed `.npz` format.

It is equivalent to `cupy.savez()` function except the output file is compressed.

See also:

`cupy.savez()` for more detail, `numpy.savez_compressed()`

## Text files

<code>loadtxt(*args, **kwargs)</code>	Load data from a text file.
<code>savetxt(fname, X, *args, **kwargs)</code>	Save an array to a text file.
<code>genfromtxt(*args, **kwargs)</code>	Load data from text file, with missing values handled as specified.
<code>fromstring(*args, **kwargs)</code>	A new 1-D array initialized from text data in a string.

**cupy.savetxt**`cupy.savetxt(fname, X, *args, **kwargs)`

Save an array to a text file.

---

**Note:** Uses NumPy's `savetxt`.

---

**See also:**`numpy.savetxt()`**cupy.genfromtxt**`cupy.genfromtxt(*args, **kwargs)`

Load data from text file, with missing values handled as specified.

---

**Note:** Uses NumPy's `genfromtxt` and coerces the result to a CuPy array.

---

**See also:**`numpy.genfromtxt()`**String formatting**

<code>array2string(a, *args, **kwargs)</code>	Return a string representation of an array.
<code>array_repr(arr[, max_line_width, precision, ...])</code>	Returns the string representation of an array.
<code>array_str(arr[, max_line_width, precision, ...])</code>	Returns the string representation of the content of an array.
<code>format_float_positional(x, *args, **kwargs)</code>	Format a floating-point scalar as a decimal string in positional notation.
<code>format_float_scientific(x, *args, **kwargs)</code>	Format a floating-point scalar as a decimal string in scientific notation.

**cupy.array2string**`cupy.array2string(a, *args, **kwargs)`

Return a string representation of an array.

**See also:**`numpy.array2string()`

## cupy.array\_repr

`cupy.array_repr(arr, max_line_width=None, precision=None, suppress_small=None)`

Returns the string representation of an array.

### Parameters

- **arr** (*array\_like*) – Input array. It should be able to feed to `cupy.asnumpy()`.
- **max\_line\_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress\_small** (*bool*) – If True, very small numbers are printed as zeros

### Returns

The string representation of `arr`.

### Return type

str

See also:

`numpy.array_repr()`

## cupy.array\_str

`cupy.array_str(arr, max_line_width=None, precision=None, suppress_small=None)`

Returns the string representation of the content of an array.

### Parameters

- **arr** (*array\_like*) – Input array. It should be able to feed to `cupy.asnumpy()`.
- **max\_line\_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress\_small** (*bool*) – If True, very small number are printed as zeros.

See also:

`numpy.array_str()`

## cupy.format\_float\_positional

`cupy.format_float_positional(x, *args, **kwargs)`

Format a floating-point scalar as a decimal string in positional notation.

See `numpy.format_float_positional()` for the list of arguments.

See also:

`numpy.format_float_positional()`



**cupy.format\_float\_scientific**

`cupy.format_float_scientific(x, *args, **kwargs)`

Format a floating-point scalar as a decimal string in scientific notation.

See `numpy.format_float_scientific()` for the list of arguments.

**See also:**

`numpy.format_float_scientific()`

**Base-n representations**

<code>binary_repr(num[, width])</code>	Return the binary representation of the input number as a string.
<code>base_repr(number[, base, padding])</code>	Return a string representation of a number in the given base system.

**cupy.base\_repr**

`cupy.base_repr(number, base=2, padding=0)`

Return a string representation of a number in the given base system.

**See also:**

`numpy.base_repr()`

**5.3.9 Linear algebra (cupy.linalg)**


---

**Hint:** NumPy API Reference: Linear algebra (`numpy.linalg`)

---

**See also:**

*Linear algebra (cupyx.scipy.linalg)*

**Matrix and vector products**

<code>dot(a, b[, out])</code>	Returns a dot product of two arrays.
<code>vdot(a, b)</code>	Returns the dot product of two vectors.
<code>inner(a, b)</code>	Returns the inner product of two arrays.
<code>outer(a, b[, out])</code>	Returns the outer product of two vectors.
<code>matmul</code>	<code>matmul(x1, x2, /, out=None, **kwargs)</code>
<code>tensordot(a, b[, axes])</code>	Returns the tensor dot product of two arrays along specified axes.
<code>einsum(subscripts, *operands[, dtype, optimize])</code>	Evaluates the Einstein summation convention on the operands.
<code>linalg.matrix_power(M, n)</code>	Raise a square matrix to the (integer) power <i>n</i> .
<code>kron(a, b)</code>	Returns the kronecker product of two arrays.

## cupy.dot

`cupy.dot(a, b, out=None)`

Returns a dot product of two arrays.

For arrays with more than one axis, it computes the dot product along the last axis of `a` and the second-to-last axis of `b`. This is just a matrix product if the both arrays are 2-D. For 1-D arrays, it uses their unique axis as an axis to take dot product over.

### Parameters

- **a** (`cupy.ndarray`) – The left argument.
- **b** (`cupy.ndarray`) – The right argument.
- **out** (`cupy.ndarray`) – Output array.

### Returns

The dot product of `a` and `b`.

### Return type

*cupy.ndarray*

### See also:

`numpy.dot()`

## cupy.vdot

`cupy.vdot(a, b)`

Returns the dot product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs inner product of these vectors.

### Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.

### Returns

Zero-dimensional array of the dot product result.

### Return type

*cupy.ndarray*

### See also:

`numpy.vdot()`

## cupy.inner

`cupy.inner(a, b)`

Returns the inner product of two arrays.

It uses the last axis of each argument to take sum product.

### Parameters

- **a** (`cupy.ndarray`) – The first argument.

- **b** (`cupy.ndarray`) – The second argument.

**Returns**

The inner product of `a` and `b`.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.inner()`

**cupy.outer**

`cupy.outer(a, b, out=None)`

Returns the outer product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs outer product of these vectors.

**Parameters**

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **out** (`cupy.ndarray`) – Output array.

**Returns**

2-D array of the outer product of `a` and `b`.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.outer()`

**cupy.tensordot**

`cupy.tensordot(a, b, axes=2)`

Returns the tensor dot product of two arrays along specified axes.

This is equivalent to compute dot product along the specified axes which are treated as one axis by reshaping.

**Parameters**

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **axes** –
  - If it is an integer, then `axes` axes at the last of `a` and the first of `b` are used.
  - If it is a pair of sequences of integers, then these two sequences specify the list of axes for `a` and `b`. The corresponding axes are paired for sum-product.

**Returns**

The tensor dot product of `a` and `b` along the axes specified by `axes`.

**Return type**

`cupy.ndarray`

See also:

`numpy.tensordot()`

## `cupy.einsum`

`cupy.einsum(subscripts, *operands, dtype=None, optimize=False)`

Evaluates the Einstein summation convention on the operands. Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations.

---

### Note:

- Memory contiguity of the returned array is not always compatible with that of `numpy.einsum()`.
  - `out`, `order`, and `casting` options are not supported.
  - If `CUPY_ACCELERATORS` includes `cutensornet`, the `einsum` calculation will be performed by the cuTensorNet backend if possible.
    - The support of the `optimize` option is limited (currently, only `False`, ‘cutensornet’, or a custom path for pairwise contraction is supported, and the maximum intermediate size is ignored). If you need finer control for path optimization, consider replacing `cupy.einsum()` by `cuquantum.contract()` instead.
    - Requires cuQuantum Python (v22.03+).
  - If `CUPY_ACCELERATORS` includes `cutensor`, `einsum` will be accelerated by the cuTENSOR backend whenever possible.
- 

### Parameters

- **subscripts** (*str*) – Specifies the subscripts for summation.
- **operands** (*sequence of arrays*) – These are the arrays for the operation.
- **dtype** – If provided, forces the calculation to use the data type specified. Default is `None`.
- **optimize** – Valid options include `{False, True, ‘greedy’, ‘optimal’}`. Controls if intermediate optimization should occur. No optimization will occur if `False`, and `True` will default to the ‘greedy’ algorithm. Also accepts an explicit contraction list from `numpy.einsum_path()`. Defaults to `False`. If a pair is supplied, the second argument is assumed to be the maximum intermediate size created.

### Returns

The calculation based on the Einstein summation convention.

### Return type

`cupy.ndarray`

See also:

`numpy.einsum()`

## cupy.linalg.matrix\_power

`cupy.linalg.matrix_power(M, n)`

Raise a square matrix to the (integer) power  $n$ .

### Parameters

- **M** (`ndarray`) – Matrix to raise by power  $n$ .
- **n** (`~int`) – Power to raise matrix to.

### Returns

Output array.

### Return type

`ndarray`

..seealso: `numpy.linalg.matrix_power()`

## cupy.kron

`cupy.kron(a, b)`

Returns the kronecker product of two arrays.

### Parameters

- **a** (`ndarray`) – The first argument.
- **b** (`ndarray`) – The second argument.

### Returns

Output array.

### Return type

`ndarray`

See also:

`numpy.kron()`

## Decompositions

<code>linalg.cholesky(a)</code>	Cholesky decomposition.
<code>linalg.qr(a[, mode])</code>	QR decomposition.
<code>linalg.svd(a[, full_matrices, compute_uv])</code>	Singular Value Decomposition.

## cupy.linalg.cholesky

`cupy.linalg.cholesky(a)`

Cholesky decomposition.

Decompose a given two-dimensional square matrix into  $L * L.H$ , where  $L$  is a lower-triangular matrix and  $.H$  is a conjugate transpose operator.

### Parameters

**a** (`cupy.ndarray`) – Hermitian (symmetric if all elements are real), positive-definite input matrix with dimension  $(\dots, M, M)$ .

**Returns**

The lower-triangular matrix of shape  $(\dots, M, M)$ .

**Return type**

*cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.cholesky()`

**cupy.linalg.qr**

`cupy.linalg.qr(a, mode='reduced')`

QR decomposition.

Decompose a given two-dimensional matrix into  $Q * R$ , where  $Q$  is an orthonormal and  $R$  is an upper-triangular matrix.

**Parameters**

- **a** (*cupy.ndarray*) – The input matrix.
- **mode** (*str*) – The mode of decomposition. Currently ‘reduced’, ‘complete’, ‘r’, and ‘raw’ modes are supported. The default mode is ‘reduced’, in which matrix  $A = (\dots, M, N)$  is decomposed into  $Q, R$  with dimensions  $(\dots, M, K), (\dots, K, N)$ , where  $K = \min(M, N)$ .

**Returns**

Although the type of returned object depends on the mode, it returns a tuple of  $(Q, R)$  by default. For details, please see the document of `numpy.linalg.qr()`.

**Return type**

*cupy.ndarray*, or tuple of *ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.qr()`

## cupy.linalg.svd

`cupy.linalg.svd(a, full_matrices=True, compute_uv=True)`

Singular Value Decomposition.

Factorizes the matrix `a` as  $u * \text{np.diag}(s) * v$ , where `u` and `v` are unitary and `s` is an one-dimensional array of `a`'s singular values.

### Parameters

- **a** (`cupy.ndarray`) – The input matrix with dimension  $(\dots, M, N)$ .
- **full\_matrices** (`bool`) – If True, it returns `u` and `v` with dimensions  $(\dots, M, M)$  and  $(\dots, N, N)$ . Otherwise, the dimensions of `u` and `v` are  $(\dots, M, K)$  and  $(\dots, K, N)$ , respectively, where  $K = \min(M, N)$ .
- **compute\_uv** (`bool`) – If False, it only returns singular values.

### Returns

A tuple of (`u`, `s`, `v`) such that  $a = u * \text{np.diag}(s) * v$ .

### Return type

tuple of `cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupy.errstate()` or `cupy.seterr()`.

**Note:** On CUDA, when `a.ndim > 2` and the matrix dimensions  $\leq 32$ , a fast code path based on Jacobian method (`gesvdj`) is taken. Otherwise, a QR method (`gesvd`) is used.

On ROCm, there is no such a fast code path that switches the underlying algorithm.

### See also:

`numpy.linalg.svd()`

## Matrix eigenvalues

<code>linalg.eigh(a[, UPLO])</code>	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.
<code>linalg.eigvalsh(a[, UPLO])</code>	Compute the eigenvalues of a complex Hermitian or real symmetric matrix.

## cupy.linalg.eigh

cupy.linalg.eigh(*a*, *UPLO*='L')

Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of *a*, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

### Parameters

- **a** ([cupy.ndarray](#)) – A symmetric 2-D square matrix (*M*, *M*) or a batch of symmetric 2-D square matrices (... , *M*, *M*).
- **UPLO** (*str*) – Select from 'L' or 'U'. It specifies which part of *a* is used. 'L' uses the lower triangular part of *a*, and 'U' uses the upper triangular part of *a*.

### Returns

Returns a tuple (*w*, *v*). *w* contains eigenvalues and *v* contains eigenvectors. *v*[:, *i*] is an eigenvector corresponding to an eigenvalue *w*[*i*]. For batch input, *v*[*k*, :, *i*] is an eigenvector corresponding to an eigenvalue *w*[*k*, *i*] of *a*[*k*].

### Return type

tuple of [ndarray](#)

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

[numpy.linalg.eigh\(\)](#)

## cupy.linalg.eigvalsh

cupy.linalg.eigvalsh(*a*, *UPLO*='L')

Compute the eigenvalues of a complex Hermitian or real symmetric matrix.

Main difference from `eigh`: the eigenvectors are not computed.

### Parameters

- **a** ([cupy.ndarray](#)) – A symmetric 2-D square matrix (*M*, *M*) or a batch of symmetric 2-D square matrices (... , *M*, *M*).
- **UPLO** (*str*) – Select from 'L' or 'U'. It specifies which part of *a* is used. 'L' uses the lower triangular part of *a*, and 'U' uses the upper triangular part of *a*.

### Returns

Returns eigenvalues as a vector *w*. For batch input, *w*[*k*] is a vector of eigenvalues of matrix *a*[*k*].

### Return type

[cupy.ndarray](#)



**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.eigvalsh()`

## Norms and other numbers

<code>linalg.norm(x[, ord, axis, keepdims])</code>	Returns one of matrix norms specified by <code>ord</code> parameter.
<code>linalg.det(a)</code>	Returns the determinant of an array.
<code>linalg.matrix_rank(M[, tol])</code>	Return matrix rank of array using SVD method
<code>linalg.slogdet(a)</code>	Returns sign and logarithm of the determinant of an array.
<code>trace(a[, offset, axis1, axis2, dtype, out])</code>	Returns the sum along the diagonals of an array.

### cupy.linalg.norm

`cupy.linalg.norm(x, ord=None, axis=None, keepdims=False)`

Returns one of matrix norms specified by `ord` parameter.

See `numpy.linalg.norm` for more detail.

#### Parameters

- **x** (`cupy.ndarray`) – Array to take norm. If `axis` is `None`, `x` must be 1-D or 2-D.
- **ord** (*non-zero int, inf, -inf, 'fro'*) – Norm type.
- **axis** (*int, 2-tuple of ints, None*) – 1-D or 2-D norm is computed over `axis`.
- **keepdims** (*bool*) – If this is set `True`, the axes which are normed over are left.

#### Returns

`cupy.ndarray`

### cupy.linalg.det

`cupy.linalg.det(a)`

Returns the determinant of an array.

#### Parameters

**a** (`cupy.ndarray`) – The input matrix with dimension `(..., N, N)`.

#### Returns

Determinant of `a`. Its shape is `a.shape[:-2]`.

#### Return type

`cupy.ndarray`

See also:

`numpy.linalg.det()`

## cupy.linalg.matrix\_rank

cupy.linalg.matrix\_rank(*M*, *tol=None*)

Return matrix rank of array using SVD method

### Parameters

- **M** (`cupy.ndarray`) – Input array. Its *ndim* must be less than or equal to 2.
- **tol** (*None* or `float`) – Threshold of singular value of *M*. When *tol* is *None*, and *eps* is the epsilon value for datatype of *M*, then *tol* is set to  $S_{\max}() * \max(M.\text{shape}) * \text{eps}$ , where *S* is the singular value of *M*. It obeys `numpy.linalg.matrix_rank()`.

### Returns

Rank of *M*.

### Return type

`cupy.ndarray`

See also:

`numpy.linalg.matrix_rank()`

## cupy.linalg.slogdet

cupy.linalg.slogdet(*a*)

Returns sign and logarithm of the determinant of an array.

It calculates the natural logarithm of the determinant of a given value.

### Parameters

**a** (`cupy.ndarray`) – The input matrix with dimension  $(\dots, N, N)$ .

### Returns

It returns a tuple (*sign*, *logdet*). *sign* represents each sign of the determinant as a real number 0, 1 or -1. 'logdet' represents the natural logarithm of the absolute of the determinant. If the determinant is zero, *sign* will be 0 and *logdet* will be -inf. The shapes of both *sign* and *logdet* are equal to *a.shape[:-2]*.

### Return type

tuple of `ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**Warning:** To produce the same results as `numpy.linalg.slogdet()` for singular inputs, set the *linalg* configuration to *raise*.

See also:

`numpy.linalg.slogdet()`

## cupy.trace

`cupy.trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Returns the sum along the diagonals of an array.

It computes the sum along the diagonals at `axis1` and `axis2`.

### Parameters

- **a** (`cupy.ndarray`) – Array to take trace.
- **offset** (`int`) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.
- **axis1** (`int`) – The first axis along which the trace is taken.
- **axis2** (`int`) – The second axis along which the trace is taken.
- **dtype** – Data type specifier of the output.
- **out** (`cupy.ndarray`) – Output array.

### Returns

The trace of `a` along axes (`axis1`, `axis2`).

### Return type

`cupy.ndarray`

See also:

`numpy.trace()`

## Solving equations and inverting matrices

<code>linalg.solve(a, b)</code>	Solves a linear matrix equation.
<code>linalg.tensorsolve(a, b[, axes])</code>	Solves tensor equations denoted by $\mathbf{ax} = \mathbf{b}$ .
<code>linalg.lstsq(a, b[, rcond])</code>	Return the least-squares solution to a linear matrix equation.
<code>linalg.inv(a)</code>	Computes the inverse of a matrix.
<code>linalg.pinv(a[, rcond])</code>	Compute the Moore-Penrose pseudoinverse of a matrix.
<code>linalg.tensorinv(a[, ind])</code>	Computes the inverse of a tensor.

## cupy.linalg.solve

`cupy.linalg.solve(a, b)`

Solves a linear matrix equation.

It computes the exact solution of  $\mathbf{x}$  in  $\mathbf{ax} = \mathbf{b}$ , where `a` is a square and full rank matrix.

### Parameters

- **a** (`cupy.ndarray`) – The matrix with dimension  $(\dots, \mathbf{M}, \mathbf{M})$ .
- **b** (`cupy.ndarray`) – The matrix with dimension  $(\dots, \mathbf{M})$  or  $(\dots, \mathbf{M}, \mathbf{K})$ .

### Returns

The matrix with dimension  $(\dots, \mathbf{M})$  or  $(\dots, \mathbf{M}, \mathbf{K})$ .

**Return type***cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**`numpy.linalg.solve()`**cupy.linalg.tensorsolve**`cupy.linalg.tensorsolve(a, b, axes=None)`

Solves tensor equations denoted by  $ax = b$ .

Suppose that  $b$  is equivalent to `cupy.tensordot(a, x)`. This function computes tensor  $x$  from  $a$  and  $b$ .

**Parameters**

- **a** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **b** (`cupy.ndarray`) – The tensor with `len(shape) >= 1`
- **axes** (*tuple of ints*) – Axes in  $a$  to reorder to the right before inversion.

**Returns**

The tensor with shape  $Q$  such that  $b.shape + Q == a.shape$ .

**Return type***cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**`numpy.linalg.tensorsolve()`**cupy.linalg.lstsq**`cupy.linalg.lstsq(a, b, rcond='warn')`

Return the least-squares solution to a linear matrix equation.

Solves the equation  $ax = b$  by computing a vector  $x$  that minimizes the Euclidean 2-norm  $\|b - ax\|^2$ . The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of  $a$  can be less than, equal to, or greater than its number of linearly independent columns). If  $a$  is square and of full rank, then  $x$  (but for round-off error) is the “exact” solution of the equation.

**Parameters**

- **a** (`cupy.ndarray`) – “Coefficient” matrix with dimension  $(M, N)$
- **b** (`cupy.ndarray`) – “Dependent variable” values with dimension  $(M,)$  or  $(M, K)$

- **rcond** (*float*) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by *s*, and sets all singular values smaller than *s* to zero.

**Returns**

A tuple of (*x*, *residuals*, *rank*, *s*). Note *x* is the least-squares solution with shape (N,) or (N, K) depending if *b* was two-dimensional. The sums of *residuals* is the squared Euclidean 2-norm for each column in  $b - a*x$ . The *residuals* is an empty array if the rank of *a* is  $< N$  or  $M \leq N$ , but iff *b* is 1-dimensional, this is a (1,) shape array. Otherwise the shape is (K,). The rank of matrix *a* is an integer. The singular values of *a* are *s*.

**Return type**

*tuple*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.lstsq()`

**cupy.linalg.inv**

`cupy.linalg.inv(a)`

Computes the inverse of a matrix.

This function computes matrix *a\_inv* from n-dimensional regular matrix *a* such that `dot(a, a_inv) == eye(n)`.

**Parameters**

*a* (`cupy.ndarray`) – The regular matrix

**Returns**

The inverse of a matrix.

**Return type**

*cupy.ndarray*

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

**See also:**

`numpy.linalg.inv()`

## cupy.linalg.pinv

`cupy.linalg.pinv(a, rcond=1e-15)`

Compute the Moore-Penrose pseudoinverse of a matrix.

It computes a pseudoinverse of a matrix `a`, which is a generalization of the inverse matrix with Singular Value Decomposition (SVD). Note that it automatically removes small singular values for stability.

### Parameters

- **a** (`cupy.ndarray`) – The matrix with dimension  $(\dots, M, N)$
- **rcond** (`float` or `cupy.ndarray`) – Cutoff parameter for small singular values. For stability it computes the largest singular value denoted by  $s$ , and sets all singular values smaller than  $\text{rcond} * s$  to zero. Broadcasts against the stack of matrices.

### Returns

The pseudoinverse of `a` with dimension  $(\dots, N, M)$ .

### Return type

`cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.pinv()`

## cupy.linalg.tensorinv

`cupy.linalg.tensorinv(a, ind=2)`

Computes the inverse of a tensor.

This function computes tensor `a_inv` from tensor `a` such that `tensordot(a_inv, a, ind) == I`, where `I` denotes the identity tensor.

### Parameters

- **a** (`cupy.ndarray`) – The tensor such that `prod(a.shape[:ind]) == prod(a.shape[ind:])`.
- **ind** (`int`) – The positive number used in axes option of `tensordot`.

### Returns

The inverse of a tensor whose shape is equivalent to `a.shape[ind:] + a.shape[:ind]`.

### Return type

`cupy.ndarray`

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the `linalg` configuration to a value that is not `ignore` in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

`numpy.linalg.tensorinv()`

### 5.3.10 Logic functions

---

**Hint:** NumPy API Reference: Logic functions

---

#### Truth value testing

<code>all(a[, axis, out, keepdims])</code>	Tests whether all array elements along a given axis evaluate to True.
<code>any(a[, axis, out, keepdims])</code>	Tests whether any array elements along a given axis evaluate to True.
<code>union1d(arr1, arr2)</code>	Find the union of two arrays.

#### `cupy.all`

`cupy.all(a, axis=None, out=None, keepdims=False)`

Tests whether all array elements along a given axis evaluate to True.

##### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (*int or tuple of ints*) – Along which axis to compute all. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

##### Returns

**y** – An array reduced of the input array along the axis.

##### Return type

`cupy.ndarray`

See also:

`numpy.all`

#### `cupy.any`

`cupy.any(a, axis=None, out=None, keepdims=False)`

Tests whether any array elements along a given axis evaluate to True.

##### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (*int or tuple of ints*) – Along which axis to compute all. The flattened array is used by default.

- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns**

**y** – An array reduced of the input array along the axis.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.any`

## `cupy.union1d`

`cupy.union1d(arr1, arr2)`

Find the union of two arrays.

Returns the unique, sorted array of values that are in either of the two input arrays.

**Parameters**

- **arr1** (`cupy.ndarray`) – Input arrays. They are flattened if they are not already 1-D.
- **arr2** (`cupy.ndarray`) – Input arrays. They are flattened if they are not already 1-D.

**Returns**

**union1d** – Sorted union of the input arrays.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.union1d`

## Array contents

<code>isfinite(x, /[, out, casting, dtype])</code>	Tests finiteness elementwise.
<code>isinf(x, /[, out, casting, dtype])</code>	Tests if each element is the positive or negative infinity.
<code>isnan(x, /[, out, casting, dtype])</code>	Tests if each element is a NaN.
<code>isneginf(x[, out])</code>	Test element-wise for negative infinity, return result as bool array.
<code>isposinf(x[, out])</code>	Test element-wise for positive infinity, return result as bool array.

## `cupy.isneginf`

`cupy.isneginf(x, out=None)`

Test element-wise for negative infinity, return result as bool array.

**Parameters**

- **x** (`cupy.ndarray`) – Input array.



- **out** (`cupy.ndarray`, *optional*) – A location into which the result is stored. If provided, it should have a shape that input broadcasts to. By default, `None`, a freshly- allocated boolean array, is returned.

**Returns**

**y** – Boolean array of same shape as **x**.

**Return type**

`cupy.ndarray`

**Examples**

```
>>> cupy.isneginf(0)
array(False)
>>> cupy.isneginf(-cupy.inf)
array(True)
>>> cupy.isneginf(cupy.array([-cupy.inf, -4, cupy.nan, 0, 4, cupy.inf]))
array([ True, False, False, False, False, False])
```

**See also:**

`numpy.isneginf`

**cupy.isposinf**

`cupy.isposinf(x, out=None)`

Test element-wise for positive infinity, return result as bool array.

**Parameters**

- **x** (`cupy.ndarray`) – Input array.
- **out** (`cupy.ndarray`) – A location into which the result is stored. If provided, it should have a shape that input broadcasts to. By default, `None`, a freshly- allocated boolean array, is returned.

**Returns**

**y** – Boolean array of same shape as **x**.

**Return type**

`cupy.ndarray`

**Examples**

```
>>> cupy.isposinf(0)
array(False)
>>> cupy.isposinf(cupy.inf)
array(True)
>>> cupy.isposinf(cupy.array([-cupy.inf, -4, cupy.nan, 0, 4, cupy.inf]))
array([False, False, False, False, False,  True])
```

**See also:**

`numpy.isposinf`

## Array type testing

<code>iscomplex(x)</code>	Returns a bool array, where True if input element is complex.
<code>iscomplexobj(x)</code>	Check for a complex type or an array of complex numbers.
<code>isfortran(a)</code>	Returns True if the array is Fortran contiguous but <i>not</i> C contiguous.
<code>isreal(x)</code>	Returns a bool array, where True if input element is real.
<code>isrealobj(x)</code>	Return True if x is a not complex type or an array of complex numbers.
<code>isscalar(element)</code>	Returns True if the type of num is a scalar type.

### `cupy.iscomplex`

`cupy.iscomplex(x)`

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

**Parameters**

**x** (`cupy.ndarray`) – Input array.

**Returns**

Boolean array of the same shape as **x**.

**Return type**

`cupy.ndarray`

**See also:**

`isreal()`, `iscomplexobj()`

### Examples

```
>>> cupy.iscomplex(cupy.array([1+1j, 1+0j, 4.5, 3, 2, 2j]))
array([ True, False, False, False, False,  True])
```

### `cupy.iscomplexobj`

`cupy.iscomplexobj(x)`

Check for a complex type or an array of complex numbers.

The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, `iscomplexobj` evaluates to True.

**Parameters**

**x** (`cupy.ndarray`) – Input array.

**Returns**

The return value, True if **x** is of a complex type or has at least one complex element.

**Return type**

`bool`

See also:

`isrealobj()`, `iscomplex()`

### Examples

```
>>> cupy.iscomplexobj(cupy.array([3, 1+0j, True]))
True
>>> cupy.iscomplexobj(cupy.array([3, 1, True]))
False
```

## cupy.isfortran

`cupy.isfortran(a)`

Returns True if the array is Fortran contiguous but *not* C contiguous.

If you only want to check if an array is Fortran contiguous use `a.flags.f_contiguous` instead.

#### Parameters

**a** (`cupy.ndarray`) – Input array.

#### Returns

The return value, True if a is Fortran contiguous but not C contiguous.

#### Return type

`bool`

See also:

`isfortran()`

### Examples

`cupy.array` allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

```
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(a)
False
```

```
>>> b = cupy.array([[1, 2, 3], [4, 5, 6]], order='F')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(b)
True
```

The transpose of a C-ordered array is a FORTRAN-ordered array.

```
>>> a = cupy.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> cupy.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> cupy.isfortran(b)
True
```

C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.

```
>>> cupy.isfortran(np.array([1, 2], order='F'))
False
```

## cupy.isreal

### cupy.isreal(x)

Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

#### Parameters

**x** ([cupy.ndarray](#)) – Input array.

#### Returns

Boolean array of same shape as x.

#### Return type

*cupy.ndarray*

#### See also:

[iscomplex\(\)](#), [isrealobj\(\)](#)

## Examples

```
>>> cupy.isreal(cp.array([1+1j, 1+0j, 4.5, 3, 2, 2j]))
array([False,  True,  True,  True,  True, False])
```

## cupy.isrealobj

`cupy.isrealobj(x)`

Return True if *x* is a not complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, *isrealobj* evaluates to False if the data type is complex.

### Parameters

**x** (`cupy.ndarray`) – The input can be of any type and shape.

### Returns

The return value, False if *x* is of a complex type.

### Return type

`bool`

**See also:**

`iscomplexobj()`, `isreal()`

## Examples

```

>>> cupy.isrealobj(cupy.array([3, 1+0j, True]))
False
>>> cupy.isrealobj(cupy.array([3, 1, True]))
True

```

## cupy.isscalar

`cupy.isscalar(element)`

Returns True if the type of *num* is a scalar type.

**See also:**

`numpy.isscalar()`

## Logic operations

<code>logical_and(x1, x2, /[, out, casting, dtype])</code>	Computes the logical AND of two arrays.
<code>logical_or(x1, x2, /[, out, casting, dtype])</code>	Computes the logical OR of two arrays.
<code>logical_not(x, /[, out, casting, dtype])</code>	Computes the logical NOT of an array.
<code>logical_xor(x1, x2, /[, out, casting, dtype])</code>	Computes the logical XOR of two arrays.

## Comparison

<code>allclose(a, b[, rtol, atol, equal_nan])</code>	Returns True if two arrays are element-wise equal within a tolerance.
<code>isclose(a, b[, rtol, atol, equal_nan])</code>	Returns a boolean array where two arrays are equal within a tolerance.
<code>array_equal(a1, a2[, equal_nan])</code>	Returns True if two arrays are element-wise exactly equal.
<code>array_equiv(a1, a2)</code>	Returns True if all elements are equal or shape consistent, i.e., one input array can be broadcasted to create the same shape as the other.
<code>greater(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 > x2$ .
<code>greater_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 \geq x2$ .
<code>less(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 < x2$ .
<code>less_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 \leq x2$ .
<code>equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 == x2$ .
<code>not_equal(x1, x2, /[, out, casting, dtype])</code>	Tests elementwise if $x1 \neq x2$ .

## cupy.allclose

`cupy.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)`

Returns True if two arrays are element-wise equal within a tolerance.

Two values in `a` and `b` are considered equal when the following equation is satisfied.

$$|a - b| \leq \text{atol} + \text{rtol}|b|$$

### Parameters

- **a** (`cupy.ndarray`) – Input array to compare.
- **b** (`cupy.ndarray`) – Input array to compare.
- **rtol** (`float`) – The relative tolerance.
- **atol** (`float`) – The absolute tolerance.
- **equal\_nan** (`bool`) – If True, NaN's in `a` will be considered equal to NaN's in `b`.

### Returns

A boolean 0-dim array. If its value is `True`, two arrays are element-wise equal within a tolerance.

### Return type

`cupy.ndarray`

See also:

`numpy.allclose()`

## cupy.isclose

`cupy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)`

Returns a boolean array where two arrays are equal within a tolerance.

Two values in `a` and `b` are considered equal when the following equation is satisfied.

$$|a - b| \leq \text{atol} + \text{rtol}|b|$$

### Parameters

- **a** (`cupy.ndarray`) – Input array to compare.
- **b** (`cupy.ndarray`) – Input array to compare.
- **rtol** (`float`) – The relative tolerance.
- **atol** (`float`) – The absolute tolerance.
- **equal\_nan** (`bool`) – If True, NaN's in `a` will be considered equal to NaN's in `b`.

### Returns

A boolean array storing where `a` and `b` are equal.

### Return type

`cupy.ndarray`

See also:

`numpy.isclose()`

## cupy.array\_equal

`cupy.array_equal(a1, a2, equal_nan=False)`

Returns True if two arrays are element-wise exactly equal.

### Parameters

- **a1** (`cupy.ndarray`) – Input array to compare.
- **a2** (`cupy.ndarray`) – Input array to compare.
- **equal\_nan** (`bool`) – If True, NaN's in `a1` will be considered equal to NaN's in `a2`.

### Returns

A boolean 0-dim array. If its value is True, two arrays are element-wise equal.

### Return type

`cupy.ndarray`

See also:

`numpy.array_equal()`

## cupy.array\_equiv

`cupy.array_equiv(a1, a2)`

Returns True if all elements are equal or shape consistent, i.e., one input array can be broadcasted to create the same shape as the other.

### Parameters

- **a1** (`cupy.ndarray`) – Input array.
- **a2** (`cupy.ndarray`) – Input array.

### Returns

**A boolean 0-dim array.**  
True if equivalent, otherwise False.

### Return type

*cupy.ndarray*

### See also:

`numpy.array_equiv()`

## 5.3.11 Mathematical functions

---

**Hint:** NumPy API Reference: Mathematical functions

---

### Trigonometric functions

<i>sin</i> (x, /[, out, casting, dtype])	Elementwise sine function.
<i>cos</i> (x, /[, out, casting, dtype])	Elementwise cosine function.
<i>tan</i> (x, /[, out, casting, dtype])	Elementwise tangent function.
<i>arcsin</i> (x, /[, out, casting, dtype])	Elementwise inverse-sine function (a.k.a.
<i>arccos</i> (x, /[, out, casting, dtype])	Elementwise inverse-cosine function (a.k.a.
<i>arctan</i> (x, /[, out, casting, dtype])	Elementwise inverse-tangent function (a.k.a.
<i>hypot</i> (x1, x2, /[, out, casting, dtype])	Computes the hypoteneous of orthogonal vectors of given length.
<i>arctan2</i> (x1, x2, /[, out, casting, dtype])	Elementwise inverse-tangent of the ratio of two arrays.
<i>degrees</i>	<code>rad2deg(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<i>radians</i> (x, /[, out, casting, dtype])	Converts angles from degrees to radians elementwise.
<i>unwrap</i> (p[, discontinuity, axis, period])	Unwrap by taking the complement of large deltas w.r.t.
<i>deg2rad</i>	<code>radians(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<i>rad2deg</i> (x, /[, out, casting, dtype])	Converts angles from radians to degrees elementwise.



## cupy.unwrap

`cupy.unwrap(p, discount=None, axis=-1, *, period=6.283185307179586)`

Unwrap by taking the complement of large deltas w.r.t. the period.

This unwraps a signal  $p$  by changing elements which have an absolute difference from their predecessor of more than `max(discount, period/2)` to their *period*-complementary values.

For the default case where *period* is  $2\pi$  and *discount* is  $\pi$ , this unwraps a radian phase  $p$  such that adjacent differences are never greater than  $\pi$  by adding  $2k\pi$  for some integer  $k$ .

### Parameters

- **p** (`cupy.ndarray`) – Input array. *discount* (float): Maximum discontinuity between values, default is `period/2`. Values below `period/2` are treated as if they were `period/2`. To have an effect different from the default, *discount* should be larger than `period/2`.
- **axis** (`int`) – Axis along which unwrap will operate, default is the last axis.
- **period** – float, optional Size of the range over which the input wraps. By default, it is  $2\pi$ .

### Returns

The result array.

### Return type

`cupy.ndarray`

See also:

`numpy.unwrap()`

## Hyperbolic functions

<code>sinh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic sine function.
<code>cosh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic cosine function.
<code>tanh(x, /[, out, casting, dtype])</code>	Elementwise hyperbolic tangent function.
<code>arcsinh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic sine function.
<code>arccosh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic cosine function.
<code>arctanh(x, /[, out, casting, dtype])</code>	Elementwise inverse of hyperbolic tangent function.

## Rounding

<code>around(a[, decimals, out])</code>	Rounds to the given number of decimals.
<code>round_(a[, decimals, out])</code>	
<code>rint(x, /[, out, casting, dtype])</code>	Rounds each element of an array to the nearest integer.
<code>fix(x, /[, out, casting, dtype])</code>	If given value $x$ is positive, it return $\text{floor}(x)$ .
<code>floor(x, /[, out, casting, dtype])</code>	Rounds each element of an array to its floor integer.
<code>ceil(x, /[, out, casting, dtype])</code>	Rounds each element of an array to its ceiling integer.
<code>trunc(x, /[, out, casting, dtype])</code>	Rounds each element of an array towards zero.

## cupy.around

`cupy.around(a, decimals=0, out=None)`

Rounds to the given number of decimals.

### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **decimals** (`int`) – Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.
- **out** (`cupy.ndarray`) – Output array.

### Returns

Rounded array.

### Return type

*cupy.ndarray*

See also:

`numpy.around()`

## cupy.round\_

`cupy.round_(a, decimals=0, out=None)`

## cupy.fix

`cupy.fix(x, /, out=None, *, casting='same_kind', dtype=None)`

If given value **x** is positive, it return `floor(x)`.

Else, it return `ceil(x)`.

See also:

`numpy.fix()`

## Sums, products, differences

<code>prod(a[, axis, dtype, out, keepdims])</code>	Returns the product of an array along given axes.
<code>sum(a[, axis, dtype, out, keepdims])</code>	Returns the sum of an array along given axes.
<code>nanprod(a[, axis, dtype, out, keepdims])</code>	Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.
<code>nansum(a[, axis, dtype, out, keepdims])</code>	Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.
<code>cumprod(a[, axis, dtype, out])</code>	Returns the cumulative product of an array along a given axis.
<code>cumsum(a[, axis, dtype, out])</code>	Returns the cumulative sum of an array along a given axis.
<code>nancumprod(a[, axis, dtype, out])</code>	Returns the cumulative product of an array along a given axis treating Not a Numbers (NaNs) as one.
<code>nancumsum(a[, axis, dtype, out])</code>	Returns the cumulative sum of an array along a given axis treating Not a Numbers (NaNs) as zero.
<code>diff(a[, n, axis, prepend, append])</code>	Calculate the n-th discrete difference along the given axis.
<code>gradient(f, *varargs[, axis, edge_order])</code>	Return the gradient of an N-dimensional array.
<code>ediff1d(arr[, to_end, to_begin])</code>	Calculates the difference between consecutive elements of an array.
<code>cross(a, b[, axisa, axisb, axisc, axis])</code>	Returns the cross product of two vectors.
<code>trapz(y[, x, dx, axis])</code>	Integrate along the given axis using the composite trapezoidal rule.

## cupy.prod

`cupy.prod(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the product of an array along given axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to take product.
- **axis** (`int` or *sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.

### Returns

The result array.

### Return type

`cupy.ndarray`

See also:

`numpy.prod()`

## cupy.sum

`cupy.sum(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the sum of an array along given axes.

### Parameters

- **a** (`cupy.ndarray`) – Array to take sum.
- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

### Returns

The result array.

### Return type

*cupy.ndarray*

### See also:

`numpy.sum()`

## cupy.nanprod

`cupy.nanprod(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.

### Parameters

- **a** (`cupy.ndarray`) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

### Returns

The result array.

### Return type

*cupy.ndarray*

### See also:

`numpy.nanprod()`

**cupy.nansum**

`cupy.nansum(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.

**Parameters**

- **a** (`cupy.ndarray`) – Array to take sum.
- **axis** (`int` or *sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.

**Returns**

The result array.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.nansum()`

**cupy.cumprod**

`cupy.cumprod(a, axis=None, dtype=None, out=None)`

Returns the cumulative product of an array along a given axis.

**Parameters**

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative product is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

**Returns**

The result array.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.cumprod()`

## cupy.cumsum

`cupy.cumsum(a, axis=None, dtype=None, out=None)`

Returns the cumulative sum of an array along a given axis.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative sum is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

### Returns

The result array.

### Return type

`cupy.ndarray`

### See also:

`numpy.cumsum()`

## cupy.nancumprod

`cupy.nancumprod(a, axis=None, dtype=None, out=None)`

Returns the cumulative product of an array along a given axis treating Not a Numbers (NaNs) as one.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative product is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

### Returns

The result array.

### Return type

`cupy.ndarray`

### See also:

`numpy.nancumprod()`

## cupy.nancumsum

`cupy.nancumsum(a, axis=None, dtype=None, out=None)`

Returns the cumulative sum of an array along a given axis treating Not a Numbers (NaNs) as zero.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **axis** (`int`) – Axis along which the cumulative sum is taken. If it is not specified, the input is flattened.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.

### Returns

The result array.

### Return type

`cupy.ndarray`

### See also:

`numpy.nancumsum()`

## cupy.diff

`cupy.diff(a, n=1, axis=-1, prepend=None, append=None)`

Calculate the n-th discrete difference along the given axis.

### Parameters

- **a** (`cupy.ndarray`) – Input array.
- **n** (`int`) – The number of times values are differenced. If zero, the input is returned as-is.
- **axis** (`int`) – The axis along which the difference is taken, default is the last axis.
- **prepend** (`int`, `float`, `cupy.ndarray`) – Value to prepend to a.
- **append** (`int`, `float`, `cupy.ndarray`) – Value to append to a.

### Returns

The result array.

### Return type

`cupy.ndarray`

### See also:

`numpy.diff()`

## cupy.gradient

`cupy.gradient(f, *varargs, axis=None, edge_order=1)`

Return the gradient of an N-dimensional array.

The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries. The returned gradient hence has the same shape as the input array.

### Parameters

- **f** (`cupy.ndarray`) – An N-dimensional array containing samples of a scalar function.
- **varargs** (*list of scalar or array, optional*) – Spacing between f values. Default unitary spacing for all dimensions. Spacing can be specified using:
  1. single scalar to specify a sample distance for all dimensions.
  2. N scalars to specify a constant sample distance for each dimension. i.e. *dx, dy, dz, ...*
  3. N arrays to specify the coordinates of the values along each dimension of F. The length of the array must match the size of the corresponding dimension
  4. Any combination of N scalars/arrays with the meaning of 2. and 3.
- If *axis* is given, the number of varargs must equal the number of axes. Default: 1.
- **edge\_order** (*{1, 2}, optional*) – The gradient is calculated using N-th order accurate differences at the boundaries. Default: 1.
- **axis** (*None or int or tuple of ints, optional*) – The gradient is calculated only along the given axis or axes. The default (*axis = None*) is to calculate the gradient for all the axes of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

### Returns

A set of ndarrays (or a single ndarray if there is only one dimension) corresponding to the derivatives of f with respect to each dimension. Each derivative has the same shape as f.

### Return type

gradient (`cupy.ndarray` or list of `cupy.ndarray`)

See also:

`numpy.gradient()`

## cupy.ediff1d

`cupy.ediff1d(arr, to_end=None, to_begin=None)`

Calculates the difference between consecutive elements of an array.

### Parameters

- **arr** (`cupy.ndarray`) – Input array.
- **to\_end** (`cupy.ndarray`, *optional*) – Numbers to append at the end of the returned differences.
- **to\_begin** (`cupy.ndarray`, *optional*) – Numbers to prepend at the beginning of the returned differences.



**Returns**

New array consisting differences among succeeding elements.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.ediff1d()`

**cupy.cross**

`cupy.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)`

Returns the cross product of two vectors.

The cross product of **a** and **b** in  $R^3$  is a vector perpendicular to both **a** and **b**. If **a** and **b** are arrays of vectors, the vectors are defined by the last axis of **a** and **b** by default, and these axes can have dimensions 2 or 3. Where the dimension of either **a** or **b** is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

**Parameters**

- **a** (*cupy.ndarray*) – Components of the first vector(s).
- **b** (*cupy.ndarray*) – Components of the second vector(s).
- **axisa** (*int*, *optional*) – Axis of **a** that defines the vector(s). By default, the last axis.
- **axisb** (*int*, *optional*) – Axis of **b** that defines the vector(s). By default, the last axis.
- **axisc** (*int*, *optional*) – Axis of **c** containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.
- **axis** (*int*, *optional*) – If defined, the axis of **a**, **b** and **c** that defines the vector(s) and cross product(s). Overrides **axisa**, **axisb** and **axisc**.

**Returns**

Vector cross product(s).

**Return type**

*cupy.ndarray*

**See also:**

`numpy.cross()`

**cupy.trapz**

`cupy.trapz(y, x=None, dx=1.0, axis=-1)`

Integrate along the given axis using the composite trapezoidal rule. Integrate  $y(x)$  along the given axis.

**Parameters**

- **y** (*cupy.ndarray*) – Input array to integrate.
- **x** (*cupy.ndarray*) – Sample points over which to integrate. If **None** equal spacing  $dx$  is assumed.
- **dx** (*float*) – Spacing between sample points, used if **x** is **None**, default is 1.

- **axis** (*int*) – The axis along which the integral is taken, default is the last axis.

**Returns**

Definite integral as approximated by the trapezoidal rule.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.trapz()`

## Exponents and logarithms

<code>exp(x, /[, out, casting, dtype])</code>	Elementwise exponential function.
<code>expm1(x, /[, out, casting, dtype])</code>	Computes $\exp(x) - 1$ elementwise.
<code>exp2(x, /[, out, casting, dtype])</code>	Elementwise exponentiation with base 2.
<code>log(x, /[, out, casting, dtype])</code>	Elementwise natural logarithm function.
<code>log10(x, /[, out, casting, dtype])</code>	Elementwise common logarithm function.
<code>log2(x, /[, out, casting, dtype])</code>	Elementwise binary logarithm function.
<code>log1p(x, /[, out, casting, dtype])</code>	Computes $\log(1 + x)$ elementwise.
<code>logaddexp(x1, x2, /[, out, casting, dtype])</code>	Computes $\log(\exp(x1) + \exp(x2))$ elementwise.
<code>logaddexp2(x1, x2, /[, out, casting, dtype])</code>	Computes $\log_2(\exp_2(x1) + \exp_2(x2))$ elementwise.

## Other special functions

<code>i0(x, /[, out, casting, dtype])</code>	Modified Bessel function of the first kind, order 0.
<code>sinc(x, /[, out, casting, dtype])</code>	Elementwise sinc function.

### cupy.i0

`cupy.i0(x, /, out=None, *, casting='same_kind', dtype=None)`

Modified Bessel function of the first kind, order 0.

**See also:**

`numpy.i0()`

### cupy.sinc

`cupy.sinc(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise sinc function.

**See also:**

`numpy.sinc()`

## Floating point routines

<code>signbit(x, /[, out, casting, dtype])</code>	Tests elementwise if the sign bit is set (i.e.
<code>copysign(x1, x2, /[, out, casting, dtype])</code>	Returns the first argument with the sign bit of the second elementwise.
<code>frexp(x[, out1, out2], / [[, out, casting, ...]])</code>	Decomposes each element to mantissa and two's exponent.
<code>ldexp(x1, x2, /[, out, casting, dtype])</code>	Computes $x1 * 2^{**} x2$ elementwise.
<code>nextafter(x1, x2, /[, out, casting, dtype])</code>	Computes the nearest neighbor float values towards the second argument.

## Rational routines

<code>lcm(x1, x2, /[, out, casting, dtype])</code>	Computes lcm of $x1$ and $x2$ elementwise.
<code>gcd(x1, x2, /[, out, casting, dtype])</code>	Computes gcd of $x1$ and $x2$ elementwise.

## Arithmetic operations

<code>add(x1, x2, /[, out, casting, dtype])</code>	Adds two arrays elementwise.
<code>reciprocal(x, /[, out, casting, dtype])</code>	Computes $1 / x$ elementwise.
<code>positive(x, /[, out, casting, dtype])</code>	Takes numerical positive elementwise.
<code>negative(x, /[, out, casting, dtype])</code>	Takes numerical negative elementwise.
<code>multiply(x1, x2, /[, out, casting, dtype])</code>	Multiplies two arrays elementwise.
<code>divide</code>	<code>true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>power(x1, x2, /[, out, casting, dtype])</code>	Computes $x1^{**} x2$ elementwise.
<code>subtract(x1, x2, /[, out, casting, dtype])</code>	Subtracts arguments elementwise.
<code>true_divide(x1, x2, /[, out, casting, dtype])</code>	Elementwise true division (i.e.
<code>floor_divide(x1, x2, /[, out, casting, dtype])</code>	Elementwise floor division (i.e.
<code>float_power(x1, x2, /[, out, casting, dtype])</code>	First array elements raised to powers from second array, element-wise.
<code>fmod(x1, x2, /[, out, casting, dtype])</code>	Computes the remainder of C division elementwise.
<code>mod(x1, x2, /[, out, casting, dtype])</code>	Computes the remainder of Python division elementwise.
<code>modf(x[, out1, out2], / [[, out, casting, dtype]])</code>	Extracts the fractional and integral parts of an array elementwise.
<code>remainder</code>	<code>mod(x1, x2, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>divmod(x1, x2[, out1, out2], / [[, out, ...]])</code>	

## Handling complex numbers

<code>angle(z[, deg])</code>	Returns the angle of the complex argument.
<code>real(val)</code>	Returns the real part of the elements of the array.
<code>imag(val)</code>	Returns the imaginary part of the elements of the array.
<code>conj</code>	<code>conjugate(x, /, out=None, *, casting='same_kind', dtype=None)</code>
<code>conjugate(x, /[, out, casting, dtype])</code>	Returns the complex conjugate, element-wise.

### `cupy.angle`

`cupy.angle(z, deg=False)`

Returns the angle of the complex argument.

**See also:**

`numpy.angle()`

### `cupy.real`

`cupy.real(val)`

Returns the real part of the elements of the array.

**See also:**

`numpy.real()`

### `cupy.imag`

`cupy.imag(val)`

Returns the imaginary part of the elements of the array.

**See also:**

`numpy.imag()`

## Miscellaneous

<code>convolve(a, v[, mode])</code>	Returns the discrete, linear convolution of two one-dimensional sequences.
<code>clip(a, a_min, a_max[, out])</code>	Clips the values of an array to a given interval.
<code>sqrt(x, /[, out, casting, dtype])</code>	Elementwise square root function.
<code>cbrt(x, /[, out, casting, dtype])</code>	Elementwise cube root function.
<code>square(x, /[, out, casting, dtype])</code>	Elementwise square function.
<code>absolute(x, /[, out, casting, dtype])</code>	Elementwise absolute value function.
<code>fabs(x, /[, out, casting, dtype])</code>	Calculates absolute values element-wise.
<code>sign(x, /[, out, casting, dtype])</code>	Elementwise sign function.
<code>maximum(x1, x2, /[, out, casting, dtype])</code>	Takes the maximum of two arrays elementwise.
<code>minimum(x1, x2, /[, out, casting, dtype])</code>	Takes the minimum of two arrays elementwise.
<code>fmax(x1, x2, /[, out, casting, dtype])</code>	Takes the maximum of two arrays elementwise.
<code>fmin(x1, x2, /[, out, casting, dtype])</code>	Takes the minimum of two arrays elementwise.
<code>nan_to_num(x[, copy, nan, posinf, neginf])</code>	Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the <i>nan</i> , <i>posinf</i> and/or <i>neginf</i> key-words.
<code>heaviside(x1, x2, /[, out, casting, dtype])</code>	Compute the Heaviside step function.
<code>real_if_close(a[, tol])</code>	If input is complex with all imaginary parts close to zero, return real parts.
<code>interp(x, xp, fp[, left, right, period])</code>	One-dimensional linear interpolation.

## cupy.convolve

`cupy.convolve(a, v, mode='full')`

Returns the discrete, linear convolution of two one-dimensional sequences.

### Parameters

- **a** (`cupy.ndarray`) – first 1-dimensional input.
- **v** (`cupy.ndarray`) – second 1-dimensional input.
- **mode** (`str`, *optional*) – *valid*, *same*, *full*

### Returns

Discrete, linear convolution of *a* and *v*.

### Return type

`cupy.ndarray`

### See also:

`numpy.convolve()`

## cupy.clip

`cupy.clip(a, a_min, a_max, out=None)`

Clips the values of an array to a given interval.

This is equivalent to `maximum(minimum(a, a_max), a_min)`, while this function is more efficient.

### Parameters

- **a** (`cupy.ndarray`) – The source array.
- **a\_min** (*scalar*, `cupy.ndarray` or `None`) – The left side of the interval. When it is `None`, it is ignored.
- **a\_max** (*scalar*, `cupy.ndarray` or `None`) – The right side of the interval. When it is `None`, it is ignored.
- **out** (`cupy.ndarray`) – Output array.

### Returns

Clipped array.

### Return type

*cupy.ndarray*

See also:

`numpy.clip()`

### Notes

When *a\_min* is greater than *a\_max*, *clip* returns an array in which all values are equal to *a\_max*.

## cupy.nan\_to\_num

`cupy.nan_to_num(x, copy=True, nan=0.0, posinf=None, neginf=None)`

Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the *nan*, *posinf* and/or *neginf* keywords.

See also:

`numpy.nan_to_num()`

## cupy.real\_if\_close

`cupy.real_if_close(a, tol=100)`

If input is complex with all imaginary parts close to zero, return real parts. “Close to zero” is defined as *tol* \* (machine epsilon of the type for *a*).

<b>Warning:</b> This function may synchronize the device.
---

See also:

`numpy.real_if_close()`

## cupy.interp

`cupy.interp(x, xp, fp, left=None, right=None, period=None)`

One-dimensional linear interpolation.

### Parameters

- **x** (`cupy.ndarray`) – a 1D array of points on which the interpolation is performed.
- **xp** (`cupy.ndarray`) – a 1D array of points on which the function values (fp) are known.
- **fp** (`cupy.ndarray`) – a 1D array containing the function values at the the points xp.
- **left** (*float* or *complex*) – value to return if  $x < xp[0]$ . Default is `fp[0]`.
- **right** (*float* or *complex*) – value to return if  $x > xp[-1]$ . Default is `fp[-1]`.
- **period** (*None* or *float*) – a period for the x-coordinates. Parameters `left` and `right` are ignored if `period` is specified. Default is `None`.

### Returns

The interpolated values, same shape as `x`.

### Return type

*cupy.ndarray*

---

**Note:** This function may synchronize if `left` or `right` is not already on the device.

---

### See also:

`numpy.interp()`

## 5.3.12 Miscellaneous routines

---

**Hint:** NumPy API Reference: Miscellaneous routines

---

## Memory ranges

<code>byte_bounds(a)</code>	Returns pointers to the end-points of an array.
<code>shares_memory(a, b[, max_work])</code>	
<code>may_share_memory(a, b[, max_work])</code>	

---

## cupy.byte\_bounds

`cupy.byte_bounds(a)`

Returns pointers to the end-points of an array.

**Parameters**

**a** – ndarray

**Returns**

pointers to the end-points of an array

**Return type**

Tuple[int, int]

See also:

`numpy.byte_bounds()`

## cupy.shares\_memory

`cupy.shares_memory(a, b, max_work=None)`

## cupy.may\_share\_memory

`cupy.may_share_memory(a, b, max_work=None)`

## Utility

---

<code>show_config(*[, _full])</code>	Prints the current runtime configuration to standard output.
--------------------------------------	--

---

## cupy.show\_config

`cupy.show_config(*, _full=False)`

Prints the current runtime configuration to standard output.

## Matlab-like Functions

---

<code>who([vardict])</code>	Print the CuPy arrays in the given dictionary.
-----------------------------	--

---



## cupy.who

`cupy.who(vardict=None)`

Print the CuPy arrays in the given dictionary.

Prints out the name, shape, bytes and type of all of the ndarrays present in *vardict*.

If there is no dictionary passed in or *vardict* is None then returns CuPy arrays in the `globals()` dictionary (all CuPy arrays in the namespace).

### Parameters

**vardict** – (None or dict) A dictionary possibly containing ndarrays. Default is `globals()` if *None* specified

### Example

```

>>> a = cupy.arange(10)
>>> b = cupy.ones(20)
>>> cupy.who()

```

Name	Shape	Bytes	Type
a	10	80	int64
b	20	160	float64

```

Upper bound on total bytes = 240
>>> d = {'x': cupy.arange(2.0),
... 'y': cupy.arange(3.0), 'txt': 'Some str',
... 'idx': 5}
>>> cupy.who(d)

```

Name	Shape	Bytes	Type
x	2	16	float64
y	3	24	float64

```

Upper bound on total bytes = 40

```

## 5.3.13 Padding arrays

**Hint:** [NumPy API Reference: Padding arrays](#)

`pad(array, pad_width[, mode])`

Pads an array with specified widths and values.

## cupy.pad

`cupy.pad(array, pad_width, mode='constant', **kwargs)`

Pads an array with specified widths and values.

### Parameters

- **array** (`cupy.ndarray`) – The array to pad.
- **pad\_width** (*sequence, array\_like or int*) – Number of values padded to the edges of each axis. ((before\_1, after\_1), ... (before\_N, after\_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes. You cannot specify `cupy.ndarray`.
- **mode** (*str or function, optional*) – One of the following string values or a user supplied function
  - 'constant' (default)**  
Pads with a constant value.
  - 'edge'**  
Pads with the edge values of array.
  - 'linear\_ramp'**  
Pads with the linear ramp between end\_value and the array edge value.
  - 'maximum'**  
Pads with the maximum value of all or part of the vector along each axis.
  - 'mean'**  
Pads with the mean value of all or part of the vector along each axis.
  - 'median'**  
Pads with the median value of all or part of the vector along each axis. (Not Implemented)
  - 'minimum'**  
Pads with the minimum value of all or part of the vector along each axis.
  - 'reflect'**  
Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
  - 'symmetric'**  
Pads with the reflection of the vector mirrored along the edge of the array.
  - 'wrap'**  
Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.
  - 'empty'**  
Pads with undefined values.
  - <function>**  
Padding function, see Notes.
- **stat\_length** (*sequence or int, optional*) – Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value. ((before\_1, after\_1), ... (before\_N, after\_N)) unique statistic lengths for each axis. ((before, after),) yields same before and after statistic lengths for each axis. (stat\_length,) or int is a shortcut for before = after = statistic length for all axes. Default is None, to use the entire axis. You cannot specify `cupy.ndarray`.

- **constant\_values** (*sequence or scalar, optional*) – Used in ‘constant’. The values to set the padded values for each axis. ((before\_1, after\_1), ... (before\_N, after\_N)) unique pad constants for each axis. ((before, after),) yields same before and after constants for each axis. (constant,) or constant is a shortcut for before = after = constant for all axes. Default is 0. You cannot specify `cupy.ndarray`.
- **end\_values** (*sequence or scalar, optional*) – Used in ‘linear\_ramp’. The values used for the ending value of the linear\_ramp and that will form the edge of the padded array. ((before\_1, after\_1), ... (before\_N, after\_N)) unique end values for each axis. ((before, after),) yields same before and after end values for each axis. (constant,) or constant is a shortcut for before = after = constant for all axes. Default is 0. You cannot specify `cupy.ndarray`.
- **reflect\_type** ({‘even’, ‘odd’}, *optional*) – Used in ‘reflect’, and ‘symmetric’. The ‘even’ style is the default with an unaltered reflection around the edge value. For the ‘odd’ style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

**Returns**

Padded array with shape extended by `pad_width`.

**Return type**

*cupy.ndarray*

---

**Note:** For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should modify a rank 1 array in-place. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, kwargs)
```

where

**vector** (`cupy.ndarray`)

A rank 1 array already padded with zeros. Padded values are `vector[:iaxis_pad_width[0]]` and `vector[-iaxis_pad_width[1]:]`.

**iaxis\_pad\_width** (`tuple`)

A 2-tuple of ints, `iaxis_pad_width[0]` represents the number of values padded at the beginning of vector where `iaxis_pad_width[1]` represents the number of values padded at the end of vector.

**iaxis** (`int`)

The axis currently being calculated.

**kwargs** (`dict`)

Any keyword arguments the function requires.

---

## Examples

```
>>> a = cupy.array([1, 2, 3, 4, 5])
>>> cupy.pad(a, (2, 3), 'constant', constant_values=(4, 6))
array([4, 4, 1, ..., 6, 6, 6])
```

```
>>> cupy.pad(a, (2, 3), 'edge')
array([1, 1, 1, ..., 5, 5, 5])
```

```
>>> cupy.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> cupy.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> cupy.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = cupy.array([[1, 2], [3, 4]])
>>> cupy.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 3, 4, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = cupy.array([1, 2, 3, 4, 5])
>>> cupy.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> cupy.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
>>> cupy.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> cupy.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> cupy.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = cupy.arange(6)
>>> a = a.reshape((2, 3))
```

(continues on next page)

(continued from previous page)

```

>>> cupy.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> cupy.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100,  0,  1,  2, 100, 100],
       [100, 100,  3,  4,  5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])

```

### 5.3.14 Polynomials

**Hint:** [NumPy API Reference: Polynomials](#)

#### Power Series (`cupy.polynomial.polynomial`)

**Hint:** [NumPy API Reference: Power Series \(numpy.polynomial.polynomial\)](#)

#### Misc Functions

<code>polyvander(x, deg)</code>	Computes the Vandermonde matrix of given degree.
<code>polycompanion(c)</code>	Computes the companion matrix of c.
<code>polyval(x, c[, tensor])</code>	Evaluate a polynomial at points x.
<code>polyvalfromroots(x, r[, tensor])</code>	Evaluate a polynomial specified by its roots at points x.

#### `cupy.polynomial.polynomial.polyvander`

`cupy.polynomial.polynomial.polyvander(x, deg)`  
 Computes the Vandermonde matrix of given degree.

##### Parameters

- **x** (`cupy.ndarray`) – array of points
- **deg** (`int`) – degree of the resulting matrix.

##### Returns

The Vandermonde matrix

##### Return type

`cupy.ndarray`

See also:

`numpy.polynomial.polynomial.polyvander()`

### `cupy.polynomial.polynomial.polycompanion`

`cupy.polynomial.polynomial.polycompanion(c)`

Computes the companion matrix of *c*.

**Parameters**

**c** (`cupy.ndarray`) – 1-D array of polynomial coefficients ordered from low to high degree.

**Returns**

Companion matrix of dimensions (deg, deg).

**Return type**

`cupy.ndarray`

See also:

`numpy.polynomial.polynomial.polycompanion()`

### `cupy.polynomial.polynomial.polyval`

`cupy.polynomial.polynomial.polyval(x, c, tensor=True)`

Evaluate a polynomial at points *x*.

If *c* is of length  $n + 1$ , this function returns the value

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then  $p(x)$  will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be `c.shape[1:] + x.shape`. If *tensor* is false the shape will be `c.shape[1:]`. Note that scalars have shape `(,)`.

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

- **x** (*array\_like, compatible object*) – If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with themselves and with the elements of *c*.
- **c** (*array\_like*) – Array of coefficients ordered so that the coefficients for terms of degree *n* are contained in `c[n]`. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.
- **tensor** (*boolean, optional*) – If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

**Returns**

**values** – The shape of the returned array is described above.

**Return type**

*ndarray*, compatible object

**See also:**

`numpy.polynomial.polynomial.polyval`

**Notes**

The evaluation uses Horner's method.

**`cupy.polynomial.polynomial.polyvalfromroots`**

`cupy.polynomial.polynomial.polyvalfromroots(x, r, tensor=True)`

Evaluate a polynomial specified by its roots at points  $x$ .

If  $r$  is of length  $N$ , this function returns the value

$$p(x) = \prod_{n=1}^N (x - r_n)$$

The parameter  $x$  is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either  $x$  or its elements must support multiplication and addition both with themselves and with the elements of  $r$ .

If  $r$  is a 1-D array, then  $p(x)$  will have the same shape as  $x$ . If  $r$  is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is `True` the shape will be  $r.\text{shape}[1:] + x.\text{shape}$ ; that is, each polynomial is evaluated at every value of  $x$ . If *tensor* is `False`, the shape will be  $r.\text{shape}[1:]$ ; that is, each polynomial is evaluated only for the corresponding broadcast value of  $x$ . Note that scalars have shape  $(,)$ .

**Parameters**

- **$x$  (array\_like, compatible object)** – If  $x$  is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case,  $x$  or its elements must support addition and multiplication with themselves and with the elements of  $r$ .
- **$r$  (array\_like)** – Array of roots. If  $r$  is multidimensional the first index is the root index, while the remaining indices enumerate multiple polynomials. For instance, in the two dimensional case the roots of each polynomial may be thought of as stored in the columns of  $r$ .
- ***tensor* (boolean, optional)** – If `True`, the shape of the roots array is extended with ones on the right, one for each dimension of  $x$ . Scalars have dimension 0 for this action. The result is that every column of coefficients in  $r$  is evaluated for every element of  $x$ . If `False`,  $x$  is broadcast over the columns of  $r$  for the evaluation. This keyword is useful when  $r$  is multidimensional. The default value is `True`.

**Returns**

**values** – The shape of the returned array is described above.

**Return type**

*ndarray*, compatible object

**See also:**

`numpy.polynomial.polynomial.polyvalfromroots`

## Polyutils

---

**Hint:** NumPy API Reference: Polyutils

---

## Functions

---

<code>as_series(alist[, trim])</code>	Returns argument as a list of 1-d arrays.
<code>trimseq(seq)</code>	Removes small polynomial series coefficients.
<code>trimcoef(c[, tol])</code>	Removes small trailing coefficients from a polynomial.

---

### `cupy.polynomial.polyutils.as_series`

`cupy.polynomial.polyutils.as_series(alist, trim=True)`

Returns argument as a list of 1-d arrays.

**Parameters**

- **alist** (`cupy.ndarray` or *list of `cupy.ndarray`*) – 1-D or 2-D input array.
- **trim** (*bool, optional*) – trim trailing zeros.

**Returns**

list of 1-D arrays.

**Return type**

list of *cupy.ndarray*

**See also:**

`numpy.polynomial.polyutils.as_series()`

### `cupy.polynomial.polyutils.trimseq`

`cupy.polynomial.polyutils.trimseq(seq)`

Removes small polynomial series coefficients.

**Parameters**

**seq** (`cupy.ndarray`) – input array.

**Returns**

input array with trailing zeros removed. If the resulting output is empty, it returns the first element.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.polynomial.polyutils.trimseq()`



**cupy.polynomial.polyutils.trimcoef**

`cupy.polynomial.polyutils.trimcoef(c, tol=0)`

Removes small trailing coefficients from a polynomial.

**Parameters**

- **c** (`cupy.ndarray`) – 1d array of coefficients from lowest to highest order.
- **tol** (*number, optional*) – trailing coefficients whose absolute value are less than or equal to tol are trimmed.

**Returns**

trimmed 1d array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.polynomial.polyutils.trimcoef()`

**Poly1d**


---

**Hint:** NumPy API Reference: Poly1d

---

**Basics**

<code>poly1d(c_or_r[, r, variable])</code>	A one-dimensional polynomial class.
<code>cupy.poly(seq_of_zeros)</code>	Computes the coefficients of a polynomial with the given roots sequence.
<code>polyval(p, x)</code>	Evaluates a polynomial at specific values.
<code>roots(p)</code>	Computes the roots of a polynomial with given coefficients.

**cupy.poly1d**

**class** `cupy.poly1d(c_or_r, r=False, variable=None)`

A one-dimensional polynomial class.

---

**Note:** This is a counterpart of an old polynomial class in NumPy. Note that the new NumPy polynomial API (`numpy.polynomial.polynomial`) has different convention, e.g. order of coefficients is reversed.

---

**Parameters**

- **c\_or\_r** (*array\_like*) – The polynomial’s coefficients in decreasing powers
- **r** (*bool, optional*) – If True, `c_or_r` specifies the polynomial’s roots; the default is False.
- **variable** (*str, optional*) – Changes the variable used when printing the polynomial from `x` to `variable`

See also:

`numpy.poly1d`

## Methods

**`__call__`**(*\*args, \*\*kwargs*)

Call self as a function.

**`__getitem__`**(*key, /*)

Return self[key].

**`__setitem__`**(*key, value, /*)

Set self[key] to value.

**`__len__`**()

Return len(self).

**`__iter__`**()

Implement iter(self).

**`deriv`**(*self, m=1*)

**`get`**(*self, stream=None*)

Returns a copy of poly1d object on host memory.

### Parameters

**`stream`** (`cupy.cuda.Stream`) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.

### Returns

Copy of poly1d object on host memory.

### Return type

`numpy.poly1d`

**`integ`**(*self, m=1, k=0*)

**`set`**(*self, polyin, stream=None*)

Copies a poly1d object on the host memory to `cupy.poly1d`.

### Parameters

- **`polyin`** (`numpy.poly1d`) – The source object on the host memory.
- **`stream`** (`cupy.cuda.Stream`) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous. The default uses CUDA stream object of the current context.

**`__eq__`**(*value, /*)

Return self==value.

**`__ne__`**(*value, /*)

Return self!=value.

**`__lt__`**(*value, /*)

Return self<value.

```
__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

### Attributes

```
c
coef
coefficients
coeffs
o
order
r
roots
variable
```

### cupy.poly

`cupy.poly(seq_of_zeros)`

Computes the coefficients of a polynomial with the given roots sequence.

#### Parameters

**seq\_of\_zeros** ([cupy.ndarray](#)) – a sequence of polynomial roots.

#### Returns

polynomial coefficients from highest to lowest degree.

#### Return type

[cupy.ndarray](#)

**Warning:** This function doesn't support general 2d square arrays currently. Only complex Hermitian and real symmetric 2d arrays are allowed.

#### See also:

[numpy.poly\(\)](#)

## cupy.polyval

`cupy.polyval(p, x)`

Evaluates a polynomial at specific values.

### Parameters

- **p** (`cupy.ndarray` or `cupy.poly1d`) – input polynomial.
- **x** (`scalar`, `cupy.ndarray`) – values at which the polynomial
- **evaluated.** (*is*) –

### Returns

polynomial evaluated at x.

### Return type

*cupy.ndarray* or *cupy.poly1d*

**Warning:** This function doesn't currently support `poly1d` values to evaluate.

See also:

`numpy.polyval()`

## cupy.roots

`cupy.roots(p)`

Computes the roots of a polynomial with given coefficients.

### Parameters

**p** (`cupy.ndarray` or `cupy.poly1d`) – polynomial coefficients.

### Returns

polynomial roots.

### Return type

*cupy.ndarray*

**Warning:** This function doesn't support currently polynomial coefficients whose companion matrices are general 2d square arrays. Only those with complex Hermitian or real symmetric 2d arrays are allowed.

The current *cupy.roots* doesn't guarantee the order of results.

See also:

`numpy.roots()`

## Fitting

<code>polyfit(x, y, deg[, rcond, full, w, cov])</code>	Returns the least squares fit of polynomial of degree <code>deg</code> to the data <code>y</code> sampled at <code>x</code> .
--	---

### cupy.polyfit

`cupy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)`

Returns the least squares fit of polynomial of degree `deg` to the data `y` sampled at `x`.

#### Parameters

- **x** (`cupy.ndarray`) – x-coordinates of the sample points of shape (M,).
- **y** (`cupy.ndarray`) – y-coordinates of the sample points of shape (M,) or (M, K).
- **deg** (`int`) – degree of the fitting polynomial.
- **rcond** (`float`, *optional*) – relative condition number of the fit. The default value is `len(x) * eps`.
- **full** (`bool`, *optional*) – indicator of the return value nature. When `False` (default), only the coefficients are returned. When `True`, diagnostic information is also returned.
- **w** (`cupy.ndarray`, *optional*) – weights applied to the y-coordinates of the sample points of shape (M,).
- **cov** (`bool` or `str`, *optional*) – if given, returns the coefficients along with the covariance matrix.

#### Returns

**p** (`cupy.ndarray` of shape `(deg + 1,)` or `(deg + 1, K)`):

Polynomial coefficients from highest to lowest degree

**residuals, rank, singular\_values, rcond** (`cupy.ndarray`, `int`, `cupy.ndarray`, `float`):

Present only if `full=True`. Sum of squared residuals of the least-squares fit, rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of `rcond`.

**V** (`cupy.ndarray` of shape `(M, M)` or `(M, M, K)`):

Present only if `full=False` and `cov=True`. The covariance matrix of the polynomial coefficient estimates.

#### Return type

`cupy.ndarray` or `tuple`

**Warning:** `numpy.RankWarning`: The rank of the coefficient matrix in the least-squares fit is deficient. It is raised if `full=False`.

See also:

`numpy.polyfit()`

## Arithmetic

<code>polyadd(a1, a2)</code>	Computes the sum of two polynomials.
<code>polysub(a1, a2)</code>	Computes the difference of two polynomials.
<code>polymul(a1, a2)</code>	Computes the product of two polynomials.

### `cupy.polyadd`

`cupy.polyadd(a1, a2)`

Computes the sum of two polynomials.

**Parameters**

- **a1** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – first input polynomial.
- **a2** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – second input polynomial.

**Returns**

The sum of the inputs.

**Return type**

`cupy.ndarray` or `cupy.poly1d`

See also:

`numpy.polyadd()`

### `cupy.polysub`

`cupy.polysub(a1, a2)`

Computes the difference of two polynomials.

**Parameters**

- **a1** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – first input polynomial.
- **a2** (*scalar*, `cupy.ndarray` or `cupy.poly1d`) – second input polynomial.

**Returns**

The difference of the inputs.

**Return type**

`cupy.ndarray` or `cupy.poly1d`

See also:

`numpy.polysub()`

## cupy.polymul

`cupy.polymul(a1, a2)`

Computes the product of two polynomials.

### Parameters

- **a1** (*scalar, cupy.ndarray or cupy.poly1d*) – first input polynomial.
- **a2** (*scalar, cupy.ndarray or cupy.poly1d*) – second input polynomial.

### Returns

The product of the inputs.

### Return type

*cupy.ndarray or cupy.poly1d*

See also:

`numpy.polymul()`

## 5.3.15 Random sampling (cupy.random)

Differences between `cupy.random` and `numpy.random`:

- Most functions under `cupy.random` support the `dtype` option, which do not exist in the corresponding NumPy APIs. This option enables generation of float32 values directly without any space overhead.
- `cupy.random.default_rng()` uses XORWOW bit generator by default.
- Random states cannot be serialized. See the description below for details.
- CuPy does not guarantee that the same number generator is used across major versions. This means that numbers generated by `cupy.random` by new major version may not be the same as the previous one, even if the same seed and distribution are used.

### New Random Generator API

---

**Hint:** NumPy API Reference: Random sampling (`numpy.random`)

---

### Random Generator

---

**Hint:** NumPy API Reference: Random Generator

---

<code>default_rng([seed])</code>	Construct a new Generator with the default BitGenerator (XORWOW).
<code>Generator(bit_generator)</code>	Container for the BitGenerators.

## cupy.random.default\_rng

`cupy.random.default_rng(seed=None)`

Construct a new Generator with the default BitGenerator (XORWOW).

### Parameters

**seed** (*None*, *int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *cupy.random.BitGenerator*, *cupy.random.Generator*, *optional*) – A seed to initialize the *cupy.random.BitGenerator*. If an *int* or *array\_like[ints]* or *None* is passed, then it will be passed to *numpy.random.SeedSequence* to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance. Additionally, when passed *:class: 'BitGenerator'*, it will be wrapped by *Generator*. If passed a *Generator*, it will be returned unaltered.

### Returns

The initialized generator object.

### Return type

*Generator*

## cupy.random.Generator

**class** `cupy.random.Generator(bit_generator)`

Container for the BitGenerators.

*Generator* exposes a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument *size* that defaults to *None*. If *size* is *None*, then a single value is generated and returned. If *size* is an integer, then a 1-D array filled with generated values is returned. If *size* is a tuple, then an array with that shape is filled and returned. The function `numpy.random.default_rng()` will instantiate a *Generator* with numpy's default *BitGenerator*. **No Compatibility Guarantee** *Generator* does not provide a version compatibility guarantee. In particular, as better algorithms evolve the bit stream may change.

### Parameters

**bit\_generator** – (*cupy.random.BitGenerator*): BitGenerator to use as the core generator.

## Methods

**beta**(*self*, *a*, *b*, *size=None*, *dtype=numpy.float64*)

Beta distribution.

Returns an array of samples drawn from the beta distribution. Its probability density function is defined as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}.$$

### Parameters

- **a** (*float*) – Parameter of the beta distribution  $\alpha$ .
- **b** (*float*) – Parameter of the beta distribution  $\beta$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.



**Returns**

Samples drawn from the beta distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.Generator.beta()`

**binomial**(*self*, *n*, *p*, *size=None*)

Binomial distribution.

Returns an array of samples drawn from the binomial distribution. Its probability mass function is defined as

$$f(x) = \binom{n}{x} p^x (1-p)^{(n-x)}.$$

**Parameters**

- **n** (*int* or *cupy.ndarray of ints*) – Parameter of the distribution,  $\geq 0$ . Floats are also accepted, but they will be truncated to integers.
- **p** (*float* or *cupy.ndarray of floats*) – Parameter of the distribution,  $\geq 0$  and  $\leq 1$ .
- **size** (*int* or *tuple of ints, optional*) – The shape of the output array. If *None* (default), a single value is returned if *n* and *p* are both scalars. Otherwise, `cupy.broadcast(n, p).size` samples are drawn.

**Returns**

Samples drawn from the binomial distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.Generator.binomial()`

**chisquare**(*self*, *df*, *size=None*)

Chi-square distribution.

Returns an array of samples drawn from the chi-square distribution. Its probability density function is defined as

$$f(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}.$$

**Parameters**

- **df** (*float* or *array\_like of floats*) – Degree of freedom *k*.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.

**Returns**

Samples drawn from the chi-square distribution.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.Generator.chisquare()`

**dirichlet**(*self*, *alpha*, *size=None*)

Dirichlet distribution.

Returns an array of samples drawn from the dirichlet distribution. Its probability density function is defined as

$$f(x) = \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K x_i^{\alpha_i - 1}.$$

#### Parameters

- **alpha** (*array*) – Parameters of the dirichlet distribution  $\alpha$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, array of alpha. shape is generated

#### Returns

Samples drawn from the dirichlet distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.Generator.dirichlet()`

**exponential**(*self*, *scale=1.0*, *size=None*)

Exponential distribution.

Returns an array of samples drawn from the exponential distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right).$$

#### Parameters

- **scale** (*float or array\_like of floats*) – The scale parameter  $\beta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.

#### Returns

Samples drawn from the exponential distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.Generator.exponential()`

**f**(*self*, *dfnum*, *dfden*, *size=None*)

F distribution.

Returns an array of samples drawn from the f distribution. Its probability density function is defined as

$$f(x) = \frac{1}{B\left(\frac{d_1}{2}, \frac{d_2}{2}\right)} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2}x\right)^{-\frac{d_1+d_2}{2}}.$$

**Parameters**

- **dfnum** (*float or array\_like of floats*) – Degrees of freedom in numerator,  $d_1$ .
- **dfden** (*float or array\_like of floats*) – Degrees of freedom in denominator,  $d_2$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.

**Returns**

Samples drawn from the f distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.Generator.f()`

**gamma**(*self, shape, scale=1.0, size=None*)

Gamma distribution.

Returns an array of samples drawn from the gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}.$$

**Parameters**

- **shape** (*float or array\_like of float*) – The shape of the gamma distribution. Must be non-negative.
- **scale** (*float or array\_like of float*) – The scale of the gamma distribution. Must be non-negative. Default equals to 1
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.

**See also:**

- `numpy.random.Generator.gamma()`

**geometric**(*self, p, size=None*)

Geometric distribution.

Returns an array of samples drawn from the geometric distribution. Its probability mass function is defined as

$$f(x) = p(1 - p)^{k-1}.$$

**Parameters**

- **p** (*float or cupy.ndarray of floats*) – Success probability of the geometric distribution.
- **size** (*int or tuple of ints, optional*) – The shape of the output array. If `None` (default), a single value is returned if `p` is scalar. Otherwise, `p.size` samples are drawn.

**Returns**

Samples drawn from the geometric distribution.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.Generator.geometric()`

**hypergeometric**(*self*, *ngood*, *nbad*, *nsample*, *size=None*)

Hypergeometric distribution.

Returns an array of samples drawn from the hypergeometric distribution. Its probability mass function is defined as

$$f(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}}.$$

#### Parameters

- **ngood** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $n$ .
- **nbad** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $m$ .
- **nsample** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $N$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.

#### Returns

Samples drawn from the hypergeometric distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.Generator.hypergeometric()`

**integers**(*self*, *low*, *high=None*, *size=None*, *dtype=numpy.int64*, *endpoint=False*)

Returns a scalar or an array of integer values over an interval.

Each element of returned values are independently sampled from uniform distribution over the [*low*, *high*) or [*low*, *high*] intervals.

#### Parameters

- **low** (*int*) – If *high* is not *None*, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to 0.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None* or *int* or *tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.
- **endpoint** (*bool*) – If *True*, sample from [*low*, *high*]. Defaults to *False*

#### Returns

If *size* is *None*, it is single integer sampled. If *size* is integer, it is the 1D-array of length *size* element. Otherwise, it is the array whose shape specified by *size*.

#### Return type

*int* or *cupy.ndarray* of *ints*

See also:

- `numpy.random.Generator.integers()`

**logseries**(*self*, *p*, *size=None*)

Log series distribution.

Returns an array of samples drawn from the log series distribution. Its probability mass function is defined as

$$f(x) = \frac{-p^x}{x \ln(1 - p)}.$$

#### Parameters

- **p** (*float* or *cupy.ndarray* of *floats*) – Parameter of the log series distribution. Must be in the range (0, 1).
- **size** (*int* or *tuple* of *ints*, *optional*) – The shape of the output array. If *None* (default), a single value is returned if *p* is scalar. Otherwise, *p.size* samples are drawn.

#### Returns

Samples drawn from the log series distribution.

#### Return type

*cupy.ndarray*

#### See also:

`numpy.random.Generator.logseries()`

**poisson**(*self*, *lam=1.0*, *size=None*)

Poisson distribution.

Returns an array of samples drawn from the poisson distribution. Its probability mass function is defined as

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}.$$

#### Parameters

- **lam** (*float* or *array\_like* of *floats*) – Parameter of the poisson distribution  $\lambda$ .
- **size** (*int* or *tuple* of *ints*) – The shape of the array. If *None*, this function generate an array whose shape is *lam.shape*.

#### Returns

Samples drawn from the poisson distribution.

#### Return type

*cupy.ndarray*

#### See also:

`numpy.random.Generator.poisson()`

**power**(*self*, *a*, *size=None*)

Power distribution.

Returns an array of samples drawn from the power distribution. Its probability density function is defined as

$$f(x) = ax^{a-1}.$$

#### Parameters

- **a** (*float* or *array\_like of floats*) – Parameter of the power distribution  $a$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.

**Returns**

Samples drawn from the power distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.Generator.power()`

**random**(*self*, *size=None*, *dtype=numpy.float64*, *out=None*)

Return random floats in the half-open interval  $[0.0, 1.0)$ .

Results are from the “continuous uniform” distribution over the stated interval. To sample  $Unif[a, b)$ ,  $b > a$  multiply the output of *random* by  $(b-a)$  and add  $a$ :

$$(b - a) * \text{random}() + a$$
**Parameters**

- **size** (*None* or *int* or *tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.
- **out** (*cupy.ndarray*, *optional*) – If specified, values will be written to this array

**Returns**

Samples uniformly drawn from the  $[0, 1)$  interval

**Return type**

*cupy.ndarray*

**See also:**

- `numpy.random.Generator.random()`

**standard\_exponential**(*self*, *size=None*, *dtype=numpy.float64*, *method='inv'*, *out=None*)

Standard exponential distribution.

Returns an array of samples drawn from the standard exponential distribution. Its probability density function is defined as

$$f(x) = e^{-x}.$$

**Parameters**

- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.
- **method** (*str*) – Method to sample. Currently only 'inv', sampling from the default inverse CDF, is supported.

- **out** (`cupy.ndarray`, *optional*) – If specified, values will be written to this array

**Returns**

Samples drawn from the standard exponential distribution.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.random.Generator.standard_exponential()`

**standard\_gamma**(*self*, *shape*, *size=None*, *dtype=numpy.float64*, *out=None*)

Standard gamma distribution.

Returns an array of samples drawn from the standard gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)} x^{k-1} e^{-x}.$$

**Parameters**

- **shape** (*float or array\_like of float*) – The shape of the gamma distribution. Must be non-negative.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`, *optional*) – If specified, values will be written to this array

**See also:**

- `numpy.random.Generator.standard_gamma()`

**standard\_normal**(*self*, *size=None*, *dtype=numpy.float64*, *out=None*)

Standard normal distribution.

Returns an array of samples drawn from the standard normal distribution.

**Parameters**

- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`, *optional*) – If specified, values will be written to this array

**Returns**

Samples drawn from the standard normal distribution.

**Return type**

`cupy.ndarray`

**See also:**

- `numpy.random.Generator.standard_normal()`

**uniform**(*self*, *low*=0.0, *high*=1.0, *size*=None, *dtype*=numpy.float64)

Draw samples from a uniform distribution. Samples are uniformly distributed over the half-open interval [*low*, *high*) (includes *low*, but excludes *high*). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

#### Parameters

- **low** (*float* or *array\_like of floats*, *optional*) – Lower boundary of the output interval. All values generated will be greater than or equal to *low*. The default value is 0.
- **high** (*float* or *array\_like of floats*) – Upper boundary of the output interval. All values generated will be less than *high*. The high limit may be included in the returned array of floats due to floating-point rounding in the equation  $\text{low} + (\text{high} - \text{low}) * \text{random}()$ . *high* - *low* must be non-negative. The default value is 1.0.
- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. If *size* is *None* (default), a single value is returned if *low* and *high* are both scalars. Otherwise, `cupy.broadcast(low, high).size` samples are drawn.

#### Returns

**out** – Drawn samples from the parameterized uniform distribution.

#### Return type

*ndarray* or scalar

#### See also:

- meth:*numpy.random.Generator.uniform*
- meth:*integers*: Discrete uniform distribution, yielding integers.
- meth:*random*: Floats uniformly distributed over [0, 1).

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.



## Bit Generators

---

**Hint:** NumPy API Reference: Bit Generators

---

<i>BitGenerator</i> ([seed])	Generic BitGenerator.
------------------------------	-----------------------

---

### cupy.random.BitGenerator

**class** cupy.random.**BitGenerator**(seed=None)

Generic BitGenerator.

Base Class for generic BitGenerators, which provide a stream of random bits based on different algorithms. Must be overridden.

#### Parameters

**seed** (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence`` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.

#### Methods

**random\_raw**(self, size=None, output=True)

**\_\_eq\_\_**(value, /)

Return self==value.

**\_\_ne\_\_**(value, /)

Return self!=value.

**\_\_lt\_\_**(value, /)

Return self<value.

**\_\_le\_\_**(value, /)

Return self<=value.

**\_\_gt\_\_**(value, /)

Return self>value.

**\_\_ge\_\_**(value, /)

Return self>=value.

CuPy provides the following bit generator implementations:

<i>XORWOW</i> ([seed, size])	BitGenerator that uses cuRAND XORWOW device generator.
<i>MRG32k3a</i> ([seed, size])	BitGenerator that uses cuRAND MRG32k3a device generator.
<i>Philox4x3210</i> ([seed, size])	BitGenerator that uses cuRAND Philox4x3210 device generator.

---

## cupy.random.XORWOW

**class** cupy.random.XORWOW(*seed=None*, \*, *size=-1*)

BitGenerator that uses cuRAND XORWOW device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*None*, *int*, *array\_like[ints]*, *numpy.random.SeedSequence*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

### Methods

**random\_raw**(*self*, *size=None*, *output=True*)

Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. Default is *None*, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

### Returns

Drawn samples.

### Return type

*cupy.ndarray*

---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

**state**(*self*)

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

`__gt__(value, /)`

Return self > value.

`__ge__(value, /)`

Return self >= value.

## Attributes

`generator = 0`

## `cupy.random.MRG32k3a`

`class cupy.random.MRG32k3a(seed=None, *, size=-1)`

BitGenerator that uses cuRAND MRG32k3a device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence`` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

## Methods

`random_raw(self, size=None, output=True)`

Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. Default is *None*, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

### Returns

Drawn samples.

### Return type

*cupy.ndarray*

---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

`state(self)`

```
__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

### Attributes

**generator** = 1

## cupy.random.Philox4x3210

**class** cupy.random.Philox4x3210(*seed=None, \*, size=-1*)

BitGenerator that uses cuRAND Philox4x3210 device generator.

This generator allocates the state using the cuRAND device API.

### Parameters

- **seed** (*int*, *array\_like[ints]*, *numpy.random.SeedSequence*, *optional*) – A seed to initialize the *BitGenerator*. If *None*, then fresh, unpredictable entropy will be pulled from the OS. If an *int* or *array\_like[ints]* is passed, then it will be passed to `~numpy.random.SeedSequence`` to derive the initial *BitGenerator* state. One may also pass in a *SeedSequence* instance.
- **size** (*int*) – Maximum number of samples that can be generated at once. defaults to 1000 \* 256.

### Methods

**random\_raw**(*self, size=None, output=True*)

Return randoms as generated by the underlying BitGenerator.

### Parameters

- **size** (*int* or *tuple of ints*, *optional*) – Output shape. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. Default is *None*, in which case a single value is returned.
- **output** (*bool*, *optional*) – Output values. Used for performance testing since the generated values are not returned.

### Returns

Drawn samples.

**Return type***cupy.ndarray*


---

**Note:** This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG. See the class docstring for the number of bits returned.

---

**state**(*self*)**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self&lt;value.

**\_\_le\_\_**(*value*, /)

Return self&lt;=value.

**\_\_gt\_\_**(*value*, /)

Return self&gt;value.

**\_\_ge\_\_**(*value*, /)

Return self&gt;=value.

**Attributes****generator** = 2**Legacy Random Generation****Hint:**

- [NumPy API Reference: Legacy Random Generation](#)
- [NumPy 1.16 Reference](#)

*RandomState*([seed, method])

Portable container of a pseudo-random number generator.

## cupy.random.RandomState

**class** `cupy.random.RandomState(seed=None, method=None)`

Portable container of a pseudo-random number generator.

An instance of this class holds the state of a random number generator. The state is available only on the device which has been current at the initialization of the instance.

Functions of `cupy.random` use global instances of this class. Different instances are used for different devices. The global state for the current device can be obtained by the `cupy.random.get_random_state()` function.

### Parameters

- **seed** (*None* or *int*) – Seed of the random number generator. See the `seed()` method for detail.
- **method** (*int*) – Method of the random number generator. Following values are available:

```
cupy.cuda.curand.CURAND_RNG_PSEUDO_DEFAULT
cupy.cuda.curand.CURAND_RNG_PSEUDO_XORWOW
cupy.cuda.curand.CURAND_RNG_PSEUDO_MRG32K3A
cupy.cuda.curand.CURAND_RNG_PSEUDO_MTGP32
cupy.cuda.curand.CURAND_RNG_PSEUDO_MT19937
cupy.cuda.curand.CURAND_RNG_PSEUDO_PHILOX4_32_10
```

### Methods

**beta**(*a*, *b*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the beta distribution.

**See also:**

- `cupy.random.beta()` for full documentation
- `numpy.random.RandomState.beta()`

**binomial**(*n*, *p*, *size=None*, *dtype=<class 'int'>*)

Returns an array of samples drawn from the binomial distribution.

**See also:**

- `cupy.random.binomial()` for full documentation
- `numpy.random.RandomState.binomial()`

**chisquare**(*df*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the chi-square distribution.

**See also:**

- `cupy.random.chisquare()` for full documentation
- `numpy.random.RandomState.chisquare()`

**choice**(*a*, *size=None*, *replace=True*, *p=None*)

Returns an array of random values from a given 1-D array.

**See also:**

- [`cupy.random.choice\(\)`](#) for full documentation
- `numpy.random.choice()`

**dirichlet**(*alpha*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the dirichlet distribution.

**See also:**

- [`cupy.random.dirichlet\(\)`](#) for full documentation
- `numpy.random.RandomState.dirichlet()`

**exponential**(*scale=1.0*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from a exponential distribution.

**Warning:** This function may synchronize the device.

**See also:**

- [`cupy.random.exponential\(\)`](#) for full documentation
- `numpy.random.RandomState.exponential()`

**f**(*dfnum*, *dfden*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from the f distribution.

**See also:**

- [`cupy.random.f\(\)`](#) for full documentation
- `numpy.random.RandomState.f()`

**gamma**(*shape*, *scale=1.0*, *size=None*, *dtype=<class 'float'>*)

Returns an array of samples drawn from a gamma distribution.

**See also:**

- [`cupy.random.gamma\(\)`](#) for full documentation
- `numpy.random.RandomState.gamma()`

**geometric**(*p*, *size=None*, *dtype=<class 'int'>*)

Returns an array of samples drawn from the geometric distribution.

**See also:**

- [`cupy.random.geometric\(\)`](#) for full documentation
- `numpy.random.RandomState.geometric()`

**gumbel**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a Gumbel distribution.

**See also:**

- `cupy.random.gumbel()` for full documentation
- `numpy.random.RandomState.gumbel()`

**hypergeometric**(*ngood, nbad, nsample, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the hypergeometric distribution.

**See also:**

- `cupy.random.hypergeometric()` for full documentation
- `numpy.random.RandomState.hypergeometric()`

**laplace**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the laplace distribution.

**See also:**

- `cupy.random.laplace()` for full documentation
- `numpy.random.RandomState.laplace()`

**logistic**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the logistic distribution.

**See also:**

- `cupy.random.logistic()` for full documentation
- `numpy.random.RandomState.logistic()`

**lognormal**(*mean=0.0, sigma=1.0, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a log normal distribution.

**See also:**

- `cupy.random.lognormal()` for full documentation
- `numpy.random.RandomState.lognormal()`

**logseries**(*p, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from a log series distribution.

<b>Warning:</b> This function may synchronize the device.
---

**See also:**

- `cupy.random.logseries()` for full documentation
- `numpy.random.RandomState.logseries()`



**multivariate\_normal**(*mean, cov, size=None, check\_valid='ignore', tol=1e-08, method='cholesky', dtype=<class 'float'>*)

Returns an array of samples drawn from the multivariate normal distribution.

**Warning:** This function calls one or more cuSOLVER routine(s) which may yield invalid results if input conditions are not met. To detect these invalid results, you can set the *linalg* configuration to a value that is not *ignore* in `cupyx.errstate()` or `cupyx.seterr()`.

See also:

- `cupy.random.multivariate_normal()` for full documentation
- `numpy.random.RandomState.multivariate_normal()`

**negative\_binomial**(*n, p, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the negative binomial distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.negative_binomial()` for full documentation
- `numpy.random.RandomState.negative_binomial()`

**noncentral\_chisquare**(*df, nonc, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the noncentral chi-square distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.noncentral_chisquare()` for full documentation
- `numpy.random.RandomState.noncentral_chisquare()`

**noncentral\_f**(*dfnum, dfden, nonc, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the noncentral F distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.noncentral_f()` for full documentation
- `numpy.random.RandomState.noncentral_f()`

**normal**(*loc=0.0, scale=1.0, size=None, dtype=<class 'float'>*)

Returns an array of normally distributed samples.

**See also:**

- [`cupy.random.normal\(\)`](#) for full documentation
- [`numpy.random.RandomState.normal\(\)`](#)

**pareto**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the pareto II distribution.

**See also:**

- [`cupy.random.pareto\(\)`](#) for full documentation
- [`numpy.random.RandomState.pareto\(\)`](#)

**permutation**(*a*)

Returns a permuted range or a permutation of an array.

**poisson**(*lam=1.0, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the poisson distribution.

**See also:**

- [`cupy.random.poisson\(\)`](#) for full documentation
- [`numpy.random.RandomState.poisson\(\)`](#)

**power**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the power distribution.

<b>Warning:</b> This function may synchronize the device.
---

**See also:**

- [`cupy.random.power\(\)`](#) for full documentation
- [`numpy.random.RandomState.power\(\)`](#)

**rand**(*\*size, \*\*kwarg*)

Returns uniform random values over the interval `[0, 1)`.

**See also:**

- [`cupy.random.rand\(\)`](#) for full documentation
- [`numpy.random.RandomState.rand\(\)`](#)

**randint**(*low, high=None, size=None, dtype=<class 'int'>*)

Returns a scalar or an array of integer values over `[low, high)`.

**See also:**

- [`cupy.random.randint\(\)`](#) for full documentation

- `numpy.random.RandomState.randint()`

**randn**(\*size, \*\*kwarg)

Returns an array of standard normal random values.

**See also:**

- `cupy.random.randn()` for full documentation
- `numpy.random.RandomState.randn()`

**random\_sample**(size=None, dtype=<class 'float'>)

Returns an array of random values over the interval `[0, 1)`.

**See also:**

- `cupy.random.random_sample()` for full documentation
- `numpy.random.RandomState.random_sample()`

**rayleigh**(scale=1.0, size=None, dtype=<class 'float'>)

Returns an array of samples drawn from a rayleigh distribution.

**Warning:** This function may synchronize the device.

**See also:**

- `cupy.random.rayleigh()` for full documentation
- `numpy.random.RandomState.rayleigh()`

**seed**(seed=None)

Resets the state of the random number generator with a seed.

**See also:**

- `cupy.random.seed()` for full documentation
- `numpy.random.RandomState.seed()`

**shuffle**(a)

Returns a shuffled array.

**See also:**

- `cupy.random.shuffle()` for full documentation
- `numpy.random.shuffle()`

**standard\_cauchy**(size=None, dtype=<class 'float'>)

Returns an array of samples drawn from the standard cauchy distribution.

**See also:**

- `cupy.random.standard_cauchy()` for full documentation
- `numpy.random.RandomState.standard_cauchy()`

**standard\_exponential**(*size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the standard exp distribution.

**See also:**

- [`cupy.random.standard\_exponential\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_exponential\(\)`](#)

**standard\_gamma**(*shape, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from a standard gamma distribution.

**See also:**

- [`cupy.random.standard\_gamma\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_gamma\(\)`](#)

**standard\_normal**(*size=None, dtype=<class 'float'>*)

Returns samples drawn from the standard normal distribution.

**See also:**

- [`cupy.random.standard\_normal\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_normal\(\)`](#)

**standard\_t**(*df, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the standard t distribution.

**See also:**

- [`cupy.random.standard\_t\(\)`](#) for full documentation
- [`numpy.random.RandomState.standard\_t\(\)`](#)

**tomaxint**(*size=None*)

Draws integers between 0 and max integer inclusive.

Return a sample of uniformly distributed random integers in the interval `[0, np.iinfo(np.int_).max]`. The `np.int_` type translates to the C long integer type and its precision is platform dependent.

**Parameters**

**size** (*int* or *tuple of ints*) – Output shape.

**Returns**

Drawn samples.

**Return type**

*cupy.ndarray*

**See also:**

[`numpy.random.RandomState.tomaxint\(\)`](#)

**triangular**(*left, mode, right, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the triangular distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.triangular()` for full documentation
- `numpy.random.RandomState.triangular()`

**uniform**(*low=0.0, high=1.0, size=None, dtype=<class 'float'>*)

Returns an array of uniformly-distributed samples over an interval.

See also:

- `cupy.random.uniform()` for full documentation
- `numpy.random.RandomState.uniform()`

**vonmises**(*mu, kappa, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the von Mises distribution.

See also:

- `cupy.random.vonmises()` for full documentation
- `numpy.random.RandomState.vonmises()`

**wald**(*mean, scale, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the Wald distribution.

See also:

- `cupy.random.wald()` for full documentation
- `numpy.random.RandomState.wald()`

**weibull**(*a, size=None, dtype=<class 'float'>*)

Returns an array of samples drawn from the weibull distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.weibull()` for full documentation
- `numpy.random.RandomState.weibull()`

**zipf**(*a, size=None, dtype=<class 'int'>*)

Returns an array of samples drawn from the Zipf distribution.

**Warning:** This function may synchronize the device.

See also:

- `cupy.random.zipf()` for full documentation
- `numpy.random.RandomState.zipf()`

```

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

## Functions in `cupy.random`

<code>beta(a, b[, size, dtype])</code>	Beta distribution.
<code>binomial(n, p[, size, dtype])</code>	Binomial distribution.
<code>bytes(length)</code>	Returns random bytes.
<code>chisquare(df[, size, dtype])</code>	Chi-square distribution.
<code>choice(a[, size, replace, p])</code>	Returns an array of random values from a given 1-D array.
<code>dirichlet(alpha[, size, dtype])</code>	Dirichlet distribution.
<code>exponential(scale[, size, dtype])</code>	Exponential distribution.
<code>f(dfnum, dfden[, size, dtype])</code>	F distribution.
<code>gamma(shape[, scale, size, dtype])</code>	Gamma distribution.
<code>geometric(p[, size, dtype])</code>	Geometric distribution.
<code>gumbel([loc, scale, size, dtype])</code>	Returns an array of samples drawn from a Gumbel distribution.
<code>hypergeometric(ngood, nbad, nsample[, size, ...])</code>	hypergeometric distribution.
<code>laplace([loc, scale, size, dtype])</code>	Laplace distribution.
<code>logistic([loc, scale, size, dtype])</code>	Logistic distribution.
<code>lognormal([mean, sigma, size, dtype])</code>	Returns an array of samples drawn from a log normal distribution.
<code>logseries(p[, size, dtype])</code>	Log series distribution.
<code>multinomial(n, pvals[, size])</code>	Returns an array from multinomial distribution.
<code>multivariate_normal(mean, cov[, size, ...])</code>	Multivariate normal distribution.
<code>negative_binomial(n, p[, size, dtype])</code>	Negative binomial distribution.
<code>noncentral_chisquare(df, nonc[, size, dtype])</code>	Noncentral chisquare distribution.
<code>noncentral_f(dfnum, dfden, nonc[, size, dtype])</code>	Noncentral F distribution.
<code>normal([loc, scale, size, dtype])</code>	Returns an array of normally distributed samples.
<code>pareto(a[, size, dtype])</code>	Pareto II or Lomax distribution.
<code>permutation(a)</code>	Returns a permuted range or a permutation of an array.
<code>poisson([lam, size, dtype])</code>	Poisson distribution.

continues on next page

Table 2 – continued from previous page

<code>power(a[, size, dtype])</code>	Power distribution.
<code>rand(*size, **kwarg)</code>	Returns an array of uniform random values over the interval $[0, 1)$ .
<code>randint(low[, high, size, dtype])</code>	Returns a scalar or an array of integer values over $[low, high]$ .
<code>randn(*size, **kwarg)</code>	Returns an array of standard normal random values.
<code>random([size, dtype])</code>	Returns an array of random values over the interval $[0, 1)$ .
<code>random_integers(low[, high, size])</code>	Return a scalar or an array of integer values over $[low, high]$ .
<code>random_sample([size, dtype])</code>	Returns an array of random values over the interval $[0, 1)$ .
<code>ranf([size, dtype])</code>	Returns an array of random values over the interval $[0, 1)$ .
<code>rayleigh([scale, size, dtype])</code>	Rayleigh distribution.
<code>sample([size, dtype])</code>	Returns an array of random values over the interval $[0, 1)$ .
<code>seed([seed])</code>	Resets the state of the random number generator with a seed.
<code>shuffle(a)</code>	Shuffles an array.
<code>standard_cauchy([size, dtype])</code>	Standard cauchy distribution.
<code>standard_exponential([size, dtype])</code>	Standard exponential distribution.
<code>standard_gamma(shape[, size, dtype])</code>	Standard gamma distribution.
<code>standard_normal([size, dtype])</code>	Returns an array of samples drawn from the standard normal distribution.
<code>standard_t(df[, size, dtype])</code>	Standard Student's t distribution.
<code>triangular(left, mode, right[, size, dtype])</code>	Triangular distribution.
<code>uniform([low, high, size, dtype])</code>	Returns an array of uniformly-distributed samples over an interval.
<code>vonmises(mu, kappa[, size, dtype])</code>	von Mises distribution.
<code>wald(mean, scale[, size, dtype])</code>	Wald distribution.
<code>weibull(a[, size, dtype])</code>	weibull distribution.
<code>zipf(a[, size, dtype])</code>	Zipf distribution.

## cupy.random.beta

`cupy.random.beta(a, b, size=None, dtype=<class 'float'>)`

Beta distribution.

Returns an array of samples drawn from the beta distribution. Its probability density function is defined as

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}.$$

### Parameters

- **a** (*float*) – Parameter of the beta distribution  $\alpha$ .
- **b** (*float*) – Parameter of the beta distribution  $\beta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the beta distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.beta()`

**cupy.random.binomial**

`cupy.random.binomial(n, p, size=None, dtype=<class 'int'>)`

Binomial distribution.

Returns an array of samples drawn from the binomial distribution. Its probability mass function is defined as

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}.$$

**Parameters**

- **n** (*int*) – Trial number of the binomial distribution.
- **p** (*float*) – Success probability of the binomial distribution.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns**

Samples drawn from the binomial distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.binomial()`

**cupy.random.bytes**

`cupy.random.bytes(length)`

Returns random bytes.

---

**Note:** This function is just a wrapper for `numpy.random.bytes`. The resulting bytes are generated on the host (NumPy), not GPU.

---

**See also:**

`numpy.random.bytes`



**cupy.random.chisquare**

`cupy.random.chisquare(df, size=None, dtype=<class 'float'>)`

Chi-square distribution.

Returns an array of samples drawn from the chi-square distribution. Its probability density function is defined as

$$f(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}.$$

**Parameters**

- **df** (*int* or *array\_like of ints*) – Degree of freedom  $k$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the chi-square distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.chisquare()`

**cupy.random.choice**

`cupy.random.choice(a, size=None, replace=True, p=None)`

Returns an array of random values from a given 1-D array.

Each element of the returned array is independently sampled from `a` according to `p` or uniformly.

---

**Note:** Currently `p` is not supported when `replace=False`.

---

**Parameters**

- **a** (*1-D array-like* or *int*) – If an array-like, a random sample is generated from its elements. If an int, the random sample is generated as if `a` was `cupy.arange(n)`
- **size** (*int* or *tuple of ints*) – The shape of the array.
- **replace** (*boolean*) – Whether the sample is with or without replacement.
- **p** (*1-D array-like*) – The probabilities associated with each entry in `a`. If not given the sample assumes a uniform distribution over all entries in `a`.

**Returns**

An array of `a` values distributed according to `p` or uniformly.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.choice()`

### cupy.random.dirichlet

cupy.random.**dirichlet**(*alpha*, *size=None*, *dtype=<class 'float'>*)

Dirichlet distribution.

Returns an array of samples drawn from the dirichlet distribution. Its probability density function is defined as

$$f(x) = \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K x_i^{\alpha_i-1}.$$

#### Parameters

- **alpha** (*array*) – Parameters of the dirichlet distribution  $\alpha$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the dirichlet distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.dirichlet()`

### cupy.random.exponential

cupy.random.**exponential**(*scale*, *size=None*, *dtype=<class 'float'>*)

Exponential distribution.

Returns an array of samples drawn from the exponential distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right).$$

#### Parameters

- **scale** (*float* or *array\_like of floats*) – The scale parameter  $\beta$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the exponential distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.exponential()`

**cupy.random.f**

`cupy.random.f(dfnum, dfden, size=None, dtype=<class 'float'>)`

F distribution.

Returns an array of samples drawn from the f distribution. Its probability density function is defined as

$$f(x) = \frac{1}{B(\frac{d_1}{2}, \frac{d_2}{2})} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2}x\right)^{-\frac{d_1+d_2}{2}}.$$

**Parameters**

- **dfnum** (*float* or *array\_like of floats*) – Parameter of the f distribution  $d_1$ .
- **dfden** (*float* or *array\_like of floats*) – Parameter of the f distribution  $d_2$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the f distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.f()`

**cupy.random.gamma**

`cupy.random.gamma(shape, scale=1.0, size=None, dtype=<class 'float'>)`

Gamma distribution.

Returns an array of samples drawn from the gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}.$$

**Parameters**

- **shape** (*array*) – Parameter of the gamma distribution  $k$ .
- **scale** (*array*) – Parameter of the gamma distribution  $\theta$
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns: `cupy.ndarray`: Samples drawn from the gamma distribution.

**See also:**

`numpy.random.gamma()`

## cupy.random.geometric

`cupy.random.geometric(p, size=None, dtype=<class 'int'>)`

Geometric distribution.

Returns an array of samples drawn from the geometric distribution. Its probability mass function is defined as

$$f(x) = p(1 - p)^{k-1}.$$

### Parameters

- **p** (*float*) – Success probability of the geometric distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

### Returns

Samples drawn from the geometric distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.geometric()`

## cupy.random.gumbel

`cupy.random.gumbel(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from a Gumbel distribution.

The samples are drawn from a Gumbel distribution with location `loc` and scale `scale`. Its probability density function is defined as

$$f(x) = \frac{1}{\eta} \exp \left\{ -\frac{x - \mu}{\eta} \right\} \exp \left[ -\exp \left\{ -\frac{x - \mu}{\eta} \right\} \right],$$

where  $\mu$  is `loc` and  $\eta$  is `scale`.

### Parameters

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $\eta$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

### Returns

Samples drawn from the Gumbel distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.gumbel()`

**cupy.random.hypergeometric**

`cupy.random.hypergeometric(ngood, nbad, nsample, size=None, dtype=<class 'int'>)`

hypergeometric distribution.

Returns an array of samples drawn from the hypergeometric distribution. Its probability mass function is defined as

$$f(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}}.$$

**Parameters**

- **ngood** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $n$ .
- **nbad** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $m$ .
- **nsample** (*int* or *array\_like of ints*) – Parameter of the hypergeometric distribution  $N$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns**

Samples drawn from the hypergeometric distribution.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.hypergeometric()`

**cupy.random.laplace**

`cupy.random.laplace(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Laplace distribution.

Returns an array of samples drawn from the laplace distribution. Its probability density function is defined as

$$f(x) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right).$$

**Parameters**

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $b$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the laplace distribution.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.laplace()`

### `cupy.random.logistic`

`cupy.random.logistic(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Logistic distribution.

Returns an array of samples drawn from the logistic distribution. Its probability density function is defined as

$$f(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}.$$

#### Parameters

- **loc** (*float*) – The location of the mode  $\mu$ .
- **scale** (*float*) – The scale parameter  $s$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the logistic distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.logistic()`

### `cupy.random.lognormal`

`cupy.random.lognormal(mean=0.0, sigma=1.0, size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from a log normal distribution.

The samples are natural log of samples drawn from a normal distribution with mean `mean` and deviation `sigma`.

#### Parameters

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the log normal distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.lognormal()`

### cupy.random.logseries

`cupy.random.logseries(p, size=None, dtype=<class 'int'>)`

Log series distribution.

Returns an array of samples drawn from the log series distribution. Its probability mass function is defined as

$$f(x) = \frac{-p^x}{x \ln(1-p)}.$$

#### Parameters

- **p** (*float*) – Parameter of the log series distribution  $p$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

#### Returns

Samples drawn from the log series distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.logseries()`

### cupy.random.multinomial

`cupy.random.multinomial(n, pvals, size=None)`

Returns an array from multinomial distribution.

#### Parameters

- **n** (*int*) – Number of trials.
- **pvals** (*cupy.ndarray*) – Probabilities of each of the  $p$  different outcomes. The sum of these values must be 1.
- **size** (*int or tuple of ints or None*) – Shape of a sample in each trial. For example when `size` is `(a, b)`, shape of returned value is `(a, b, p)` where  $p$  is `len(pvals)`. If `size` is `None`, it is treated as `()`. So, shape of returned value is `(p,)`.

#### Returns

An array drawn from multinomial distribution.

#### Return type

*cupy.ndarray*

---

**Note:** It does not support `sum(pvals) < 1` case.

---

See also:

`numpy.random.multinomial()`

## cupy.random.multivariate\_normal

`cupy.random.multivariate_normal(mean, cov, size=None, check_valid='ignore', tol=1e-08, method='cholesky', dtype=<class 'float'>)`

Multivariate normal distribution.

Returns an array of samples drawn from the multivariate normal distribution. Its probability density function is defined as

$$f(x) = \frac{1}{(2\pi|\Sigma|)^{(n/2)}} \exp\left(-\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu)\right).$$

### Parameters

- **mean** (1-D array\_like, of length  $N$ ) – Mean of the multivariate normal distribution  $\mu$ .
- **cov** (2-D array\_like, of shape  $(N, N)$ ) – Covariance matrix  $\Sigma$  of the multivariate normal distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (int or tuple of ints) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **check\_valid** ('warn', 'raise', 'ignore') – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (float) – Tolerance when checking the singular values in covariance matrix.
- **method** – { 'cholesky', 'eigh', 'svd' }, optional The cov input is used to compute a factor matrix  $A$  such that  $A @ A.T = cov$ . This argument is used to select the method used to compute the factor matrix  $A$ . The default method 'cholesky' is the fastest, while 'svd' is the slowest but more robust than the fastest method. The method *eigh* uses eigen decomposition to compute  $A$  and is faster than svd but slower than cholesky.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

### Returns

Samples drawn from the multivariate normal distribution.

### Return type

*cupy.ndarray*

---

**Note:** Default *method* is set to fastest, 'cholesky', unlike `numpy` which defaults to 'svd'. Cholesky decomposition in CuPy will fail silently if the input covariance matrix is not positive definite and give invalid results, unlike in `numpy`, where an invalid covariance matrix will raise an exception. Setting *check\_valid* to 'raise' will replicate `numpy` behavior by checking the input, but will also force device synchronization. If validity of input is unknown, setting *method* to 'einh' or 'svd' and *check\_valid* to 'warn' will use cholesky decomposition for positive definite matrices, and fallback to the specified *method* for other matrices (i.e., not positive semi-definite), and will warn if decomposition is suspect.

---

### See also:

`numpy.random.multivariate_normal()`



### cupy.random.negative\_binomial

`cupy.random.negative_binomial(n, p, size=None, dtype=<class 'int'>)`

Negative binomial distribution.

Returns an array of samples drawn from the negative binomial distribution. Its probability mass function is defined as

$$f(x) = \binom{x+n-1}{n-1} p^n (1-p)^x.$$

#### Parameters

- **n** (*int*) – Parameter of the negative binomial distribution  $n$ .
- **p** (*float*) – Parameter of the negative binomial distribution  $p$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

#### Returns

Samples drawn from the negative binomial distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.negative_binomial()`

### cupy.random.noncentral\_chisquare

`cupy.random.noncentral_chisquare(df, nonc, size=None, dtype=<class 'float'>)`

Noncentral chisquare distribution.

Returns an array of samples drawn from the noncentral chisquare distribution. Its probability density function is defined as

$$f(x) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{k/4-1/2} I_{k/2-1}(\sqrt{\lambda x}),$$

where  $I$  is the modified Bessel function of the first kind.

#### Parameters

- **df** (*float*) – Parameter of the noncentral chisquare distribution  $k$ .
- **nonc** (*float*) – Parameter of the noncentral chisquare distribution  $\lambda$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the noncentral chisquare distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.noncentral_chisquare()`

### cupy.random.noncentral\_f

`cupy.random.noncentral_f(dfnum, dfden, nonc, size=None, dtype=<class 'float'>)`

Noncentral F distribution.

Returns an array of samples drawn from the noncentral F distribution.

Reference: [https://en.wikipedia.org/wiki/Noncentral\\_F-distribution](https://en.wikipedia.org/wiki/Noncentral_F-distribution)

#### Parameters

- **dfnum** (*float*) – Parameter of the noncentral F distribution.
- **dfden** (*float*) – Parameter of the noncentral F distribution.
- **nonc** (*float*) – Parameter of the noncentral F distribution.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the noncentral F distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.noncentral_f()`

### cupy.random.normal

`cupy.random.normal(loc=0.0, scale=1.0, size=None, dtype=<class 'float'>)`

Returns an array of normally distributed samples.

#### Parameters

- **loc** (*float* or *array\_like of floats*) – Mean of the normal distribution.
- **scale** (*float* or *array\_like of floats*) – Standard deviation of the normal distribution.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Normally distributed samples.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.normal()`

**cupy.random.pareto**

`cupy.random.pareto(a, size=None, dtype=<class 'float'>)`

Pareto II or Lomax distribution.

Returns an array of samples drawn from the Pareto II distribution. Its probability density function is defined as

$$f(x) = \alpha(1+x)^{-(\alpha+1)}.$$

**Parameters**

- **a** (*float* or *array\_like of floats*) – Parameter of the Pareto II distribution  $\alpha$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, this function generate an array whose shape is `a.shape`.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the Pareto II distribution.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.pareto()`

**cupy.random.permutation**

`cupy.random.permutation(a)`

Returns a permuted range or a permutation of an array.

**Parameters**

**a** (*int* or *cupy.ndarray*) – The range or the array to be shuffled.

**Returns**

If `a` is an integer, it is permutation range between 0 and `a - 1`. Otherwise, it is a permutation of `a`.

**Return type**

*cupy.ndarray*

See also:

`numpy.random.permutation()`

**cupy.random.poisson**

`cupy.random.poisson(lam=1.0, size=None, dtype=<class 'int'>)`

Poisson distribution.

Returns an array of samples drawn from the poisson distribution. Its probability mass function is defined as

$$f(x) = \frac{\lambda^x e^{-\lambda}}{k!}.$$

**Parameters**

- **lam** (*array\_like of floats*) – Parameter of the poisson distribution  $\lambda$ .

- **size** (*int* or *tuple of ints*) – The shape of the array. If None, this function generate an array whose shape is *lam.shape*.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

**Returns**

Samples drawn from the poisson distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.poisson()`

## **cupy.random.power**

`cupy.random.power(a, size=None, dtype=<class 'float'>)`

Power distribution.

Returns an array of samples drawn from the power distribution. Its probability density function is defined as

$$f(x) = ax^{a-1}.$$

**Parameters**

- **a** (*float*) – Parameter of the power distribution *a*.
- **size** (*int* or *tuple of ints*) – The shape of the array. If None, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the power distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.power()`

## **cupy.random.rand**

`cupy.random.rand(*size, **kwarg)`

Returns an array of uniform random values over the interval `[0, 1)`.

Each element of the array is uniformly distributed on the half-open interval `[0, 1)`. All elements are identically and independently distributed (i.i.d.).

**Parameters**

- **size** (*ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns**

A random array.

**Return type***cupy.ndarray***See also:**`numpy.random.rand()`**Example**

```
>>> cupy.random.rand(3, 2)
array([[0.86476479, 0.05633727], # random
       [0.27283185, 0.38255354], # random
       [0.16592278, 0.75150313]]) # random

>>> cupy.random.rand(3, 2, dtype=cupy.float32)
array([[0.9672306 , 0.9590486 ], # random
       [0.6851264 , 0.70457625], # random
       [0.22382522, 0.36055237]], dtype=float32) # random
```

**cupy.random.randint**`cupy.random.randint(low, high=None, size=None, dtype='l')`Returns a scalar or an array of integer values over `[low, high)`.Each element of returned values are independently sampled from uniform distribution over left-close and right-open interval `[low, high)`.**Parameters**

- **low** (*int*) – If `high` is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to 0.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.
- **dtype** – Data type specifier.

**Returns**If `size` is `None`, it is single integer sampled. If `size` is integer, it is the 1D-array of length `size` element. Otherwise, it is the array whose shape specified by `size`.**Return type***int* or *cupy.ndarray* of ints**cupy.random.randn**`cupy.random.randn(*size, **kwarg)`

Returns an array of standard normal random values.

Each element of the array is normally distributed with zero mean and unit variance. All elements are identically and independently distributed (i.i.d.).

**Parameters**

- **size** (*ints*) – The shape of the array.

- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

**Returns**

An array of standard normal random values.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.randn()`

---

**Example**

```
>>> cupy.random.randn(3, 2)
array([[0.41193321, 1.59579542],      # random
       [0.47904589, 0.18566376],      # random
       [0.59748424, 2.32602829]])      # random

>>> cupy.random.randn(3, 2, dtype=cupy.float32)
array([[ 0.1373886 ,  2.403238  ],      # random
       [ 0.84020025,  1.5089266 ],      # random
       [-1.2268474 , -0.48219103]], dtype=float32) # random
```

---

**cupy.random.random**

`cupy.random.random(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of [`cupy.random.rand\(\)`](#).

**Parameters**

- **size** (*int* or *tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

An array of uniformly distributed random values.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.random_sample()`

### cupy.random.random\_integers

`cupy.random.random_integers(low, high=None, size=None)`

Return a scalar or an array of integer values over `[low, high]`

Each element of returned values are independently sampled from uniform distribution over closed interval `[low, high]`.

#### Parameters

- **low** (*int*) – If **high** is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and the lower bound is set to 1.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.

#### Returns

If **size** is `None`, it is single integer sampled. If **size** is integer, it is the 1D-array of length **size** element. Otherwise, it is the array whose shape specified by **size**.

#### Return type

*int* or *cupy.ndarray* of ints

### cupy.random.random\_sample

`cupy.random.random_sample(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

#### Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

An array of uniformly distributed random values.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.random_sample()`

### cupy.random.ranf

`cupy.random.ranf(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

#### Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

An array of uniformly distributed random values.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.random_sample()`

**cupy.random.rayleigh**

`cupy.random.rayleigh(scale=1.0, size=None, dtype=<class 'float'>)`

Rayleigh distribution.

Returns an array of samples drawn from the rayleigh distribution. Its probability density function is defined as

$$f(x) = \frac{x}{\sigma^2} e^{\frac{-x^2}{2\sigma^2}}, x \geq 0.$$

**Parameters**

- **scale** (*array*) – Parameter of the rayleigh distribution  $\sigma$ .
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the rayleigh distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.rayleigh()`

**cupy.random.sample**

`cupy.random.sample(size=None, dtype=<class 'float'>)`

Returns an array of random values over the interval `[0, 1)`.

This is a variant of `cupy.random.rand()`.

**Parameters**

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

An array of uniformly distributed random values.

**Return type**

*cupy.ndarray*



See also:

`numpy.random.random_sample()`

### `cupy.random.seed`

`cupy.random.seed(seed=None)`

Resets the state of the random number generator with a seed.

This function resets the state of the global random number generator for the current device. Be careful that generators for other devices are not affected.

#### Parameters

**seed** (*None* or *int*) – Seed for the random number generator. If *None*, it uses `os.urandom()` if available or `time.time()` otherwise. Note that this function does not support seeding by an integer array.

### `cupy.random.shuffle`

`cupy.random.shuffle(a)`

Shuffles an array.

#### Parameters

**a** (`cupy.ndarray`) – The array to be shuffled.

See also:

`numpy.random.shuffle()`

### `cupy.random.standard_cauchy`

`cupy.random.standard_cauchy(size=None, dtype=<class 'float'>)`

Standard cauchy distribution.

Returns an array of samples drawn from the standard cauchy distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

#### Parameters

- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the standard cauchy distribution.

#### Return type

`cupy.ndarray`

See also:

`numpy.random.standard_cauchy()`

### cupy.random.standard\_exponential

`cupy.random.standard_exponential(size=None, dtype=<class 'float'>)`

Standard exponential distribution.

Returns an array of samples drawn from the standard exponential distribution. Its probability density function is defined as

$$f(x) = e^{-x}.$$

#### Parameters

- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the standard exponential distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.standard_exponential()`

### cupy.random.standard\_gamma

`cupy.random.standard_gamma(shape, size=None, dtype=<class 'float'>)`

Standard gamma distribution.

Returns an array of samples drawn from the standard gamma distribution. Its probability density function is defined as

$$f(x) = \frac{1}{\Gamma(k)} x^{k-1} e^{-x}.$$

#### Parameters

- **shape** (*array*) – Parameter of the gamma distribution  $k$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

#### Returns

Samples drawn from the standard gamma distribution.

#### Return type

*cupy.ndarray*

See also:

`numpy.random.standard_gamma()`

**cupy.random.standard\_normal**

`cupy.random.standard_normal(size=None, dtype=<class 'float'>)`

Returns an array of samples drawn from the standard normal distribution.

This is a variant of `cupy.random.randn()`.

**Parameters**

- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

**Returns**

Samples drawn from the standard normal distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.standard_normal()`

**cupy.random.standard\_t**

`cupy.random.standard_t(df, size=None, dtype=<class 'float'>)`

Standard Student's t distribution.

Returns an array of samples drawn from the standard Student's t distribution. Its probability density function is defined as

$$f(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})} \left(1 + \frac{x^2}{\nu}\right)^{-\left(\frac{\nu+1}{2}\right)}.$$

**Parameters**

- **df** (*float* or *array\_like of floats*) – Degree of freedom  $\nu$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the standard Student's t distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.standard_t()`

## cupy.random.triangular

cupy.random.**triangular**(*left*, *mode*, *right*, *size=None*, *dtype=<class 'float'>*)

Triangular distribution.

Returns an array of samples drawn from the triangular distribution. Its probability density function is defined as

$$f(x) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

### Parameters

- **left** (*float*) – Lower limit  $l$ .
- **mode** (*float*) – The value where the peak of the distribution occurs.  $m$ .
- **right** (*float*) – Higher Limit  $r$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

### Returns

Samples drawn from the triangular distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.triangular()`

## cupy.random.uniform

cupy.random.**uniform**(*low=0.0*, *high=1.0*, *size=None*, *dtype=<class 'float'>*)

Returns an array of uniformly-distributed samples over an interval.

Samples are drawn from a uniform distribution over the half-open interval [*low*, *high*). The samples may contain the high limit due to floating-point rounding.

### Parameters

- **low** (*float*) – Lower end of the interval.
- **high** (*float*) – Upper end of the interval.
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

### Returns

Samples drawn from the uniform distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.uniform()`

**cupy.random.vonmises**

`cupy.random.vonmises(mu, kappa, size=None, dtype=<class 'float'>)`

von Mises distribution.

Returns an array of samples drawn from the von Mises distribution. Its probability density function is defined as

$$f(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}.$$

**Parameters**

- **mu** (*float*) – Parameter of the von Mises distribution  $\mu$ .
- **kappa** (*float*) – Parameter of the von Mises distribution  $\kappa$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the von Mises distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.vonmises()`

**cupy.random.wald**

`cupy.random.wald(mean, scale, size=None, dtype=<class 'float'>)`

Wald distribution.

Returns an array of samples drawn from the Wald distribution. Its probability density function is defined as

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}.$$

**Parameters**

- **mean** (*float*) – Parameter of the wald distribution  $\mu$ .
- **scale** (*float*) – Parameter of the wald distribution  $\lambda$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If *None*, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

**Returns**

Samples drawn from the wald distribution.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.random.wald()`

## cupy.random.weibull

`cupy.random.weibull(a, size=None, dtype=<class 'float'>)`

weibull distribution.

Returns an array of samples drawn from the weibull distribution. Its probability density function is defined as

$$f(x) = ax^{(a-1)}e^{-x^a}.$$

### Parameters

- **a** (*float*) – Parameter of the weibull distribution  $a$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

### Returns

Samples drawn from the weibull distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.weibull()`

## cupy.random.zipf

`cupy.random.zipf(a, size=None, dtype=<class 'int'>)`

Zipf distribution.

Returns an array of samples drawn from the Zipf distribution. Its probability mass function is defined as

$$f(x) = \frac{x^{-a}}{\zeta(a)},$$

where  $\zeta$  is the Riemann Zeta function.

### Parameters

- **a** (*float*) – Parameter of the beta distribution  $a$ .
- **size** (*int* or *tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.int32` and `numpy.int64` types are allowed.

### Returns

Samples drawn from the Zipf distribution.

### Return type

*cupy.ndarray*

See also:

`numpy.random.zipf()`

CuPy does not provide `cupy.random.get_state` nor `cupy.random.set_state` at this time. Use the following CuPy-specific APIs instead. Note that these functions use `cupy.random.RandomState` instance to represent the internal state, which cannot be serialized.

---

<code>get_random_state()</code>	Gets the state of the random number generator for the current device.
<code>set_random_state(rs)</code>	Sets the state of the random number generator for the current device.

---

### `cupy.random.get_random_state`

#### `cupy.random.get_random_state()`

Gets the state of the random number generator for the current device.

If the state for the current device is not created yet, this function creates a new one, initializes it, and stores it as the state for the current device.

##### Returns

The state of the random number generator for the device.

##### Return type

*RandomState*

### `cupy.random.set_random_state`

#### `cupy.random.set_random_state(rs)`

Sets the state of the random number generator for the current device.

##### Parameters

**state** (*RandomState*) – Random state to set for the current device.

## 5.3.16 Set routines

---

**Hint:** [NumPy API Reference: Set routines](#)

---

### Making proper sets

---

<code>unique(ar[, return_index, return_inverse, ...])</code>	Find the unique elements of an array.
--	---------------------------------------

---

## Boolean operations

<code>in1d(ar1, ar2[, assume_unique, invert])</code>	Tests whether each element of a 1-D array is also present in a second array.
<code>intersect1d(arr1, arr2[, assume_unique, ...])</code>	Find the intersection of two arrays.
<code>isin(element, test_elements[, ...])</code>	Calculates element in <code>test_elements</code> , broadcasting over <code>element</code> only.
<code>setdiff1d(ar1, ar2[, assume_unique])</code>	Find the set difference of two arrays.
<code>setxor1d(ar1, ar2[, assume_unique])</code>	Find the set exclusive-or of two arrays.

### cupy.in1d

`cupy.in1d(ar1, ar2, assume_unique=False, invert=False)`

Tests whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as `ar1` that is `True` where an element of `ar1` is in `ar2` and `False` otherwise.

#### Parameters

- **ar1** (`cupy.ndarray`) – Input array.
- **ar2** (`cupy.ndarray`) – The values against which to test each value of `ar1`.
- **assume\_unique** (`bool`, *optional*) – Ignored
- **invert** (`bool`, *optional*) – If `True`, the values in the returned array are inverted (that is, `False` where an element of `ar1` is in `ar2` and `True` otherwise). Default is `False`.

#### Returns

**y** – The values `ar1[in1d]` are in `ar2`.

#### Return type

`cupy.ndarray`, `bool`

### cupy.intersect1d

`cupy.intersect1d(arr1, arr2, assume_unique=False, return_indices=False)`

Find the intersection of two arrays. Returns the sorted, unique values that are in both of the input arrays.

#### Parameters

- **arr1** (`cupy.ndarray`) – Input arrays. Arrays will be flattened if they are not in 1D.
- **arr2** (`cupy.ndarray`) – Input arrays. Arrays will be flattened if they are not in 1D.
- **assume\_unique** (`bool`) – By default, `False`. If set `True`, the input arrays will be assumed to be unique, which speeds up the calculation. If set `True`, but the arrays are not unique, incorrect results and out-of-bounds indices could result.
- **return\_indices** (`bool`) – By default, `False`. If `True`, the indices which correspond to the intersection of the two arrays are returned.

#### Returns

- **intersect1d** (`cupy.ndarray`) – Sorted 1D array of common and unique elements.



- **comm1** (*cupy.ndarray*) – The indices of the first occurrences of the common values in *arr1*. Only provided if *return\_indices* is True.
- **comm2** (*cupy.ndarray*) – The indices of the first occurrences of the common values in *arr2*. Only provided if *return\_indices* is True.

See also:

`numpy.intersect1d`

## cupy.isin

`cupy.isin(element, test_elements, assume_unique=False, invert=False)`

Calculates element in `test_elements`, broadcasting over `element` only. Returns a boolean array of the same shape as `element` that is True where an element of `element` is in `test_elements` and False otherwise.

### Parameters

- **element** (*cupy.ndarray*) – Input array.
- **test\_elements** (*cupy.ndarray*) – The values against which to test each value of `element`. This argument is flattened if it is an array or array\_like.
- **assume\_unique** (*bool*, *optional*) – Ignored
- **invert** (*bool*, *optional*) – If True, the values in the returned array are inverted, as if calculating element not in `test_elements`. Default is False.

### Returns

`y` – Has the same shape as `element`. The values `element[isin]` are in `test_elements`.

### Return type

*cupy.ndarray*, *bool*

## cupy.setdiff1d

`cupy.setdiff1d(ar1, ar2, assume_unique=False)`

Find the set difference of two arrays. It returns unique values in *ar1* that are not in *ar2*.

### Parameters

- **ar1** (*cupy.ndarray*) – Input array
- **ar2** (*cupy.ndarray*) – Input array for comparison
- **assume\_unique** (*bool*) – By default, False, i.e. input arrays are not unique. If True, input arrays are assumed to be unique. This can speed up the calculation.

### Returns

**setdiff1d** – Returns a 1D array of values in *ar1* that are not in *ar2*. It always returns a sorted output for unsorted input only if *assume\_unique=False*.

### Return type

*cupy.ndarray*

See also:

`numpy.setdiff1d`

## cupy.setxor1d

`cupy.setxor1d(ar1, ar2, assume_unique=False)`

Find the set exclusive-or of two arrays.

### Parameters

- **ar1** (`cupy.ndarray`) – Input arrays. They are flattened if they are not already 1-D.
- **ar2** (`cupy.ndarray`) – Input arrays. They are flattened if they are not already 1-D.
- **assume\_unique** (`bool`) – By default, False, i.e. input arrays are not unique. If True, input arrays are assumed to be unique. This can speed up the calculation.

### Returns

**setxor1d** – Return the sorted, unique values that are in only one (not both) of the input arrays.

### Return type

`cupy.ndarray`

See also:

`numpy.setxor1d`

## 5.3.17 Sorting, searching, and counting

---

**Hint:** NumPy API Reference: Sorting, searching, and counting

---

### Sorting

<code>sort(a[, axis, kind])</code>	Returns a sorted copy of an array with a stable sorting algorithm.
<code>lexsort(keys)</code>	Perform an indirect sort using an array of keys.
<code>argsort(a[, axis, kind])</code>	Returns the indices that would sort an array with a stable sorting.
<code>msort(a)</code>	Returns a copy of an array sorted along the first axis.
<code>sort_complex(a)</code>	Sort a complex array using the real part first, then the imaginary part.
<code>partition(a, kth[, axis])</code>	Returns a partitioned copy of an array.
<code>argpartition(a, kth[, axis])</code>	Returns the indices that would partially sort an array.

## cupy.sort

`cupy.sort(a, axis=-1, kind=None)`

Returns a sorted copy of an array with a stable sorting algorithm.

### Parameters

- **a** (`cupy.ndarray`) – Array to be sorted.
- **axis** (`int` or `None`) – Axis along which to sort. Default is -1, which means sort along the last axis. If None is supplied, the array is flattened before sorting.

- **kind** – Default is *None*, which is equivalent to ‘stable’. Unlike in NumPy any other options are not accepted here.

**Returns**

Array of the same type and shape as *a*.

**Return type**

*cupy.ndarray*

---

**Note:** For its implementation reason, `cupy.sort` currently does not support `kind` and `order` parameters that `numpy.sort` does support.

---

**See also:**

`numpy.sort()`

**cupy.lexsort**

`cupy.lexsort(keys)`

Perform an indirect sort using an array of keys.

**Parameters**

**keys** (*cupy.ndarray*) – (*k*, *N*) array containing *k* (*N*,)-shaped arrays. The *k* different “rows” to be sorted. The last row is the primary sort key.

**Returns**

Array of indices that sort the keys.

**Return type**

*cupy.ndarray*

---

**Note:** For its implementation reason, `cupy.lexsort` currently supports only keys with their rank of one or two and does not support `axis` parameter that `numpy.lexsort` supports.

---

**See also:**

`numpy.lexsort()`

**cupy.argsort**

`cupy.argsort(a, axis=-1, kind=None)`

Returns the indices that would sort an array with a stable sorting.

**Parameters**

- **a** (*cupy.ndarray*) – Array to sort.
- **axis** (*int* or *None*) – Axis along which to sort. Default is -1, which means sort along the last axis. If *None* is supplied, the array is flattened before sorting.
- **kind** – Default is *None*, which is equivalent to ‘stable’. Unlike in NumPy any other options are not accepted here.

**Returns**

Array of indices that sort *a*.

**Return type***cupy.ndarray*

---

**Note:** For its implementation reason, `cupy.argsort` does not support `kind` and `order` parameters.

---

**See also:**`numpy.argsort()`**cupy.msort**`cupy.msort(a)`

Returns a copy of an array sorted along the first axis.

**Parameters**

**a** (`cupy.ndarray`) – Array to be sorted.

**Returns**

Array of the same type and shape as **a**.

**Return type***cupy.ndarray***See also:**`numpy.msort()`**cupy.sort\_complex**`cupy.sort_complex(a)`

Sort a complex array using the real part first, then the imaginary part.

**Parameters**

**a** (`cupy.ndarray`) – Array to be sorted.

**Returns**

sorted complex array.

**Return type***cupy.ndarray***See also:**`numpy.sort_complex()`**cupy.partition**`cupy.partition(a, kth, axis=-1)`

Returns a partitioned copy of an array.

Creates a copy of the array whose elements are rearranged such that the value of the element in *k*-th position would occur in that position in a sorted array. All of the elements before the new *k*-th element are less than or equal to the elements after the new *k*-th element.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be sorted.
- **kth** (`int` or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (`int` or `None`) – Axis along which to sort. Default is -1, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns**

Array of the same type and shape as a.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.partition()`

### `cupy.argpartition`

`cupy.argpartition(a, kth, axis=-1)`

Returns the indices that would partially sort an array.

**Parameters**

- **a** (`cupy.ndarray`) – Array to be sorted.
- **kth** (`int` or *sequence of ints*) – Element index to partition by. If supplied with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.
- **axis** (`int` or `None`) – Axis along which to sort. Default is -1, which means sort along the last axis. If `None` is supplied, the array is flattened before sorting.

**Returns**

Array of the same type and shape as a.

**Return type**

`cupy.ndarray`

---

**Note:** For its implementation reason, `cupy.argpartition` fully sorts the given array as `cupy.argsort` does. It also does not support `kind` and `order` parameters that `numpy.argpartition` supports.

---

**See also:**

`numpy.argpartition()`

**See also:**

`cupy.ndarray.sort()`

## Searching

<code>argmax(a[, axis, dtype, out, keepdims])</code>	Returns the indices of the maximum along an axis.
<code>nanargmax(a[, axis, dtype, out, keepdims])</code>	Return the indices of the maximum values in the specified axis ignoring NaNs.
<code>argmin(a[, axis, dtype, out, keepdims])</code>	Returns the indices of the minimum along an axis.
<code>nanargmin(a[, axis, dtype, out, keepdims])</code>	Return the indices of the minimum values in the specified axis ignoring NaNs.
<code>argwhere(a)</code>	Return the indices of the elements that are non-zero.
<code>nonzero(a)</code>	Return the indices of the elements that are non-zero.
<code>flatnonzero(a)</code>	Return indices that are non-zero in the flattened version of a.
<code>where(condition[, x, y])</code>	Return elements, either from x or y, depending on condition.
<code>searchsorted(a, v[, side, sorter])</code>	Finds indices where elements should be inserted to maintain order.
<code>extract(condition, a)</code>	Return the elements of an array that satisfy some condition.

### cupy.argmax

`cupy.argmax(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the indices of the maximum along an axis.

#### Parameters

- **a** (`cupy.ndarray`) – Array to take argmax.
- **axis** (`int`) – Along which axis to find the maximum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis `axis` is preserved as an axis of length one.

#### Returns

The indices of the maximum of a along an axis.

#### Return type

`cupy.ndarray`

---

**Note:** `dtype` and `keepdim` arguments are specific to CuPy. They are not in NumPy.

---

---

**Note:** `axis` argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

#### See also:

`numpy.argmax()`

## cupy.nanargmax

`cupy.nanargmax(a, axis=None, dtype=None, out=None, keepdims=False)`

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slice -1 is returned. Subclass cannot be passed yet, subok=True still unsupported

### Parameters

- **a** (`cupy.ndarray`) – Array to take nanargmax.
- **axis** (`int`) – Along which axis to find the maximum. a is flattened by default.

### Returns

The indices of the maximum of a along an axis ignoring NaN values.

### Return type

`cupy.ndarray`

---

**Note:** For performance reasons, `cupy.nanargmax` returns out of range values for all-NaN slice whereas `numpy.nanargmax` raises `ValueError`

---

### See also:

`numpy.nanargmax()`

## cupy.argmin

`cupy.argmin(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the indices of the minimum along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to take argmin.
- **axis** (`int`) – Along which axis to find the minimum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis axis is preserved as an axis of length one.

### Returns

The indices of the minimum of a along an axis.

### Return type

`cupy.ndarray`

---

**Note:** `dtype` and `keepdim` arguments are specific to CuPy. They are not in NumPy.

---



---

**Note:** `axis` argument accepts a tuple of ints, but this is specific to CuPy. NumPy does not support it.

---

### See also:

`numpy.argmin()`

## cupy.nanargmin

`cupy.nanargmin(a, axis=None, dtype=None, out=None, keepdims=False)`

Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slice -1 is returned. Subclass cannot be passed yet, subok=True still unsupported

### Parameters

- **a** (`cupy.ndarray`) – Array to take nanargmin.
- **axis** (`int`) – Along which axis to find the minimum. a is flattened by default.

### Returns

The indices of the minimum of a along an axis ignoring NaN values.

### Return type

`cupy.ndarray`

---

**Note:** For performance reasons, `cupy.nanargmin` returns out of range values for all-NaN slice whereas `numpy.nanargmin` raises `ValueError`

---

### See also:

`numpy.nanargmin()`

## cupy.argwhere

`cupy.argwhere(a)`

Return the indices of the elements that are non-zero.

Returns a (N, ndim) dimensional array containing the indices of the non-zero elements. Where *N* is number of non-zero elements and *ndim* is dimension of the given array.

### Parameters

**a** (`cupy.ndarray`) – array

### Returns

Indices of elements that are non-zero.

### Return type

`cupy.ndarray`

### See also:

`numpy.argwhere()`

## cupy.flatnonzero

`cupy.flatnonzero(a)`

Return indices that are non-zero in the flattened version of a.

This is equivalent to `a.ravel().nonzero()[0]`.

### Parameters

**a** (`cupy.ndarray`) – input array



**Returns**

Output array, containing the indices of the elements of `a.ravel()` that are non-zero.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

**See also:**

`numpy.flatnonzero()`

**cupy.searchsorted**

`cupy.searchsorted(a, v, side='left', sorter=None)`

Finds indices where elements should be inserted to maintain order.

Find the indices into a sorted array `a` such that, if the corresponding elements in `v` were inserted before the indices, the order of `a` would be preserved.

**Parameters**

- **a** (*cupy.ndarray*) – Input array. If `sorter` is `None`, then it must be sorted in ascending order, otherwise `sorter` must be an array of indices that sort it.
- **v** (*cupy.ndarray*) – Values to insert into `a`.
- **side** – { 'left', 'right' } If `left`, return the index of the first suitable location found. If `right`, return the last such index. If there is no suitable index, return either 0 or length of `a`.
- **sorter** – 1-D array\_like Optional array of integer indices that sort array `a` into ascending order. They are typically the result of `argsort()`.

**Returns**

Array of insertion points with the same shape as `v`.

**Return type**

*cupy.ndarray*

---

**Note:** When `a` is not in ascending order, behavior is undefined.

---

**See also:**

`numpy.searchsorted()`

**cupy.extract**

`cupy.extract(condition, a)`

Return the elements of an array that satisfy some condition.

This is equivalent to `np.compress(ravel(condition), ravel(arr))`. If `condition` is boolean, `np.extract` is equivalent to `arr[condition]`.

**Parameters**

- **condition** (*int or array\_like*) – An array whose nonzero or True entries indicate the elements of array to extract.
- **a** (*cupy.ndarray*) – Input array of the same size as condition.

**Returns**

Rank 1 array of values from arr where condition is True.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

**See also:**

`numpy.extract()`

## Counting

---

<code>count_nonzero(a[, axis])</code>	Counts the number of non-zero values in the array.
---------------------------------------	--

---

### `cupy.count_nonzero`

`cupy.count_nonzero(a, axis=None)`

Counts the number of non-zero values in the array.

---

**Note:** `numpy.count_nonzero()` returns *int* value when *axis=None*, but `cupy.count_nonzero()` returns zero-dimensional array to reduce CPU-GPU synchronization.

---

**Parameters**

- **a** (*cupy.ndarray*) – The array for which to count non-zeros.
- **axis** (*int or tuple, optional*) – Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of a

**Returns**

Number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.

**Return type**

*cupy.ndarray* of *int*

### 5.3.18 Statistics

---

**Hint:** [NumPy API Reference: Statistics](#)

---

#### Order statistics

<code>amin(a[, axis, out, keepdims])</code>	Returns the minimum of an array or the minimum along an axis.
<code>amax(a[, axis, out, keepdims])</code>	Returns the maximum of an array or the maximum along an axis.
<code>nanmin(a[, axis, out, keepdims])</code>	Returns the minimum of an array along an axis ignoring NaN.
<code>nanmax(a[, axis, out, keepdims])</code>	Returns the maximum of an array along an axis ignoring NaN.
<code>ptp(a[, axis, out, keepdims])</code>	Returns the range of values (maximum - minimum) along an axis.
<code>percentile(a, q[, axis, out, ...])</code>	Computes the q-th percentile of the data along the specified axis.
<code>quantile(a, q[, axis, out, overwrite_input, ...])</code>	Computes the q-th quantile of the data along the specified axis.

---

#### cupy.amin

`cupy.amin(a, axis=None, out=None, keepdims=False)`

Returns the minimum of an array or the minimum along an axis.

---

**Note:** When at least one element is NaN, the corresponding min value will be NaN.

---

#### Parameters

- **a** (`cupy.ndarray`) – Array to take the minimum.
- **axis** (`int`) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

#### Returns

The minimum of `a`, along the axis if specified.

#### Return type

`cupy.ndarray`

---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have one or more NaN elements.

---

See also:

`numpy.amin()`

## **cupy.amax**

`cupy.amax(a, axis=None, out=None, keepdims=False)`

Returns the maximum of an array or the maximum along an axis.

---

**Note:** When at least one element is NaN, the corresponding min value will be NaN.

---

### **Parameters**

- **a** (`cupy.ndarray`) – Array to take the maximum.
- **axis** (`int`) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

### **Returns**

The maximum of **a**, along the axis if specified.

### **Return type**

`cupy.ndarray`

---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have one or more NaN elements.

---

See also:

`numpy.amax()`

## **cupy.nanmin**

`cupy.nanmin(a, axis=None, out=None, keepdims=False)`

Returns the minimum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a `RuntimeWarning` is raised and NaN is returned.

### **Parameters**

- **a** (`cupy.ndarray`) – Array to take the minimum.
- **axis** (`int`) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

### **Returns**

The minimum of **a**, along the axis if specified.

### **Return type**

`cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.nanmin()`

## cupy.nanmax

`cupy.nanmax(a, axis=None, out=None, keepdims=False)`

Returns the maximum of an array along an axis ignoring NaN.

When there is a slice whose elements are all NaN, a `RuntimeWarning` is raised and NaN is returned.

### Parameters

- **a** (`cupy.ndarray`) – Array to take the maximum.
- **axis** (`int`) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

### Returns

The maximum of a, along the axis if specified.

### Return type

`cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.nanmax()`

## cupy.ptp

`cupy.ptp(a, axis=None, out=None, keepdims=False)`

Returns the range of values (maximum - minimum) along an axis.

---

**Note:** The name of the function comes from the acronym for ‘peak to peak’.

When at least one element is NaN, the corresponding ptp value will be NaN.

---

### Parameters

- **a** (`cupy.ndarray`) – Array over which to take the range.
- **axis** (`int`) – Axis along which to take the minimum. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is retained as an axis of size one.

**Returns**

The minimum of `a`, along the axis if specified.

**Return type**

*cupy.ndarray*

---

**Note:** When cuTENSOR accelerator is used, the output value might be collapsed for reduction axes that have one or more NaN elements.

---

**See also:**

`numpy.amin()`

**cupy.percentile**

`cupy.percentile(a, q, axis=None, out=None, overwrite_input=False, method='linear', keepdims=False, *, interpolation=None)`

Computes the q-th percentile of the data along the specified axis.

**Parameters**

- **a** (*cupy.ndarray*) – Array for which to compute percentiles.
- **q** (*float, tuple of floats or cupy.ndarray*) – Percentiles to compute in the range between 0 and 100 inclusive.
- **axis** (*int or tuple of ints*) – Along which axis or axes to compute the percentiles. The flattened array is used by default.
- **out** (*cupy.ndarray*) – Output array.
- **overwrite\_input** (*bool*) – If True, then allow the input array *a* to be modified by the intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.
- **method** (*str*) – Interpolation method when a quantile lies between two data points. `linear` interpolation is used by default. Supported interpolations are `lower`, `higher`, `midpoint`, `nearest` and `linear`.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.
- **interpolation** (*str*) – Deprecated name for the method keyword argument.

**Returns**

The percentiles of `a`, along the axis if specified.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.percentile()`

## cupy.quantile

`cupy.quantile(a, q, axis=None, out=None, overwrite_input=False, method='linear', keepdims=False, *, interpolation=None)`

Computes the q-th quantile of the data along the specified axis.

### Parameters

- **a** (`cupy.ndarray`) – Array for which to compute quantiles.
- **q** (`float`, *tuple of floats* or `cupy.ndarray`) – Quantiles to compute in the range between 0 and 1 inclusive.
- **axis** (`int` or *tuple of ints*) – Along which axis or axes to compute the quantiles. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **overwrite\_input** (`bool`) – If True, then allow the input array *a* to be modified by the intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.
- **method** (`str`) – Interpolation method when a quantile lies between two data points. `linear` interpolation is used by default. Supported interpolations are `lower`, `higher`, `midpoint`, `nearest` and `linear`.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.
- **interpolation** (`str`) – Deprecated name for the method keyword argument.

### Returns

The quantiles of *a*, along the axis if specified.

### Return type

`cupy.ndarray`

See also:

`numpy.quantile()`

## Averages and variances

<code>median(a[, axis, out, overwrite_input, keepdims])</code>	Compute the median along the specified axis.
<code>average(a[, axis, weights, returned, keepdims])</code>	Returns the weighted average along an axis.
<code>mean(a[, axis, dtype, out, keepdims])</code>	Returns the arithmetic mean along an axis.
<code>std(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the standard deviation along an axis.
<code>var(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the variance along an axis.
<code>nanmedian(a[, axis, out, overwrite_input, ...])</code>	Compute the median along the specified axis, while ignoring NaNs.
<code>nanmean(a[, axis, dtype, out, keepdims])</code>	Returns the arithmetic mean along an axis ignoring NaN values.
<code>nanstd(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the standard deviation along an axis ignoring NaN values.
<code>nanvar(a[, axis, dtype, out, ddof, keepdims])</code>	Returns the variance along an axis ignoring NaN values.

## cupy.median

`cupy.median(a, axis=None, out=None, overwrite_input=False, keepdims=False)`

Compute the median along the specified axis.

Returns the median of the array elements.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute the median.
- **axis** (`int`, *sequence of int or None*) – Axis along which the medians are computed. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **overwrite\_input** (`bool`) – If `True`, then allow use of memory of input array `a` for calculations. The input array will be modified by the call to `median`. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is `False`. If `overwrite_input` is `True` and `a` is not already an ndarray, an error will be raised.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

### Returns

The median of `a`, along the axis if specified.

### Return type

`cupy.ndarray`

See also:

`numpy.median()`

## cupy.average

`cupy.average(a, axis=None, weights=None, returned=False, *, keepdims=False)`

Returns the weighted average along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute average.
- **axis** (`int`) – Along which axis to compute average. The flattened array is used by default.
- **weights** (`cupy.ndarray`) – Array of weights where each element corresponds to the value in `a`. If `None`, all the values in `a` have a weight equal to one.
- **returned** (`bool`) – If `True`, a tuple of the average and the sum of weights is returned, otherwise only the average is returned.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

### Returns

The average of the input array along the axis and the sum of weights.

### Return type

`cupy.ndarray` or tuple of `cupy.ndarray`



**Warning:** This function may synchronize the device if `weight` is given.

See also:

`numpy.average()`

## cupy.mean

`cupy.mean(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the arithmetic mean along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute mean.
- **axis** (`int`, *sequence of int or None*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

### Returns

The mean of the input array along the axis.

### Return type

`cupy.ndarray`

See also:

`numpy.mean()`

## cupy.std

`cupy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the standard deviation along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute standard deviation.
- **axis** (`int`) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If True, the axis is remained as an axis of size one.

### Returns

The standard deviation of the input array along the axis.

### Return type

`cupy.ndarray`

See also:

`numpy.std()`

## cupy.var

`cupy.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the variance along an axis.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute variance.
- **axis** (`int`) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

### Returns

The variance of the input array along the axis.

### Return type

`cupy.ndarray`

### See also:

`numpy.var()`

## cupy.nanmedian

`cupy.nanmedian(a, axis=None, out=None, overwrite_input=False, keepdims=False)`

Compute the median along the specified axis, while ignoring NaNs.

Returns the median of the array elements.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute the median.
- **axis** (`int`, *sequence of `int` or `None`*) – Axis along which the medians are computed. The flattened array is used by default.
- **out** (`cupy.ndarray`) – Output array.
- **overwrite\_input** (`bool`) – If `True`, then allow use of memory of input array `a` for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is `False`. If `overwrite_input` is `True` and `a` is not already an ndarray, an error will be raised.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

### Returns

The median of `a`, along the axis if specified.

### Return type

`cupy.ndarray`

### See also:

`numpy.nanmedian()`

**cupy.nanmean**

`cupy.nanmean(a, axis=None, dtype=None, out=None, keepdims=False)`

Returns the arithmetic mean along an axis ignoring NaN values.

**Parameters**

- **a** (`cupy.ndarray`) – Array to compute mean.
- **axis** (`int`, *sequence of int or None*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns**

The mean of the input array along the axis ignoring NaNs.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.nanmean()`

**cupy.nanstd**

`cupy.nanstd(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the standard deviation along an axis ignoring NaN values.

**Parameters**

- **a** (`cupy.ndarray`) – Array to compute standard deviation.
- **axis** (`int`) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

**Returns**

The standard deviation of the input array along the axis.

**Return type**

`cupy.ndarray`

**See also:**

`numpy.nanstd()`

## cupy.nanvar

`cupy.nanvar(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Returns the variance along an axis ignoring NaN values.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute variance.
- **axis** (`int`) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (`bool`) – If `True`, the axis is remained as an axis of size one.

### Returns

The variance of the input array along the axis.

### Return type

`cupy.ndarray`

See also:

`numpy.nanvar()`

## Correlations

<code>corrcoef(a[, y, rowvar, bias, ddof, dtype])</code>	Returns the Pearson product-moment correlation coefficients of an array.
<code>correlate(a, v[, mode])</code>	Returns the cross-correlation of two 1-dimensional sequences.
<code>cov(a[, y, rowvar, bias, ddof, fweights, ...])</code>	Returns the covariance matrix of an array.

## cupy.corrcoef

`cupy.corrcoef(a, y=None, rowvar=True, bias=None, ddof=None, *, dtype=None)`

Returns the Pearson product-moment correlation coefficients of an array.

### Parameters

- **a** (`cupy.ndarray`) – Array to compute the Pearson product-moment correlation coefficients.
- **y** (`cupy.ndarray`) – An additional set of variables and observations.
- **rowvar** (`bool`) – If `True`, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (`None`) – Has no effect, do not use.
- **ddof** (`None`) – Has no effect, do not use.
- **dtype** – Data type specifier. By default, the return data-type will have at least `numpy.float64` precision.

### Returns

The Pearson product-moment correlation coefficients of the input array.

**Return type***cupy.ndarray***See also:**`numpy.corrcoef()`**cupy.correlate**`cupy.correlate(a, v, mode='valid')`

Returns the cross-correlation of two 1-dimensional sequences.

**Parameters**

- **a** (`cupy.ndarray`) – first 1-dimensional input.
- **v** (`cupy.ndarray`) – second 1-dimensional input.
- **mode** (*str*, optional) – *valid*, *same*, *full*

**Returns**

Discrete cross-correlation of a and v.

**Return type***cupy.ndarray***See also:**`numpy.correlate()`**cupy.cov**`cupy.cov(a, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None, *, dtype=None)`

Returns the covariance matrix of an array.

This function currently does not support `fweights` and `aweights` options.**Parameters**

- **a** (`cupy.ndarray`) – Array to compute covariance matrix.
- **y** (`cupy.ndarray`) – An additional set of variables and observations.
- **rowvar** (*bool*) – If `True`, then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed.
- **bias** (*bool*) – If `False`, normalization is by  $(N - 1)$ , where  $N$  is the number of observations given (unbiased estimate). If `True`, then normalization is by  $N$ .
- **ddof** (*int*) – If not `None` the default value implied by `bias` is overridden. Note that `ddof=1` will return the unbiased estimate and `ddof=0` will return the simple average.
- **fweights** (`cupy.ndarray`, *int*) – 1-D array of integer frequency weights. the number of times each observation vector should be repeated. It is required that `fweights >= 0`. However, the function will not error when `fweights < 0` for performance reasons.
- **aweights** (`cupy.ndarray`) – 1-D array of observation vector weights. These relative weights are typically large for observations considered “important” and smaller for observations considered less “important”. If `ddof=0` the array of weights can be used to assign probabilities to observation vectors. It is required that `aweights >= 0`. However, the function will not error when `aweights < 0` for performance reasons.

- **dtype** – Data type specifier. By default, the return data-type will have at least *numpy.float64* precision.

**Returns**

The covariance matrix of the input array.

**Return type**

*cupy.ndarray*

**See also:**

*numpy.cov()*

## Histograms

<i>histogram</i> (x[, bins, range, weights, density])	Computes the histogram of a set of data.
<i>histogram2d</i> (x, y[, bins, range, weights, ...])	Compute the bi-dimensional histogram of two data samples.
<i>histogramdd</i> (sample[, bins, range, weights, ...])	Compute the multidimensional histogram of some data.
<i>bincount</i> (x[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
<i>digitize</i> (x, bins[, right])	Finds the indices of the bins to which each value in input array belongs.

## cupy.histogram

**cupy.histogram**(x, bins=10, range=None, weights=None, density=False)

Computes the histogram of a set of data.

**Parameters**

- **x** (*cupy.ndarray*) – Input array.
- **bins** (*int* or *cupy.ndarray*) – If bins is an int, it represents the number of bins. If bins is an *ndarray*, it represents a bin edges.
- **range** (*2-tuple of float, optional*) – The lower and upper range of the bins. If not provided, range is simply (*x.min()*, *x.max()*). Values outside the range are ignored. The first element of the range must be less than or equal to the second. *range* affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within *range*, the bin count will fill the entire range including portions containing no data.
- **density** (*bool, optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, *bin\_count* / *sample\_count* / *bin\_volume*.
- **weights** (*cupy.ndarray, optional*) – An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1).

**Returns**

(*hist*, *bin\_edges*) where *hist* is a *cupy.ndarray* storing the values of the histogram, and *bin\_edges* is a *cupy.ndarray* storing the bin edges.

**Return type**

*tuple*

**Warning:** This function may synchronize the device.

See also:

`numpy.histogram()`

## `cupy.histogram2d`

`cupy.histogram2d(x, y, bins=10, range=None, weights=None, density=None)`

Compute the bi-dimensional histogram of two data samples.

### Parameters

- **x** (`cupy.ndarray`) – The first array of samples to be histogrammed.
- **y** (`cupy.ndarray`) – The second array of samples to be histogrammed.
- **bins** (*int or tuple of int or cupy.ndarray*) – The bin specification:
  - A sequence of arrays describing the monotonically increasing bin edges along each dimension.
  - The number of bins for each dimension (nx, ny)
  - The number of bins for all dimensions (nx=ny=bins).
- **range** (*sequence, optional*) – A sequence of length two, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of None in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, None, is equivalent to passing a tuple of two None values.
- **weights** (`cupy.ndarray`) – An array of values  $w_i$  weighing each sample  $(x_i, y_i)$ . The values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.
- **density** (*bool, optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin,  $\text{bin\_count} / \text{sample\_count} / \text{bin\_volume}$ .

### Returns

#### **H** (`cupy.ndarray`):

The multidimensional histogram of sample x. See *normed* and *weights* for the different possible semantics.

#### **edges0** (*tuple of cupy.ndarray*):

A list of D arrays describing the bin edges for the first dimension.

#### **edges1** (*tuple of cupy.ndarray*):

A list of D arrays describing the bin edges for the second dimension.

### Return type

`tuple`

**Warning:** This function may synchronize the device.

See also:

`numpy.histogram2d()`

## `cupy.histogramdd`

`cupy.histogramdd(sample, bins=10, range=None, weights=None, density=False)`

Compute the multidimensional histogram of some data.

### Parameters

- **sample** (`cupy.ndarray`) – The data to be histogrammed. (N, D) or (D, N) array  
Note the unusual interpretation of sample when an array\_like:
  - When an array, each row is a coordinate in a D-dimensional space - such as `histogramdd(cupy.array([p1, p2, p3]))`.
  - When an array\_like, each element is the list of values for single coordinate - such as `histogramdd((X, Y, Z))`.The first form should be preferred.
- **bins** (`int` or `tuple of int` or `cupy.ndarray`) – The bin specification:
  - A sequence of arrays describing the monotonically increasing bin edges along each dimension.
  - The number of bins for each dimension (`nx, ny, ... =bins`)
  - The number of bins for all dimensions (`nx=ny=...=bins`).
- **range** (`sequence, optional`) – A sequence of length D, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of `None` in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, `None`, is equivalent to passing a tuple of D `None` values.
- **weights** (`cupy.ndarray`) – An array of values  $w_i$  weighing each sample ( $x_i, y_i, z_i, \dots$ ). The values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.
- **density** (`bool, optional`) – If `False`, the default, returns the number of samples in each bin. If `True`, returns the probability *density* function at the bin, `bin_count / sample_count / bin_volume`.

### Returns

#### **H** (`cupy.ndarray`):

The multidimensional histogram of sample *x*. See *normed* and *weights* for the different possible semantics.

#### **edges** (`list of cupy.ndarray`):

A list of D arrays describing the bin edges for each dimension.

### Return type

`tuple`

**Warning:** This function may synchronize the device.



See also:

`numpy.histogramdd()`

## `cupy.bincount`

`cupy.bincount(x, weights=None, minlength=None)`

Count number of occurrences of each value in array of non-negative ints.

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **weights** (`cupy.ndarray`) – Weights array which has the same shape as **x**.
- **minlength** (`int`) – A minimum number of bins for the output array.

### Returns

The result of binning the input array. The length of output is equal to `max(cupy.max(x) + 1, minlength)`.

### Return type

`cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`numpy.bincount()`

## `cupy.digitize`

`cupy.digitize(x, bins, right=False)`

Finds the indices of the bins to which each value in input array belongs.

---

**Note:** In order to avoid device synchronization, `digitize` does not raise an exception when the array is not monotonic

---

### Parameters

- **x** (`cupy.ndarray`) – Input array.
- **bins** (`cupy.ndarray`) – Array of bins. It has to be 1-dimensional and monotonic increasing or decreasing.
- **right** (`bool`) – Indicates whether the intervals include the right or the left bin edge.

### Returns

Output array of indices, of same shape as **x**.

### Return type

`cupy.ndarray`

See also:

`numpy.digitize()`

### 5.3.19 Test support (cupy.testing)

---

**Hint:** NumPy API Reference: Test support ([numpy.testing](#))

---

#### Asserts

---

**Hint:** These APIs can accept both [numpy.ndarray](#) and [cupy.ndarray](#).

---

<a href="#">assert_array_almost_equal</a> (x, y[, decimal, ...])	Raises an AssertionError if objects are not equal up to desired precision.
<a href="#">assert_allclose</a> (actual, desired[, rtol, ...])	Raises an AssertionError if objects are not equal up to desired tolerance.
<a href="#">assert_array_almost_equal_nulp</a> (x, y[, nulp])	Compare two arrays relatively to their spacing.
<a href="#">assert_array_max_ulp</a> (a, b[, maxulp, dtype])	Check that all items of arrays differ in at most N Units in the Last Place.
<a href="#">assert_array_equal</a> (x, y[, err_msg, verbose, ...])	Raises an AssertionError if two array_like objects are not equal.
<a href="#">assert_array_less</a> (x, y[, err_msg, verbose])	Raises an AssertionError if array_like objects are not ordered by less than.

#### cupy.testing.assert\_array\_almost\_equal

`cupy.testing.assert_array_almost_equal(x, y, decimal=6, err_msg="", verbose=True)`

Raises an AssertionError if objects are not equal up to desired precision.

##### Parameters

- **x** ([numpy.ndarray](#) or [cupy.ndarray](#)) – The actual object to check.
- **y** ([numpy.ndarray](#) or [cupy.ndarray](#)) – The desired, expected object.
- **decimal** ([int](#)) – Desired precision.
- **err\_msg** ([str](#)) – The error message to be printed in case of failure.
- **verbose** ([bool](#)) – If True, the conflicting values are appended to the error message.

See also:

[numpy.testing.assert\\_array\\_almost\\_equal\(\)](#)

### cupy.testing.assert\_allclose

`cupy.testing.assert_allclose(actual, desired, rtol=1e-07, atol=0, err_msg="", verbose=True)`

Raises an AssertionError if objects are not equal up to desired tolerance.

#### Parameters

- **actual** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **desired** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **rtol** (*float*) – Relative tolerance.
- **atol** (*float*) – Absolute tolerance.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_allclose()`

### cupy.testing.assert\_array\_almost\_equal\_nulp

`cupy.testing.assert_array_almost_equal_nulp(x, y, nulp=1)`

Compare two arrays relatively to their spacing.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.

See also:

`numpy.testing.assert_array_almost_equal_nulp()`

### cupy.testing.assert\_array\_max\_ulp

`cupy.testing.assert_array_max_ulp(a, b, maxulp=1, dtype=None)`

Check that all items of arrays differ in at most N Units in the Last Place.

#### Parameters

- **a** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **b** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **maxulp** (*int*) – The maximum number of units in the last place that elements of a and b can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert a and b to if given.

See also:

`numpy.testing.assert_array_max_ulp()`

### cupy.testing.assert\_array\_equal

`cupy.testing.assert_array_equal(x, y, err_msg="", verbose=True, strides_check=False, **kwargs)`

Raises an `AssertionError` if two array\_like objects are not equal.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The desired, expected object.
- **strides\_check** (*bool*) – If `True`, consistency of strides is also checked.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **strict** (*bool*) – If `True`, raise an `AssertionError` when either the shape or the data type of the array\_like objects does not match. Requires NumPy version 1.24 or above.

See also:

`numpy.testing.assert_array_equal()`

### cupy.testing.assert\_array\_less

`cupy.testing.assert_array_less(x, y, err_msg="", verbose=True)`

Raises an `AssertionError` if array\_like objects are not ordered by less than.

#### Parameters

- **x** (*numpy.ndarray* or *cupy.ndarray*) – The smaller object to check.
- **y** (*numpy.ndarray* or *cupy.ndarray*) – The larger object to compare.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_array_less()`

## CuPy-specific APIs

### Asserts

---

<code>assert_array_list_equal(xlist, ylist[, ...])</code>	Compares lists of arrays pairwise with <code>assert_array_equal</code> .
---	--

---

## cupy.testing.assert\_array\_list\_equal

`cupy.testing.assert_array_list_equal(xlist, ylist, err_msg="", verbose=True)`

Compares lists of arrays pairwise with `assert_array_equal`.

### Parameters

- **x** (*array\_like*) – Array of the actual objects.
- **y** (*array\_like*) – Array of the desired, expected objects.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

Each element of `x` and `y` must be either `numpy.ndarray` or `cupy.ndarray`. `x` and `y` must have same length. Otherwise, this function raises `AssertionError`. It compares elements of `x` and `y` pairwise with `assert_array_equal()` and raises error if at least one pair is not equal.

See also:

`numpy.testing.assert_array_equal()`

## NumPy-CuPy Consistency Check

The following decorators are for testing consistency between CuPy's functions and corresponding NumPy's ones.

<code>numpy_cupy_allclose([rtol, atol, err_msg, ...])</code>	Decorator that checks NumPy results and CuPy ones are close.
<code>numpy_cupy_array_almost_equal([decimal, ...])</code>	Decorator that checks NumPy results and CuPy ones are almost equal.
<code>numpy_cupy_array_almost_equal_nulp([nulp, ...])</code>	Decorator that checks results of NumPy and CuPy are equal w.r.t.
<code>numpy_cupy_array_max_ulp([maxulp, dtype, ...])</code>	Decorator that checks results of NumPy and CuPy ones are equal w.r.t.
<code>numpy_cupy_array_equal([err_msg, verbose, ...])</code>	Decorator that checks NumPy results and CuPy ones are equal.
<code>numpy_cupy_array_list_equal([err_msg, ...])</code>	Decorator that checks the resulting lists of NumPy and CuPy's one are equal.
<code>numpy_cupy_array_less([err_msg, verbose, ...])</code>	Decorator that checks the CuPy result is less than NumPy result.

## cupy.testing.numpy\_cupy\_allclose

`cupy.testing.numpy_cupy_allclose(rtol=1e-07, atol=0, err_msg="", verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None, contiguous_check=True, *, _check_sparse_format=True)`

Decorator that checks NumPy results and CuPy ones are close.

### Parameters

- **rtol** (*float* or *dict*) – Relative tolerance. Besides a float value, a dictionary that maps a dtypes to a float value can be supplied to adjust tolerance per dtype. If the dictionary has

'default' string as its key, its value is used as the default tolerance in case any dtype keys do not match.

- **atol** (*float* or *dict*) – Absolute tolerance. Besides a float value, a dictionary can be supplied as *rtol*.
- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If *True*, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either *numpy* or *cupy* module.
- **type\_check** (*bool*) – If *True*, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is *True* all error types are acceptable. If it is *False* no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either *scipy.sparse* or *cupyx.scipy.sparse* module. If *None*, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either *scipy* or *cupyx.scipy* module. If *None*, no argument is given for the modules.
- **contiguous\_check** (*bool*) – If *True*, consistency of contiguity is also checked.

Decorated test fixture is required to return the arrays whose values are close between *numpy* case and *cupy* case. For example, this test case checks *numpy.zeros* and *cupy.zeros* should return same value.

```
>>> import unittest
>>> from cupy import testing
>>> class TestFoo(unittest.TestCase):
...
...     @testing.numpy_cupy_allclose()
...     def test_foo(self, xp):
...         # ...
...         # Prepare data with xp
...         # ...
...
...         xp_result = xp.zeros(10)
...         return xp_result
```

See also:

[`cupy.testing.assert\_allclose\(\)`](#)

### `cupy.testing.numpy_cupy_array_almost_equal`

`cupy.testing.numpy_cupy_array_almost_equal(decimal=6, err_msg="", verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks NumPy results and CuPy ones are almost equal.

#### Parameters

- **decimal** (*int*) – Desired precision.
- **err\_msg** (*str*) – The error message to be printed in case of failure.

- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If True, consistency of dtype is also checked.
- **accept\_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is True all error types are acceptable. If it is False no error is acceptable.
- **sp\_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If None, no argument is given for the modules.
- **scipy\_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If None, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `cupy.testing.assert_array_almost_equal()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_almost_equal()`

### `cupy.testing.numpy_cupy_array_almost_equal_nulp`

```
cupy.testing.numpy_cupy_array_almost_equal_nulp(nulp=1, name='xp', type_check=True,
                                                accept_error=False, sp_name=None,
                                                scipy_name=None)
```

Decorator that checks results of NumPy and CuPy are equal w.r.t. spacing.

#### Parameters

- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If True, consistency of dtype is also checked.
- **accept\_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is True, all error types are acceptable. If it is False, no error is acceptable.
- **sp\_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If None, no argument is given for the modules.
- **scipy\_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If None, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `cupy.testing.assert_array_almost_equal_nulp()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_almost_equal_nulp()`

### cupy.testing.numpy\_cupy\_array\_max\_ulp

`cupy.testing.numpy_cupy_array_max_ulp(maxulp=1, dtype=None, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks results of NumPy and CuPy ones are equal w.r.t. ulp.

#### Parameters

- **maxulp** (*int*) – The maximum number of units in the last place that elements of resulting two arrays can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert the resulting two array to if given.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept\_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy\_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.

Decorated test fixture is required to return the same arrays in the sense of `assert_array_max_ulp()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_max_ulp()`

### cupy.testing.numpy\_cupy\_array\_equal

`cupy.testing.numpy_cupy_array_equal(err_msg='', verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None, strides_check=False)`

Decorator that checks NumPy results and CuPy ones are equal.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept\_error** (*bool, Exception or tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is `True` all error types are acceptable. If it is `False` no error is acceptable.
- **sp\_name** (*str or None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If `None`, no argument is given for the modules.
- **scipy\_name** (*str or None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If `None`, no argument is given for the modules.



- **strides\_check** (*bool*) – If True, consistency of strides is also checked.

Decorated test fixture is required to return the same arrays in the sense of `numpy_cupy_array_equal()` (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_equal()`

### `cupy.testing.numpy_cupy_array_list_equal`

`cupy.testing.numpy_cupy_array_list_equal(err_msg="", verbose=True, name='xp', sp_name=None, scipy_name=None)`

Decorator that checks the resulting lists of NumPy and CuPy's one are equal.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If *None*, no argument is given for the modules.
- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx.scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the same list of arrays (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

`cupy.testing.assert_array_list_equal()`

### `cupy.testing.numpy_cupy_array_less`

`cupy.testing.numpy_cupy_array_less(err_msg="", verbose=True, name='xp', type_check=True, accept_error=False, sp_name=None, scipy_name=None)`

Decorator that checks the CuPy result is less than NumPy result.

#### Parameters

- **err\_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type\_check** (*bool*) – If True, consistency of dtype is also checked.
- **accept\_error** (*bool*, *Exception* or *tuple of Exception*) – Specify acceptable errors. When both NumPy test and CuPy test raises the same type of errors, and the type of the errors is specified with this argument, the errors are ignored and not raised. If it is True all error types are acceptable. If it is False no error is acceptable.
- **sp\_name** (*str* or *None*) – Argument name whose value is either `scipy.sparse` or `cupyx.scipy.sparse` module. If *None*, no argument is given for the modules.

- **scipy\_name** (*str* or *None*) – Argument name whose value is either `scipy` or `cupyx`. `scipy` module. If *None*, no argument is given for the modules.

Decorated test fixture is required to return the smaller array when `xp` is `cupy` than the one when `xp` is `numpy`.

See also:

`cupy.testing.assert_array_less()`

## Parameterized dtype Test

The following decorators offer the standard way for parameterized test with respect to single or the combination of dtype(s).

<code>for_dtypes(dtypes[, name])</code>	Decorator for parameterized dtype test.
<code>for_all_dtypes([name, no_float16, no_bool, ...])</code>	Decorator that checks the fixture with all dtypes.
<code>for_float_dtypes([name, no_float16])</code>	Decorator that checks the fixture with float dtypes.
<code>for_signed_dtypes([name])</code>	Decorator that checks the fixture with signed dtypes.
<code>for_unsigned_dtypes([name])</code>	Decorator that checks the fixture with unsigned dtypes.
<code>for_int_dtypes([name, no_bool])</code>	Decorator that checks the fixture with integer and optionally bool dtypes.
<code>for_complex_dtypes([name])</code>	Decorator that checks the fixture with complex dtypes.
<code>for_dtypes_combination(types[, names, full])</code>	Decorator that checks the fixture with a product set of dtypes.
<code>for_all_dtypes_combination([names, ...])</code>	Decorator that checks the fixture with a product set of all dtypes.
<code>for_signed_dtypes_combination([names, full])</code>	Decorator for parameterized test w.r.t.
<code>for_unsigned_dtypes_combination([names, full])</code>	Decorator for parameterized test w.r.t.
<code>for_int_dtypes_combination([names, no_bool, ...])</code>	Decorator for parameterized test w.r.t.

## `cupy.testing.for_dtypes`

`cupy.testing.for_dtypes(dtypes, name='dtype')`

Decorator for parameterized dtype test.

### Parameters

- **dtypes** (*list of dtypes*) – dtypes to be tested.
- **name** (*str*) – Argument name to which specified dtypes are passed.

This decorator adds a keyword argument specified by `name` to the test fixture. Then, it runs the fixtures in parallel by passing the each element of `dtypes` to the named argument.

## cupy.testing.for\_all\_dtypes

`cupy.testing.for_all_dtypes(name='dtype', no_float16=False, no_bool=False, no_complex=False)`

Decorator that checks the fixture with all dtypes.

### Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.
- **no\_complex** (*bool*) – If True, `numpy.complex64` and `numpy.complex128` are omitted from candidate dtypes.

dtypes to be tested: `numpy.complex64` (optional), `numpy.complex128` (optional), `numpy.float16` (optional), `numpy.float32`, `numpy.float64`, `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

The usage is as follows. This test fixture checks if `cPickle` successfully reconstructs `cupy.ndarray` for various dtypes. `dtype` is an argument inserted by the decorator.

```
>>> import unittest
>>> from cupy import testing
>>> class TestNpz(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     def test_pickle(self, dtype):
...         a = testing.shaped_arange((2, 3, 4), dtype=dtype)
...         s = pickle.dumps(a)
...         b = pickle.loads(s)
...         testing.assert_array_equal(a, b)
```

Typically, we use this decorator in combination with decorators that check consistency between NumPy and CuPy like `cupy.testing.numpy_cupy_allclose()`. The following is such an example.

```
>>> import unittest
>>> from cupy import testing
>>> class TestMean(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     @testing.numpy_cupy_allclose()
...     def test_mean_all(self, xp, dtype):
...         a = testing.shaped_arange((2, 3), xp, dtype)
...         return a.mean()
```

See also:

`cupy.testing.for_dtypes()`

### `cupy.testing.for_float_dtypes`

`cupy.testing.for_float_dtypes(name='dtype', no_float16=False)`

Decorator that checks the fixture with float dtypes.

#### Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.

dtypes to be tested are `numpy.float16` (optional), `numpy.float32`, and `numpy.float64`.

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_signed_dtypes`

`cupy.testing.for_signed_dtypes(name='dtype')`

Decorator that checks the fixture with signed dtypes.

#### Parameters

**name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, and `numpy.dtype('q')`.

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_unsigned_dtypes`

`cupy.testing.for_unsigned_dtypes(name='dtype')`

Decorator that checks the fixture with unsigned dtypes.

#### Parameters

**name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('B')`, `numpy.dtype('H')`,  
`numpy.dtype('I')`, `numpy.dtype('L')`, and `numpy.dtype('Q')`.

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_int_dtypes`

`cupy.testing.for_int_dtypes(name='dtype', no_bool=False)`

Decorator that checks the fixture with integer and optionally bool dtypes.

#### Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_complex_dtypes`

`cupy.testing.for_complex_dtypes(name='dtype')`

Decorator that checks the fixture with complex dtypes.

#### Parameters

**name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.complex64` and `numpy.complex128`.

See also:

`cupy.testing.for_dtypes()`, `cupy.testing.for_all_dtypes()`

### `cupy.testing.for_dtypes_combination`

`cupy.testing.for_dtypes_combination(types, names=('dtype'), full=None)`

Decorator that checks the fixture with a product set of dtypes.

#### Parameters

- **types** (*list of dtype*) – dtypes to be tested.
- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see the description below).

Decorator adds the keyword arguments specified by **names** to the test fixture. Then, it runs the fixtures in parallel with passing (possibly a subset of) the product set of dtypes. The range of dtypes is specified by **types**.

The combination of dtypes to be tested changes depending on the option **full**. If **full** is True, all combinations of **types** are tested. Sometimes, such an exhaustive test can be costly. So, if **full** is False, only a subset of possible combinations is randomly sampled. If **full** is None, the behavior is determined by an environment variable `CUPY_TEST_FULL_COMBINATION`. If the value is set to '1', it behaves as if **full**=True, and otherwise **full**=False.

### `cupy.testing.for_all_dtypes_combination`

`cupy.testing.for_all_dtypes_combination(names=('dtyes'), no_float16=False, no_bool=False, full=None, no_complex=False)`

Decorator that checks the fixture with a product set of all dtypes.

#### Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no\_float16** (*bool*) – If True, `numpy.float16` is omitted from candidate dtypes.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.

- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).
- **no\_complex** (*bool*) – If True, `numpy.complex64` and `numpy.complex128` are omitted from candidate dtypes.

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### **`cupy.testing.for_signed_dtypes_combination`**

`cupy.testing.for_signed_dtypes_combination(names=('dtype'), full=None)`

Decorator for parameterized test w.r.t. the product set of signed dtypes.

#### **Parameters**

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### **`cupy.testing.for_unsigned_dtypes_combination`**

`cupy.testing.for_unsigned_dtypes_combination(names=('dtype'), full=None)`

Decorator for parameterized test w.r.t. the product set of unsigned dtypes.

#### **Parameters**

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

[`cupy.testing.for\_dtypes\_combination\(\)`](#)

### **`cupy.testing.for_int_dtypes_combination`**

`cupy.testing.for_int_dtypes_combination(names=('dtype'), no_bool=False, full=None)`

Decorator for parameterized test w.r.t. the product set of int and boolean.

#### **Parameters**

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no\_bool** (*bool*) – If True, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If True, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in [`cupy.testing.for\_dtypes\_combination\(\)`](#)).

See also:

`cupy.testing.for_dtypes_combination()`

## Parameterized order Test

The following decorators offer the standard way to parameterize tests with orders.

<code>for_orders</code> (orders[, name])	Decorator to parameterize tests with order.
<code>for_CF_orders</code> ([name])	Decorator that checks the fixture with orders 'C' and 'F'.

### `cupy.testing.for_orders`

`cupy.testing.for_orders`(orders, name='order')

Decorator to parameterize tests with order.

#### Parameters

- **orders** (*list of order*) – orders to be tested.
- **name** (*str*) – Argument name to which the specified order is passed.

This decorator adds a keyword argument specified by **name** to the test fixtures. Then, the fixtures run by passing each element of **orders** to the named argument.

### `cupy.testing.for_CF_orders`

`cupy.testing.for_CF_orders`(name='order')

Decorator that checks the fixture with orders 'C' and 'F'.

#### Parameters

- **name** (*str*) – Argument name to which the specified order is passed.

See also:

`cupy.testing.for_all_dtypes()`

## 5.3.20 Window functions

---

**Hint:** [NumPy API Reference: Window functions](#)

---

## Various windows

<code>bartlett(M)</code>	Returns the Bartlett window.
<code>blackman(M)</code>	Returns the Blackman window.
<code>hamming(M)</code>	Returns the Hamming window.
<code>hanning(M)</code>	Returns the Hanning window.
<code>kaiser(M, beta)</code>	Return the Kaiser window.

### `cupy.bartlett`

`cupy.bartlett(M)`

Returns the Bartlett window.

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1} \left( \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right)$$

**Parameters**

**M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

Output ndarray.

**Return type**

ndarray

**See also:**

`numpy.bartlett()`

### `cupy.blackman`

`cupy.blackman(M)`

Returns the Blackman window.

The Blackman window is defined as

$$w(n) = 0.42 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) + 0.08 \cos\left(\frac{4\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters**

**M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

Output ndarray.

**Return type**

ndarray

**See also:**

`numpy.blackman()`



## cupy.hamming

`cupy.hamming(M)`

Returns the Hamming window.

The Hamming window is defined as

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters**

**M** (`int`) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

Output ndarray.

**Return type**

ndarray

**See also:**

`numpy.hamming()`

## cupy.hanning

`cupy.hanning(M)`

Returns the Hanning window.

The Hanning window is defined as

$$w(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

**Parameters**

**M** (`int`) – Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

Output ndarray.

**Return type**

ndarray

**See also:**

`numpy.hanning()`

## cupy.kaiser

`cupy.kaiser(M, beta)`

Return the Kaiser window. The Kaiser window is a taper formed by using a Bessel function.

$$w(n) = I_0\left(\beta \sqrt{1 - \frac{4n^2}{(M-1)^2}}\right) / I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2}$$

where  $I_0$  is the modified zeroth-order Bessel function.

**Args:**

**M (int):**

Number of points in the output window. If zero or less, an empty array is returned.

**beta (float):**

Shape parameter for window

**Returns**

The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

**Return type**

`ndarray`

**See also:**

`numpy.kaiser()`

## 5.4 Routines (SciPy)

The following pages describe SciPy-compatible routines. These functions cover a subset of [SciPy routines](#).

### 5.4.1 Discrete Fourier transforms (`cupyx.scipy.fft`)

---

**Hint:** [SciPy API Reference: Discrete Fourier transforms \(scipy.fft\)](#)

---

**See also:**

*Fast Fourier Transform with CuPy*

## Fast Fourier Transforms (FFTs)

<code>fft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional FFT.
<code>ifft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional FFT.
<code>ifft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional FFT.
<code>ifftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT.
<code>rfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT for real input.
<code>rfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional FFT for real input.
<code>irfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT for real input.
<code>rfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional FFT for real input.
<code>irfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT for real input.
<code>hfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the FFT of a signal that has Hermitian symmetry.
<code>ihfft(x[, n, axis, norm, overwrite_x, plan])</code>	Compute the FFT of a signal that has Hermitian symmetry.
<code>hfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the FFT of a two-dimensional signal that has Hermitian symmetry.
<code>ihfft2(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the Inverse FFT of a two-dimensional signal that has Hermitian symmetry.
<code>hfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the FFT of a N-dimensional signal that has Hermitian symmetry.
<code>ihfftn(x[, s, axes, norm, overwrite_x, plan])</code>	Compute the Inverse FFT of a N-dimensional signal that has Hermitian symmetry.

### cupyx.scipy.fft.fft

`cupyx.scipy.fft.fft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns**

The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.fft.fft()`

**cupyx.scipy.fft.iff**

`cupyx.scipy.fft.iff(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional inverse FFT.

**Parameters**

- **x** (*cupy.ndarray*) – Array to be transformed.
- **n** (*None* or *int*) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (*cupy.cuda.cufft.Plan1d* or *None*) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

**Returns**

The transformed array which shape is specified by `n` and type will convert to complex if that of the input is another.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.fft.iff()`

**cupyx.scipy.fft.fft2**

`cupyx.scipy.fft.fft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the two-dimensional FFT.

**Parameters**

- **x** (*cupy.ndarray*) – Array to be transformed.
- **s** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.

- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

#### See also:

`scipy.fft.fft2()`

### cupyx.scipy.fft.iff2

`cupyx.scipy.fft.iff2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the two-dimensional inverse FFT.

#### Parameters

- **x** (*cupy.ndarray*) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.ifft2()`

### `cupyx.scipy.fft.fftn`

`cupyx.scipy.fft.fftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that **plan** is defaulted to *None*, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by **s** and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.fftn()`

### `cupyx.scipy.fft.ifftn`

`cupyx.scipy.fft.ifftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional inverse FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the transformed axes of the output. If **s** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.

- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes)
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.ifftn()`

### cupyx.scipy.fft.rfft

`cupyx.scipy.fft.rfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional FFT for real input.

The returned array contains the positive frequency components of the corresponding `fft()`, up to and including the Nyquist frequency.

#### Parameters

- **x** (*cupy.ndarray*) – Array to be transformed.
- **n** (*None* or *int*) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis,
                                         value_type='R2C')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array.

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.rfft()`

## cupyx.scipy.fft.irfft

`cupyx.scipy.fft.irfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the one-dimensional inverse FFT for real input.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, n, axis,
                                         value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

### Returns

The transformed array.

### Return type

`cupy.ndarray`

### See also:

`scipy.fft.irfft()`

## cupyx.scipy.fft.rfft2

`cupyx.scipy.fft.rfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the two-dimensional FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (`None` or `tuple of ints`) – Shape to use from the input. If `s` is not given, the lengths of the input along the axes specified by `axes` are used.
- **axes** (`tuple of ints`) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:



```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='R2C')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]//2+1$ .

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.rfft2()`

### cupyx.scipy.fft.irfft2

`cupyx.scipy.fft.irfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the two-dimensional inverse FFT for real input.

#### Parameters

- **a** (*cupy.ndarray*) – Array to be transform.
- **s** (*None or tuple of ints*) – Shape of the output. If `s` is not given, they are determined from the lengths of the input along the axes specified by `axes`.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (*cupy.cuda.cufft.PlanNd or None*) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be  $2*(m-1)$  where  $m$  is the length of the final transformed axis of the input.

#### Return type

*cupy.ndarray*

See also:

`scipy.fft.irfft2()`

## cupyx.scipy.fft.rfftn

`cupyx.scipy.fft.rfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None* or *tuple of ints*) – Shape to use from the input. If *s* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of *x* can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming *x* over *axes*, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                         value_type='R2C')
```

Note that *plan* is defaulted to *None*, meaning CuPy will use an auto-generated plan behind the scene.

### Returns

The transformed array which shape is specified by *s* and type will convert to complex if the input is other. The length of the last axis transformed will be  $s[-1]//2+1$ .

### Return type

*cupy.ndarray*

See also:

`scipy.fft.rfftn()`

## cupyx.scipy.fft.irfftn

`cupyx.scipy.fft.irfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the N-dimensional inverse FFT for real input.

### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **s** (*None* or *tuple of ints*) – Shape of the output. If *s* is not given, they are determined from the lengths of the input along the axes specified by *axes*.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of *x* can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming *x* over *axes*, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, s, axes,
                                       value_type='C2R')
```

Note that `plan` is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by `s` and type will convert to complex if the input is other. If `s` is not given, the length of final transformed axis of output will be  $2*(m-1)$  where  $m$  is the length of the final transformed axis of the input.

#### Return type

*cupy.ndarray*

#### See also:

`scipy.fft.irfftn()`

### cupyx.scipy.fft.hfft

`cupyx.scipy.fft.hfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (*cupy.ndarray*) – Array to be transform.
- **n** (*None* or *int*) – Length of the transformed axis of the output. For  $n$  output points,  $n//2+1$  input points are necessary. If  $n$  is not given, it is determined from the length of the input along the axis specified by `axis`.
- **axis** (*int*) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (*None*) – This argument is currently not supported.

#### Returns

The transformed array which shape is specified by `n` and type will convert to complex if the input is other. If `n` is not given, the length of the transformed axis is  $2*(m-1)$  where  $m$  is the length of the transformed axis of the input.

#### Return type

*cupy.ndarray*

#### See also:

`scipy.fft.hfft()`

### cupyx.scipy.fft.ihfft

`cupyx.scipy.fft.ihfft(x, n=None, axis=-1, norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a signal that has Hermitian symmetry.

#### Parameters

- **a** (`cupy.ndarray`) – Array to be transform.
- **n** (*None* or `int`) – Number of points along transformation axis in the input to use. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If True, the contents of **x** can be destroyed.
- **plan** (*None*) – This argument is currently not supported.

#### Returns

The transformed array which shape is specified by **n** and type will convert to complex if the input is other. The length of the transformed axis is  $n//2+1$ .

#### Return type

`cupy.ndarray`

See also:

`scipy.fft.ihfft()`

### cupyx.scipy.fft.hfft2

`cupyx.scipy.fft.hfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a two-dimensional signal that has Hermitian symmetry.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None* or *tuple of ints*) – Shape of the real output.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (`bool`) – If True, the contents of **x** can be destroyed. (This argument is currently not supported)
- **plan** (*None*) – This argument is currently not supported.

#### Returns

The real result of the 2-D Hermitian complex real FFT.

#### Return type

`cupy.ndarray`

See also:

`scipy.fft.hfft2()`

**cupyx.scipy.fft.ihfft2**

`cupyx.scipy.fft.ihfft2(x, s=None, axes=(-2, -1), norm=None, overwrite_x=False, *, plan=None)`

Compute the Inverse FFT of a two-dimensional signal that has Hermitian symmetry.

**Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the real output.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed. (This argument is currently not supported)
- **plan** (*None*) – This argument is currently not supported.

**Returns**

The real result of the 2-D Hermitian inverse complex real FFT.

**Return type**

*cupy.ndarray*

See also:

`scipy.fft.ihfft2()`

**cupyx.scipy.fft.hfftn**

`cupyx.scipy.fft.hfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the FFT of a N-dimensional signal that has Hermitian symmetry.

**Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the real output.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is *None*, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed. (This argument is currently not supported)
- **plan** (*None*) – This argument is currently not supported.

**Returns**

The real result of the N-D Hermitian complex real FFT.

**Return type**

*cupy.ndarray*

See also:

`scipy.fft.hfftn()`

## cupyx.scipy.fft.ihfftn

`cupyx.scipy.fft.ihfftn(x, s=None, axes=None, norm=None, overwrite_x=False, *, plan=None)`

Compute the Inverse FFT of a N-dimensional signal that has Hermitian symmetry.

### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **s** (*None or tuple of ints*) – Shape of the real output.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **norm** ("backward", "ortho", or "forward") – Optional keyword to specify the normalization mode. Default is `None`, which is an alias of "backward".
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed. (This argument is currently not supported)
- **plan** (*None*) – This argument is currently not supported.

### Returns

The real result of the N-D Hermitian inverse complex real FFT.

### Return type

*cupy.ndarray*

See also:

`scipy.fft.ihfftn()`

## Discrete Cosine and Sine Transforms (DST and DCT)

<code>dct(x[, type, n, axis, norm, overwrite_x])</code>	Return the Discrete Cosine Transform of an array, x.
<code>idct(x[, type, n, axis, norm, overwrite_x])</code>	Return the Inverse Discrete Cosine Transform of an array, x.
<code>dctn(x[, type, s, axes, norm, overwrite_x])</code>	Compute a multidimensional Discrete Cosine Transform.
<code>idctn(x[, type, s, axes, norm, overwrite_x])</code>	Compute a multidimensional Discrete Cosine Transform.
<code>dst(x[, type, n, axis, norm, overwrite_x])</code>	Return the Discrete Sine Transform of an array, x.
<code>idst(x[, type, n, axis, norm, overwrite_x])</code>	Return the Inverse Discrete Sine Transform of an array, x.
<code>dstn(x[, type, s, axes, norm, overwrite_x])</code>	Compute a multidimensional Discrete Sine Transform.
<code>idstn(x[, type, s, axes, norm, overwrite_x])</code>	Compute a multidimensional Discrete Sine Transform.

## cupyx.scipy.fft.dct

`cupyx.scipy.fft.dct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`

Return the Discrete Cosine Transform of an array, x.

### Parameters

- **x** (`cupy.ndarray`) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DCT (see Notes). Default type is 2. Currently CuPy only supports types 2 and 3.

- **n** (*int*, *optional*:) – Length of the transform. If  $n < x.shape[axis]$ ,  $x$  is truncated. If  $n > x.shape[axis]$ ,  $x$  is zero-padded. The default results in  $n = x.shape[axis]$ .
- **axis** (*int*, *optional*) – Axis along which the dct is computed; the default is over the last axis (i.e.,  $axis=-1$ ).
- **norm** (*{"backward", "ortho", "forward"}*, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (*bool*, *optional*) – If True, the contents of  $x$  can be destroyed; the default is False.

**Returns**

**y** – The transformed input array.

**Return type**

*cupy.ndarray* of real

See also:

`scipy.fft.dct()`

**Notes**

For a single dimension array  $x$ , `dct(x, norm='ortho')` is equal to MATLAB `dct(x)`.

For `norm="ortho"` both the *dct* and *idct* are scaled by the same overall factor in both directions. By default, the transform is also orthogonalized which for types 1, 2 and 3 means the transform definition is modified to give orthogonality of the DCT matrix (see below).

For `norm="backward"`, there is no scaling on *dct* and the *idct* is scaled by  $1/N$  where  $N$  is the “logical” size of the DCT. For `norm="forward"` the  $1/N$  normalization is applied to the forward *dct* instead and the *idct* is unnormalized.

CuPy currently only supports DCT types 2 and 3. ‘The’ DCT generally refers to DCT type 2, and ‘the’ Inverse DCT generally refers to DCT type 3<sup>1</sup>. See the `scipy.fft.dct()` documentation for a full description of each type.

**References****cupyx.scipy.fft.idct**

`cupyx.scipy.fft.idct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`

Return the Inverse Discrete Cosine Transform of an array,  $x$ .

**Parameters**

- **x** (*cupy.ndarray*) – The input array.
- **type** (*{1, 2, 3, 4}*, *optional*) – Type of the DCT (see Notes). Default type is 2.
- **n** (*int*, *optional*) – Length of the transform. If  $n < x.shape[axis]$ ,  $x$  is truncated. If  $n > x.shape[axis]$ ,  $x$  is zero-padded. The default results in  $n = x.shape[axis]$ .
- **axis** (*int*, *optional*) – Axis along which the idct is computed; the default is over the last axis (i.e.,  $axis=-1$ ).
- **norm** (*{"backward", "ortho", "forward"}*, *optional*) – Normalization mode (see Notes). Default is “backward”.

<sup>1</sup> Wikipedia, “Discrete cosine transform”, [https://en.wikipedia.org/wiki/Discrete\\_cosine\\_transform](https://en.wikipedia.org/wiki/Discrete_cosine_transform)

- **overwrite\_x** (*bool*, *optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**idct** – The transformed input array.

**Return type**

*cupy.ndarray* of real

**See also:**

`scipy.fft.idct()`

**Notes**

For a single dimension array *x*, `idct(x, norm='ortho')` is equal to MATLAB `idct(x)`.

For `norm="ortho"` both the *dct* and *idct* are scaled by the same overall factor in both directions. By default, the transform is also orthogonalized which for types 1, 2 and 3 means the transform definition is modified to give orthogonality of the IDCT matrix (see *dct* for the full definitions).

‘The’ IDCT is the IDCT-II, which is the same as the normalized DCT-III<sup>1</sup>. See the `scipy.fft.dct()` documentation for a full description of each type. CuPy currently only supports DCT types 2 and 3.

**References****cupyx.scipy.fft.dctn**

`cupyx.scipy.fft.dctn(x, type=2, s=None, axes=None, norm=None, overwrite_x=False)`

Compute a multidimensional Discrete Cosine Transform.

**Parameters**

- **x** (*cupy.ndarray*) – The input array.
- **type** (*{1, 2, 3, 4}*, *optional*) – Type of the DCT (see Notes). Default type is 2.
- **s** (*int or array\_like of ints or None*, *optional*) – The shape of the result. If both *s* and *axes* (see below) are None, *s* is *x.shape*; if *s* is None but *axes* is not None, then *s* is `numpy.take(x.shape, axes, axis=0)`. If *s*[*i*] > *x.shape*[*i*], the *i*th dimension is padded with zeros. If *s*[*i*] < *x.shape*[*i*], the *i*th dimension is truncated to length *s*[*i*]. If any element of *s* is -1, the size of the corresponding dimension of *x* is used.
- **axes** (*int or array\_like of ints or None*, *optional*) – Axes over which the DCT is computed. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.
- **norm** (*{"backward", "ortho", "forward"}*, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (*bool*, *optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**y** – The transformed input array.

**Return type**

*cupy.ndarray* of real

---

<sup>1</sup> Wikipedia, “Discrete sine transform”, [https://en.wikipedia.org/wiki/Discrete\\_sine\\_transform](https://en.wikipedia.org/wiki/Discrete_sine_transform)



See also:

`scipy.fft.dctn()`

## Notes

For full details of the DCT types and normalization modes, as well as references, see *dct*.

## cupyx.scipy.fft.idctn

`cupyx.scipy.fft.idctn(x, type=2, s=None, axes=None, norm=None, overwrite_x=False)`

Compute a multidimensional Discrete Cosine Transform.

### Parameters

- **x** (`cupy.ndarray`) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DCT (see Notes). Default type is 2.
- **s** (`int` or *array\_like of ints* or `None`, *optional*) – The shape of the result. If both *s* and *axes* (see below) are `None`, *s* is `x.shape`; if *s* is `None` but *axes* is not `None`, then *s* is `numpy.take(x.shape, axes, axis=0)`. If `s[i] > x.shape[i]`, the *i*th dimension is padded with zeros. If `s[i] < x.shape[i]`, the *i*th dimension is truncated to length `s[i]`. If any element of *s* is -1, the size of the corresponding dimension of *x* is used.
- **axes** (`int` or *array\_like of ints* or `None`, *optional*) – Axes over which the IDCT is computed. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.
- **norm** (`{"backward", "ortho", "forward"}`, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (`bool`, *optional*) – If `True`, the contents of *x* can be destroyed; the default is `False`.

### Returns

**y** – The transformed input array.

### Return type

`cupy.ndarray` of real

See also:

`scipy.fft.idctn()`

## Notes

For full details of the IDCT types and normalization modes, as well as references, see `scipy.fft.idct()`.

## cupyx.scipy.fft.dst

`cupyx.scipy.fft.dst(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`

Return the Discrete Sine Transform of an array, *x*.

### Parameters

- **x** (`cupy.ndarray`) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DST (see Notes). Default type is 2.
- **n** (`int`, *optional*) – Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (`int`, *optional*) – Axis along which the dst is computed; the default is over the last axis (i.e., `axis=-1`).
- **norm** (`{"backward", "ortho", "forward"}`, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (`bool`, *optional*) – If True, the contents of *x* can be destroyed; the default is False.

### Returns

**dst** – The transformed input array.

### Return type

`cupy.ndarray` of real

See also:

`scipy.fft.dst()`

### Notes

For `norm="ortho"` both the *dst* and *idst* are scaled by the same overall factor in both directions. By default, the transform is also orthogonalized which for types 2 and 3 means the transform definition is modified to give orthogonality of the DST matrix (see below).

For `norm="backward"`, there is no scaling on the *dst* and the *idst* is scaled by  $1/N$  where *N* is the “logical” size of the DST.

See the `scipy.fft.dst()` documentation for a full description of each type. CuPy currently only supports DST types 2 and 3.

## cupyx.scipy.fft.idst

`cupyx.scipy.fft.idst(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`

Return the Inverse Discrete Sine Transform of an array, *x*.

### Parameters

- **x** (`cupy.ndarray`) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DST (see Notes). Default type is 2.
- **n** (`int`, *optional*) – Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

- **axis** (*int*, *optional*) – Axis along which the idst is computed; the default is over the last axis (i.e., `axis=-1`).
- **norm** (`{"backward", "ortho", "forward"}`, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (*bool*, *optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**idst** – The transformed input array.

**Return type**

*cupy.ndarray* of real

**See also:**

`scipy.fft.idst()`

**Notes**

For full details of the DST types and normalization modes, as well as references, see `scipy.fft.dst()`.

**cupyx.scipy.fft.dstn**

`cupyx.scipy.fft.dstn(x, type=2, s=None, axes=None, norm=None, overwrite_x=False)`

Compute a multidimensional Discrete Sine Transform.

**Parameters**

- **x** (*cupy.ndarray*) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DST (see Notes). Default type is 2.
- **s** (*int or array\_like of ints or None*, *optional*) – The shape of the result. If both *s* and *axes* (see below) are None, *s* is `x.shape`; if *s* is None but *axes* is not None, then *s* is `numpy.take(x.shape, axes, axis=0)`. If `s[i] > x.shape[i]`, the *i*th dimension is padded with zeros. If `s[i] < x.shape[i]`, the *i*th dimension is truncated to length `s[i]`. If any element of *s* is -1, the size of the corresponding dimension of *x* is used.
- **axes** (*int or array\_like of ints or None*, *optional*) – Axes over which the DST is computed. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.
- **norm** (`{"backward", "ortho", "forward"}`, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (*bool*, *optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**y** – The transformed input array.

**Return type**

*cupy.ndarray* of real

**See also:**

`scipy.fft.dstn()`

## Notes

For full details of the DST types and normalization modes, as well as references, see `scipy.fft.dst()`.

### cupyx.scipy.fft.idstn

`cupyx.scipy.fft.idstn(x, type=2, s=None, axes=None, norm=None, overwrite_x=False)`

Compute a multidimensional Discrete Sine Transform.

#### Parameters

- **x** (`cupy.ndarray`) – The input array.
- **type** (`{1, 2, 3, 4}`, *optional*) – Type of the DST (see Notes). Default type is 2.
- **s** (*int or array\_like of ints or None, optional*) – The shape of the result. If both *s* and *axes* (see below) are `None`, *s* is `x.shape`; if *s* is `None` but *axes* is not `None`, then *s* is `numpy.take(x.shape, axes, axis=0)`. If `s[i] > x.shape[i]`, the *i*th dimension is padded with zeros. If `s[i] < x.shape[i]`, the *i*th dimension is truncated to length `s[i]`. If any element of *s* is `-1`, the size of the corresponding dimension of *x* is used.
- **axes** (*int or array\_like of ints or None, optional*) – Axes over which the IDST is computed. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.
- **norm** (`{"backward", "ortho", "forward"}`, *optional*) – Normalization mode (see Notes). Default is “backward”.
- **overwrite\_x** (*bool, optional*) – If `True`, the contents of *x* can be destroyed; the default is `False`.

#### Returns

**y** – The transformed input array.

#### Return type

`cupy.ndarray` of real

See also:

`scipy.fft.idstn()`

## Notes

For full details of the IDST types and normalization modes, as well as references, see `scipy.fft.idst()`.

### Fast Hankel Transforms

---

<code>fht(a, dln, mu[, offset, bias])</code>	Compute the fast Hankel transform.
<code>ifht(A, dln, mu[, offset, bias])</code>	Compute the inverse fast Hankel transform.

---

## cupyx.scipy.fft.fht

`cupyx.scipy.fft.fht(a, dln, mu, offset=0.0, bias=0.0)`

Compute the fast Hankel transform.

Computes the discrete Hankel transform of a logarithmically spaced periodic sequence using the FFTLog algorithm<sup>1,2</sup>.

### Parameters

- **a** (`cupy.ndarray` (... , n)) – Real periodic input array, uniformly logarithmically spaced. For multidimensional input, the transform is performed over the last axis.
- **dln** (`float`) – Uniform logarithmic spacing of the input array.
- **mu** (`float`) – Order of the Hankel transform, any positive or negative real number.
- **offset** (`float`, *optional*) – Offset of the uniform logarithmic spacing of the output array.
- **bias** (`float`, *optional*) – Exponent of power law bias, any positive or negative real number.

### Returns

**A** – The transformed output array, which is real, periodic, uniformly logarithmically spaced, and of the same shape as the input array.

### Return type

`cupy.ndarray` (... , n)

See also:

`scipy.special.fht()`

`scipy.special.fhtoffset()`

Return an optimal offset for *fht*.

## References

## cupyx.scipy.fft.ifht

`cupyx.scipy.fft.ifht(A, dln, mu, offset=0.0, bias=0.0)`

Compute the inverse fast Hankel transform.

Computes the discrete inverse Hankel transform of a logarithmically spaced periodic sequence. This is the inverse operation to *fht*.

### Parameters

- **A** (`cupy.ndarray` (... , n)) – Real periodic input array, uniformly logarithmically spaced. For multidimensional input, the transform is performed over the last axis.
- **dln** (`float`) – Uniform logarithmic spacing of the input array.
- **mu** (`float`) – Order of the Hankel transform, any positive or negative real number.
- **offset** (`float`, *optional*) – Offset of the uniform logarithmic spacing of the output array.
- **bias** (`float`, *optional*) – Exponent of power law bias, any positive or negative real number.

<sup>1</sup> Talman J. D., 1978, J. Comp. Phys., 29, 35

<sup>2</sup> Hamilton A. J. S., 2000, MNRAS, 312, 257 (astro-ph/9905191)

**Returns**

**a** – The transformed output array, which is real, periodic, uniformly logarithmically spaced, and of the same shape as the input array.

**Return type**

*cupy.ndarray* (...) , n)

**See also:**

`scipy.special.ifht()`

`scipy.special.fhtoffset()`

Return an optimal offset for *fht*.

## Helper functions

<code>fftshift(x[, axes])</code>	Shift the zero-frequency component to the center of the spectrum.
<code>ifftshift(x[, axes])</code>	The inverse of <code>fftshift()</code> .
<code>fftfreq(n[, d])</code>	Return the FFT sample frequencies.
<code>rfftfreq(n[, d])</code>	Return the FFT sample frequencies for real input.
<code>next_fast_len(target[, real])</code>	Find the next fast size to <code>fft</code> .

## cupyx.scipy.fft.fftshift

`cupyx.scipy.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

**Parameters**

- **x** (*cupy.ndarray*) – Input array.
- **axes** (*int* or *tuple of ints*) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns**

The shifted array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.fft.fftshift()`

## cupyx.scipy.fft.ifftshift

`cupyx.scipy.fft.ifftshift(x, axes=None)`

The inverse of `fftshift()`.

**Parameters**

- **x** (*cupy.ndarray*) – Input array.
- **axes** (*int* or *tuple of ints*) – Axes over which to shift. Default is `None`, which shifts all axes.

**Returns**

The shifted array.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.fft.ifftshift()`

**cupyx.scipy.fft.fftfreq**

`cupyx.scipy.fft.fftfreq(n, d=1.0)`

Return the FFT sample frequencies.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns**

Array of length *n* containing the sample frequencies.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.fft.fftfreq()`

**cupyx.scipy.fft.rfftfreq**

`cupyx.scipy.fft.rfftfreq(n, d=1.0)`

Return the FFT sample frequencies for real input.

**Parameters**

- **n** (*int*) – Window length.
- **d** (*scalar*) – Sample spacing.

**Returns**

Array of length  $n//2+1$  containing the sample frequencies.

**Return type**

*cupy.ndarray*

**See also:**

`numpy.fft.rfftfreq()`

## cupyx.scipy.fft.next\_fast\_len

`cupyx.scipy.fft.next_fast_len(target, real=False)`

Find the next fast size to fft.

### Parameters

- **target** (*int*) – The size of input array.
- **real** (*bool*) – True if the FFT involves real input or output. This parameter is of no use, and only for compatibility to SciPy’s interface.

### Returns

The smallest fast length greater than or equal to the input value.

### Return type

*int*

See also:

`scipy.fft.next_fast_len()`

---

**Note:** It may return a different value to `scipy.fft.next_fast_len()` as pocketfft’s prime factors are different from cuFFT’s factors. For details, see the [cuFFT documentation](#).

---

## Code compatibility features

1. As with other FFT modules in CuPy, FFT functions in this module can take advantage of an existing cuFFT plan (returned by `get_fft_plan()`) to accelerate the computation. The plan can be either passed in explicitly via the keyword-only `plan` argument or used as a context manager. One exception to this are the DCT and DST transforms, which do not currently support a plan argument.
2. The boolean switch `cupy.fft.config.enable_nd_planning` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. This switch is neglected when planning manually using `get_fft_plan()`.
3. Like in `scipy.fft`, all FFT functions in this module have an optional argument `overwrite_x` (default is `False`), which has the same semantics as in `scipy.fft`: when it is set to `True`, the input array *x* *can* (not *will*) be overwritten arbitrarily. For this reason, when an in-place FFT is desired, the user should always reassign the input in the following manner: `x = cupyx.scipy.fftpack.fft(x, ..., overwrite_x=True, ...)`.
4. The `cupyx.scipy.fft` module can also be used as a backend for `scipy.fft` e.g. by installing with `scipy.fft.set_backend(cupyx.scipy.fft)`. This can allow `scipy.fft` to work with both `numpy` and `cupy` arrays. For more information, see *SciPy FFT backend*.
5. The boolean switch `cupy.fft.config.use_multi_gpus` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. Moreover, this switch is *honored* when planning manually using `get_fft_plan()`.
6. Both type II and III DCT and DST transforms are implemented. Type I and IV transforms are currently unavailable.



## 5.4.2 Legacy discrete fourier transforms (`cupyx.scipy.fftpack`)

**Note:** As of SciPy version 1.4.0, `scipy.fft` is recommended over `scipy.fftpack`. Consider using `cupyx.scipy.fft` instead.

**Hint:** SciPy API Reference: Legacy discrete Fourier transforms (`scipy.fftpack`)

### Fast Fourier Transforms (FFTs)

<code>fft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional FFT.
<code>ifft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional inverse FFT.
<code>fft2(x[, shape, axes, overwrite_x, plan])</code>	Compute the two-dimensional FFT.
<code>ifft2(x[, shape, axes, overwrite_x, plan])</code>	Compute the two-dimensional inverse FFT.
<code>fftn(x[, shape, axes, overwrite_x, plan])</code>	Compute the N-dimensional FFT.
<code>ifftn(x[, shape, axes, overwrite_x, plan])</code>	Compute the N-dimensional inverse FFT.
<code>rfft(x[, n, axis, overwrite_x, plan])</code>	Compute the one-dimensional FFT for real input.
<code>irfft(x[, n, axis, overwrite_x])</code>	Compute the one-dimensional inverse FFT for real input.
<code>get_fft_plan(a[, shape, axes, value_type])</code>	Generate a CUDA FFT plan for transforming up to three axes.

### `cupyx.scipy.fftpack.fft`

`cupyx.scipy.fftpack.fft(x, n=None, axis=-1, overwrite_x=False, plan=None)`

Compute the one-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming **x** over **axis**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### Returns

The transformed array which shape is specified by **n** and type will convert to complex if that of the input is another.

#### Return type

`cupy.ndarray`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

See also:

`scipy.fftpack.fft()`

### **cupyx.scipy.fftpack.ifft**

`cupyx.scipy.fftpack.ifft(x, n=None, axis=-1, overwrite_x=False, plan=None)`

Compute the one-dimensional inverse FFT.

#### **Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **n** (`None` or `int`) – Length of the transformed axis of the output. If **n** is not given, the length of the input along the axis specified by **axis** is used.
- **axis** (`int`) – Axis over which to compute the FFT.
- **overwrite\_x** (`bool`) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming **x** over **axis**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axis)
```

Note that *plan* is defaulted to `None`, meaning CuPy will use an auto-generated plan behind the scene.

#### **Returns**

The transformed array which shape is specified by **n** and type will convert to complex if that of the input is another.

#### **Return type**

`cupy.ndarray`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

See also:

`scipy.fftpack.ifft()`

### **cupyx.scipy.fftpack.fft2**

`cupyx.scipy.fftpack.fft2(x, shape=None, axes=(-2, -1), overwrite_x=False, plan=None)`

Compute the two-dimensional FFT.

#### **Parameters**

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (`None` or `tuple of ints`) – Shape of the transformed axes of the output. If **shape** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (`tuple of ints`) – Axes over which to compute the FFT.

- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

#### Returns

The transformed array which shape is specified by **shape** and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fftpack.fft2()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

## cupyx.scipy.fftpack.ifft2

`cupyx.scipy.fftpack.ifft2(x, shape=None, axes=(-2, -1), overwrite_x=False, plan=None)`

Compute the two-dimensional inverse FFT.

#### Parameters

- **x** (*cupy.ndarray*) – Array to be transformed.
- **shape** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If **shape** is not given, the lengths of the input along the axes specified by **axes** are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of **x** can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming **x** over **axes**, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

#### Returns

The transformed array which shape is specified by **shape** and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fftpack.ifft2()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.fftn

`cupyx.scipy.fftpack.fftn(x, shape=None, axes=None, overwrite_x=False, plan=None)`

Compute the N-dimensional FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If *shape* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of *x* can be destroyed.
- **plan** (`cupy.cuda.cufft.PlanNd` or *None*) – a cuFFT plan for transforming *x* over *axes*, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to *None*, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to *False*.

#### Returns

The transformed array which shape is specified by *shape* and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fftpack.fftn()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.ifftn

`cupyx.scipy.fftpack.ifftn(x, shape=None, axes=None, overwrite_x=False, plan=None)`

Compute the N-dimensional inverse FFT.

#### Parameters

- **x** (`cupy.ndarray`) – Array to be transformed.
- **shape** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If *shape* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*tuple of ints*) – Axes over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of *x* can be destroyed.

- **plan** (`cupy.cuda.cufft.PlanNd` or `None`) – a cuFFT plan for transforming `x` over `axes`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(x, axes)
```

Note that *plan* is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

#### Returns

The transformed array which shape is specified by `shape` and type will convert to complex if that of the input is another.

#### Return type

*cupy.ndarray*

See also:

`scipy.fftpack.ifftn()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

### cupyx.scipy.fftpack.rfft

`cupyx.scipy.fftpack.rfft(x, n=None, axis=-1, overwrite_x=False, plan=None)`

Compute the one-dimensional FFT for real input.

The returned real array contains

```
[y(0), Re(y(1)), Im(y(1)), ..., Re(y(n/2))] # if n is even
[y(0), Re(y(1)), Im(y(1)), ..., Re(y(n/2)), Im(y(n/2))] # if n is odd
```

#### Parameters

- **x** (*cupy.ndarray*) – Array to be transformed.
- **n** (*None* or *int*) – Length of the transformed axis of the output. If `n` is not given, the length of the input along the axis specified by `axis` is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite\_x** (*bool*) – If `True`, the contents of `x` can be destroyed.
- **plan** (`cupy.cuda.cufft.Plan1d` or `None`) – a cuFFT plan for transforming `x` over `axis`, which can be obtained using:

```
plan = cupyx.scipy.fftpack.get_fft_plan(
    x, axes, value_type='R2C')
```

Note that *plan* is defaulted to `None`, meaning CuPy will either use an auto-generated plan behind the scene if `cupy.fft.config.enable_nd_planning = True`, or use no cuFFT plan if it is set to `False`.

#### Returns

The transformed array.

**Return type***cupy.ndarray***See also:**`scipy.fftpack.rfft()`

---

**Note:** The argument *plan* is currently experimental and the interface may be changed in the future version.

---

**cupyx.scipy.fftpack.irfft**`cupyx.scipy.fftpack.irfft(x, n=None, axis=-1, overwrite_x=False)`

Compute the one-dimensional inverse FFT for real input.

**Parameters**

- **x** (*cupy.ndarray*) – Array to be transformed.
- **n** (*None* or *int*) – Length of the transformed axis of the output. If *n* is not given, the length of the input along the axis specified by *axis* is used.
- **axis** (*int*) – Axis over which to compute the FFT.
- **overwrite\_x** (*bool*) – If True, the contents of *x* can be destroyed.

**Returns**

The transformed array.

**Return type***cupy.ndarray***See also:**`scipy.fftpack.irfft()`

---

**Note:** This function does not support a precomputed *plan*. If you need this capability, please consider using `cupy.fft.irfft()` or `:func:`cupyx.scipy.fft.irfft``.

---

**cupyx.scipy.fftpack.get\_fft\_plan**`cupyx.scipy.fftpack.get_fft_plan(a, shape=None, axes=None, value_type='C2C')`

Generate a CUDA FFT plan for transforming up to three axes.

**Parameters**

- **a** (*cupy.ndarray*) – Array to be transform, assumed to be either C- or F- contiguous.
- **shape** (*None* or *tuple of ints*) – Shape of the transformed axes of the output. If *shape* is not given, the lengths of the input along the axes specified by *axes* are used.
- **axes** (*None* or *int* or *tuple of int*) – The axes of the array to transform. If *None*, it is assumed that all axes are transformed.

Currently, for performing N-D transform these must be a set of up to three adjacent axes, and must include either the first or the last axis of the array.

- **value\_type** (*str*) – The FFT type to perform. Acceptable values are:
  - 'C2C': complex-to-complex transform (default)
  - 'R2C': real-to-complex transform
  - 'C2R': complex-to-real transform

**Returns**

a cuFFT plan for either 1D transform (`cupy.cuda.cufft.Plan1d`) or N-D transform (`cupy.cuda.cufft.PlanNd`).

**Note:** The returned plan can not only be passed as one of the arguments of the functions in `cupyx.scipy.fftpack`, but also be used as a context manager for both `cupy.fft` and `cupyx.scipy.fftpack` functions:

```
x = cupy.random.random(16).reshape(4, 4).astype(complex)
plan = cupyx.scipy.fftpack.get_fft_plan(x)
with plan:
    y = cupy.fft.fftn(x)
    # alternatively:
    y = cupyx.scipy.fftpack.fftn(x) # no explicit plan is given!
# alternatively:
y = cupyx.scipy.fftpack.fftn(x, plan=plan) # pass plan explicitly
```

In the first case, no cuFFT plan will be generated automatically, even if `cupy.fft.config.enable_nd_planning = True` is set.

**Note:** If this function is called under the context of `set_cufft_callbacks()`, the generated plan will have callbacks enabled.

**Warning:** This API is a deviation from SciPy's, is currently experimental, and may be changed in the future version.

**Code compatibility features**

1. As with other FFT modules in CuPy, FFT functions in this module can take advantage of an existing cuFFT plan (returned by `get_fft_plan()`) to accelerate the computation. The plan can be either passed in explicitly via the `plan` argument or used as a context manager. The argument `plan` is currently experimental and the interface may be changed in the future version. The `get_fft_plan()` function has no counterpart in `scipy.fftpack`.
2. The boolean switch `cupy.fft.config.enable_nd_planning` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. This switch is neglected when planning manually using `get_fft_plan()`.
3. Like in `scipy.fftpack`, all FFT functions in this module have an optional argument `overwrite_x` (default is `False`), which has the same semantics as in `scipy.fftpack`: when it is set to `True`, the input array *x* can (not will) be overwritten arbitrarily. For this reason, when an in-place FFT is desired, the user should always reassign the input in the following manner: `x = cupyx.scipy.fftpack.fft(x, ..., overwrite_x=True, ...)`.
4. The boolean switch `cupy.fft.config.use_multi_gpus` also affects the FFT functions in this module, see *Discrete Fourier Transform (cupy.fft)*. Moreover, this switch is *honored* when planning manually using `get_fft_plan()`.

### 5.4.3 Interpolation (`cupyx.scipy.interpolate`)

---

**Hint:** SciPy API Reference: Interpolation functions (`scipy.interpolate`)

---

#### Univariate interpolation

<code>BarycentricInterpolator</code> ( <code>xi</code> [, <code>yi</code> , <code>axis</code> ])	The interpolating polynomial for a set of points.
<code>KroghInterpolator</code> ( <code>xi</code> , <code>yi</code> [, <code>axis</code> ])	Interpolating polynomial for a set of points.
<code>barycentric_interpolate</code> ( <code>xi</code> , <code>yi</code> , <code>x</code> [, <code>axis</code> ])	Convenience function for polynomial interpolation.
<code>krogh_interpolate</code> ( <code>xi</code> , <code>yi</code> , <code>x</code> [, <code>der</code> , <code>axis</code> ])	Convenience function for polynomial interpolation
<code>pchip_interpolate</code> ( <code>xi</code> , <code>yi</code> , <code>x</code> [, <code>der</code> , <code>axis</code> ])	Convenience function for pchip interpolation.
<code>CubicHermiteSpline</code> ( <code>x</code> , <code>y</code> , <code>dydx</code> [, <code>axis</code> , ...])	Piecewise-cubic interpolator matching values and first derivatives.
<code>PchipInterpolator</code> ( <code>x</code> , <code>y</code> [, <code>axis</code> , <code>extrapolate</code> ])	PCHIP 1-D monotonic cubic interpolation.
<code>Akima1DInterpolator</code> ( <code>x</code> , <code>y</code> [, <code>axis</code> ])	Akima interpolator
<code>PPoly</code> ( <code>c</code> , <code>x</code> [, <code>extrapolate</code> , <code>axis</code> ])	Piecewise polynomial in terms of coefficients and break-points The polynomial between <code>x[i]</code> and <code>x[i + 1]</code> is written in the local power basis.
<code>BPoly</code> ( <code>c</code> , <code>x</code> [, <code>extrapolate</code> , <code>axis</code> ])	Piecewise polynomial in terms of coefficients and break-points.

#### `cupyx.scipy.interpolate.BarycentricInterpolator`

**class** `cupyx.scipy.interpolate.BarycentricInterpolator`(`xi`, `yi=None`, `axis=0`)

The interpolating polynomial for a set of points.

Constructs a polynomial that passes through a given set of points. Allows evaluation of the polynomial, efficient changing of the y values to be interpolated, and updating by adding more x values. For reasons of numerical stability, this function does not compute the coefficients of the polynomial. The value `yi` need to be provided before the function is evaluated, but none of the preprocessing depends on them, so rapid updates are possible.

##### Parameters

- **`xi`** (`cupy.ndarray`) – 1-D array of x-coordinates of the points the polynomial should pass through
- **`yi`** (`cupy.ndarray`, *optional*) – The y-coordinates of the points the polynomial should pass through. If `None`, the y values will be supplied later via the `set_y` method
- **`axis`** (`int`, *optional*) – Axis in the `yi` array corresponding to the x-coordinate values

See also:

`scipy.interpolate.BarycentricInterpolator`



## Methods

**`__call__(x)`**

Evaluate the interpolating polynomial at the points `x`.

**Parameters**

**`x`** (`cupy.ndarray`) – Points to evaluate the interpolant at

**Returns**

`y` – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of `x`

**Return type**

`cupy.ndarray`

## Notes

Currently the code computes an outer product between `x` and the weights, that is, it constructs an intermediate array of size `N` by `len(x)`, where `N` is the degree of the polynomial.

**`add_xi(xi, yi=None)`**

Add more `x` values to the set to be interpolated.

The barycentric interpolation algorithm allows easy updating by adding more points for the polynomial to pass through.

**Parameters**

- **`xi`** (`cupy.ndarray`) – The `x`-coordinates of the points that the polynomial should pass through
- **`yi`** (`cupy.ndarray`, *optional*) – The `y`-coordinates of the points the polynomial should pass through. Should have shape `(xi.size, R)`; if `R > 1` then the polynomial is vector-valued. If `yi` is not given, the `y` values will be supplied later. `yi` should be given if and only if the interpolator has `y` values specified

**`set_yi(yi, axis=None)`**

Update the `y` values to be interpolated.

The barycentric interpolation algorithm requires the calculation of weights, but these depend only on the `xi`. The `yi` can be changed at any time.

**Parameters**

- **`yi`** (`cupy.ndarray`) – The `y`-coordinates of the points the polynomial should pass through. If `None`, the `y` values will be supplied later.
- **`axis`** (`int`, *optional*) – Axis in the `yi` array corresponding to the `x`-coordinate values

**`__eq__(value, /)`**

Return `self==value`.

**`__ne__(value, /)`**

Return `self!=value`.

**`__lt__(value, /)`**

Return `self<value`.

**`__le__(value, /)`**

Return `self<=value`.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

## cupyx.scipy.interpolate.KroghInterpolator

**class** cupyx.scipy.interpolate.KroghInterpolator(*xi, yi, axis=0*)

Interpolating polynomial for a set of points.

The polynomial passes through all the pairs (xi,yi). One may additionally specify a number of derivatives at each point xi; this is done by repeating the value xi and specifying the derivatives as successive yi values. Allows evaluation of the polynomial and all its derivatives. For reasons of numerical stability, this function does not compute the coefficients of the polynomial, although they can be obtained by evaluating all the derivatives.

### Parameters

- **xi** (`cupy.ndarray`, *length N*) – x-coordinate, must be sorted in increasing order
- **yi** (`cupy.ndarray`) – y-coordinate, when a xi occurs two or more times in a row, the corresponding yi's represent derivative values
- **axis** (`int`, *optional*) – Axis in the yi array corresponding to the x-coordinate values.

### Methods

`__call__(x)`

Evaluate the interpolant

#### Parameters

**x** (`cupy.ndarray`) – The points to evaluate the interpolant

#### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x

#### Return type

`cupy.ndarray`

### Notes

Input values *x* must be convertible to *float* values like *int* or *float*.

**derivative**(*x, der=1*)

Evaluate one derivative of the polynomial at the point x

#### Parameters

- **x** (`cupy.ndarray`) – Point or points at which to evaluate the derivatives
- **der** (`integer`, *optional*) – Which derivative to extract. This number includes the function value as 0th derivative

#### Returns

**d** – Derivative interpolated at the x-points. Shape of d is determined by replacing the interpolation axis in the original array with the shape of x

**Return type***cupy.ndarray***Notes**

This is computed by evaluating all derivatives up to the desired one (using `self.derivatives()`) and then discarding the rest.

**derivatives**(*x*, *der=None*)

Evaluate many derivatives of the polynomial at the point *x*.

The function produce an array of all derivative values at the point *x*.

**Parameters**

- **x** (*cupy.ndarray*) – Point or points at which to evaluate the derivatives
- **der** (*int* or *None*, *optional*) – How many derivatives to extract; *None* for all potentially nonzero derivatives (that is a number equal to the number of points). This number includes the function value as 0th derivative

**Returns**

**d** – Array with derivatives; `d[j]` contains the *j*th derivative. Shape of `d[j]` is determined by replacing the interpolation axis in the original array with the shape of *x*

**Return type***cupy.ndarray*

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(*value*, /)

Return `self<=value`.

**\_\_gt\_\_**(*value*, /)

Return `self>value`.

**\_\_ge\_\_**(*value*, /)

Return `self>=value`.

**cupyx.scipy.interpolate.barycentric\_interpolate**

**cupyx.scipy.interpolate.barycentric\_interpolate**(*xi*, *yi*, *x*, *axis=0*)

Convenience function for polynomial interpolation.

Constructs a polynomial that passes through a given set of points, then evaluates the polynomial. For reasons of numerical stability, this function does not compute the coefficients of the polynomial.

**Parameters**

- **xi** (*cupy.ndarray*) – 1-D array of coordinates of the points the polynomial should pass through

- **yi** (`cupy.ndarray`) – y-coordinates of the points the polynomial should pass through
- **x** (*scalar or* `cupy.ndarray`) – Points to evaluate the interpolator at
- **axis** (`int`, *optional*) – Axis in the yi array corresponding to the x-coordinate values

**Returns**

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape x

**Return type**

scalar or `cupy.ndarray`

**See also:**

`scipy.interpolate.barycentric_interpolate`

### `cupyx.scipy.interpolate.krogh_interpolate`

`cupyx.scipy.interpolate.krogh_interpolate(xi, yi, x, der=0, axis=0)`

Convenience function for polynomial interpolation

**Parameters**

- **xi** (`cupy.ndarray`) – x-coordinate
- **yi** (`cupy.ndarray`) – y-coordinates, of shape `(xi.size, R)`. Interpreted as vectors of length R, or scalars if R=1
- **x** (`cupy.ndarray`) – Point or points at which to evaluate the derivatives
- **der** (`int or list`, *optional*) – How many derivatives to extract; None for all potentially nonzero derivatives (that is a number equal to the number of points), or a list of derivatives to extract. This number includes the function value as 0th derivative
- **axis** (`int`, *optional*) – Axis in the yi array corresponding to the x-coordinate values

**Returns**

**d** – If the interpolator's values are R-D then the returned array will be the number of derivatives by N by R. If *x* is a scalar, the middle dimension will be dropped; if the *yi* are scalars then the last dimension will be dropped

**Return type**

`cupy.ndarray`

**See also:**

`scipy.interpolate.krogh_interpolate`

### `cupyx.scipy.interpolate.pchip_interpolate`

`cupyx.scipy.interpolate.pchip_interpolate(xi, yi, x, der=0, axis=0)`

Convenience function for pchip interpolation.

*xi* and *yi* are arrays of values used to approximate some function *f*, with *yi* = *f(xi)*. The interpolant uses monotonic cubic splines to find the value of new points *x* and the derivatives there. See `scipy.interpolate.PchipInterpolator` for details.

**Parameters**

- **xi** (*array\_like*) – A sorted list of x-coordinates, of length N.

- **yi** (*array\_like*) – A 1-D array of real values. *yi*’s length along the interpolation axis must be equal to the length of *xi*. If N-D array, use *axis* parameter to select correct axis.
- **x** (*scalar or array\_like*) – Of length M.
- **der** (*int or list, optional*) – Derivatives to extract. The 0th derivative can be included to return the function value.
- **axis** (*int, optional*) – Axis in the *yi* array corresponding to the x-coordinate values.

See also:

#### *PchipInterpolator*

PCHIP 1-D monotonic cubic interpolator.

#### Returns

**y** – The result, of length R or length M or M by R.

#### Return type

scalar or *array\_like*

### cupyx.scipy.interpolate.CubicHermiteSpline

**class** cupyx.scipy.interpolate.CubicHermiteSpline(*x, y, dydx, axis=0, extrapolate=None*)

Piecewise-cubic interpolator matching values and first derivatives.

The result is represented as a *PPoly* instance.<sup>1</sup>

#### Parameters

- **x** (*array\_like, shape (n,)*) – 1-D array containing values of the independent variable. Values must be real, finite and in strictly increasing order.
- **y** (*array\_like*) – Array containing values of the dependent variable. It can have arbitrary number of dimensions, but the length along *axis* (see below) must match the length of **x**. Values must be finite.
- **dydx** (*array\_like*) – Array containing derivatives of the dependent variable. It can have arbitrary number of dimensions, but the length along *axis* (see below) must match the length of **x**. Values must be finite.
- **axis** (*int, optional*) – Axis along which *y* is assumed to be varying. Meaning that for **x[i]** the corresponding values are `cupy.take(y, i, axis=axis)`. Default is 0.
- **extrapolate** (*{bool, 'periodic', None}, optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If *None* (default), it is set to `True`.

#### Variables

- **x** (*ndarray, shape (n,)*) – Breakpoints. The same **x** which was passed to the constructor.
- **c** (*ndarray, shape (4, n-1, ...)*) – Coefficients of the polynomials on each segment. The trailing dimensions match the dimensions of *y*, excluding *axis*. For example, if *y* is 1-D, then **c[k, i]** is a coefficient for  $(x-x[i])^{(3-k)}$  on the segment between **x[i]** and **x[i+1]**.
- **axis** (*int*) – Interpolation axis. The same *axis* which was passed to the constructor.

See also:

<sup>1</sup> Cubic Hermite spline on Wikipedia.

***Akima1DInterpolator***

Akima 1D interpolator.

***PchipInterpolator***

PCHIP 1-D monotonic cubic interpolator.

***PPoly***

Piecewise polynomial in terms of coefficients and breakpoints

**Notes**

If you want to create a higher-order spline matching higher-order derivatives, use *BPoly.from\_derivatives*.

**References****Methods**

**\_\_call\_\_**(*x*, *nu*=0, *extrapolate*=None)

Evaluate the piecewise polynomial or its derivative.

**Parameters**

- **x** (*array\_like*) – Points to evaluate the interpolant at.
- **nu** (*int*, *optional*) – Order of derivative to evaluate. Must be non-negative.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

**Returns**

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of *x*.

**Return type**

*array\_like*

**Notes**

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, [*a*, *b*), except for the last interval which is closed [*a*, *b*].

**antiderivative**(*nu*=1)

Construct a new piecewise polynomial representing the antiderivative. Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

**Parameters**

**nu** (*int*, *optional*) – Order of antiderivative to evaluate. Default is 1, i.e., compute the first integral. If negative, the derivative is returned.

**Returns**

**pp** – Piecewise polynomial of order  $k_2 = k + n$  representing the antiderivative of this polynomial.

**Return type**

*PPoly*

## Notes

The antiderivative returned by this function is continuous and continuously differentiable to order  $n-1$ , up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to `False` for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given  $x$  interval is difficult.

**classmethod** `construct_fast`(*c*, *x*, *extrapolate=None*, *axis=0*)

Construct the piecewise polynomial without making checks. Takes the same parameters as the constructor. Input arguments *c* and *x* must be arrays of the correct shape and type. The *c* array can only be of dtypes float and complex, and *x* array must have dtype float.

**derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

### Parameters

**nu** (*int*, *optional*) – Order of derivative to evaluate. Default is 1, i.e., compute the first derivative. If negative, the antiderivative is returned.

### Returns

**pp** – Piecewise polynomial of order  $k_2 = k - n$  representing the derivative of this polynomial.

### Return type

*PPoly*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**extend**(*c*, *x*)

Add additional breakpoints and coefficients to the polynomial.

### Parameters

- **c** (*ndarray*, *size* (*k*, *m*, ...)) – Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the `self.x` end points.
- **x** (*ndarray*, *size* (*m*,)) – Additional breakpoints. Must be sorted in the same order as `self.x` and either to the right or to the left of the current breakpoints.

**classmethod** `from_bernstein_basis`(*bp*, *extrapolate=None*)

Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

### Parameters

- **bp** (*BPoly*) – A Bernstein basis polynomial, as created by `BPoly`
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**classmethod** `from_spline`(*tck*, *extrapolate=None*)

Construct a piecewise polynomial from a spline

### Parameters

- **tck** – A spline, as a (knots, coefficients, degree) tuple or a `BSpline` object.

- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is *True*.

**integrate**(*a*, *b*, *extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

**Parameters**

- **a** (*float*) – Lower integration bound
- **b** (*float*) – Upper integration bound
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. If *None* (default), use *self.extrapolate*.

**Returns**

**ig** – Definite integral of the piecewise polynomial over [a, b]

**Return type**

array\_like

**roots**(*discontinuity=True*, *extrapolate=None*)

Find real roots of the piecewise polynomial.

**Parameters**

- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, *'periodic'* works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type**

ndarray

**See also:**

[\*PPoly.solve\*](#)

**solve**(*y=0.0*, *discontinuity=True*, *extrapolate=None*)

Find real solutions of the equation  $pp(x) == y$ .

**Parameters**

- **y** (*float*, *optional*) – Right-hand side. Default is zero.
- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, *'periodic'* works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.



**Return type***ndarray***Notes**

This routine works only on real-valued polynomials. If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a `nan` value. If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is `True`.

At the moment, there is not an actual implementation.

`__eq__(value, /)`

Return `self==value`.

`__ne__(value, /)`

Return `self!=value`.

`__lt__(value, /)`

Return `self<value`.

`__le__(value, /)`

Return `self<=value`.

`__gt__(value, /)`

Return `self>value`.

`__ge__(value, /)`

Return `self>=value`.

**Attributes**

**c**

**x**

**extrapolate**

**axis**

**cupyx.scipy.interpolate.PchipInterpolator**

**class** `cupyx.scipy.interpolate.PchipInterpolator(x, y, axis=0, extrapolate=None)`

PCHIP 1-D monotonic cubic interpolation.

`x` and `y` are arrays of values used to approximate some function `f`, with `y = f(x)`. The interpolant uses monotonic cubic splines to find the value of new points. (PCHIP stands for Piecewise Cubic Hermite Interpolating Polynomial).

**Parameters**

- **x** (*ndarray*) – A 1-D array of monotonically increasing real values. `x` cannot include duplicate values (otherwise `f` is overspecified)
- **y** (*ndarray*) – A 1-D array of real values. `y`'s length along the interpolation axis must be equal to the length of `x`. If N-D array, use `axis` parameter to select correct axis.

- **axis** (*int*, *optional*) – Axis in the y array corresponding to the x-coordinate values.
- **extrapolate** (*bool*, *optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

See also:

#### *CubicHermiteSpline*

Piecewise-cubic interpolator.

#### *Akima1DInterpolator*

Akima 1D interpolator.

#### *PPoly*

Piecewise polynomial in terms of coefficients and breakpoints.

## Notes

The interpolator preserves monotonicity in the interpolation data and does not overshoot if the data is not smooth.

The first derivatives are guaranteed to be continuous, but the second derivatives may jump at  $x_k$ .

Determines the derivatives at the points  $x_k$ ,  $f'_k$ , by using PCHIP algorithm<sup>1</sup>.

Let  $h_k = x_{k+1} - x_k$ , and  $d_k = (y_{k+1} - y_k)/h_k$  are the slopes at internal points  $x_k$ . If the signs of  $d_k$  and  $d_{k-1}$  are different or either of them equals zero, then  $f'_k = 0$ . Otherwise, it is given by the weighted harmonic mean

$$\frac{w_1 + w_2}{f'_k} = \frac{w_1}{d_{k-1}} + \frac{w_2}{d_k}$$

where  $w_1 = 2h_k + h_{k-1}$  and  $w_2 = h_k + 2h_{k-1}$ .

The end slopes are set using a one-sided scheme<sup>2</sup>.

## References

## Methods

`__call__(x, nu=0, extrapolate=None)`

Evaluate the piecewise polynomial or its derivative.

### Parameters

- **x** (*array\_like*) – Points to evaluate the interpolant at.
- **nu** (*int*, *optional*) – Order of derivative to evaluate. Must be non-negative.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

### Return type

*array\_like*

---

<sup>1</sup> F. N. Fritsch and J. Butland, A method for constructing local monotone piecewise cubic interpolants, SIAM J. Sci. Comput., 5(2), 300-304 (1984). [10.1137/0905021](https://doi.org/10.1137/0905021).

<sup>2</sup> see, e.g., C. Moler, Numerical Computing with Matlab, 2004. [10.1137/1.9780898717952](https://doi.org/10.1137/1.9780898717952)

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

### **antiderivative**(*nu=1*)

Construct a new piecewise polynomial representing the antiderivative. Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

#### **Parameters**

**nu** (*int*, *optional*) – Order of antiderivative to evaluate. Default is 1, i.e., compute the first integral. If negative, the derivative is returned.

#### **Returns**

**pp** – Piecewise polynomial of order  $k_2 = k + n$  representing the antiderivative of this polynomial.

#### **Return type**

*PPoly*

## Notes

The antiderivative returned by this function is continuous and continuously differentiable to order  $n-1$ , up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to `False` for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given  $x$  interval is difficult.

### **classmethod construct\_fast**(*c, x, extrapolate=None, axis=0*)

Construct the piecewise polynomial without making checks. Takes the same parameters as the constructor. Input arguments *c* and *x* must be arrays of the correct shape and type. The *c* array can only be of dtypes float and complex, and *x* array must have dtype float.

### **derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

#### **Parameters**

**nu** (*int*, *optional*) – Order of derivative to evaluate. Default is 1, i.e., compute the first derivative. If negative, the antiderivative is returned.

#### **Returns**

**pp** – Piecewise polynomial of order  $k_2 = k - n$  representing the derivative of this polynomial.

#### **Return type**

*PPoly*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**extend**(*c*, *x*)

Add additional breakpoints and coefficients to the polynomial.

### Parameters

- **c** (`ndarray`, *size* (*k*, *m*, ...)) – Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the `self.x` end points.
- **x** (`ndarray`, *size* (*m*,)) – Additional breakpoints. Must be sorted in the same order as `self.x` and either to the right or to the left of the current breakpoints.

**classmethod from\_bernstein\_basis**(*bp*, *extrapolate*=None)

Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

### Parameters

- **bp** (`BPoly`) – A Bernstein basis polynomial, as created by `BPoly`
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**classmethod from\_spline**(*tck*, *extrapolate*=None)

Construct a piecewise polynomial from a spline

### Parameters

- **tck** – A spline, as a (knots, coefficients, degree) tuple or a `BSpline` object.
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**integrate**(*a*, *b*, *extrapolate*=None)

Compute a definite integral over a piecewise polynomial.

### Parameters

- **a** (*float*) – Lower integration bound
- **b** (*float*) – Upper integration bound
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. If *None* (default), use `self.extrapolate`.

### Returns

**ig** – Definite integral of the piecewise polynomial over  $[a, b]$

### Return type

`array_like`

**roots**(*discontinuity*=`True`, *extrapolate*=None)

Find real roots of the piecewise polynomial.

### Parameters

- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type**

*ndarray*

**See also:**

*PPoly.solve*

**solve**(*y=0.0, discontinuity=True, extrapolate=None*)

Find real solutions of the equation  $pp(x) == y$ .

**Parameters**

- **y** (*float*, *optional*) – Right-hand side. Default is zero.
- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type**

*ndarray*

**Notes**

This routine works only on real-valued polynomials. If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a *nan* value. If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is *True*.

At the moment, there is not an actual implementation.

**\_\_eq\_\_**(*value, /*)

Return *self==value*.

**\_\_ne\_\_**(*value, /*)

Return *self!=value*.

**\_\_lt\_\_**(*value, /*)

Return *self<value*.

**\_\_le\_\_**(*value, /*)

Return *self<=value*.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### Attributes

`c`

`x`

`extrapolate`

`axis`

## cupyx.scipy.interpolate.Akima1DInterpolator

**class** cupyx.scipy.interpolate.Akima1DInterpolator(*x*, *y*, *axis*=0)

Akima interpolator

Fit piecewise cubic polynomials, given vectors *x* and *y*. The interpolation method by Akima uses a continuously differentiable sub-spline built from piecewise cubic polynomials. The resultant curve passes through the given data points and will appear smooth and natural<sup>1</sup>.

### Parameters

- **x** (`ndarray`, *shape* (*m*, )) – 1-D array of monotonically increasing real values.
- **y** (`ndarray`, *shape* (*m*, ...)) – N-D array of real values. The length of *y* along the first axis must be equal to the length of *x*.
- **axis** (`int`, *optional*) – Specifies the axis of *y* along which to interpolate. Interpolation defaults to the first axis of *y*.

See also:

### [CubicHermiteSpline](#)

Piecewise-cubic interpolator.

### [PchipInterpolator](#)

PCHIP 1-D monotonic cubic interpolator.

### [PPoly](#)

Piecewise polynomial in terms of coefficients and breakpoints

---

<sup>1</sup> A new method of interpolation and smooth curve fitting based on local procedures. Hiroshi Akima, J. ACM, October 1970, 17(4), 589-602.

## Notes

Use only for precise data, as the fitted curve passes through the given points exactly. This routine is useful for plotting a pleasingly smooth curve through a few given points for purposes of plotting.

## References

## Methods

`__call__(x, nu=0, extrapolate=None)`

Evaluate the piecewise polynomial or its derivative.

### Parameters

- **x** (*array\_like*) – Points to evaluate the interpolant at.
- **nu** (*int*, *optional*) – Order of derivative to evaluate. Must be non-negative.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

### Return type

*array\_like*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

`antiderivative(nu=1)`

Construct a new piecewise polynomial representing the antiderivative. Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

### Parameters

**nu** (*int*, *optional*) – Order of antiderivative to evaluate. Default is 1, i.e., compute the first integral. If negative, the derivative is returned.

### Returns

**pp** – Piecewise polynomial of order  $k_2 = k + n$  representing the antiderivative of this polynomial.

### Return type

*PPoly*

## Notes

The antiderivative returned by this function is continuous and continuously differentiable to order  $n-1$ , up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to `False` for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given `x` interval is difficult.

**classmethod** `construct_fast`(*c*, *x*, *extrapolate=None*, *axis=0*)

Construct the piecewise polynomial without making checks. Takes the same parameters as the constructor. Input arguments `c` and `x` must be arrays of the correct shape and type. The `c` array can only be of dtypes float and complex, and `x` array must have dtype float.

**derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

### Parameters

**nu** (*int*, *optional*) – Order of derivative to evaluate. Default is 1, i.e., compute the first derivative. If negative, the antiderivative is returned.

### Returns

**pp** – Piecewise polynomial of order  $k_2 = k - n$  representing the derivative of this polynomial.

### Return type

*PPoly*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**extend**(*c*, *x*, *right=True*)

Add additional breakpoints and coefficients to the polynomial.

### Parameters

- **c** (*ndarray*, *size* (*k*, *m*, ...)) – Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the `self.x` end points.
- **x** (*ndarray*, *size* (*m*,)) – Additional breakpoints. Must be sorted in the same order as `self.x` and either to the right or to the left of the current breakpoints.

**classmethod** `from_bernstein_basis`(*bp*, *extrapolate=None*)

Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

### Parameters

- **bp** (*BPoly*) – A Bernstein basis polynomial, as created by `BPoly`
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**classmethod** `from_spline`(*tck*, *extrapolate=None*)

Construct a piecewise polynomial from a spline

### Parameters

- **tck** – A spline, as a (knots, coefficients, degree) tuple or a `BSpline` object.



- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is *True*.

**integrate**(*a*, *b*, *extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

#### Parameters

- **a** (*float*) – Lower integration bound
- **b** (*float*) – Upper integration bound
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. If *None* (default), use *self.extrapolate*.

#### Returns

**ig** – Definite integral of the piecewise polynomial over [a, b]

#### Return type

array\_like

**roots**(*discontinuity=True*, *extrapolate=None*)

Find real roots of the piecewise polynomial.

#### Parameters

- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, *'periodic'* works the same as *False*. If *None* (default), use *self.extrapolate*.

#### Returns

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

#### Return type

ndarray

#### See also:

[\*PPoly.solve\*](#)

**solve**(*y=0.0*, *discontinuity=True*, *extrapolate=None*)

Find real solutions of the equation  $pp(x) == y$ .

#### Parameters

- **y** (*float*, *optional*) – Right-hand side. Default is zero.
- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, *'periodic'* works the same as *False*. If *None* (default), use *self.extrapolate*.

#### Returns

**roots** – Roots of the polynomial(s). If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type***ndarray***Notes**

This routine works only on real-valued polynomials. If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a `nan` value. If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is `True`.

At the moment, there is not an actual implementation.

**\_\_eq\_\_**(value, /)Return `self==value`.**\_\_ne\_\_**(value, /)Return `self!=value`.**\_\_lt\_\_**(value, /)Return `self<value`.**\_\_le\_\_**(value, /)Return `self<=value`.**\_\_gt\_\_**(value, /)Return `self>value`.**\_\_ge\_\_**(value, /)Return `self>=value`.**Attributes****c****x****extrapolate****axis****cupyx.scipy.interpolate.PPoly****class** cupyx.scipy.interpolate.PPoly(*c, x, extrapolate=None, axis=0*)

Piecewise polynomial in terms of coefficients and breakpoints The polynomial between `x[i]` and `x[i + 1]` is written in the local power basis:

$$S = \sum(c[m, i] * (xp - x[i]) ** (k - m) \text{ for } m \text{ in range}(k + 1))$$

where `k` is the degree of the polynomial.

**Parameters**

- **c** (*ndarray*, *shape* (`k, m, ...`)) – Polynomial coefficients, order `k` and `m` intervals.
- **x** (*ndarray*, *shape* (`m+1,` )) – Polynomial breakpoints. Must be sorted in either increasing or decreasing order.

- **extrapolate**(*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is *True*.
- **axis**(*int*, *optional*) – Interpolation axis. Default is zero.

#### Variables

- **x**(*ndarray*) – Breakpoints.
- **c**(*ndarray*) – Coefficients of the polynomials. They are reshaped to a 3-D array with the last dimension representing the trailing dimensions of the original coefficient array.
- **axis**(*int*) – Interpolation axis.

See also:

#### *BPoly*

piecewise polynomials in the Bernstein basis

#### Notes

High-order polynomials in the power basis can be numerically unstable. Precision problems can start to appear for orders larger than 20-30.

See also:

`scipy.interpolate.BSpline`

#### Methods

**\_\_call\_\_**(*x*, *nu=0*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative.

#### Parameters

- **x**(*array\_like*) – Points to evaluate the interpolant at.
- **nu**(*int*, *optional*) – Order of derivative to evaluate. Must be non-negative.
- **extrapolate**(*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. If *None* (default), use *self.extrapolate*.

#### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of *x*.

#### Return type

*array\_like*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

### **antiderivative**(*nu=1*)

Construct a new piecewise polynomial representing the antiderivative. Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

#### **Parameters**

**nu** (*int*, *optional*) – Order of antiderivative to evaluate. Default is 1, i.e., compute the first integral. If negative, the derivative is returned.

#### **Returns**

**pp** – Piecewise polynomial of order  $k_2 = k + n$  representing the antiderivative of this polynomial.

#### **Return type**

*PPoly*

## Notes

The antiderivative returned by this function is continuous and continuously differentiable to order  $n-1$ , up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to `False` for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given  $x$  interval is difficult.

### **classmethod construct\_fast**(*c, x, extrapolate=None, axis=0*)

Construct the piecewise polynomial without making checks. Takes the same parameters as the constructor. Input arguments *c* and *x* must be arrays of the correct shape and type. The *c* array can only be of dtypes float and complex, and *x* array must have dtype float.

### **derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

#### **Parameters**

**nu** (*int*, *optional*) – Order of derivative to evaluate. Default is 1, i.e., compute the first derivative. If negative, the antiderivative is returned.

#### **Returns**

**pp** – Piecewise polynomial of order  $k_2 = k - n$  representing the derivative of this polynomial.

#### **Return type**

*PPoly*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**extend**(*c*, *x*)

Add additional breakpoints and coefficients to the polynomial.

### Parameters

- **c** (`ndarray`, *size* (*k*, *m*, ...)) – Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the `self.x` end points.
- **x** (`ndarray`, *size* (*m*,)) – Additional breakpoints. Must be sorted in the same order as `self.x` and either to the right or to the left of the current breakpoints.

**classmethod from\_bernstein\_basis**(*bp*, *extrapolate*=None)

Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

### Parameters

- **bp** (`BPoly`) – A Bernstein basis polynomial, as created by `BPoly`
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**classmethod from\_spline**(*tck*, *extrapolate*=None)

Construct a piecewise polynomial from a spline

### Parameters

- **tck** – A spline, as a (knots, coefficients, degree) tuple or a `BSpline` object.
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. Default is `True`.

**integrate**(*a*, *b*, *extrapolate*=None)

Compute a definite integral over a piecewise polynomial.

### Parameters

- **a** (*float*) – Lower integration bound
- **b** (*float*) – Upper integration bound
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If *'periodic'*, periodic extrapolation is used. If *None* (default), use `self.extrapolate`.

### Returns

**ig** – Definite integral of the piecewise polynomial over  $[a, b]$

### Return type

`array_like`

**roots**(*discontinuity*=`True`, *extrapolate*=None)

Find real roots of the piecewise polynomial.

### Parameters

- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the *PPoly* object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type**

*ndarray*

**See also:**

*PPoly.solve*

**solve**(*y=0.0, discontinuity=True, extrapolate=None*)

Find real solutions of the equation  $pp(x) == y$ .

**Parameters**

- **y** (*float*, *optional*) – Right-hand side. Default is zero.
- **discontinuity** (*bool*, *optional*) – Whether to report sign changes across discontinuities at breakpoints as roots.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If *bool*, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as *False*. If *None* (default), use *self.extrapolate*.

**Returns**

**roots** – Roots of the polynomial(s). If the *PPoly* object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

**Return type**

*ndarray*

**Notes**

This routine works only on real-valued polynomials. If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a *nan* value. If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is *True*.

At the moment, there is not an actual implementation.

**\_\_eq\_\_**(*value*, /)

Return *self==value*.

**\_\_ne\_\_**(*value*, /)

Return *self!=value*.

**\_\_lt\_\_**(*value*, /)

Return *self<value*.

**\_\_le\_\_**(*value*, /)

Return *self<=value*.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### Attributes

`c`

`x`

`extrapolate`

`axis`

## cupyx.scipy.interpolate.BPoly

**class** `cupyx.scipy.interpolate.BPoly(c, x, extrapolate=None, axis=0)`

Piecewise polynomial in terms of coefficients and breakpoints.

The polynomial between `x[i]` and `x[i + 1]` is written in the

Bernstein polynomial basis:

$$S = \sum(c[a, i] * b(a, k; x) \text{ for } a \text{ in range}(k+1)),$$

where `k` is the degree of the polynomial, and:

$$b(a, k; x) = \text{binom}(k, a) * t^{**a} * (1 - t)^{**}(k - a),$$

with `t = (x - x[i]) / (x[i+1] - x[i])` and `binom` is the binomial coefficient.

### Parameters

- `c` (`ndarray`, *shape* `(k, m, ...)`) – Polynomial coefficients, order `k` and `m` intervals
- `x` (`ndarray`, *shape* `(m+1,)`) – Polynomial breakpoints. Must be sorted in either increasing or decreasing order.
- `extrapolate` (*bool*, *optional*) – If *bool*, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If ‘periodic’, periodic extrapolation is used. Default is *True*.
- `axis` (*int*, *optional*) – Interpolation axis. Default is zero.

### Variables

- `x` (`ndarray`) – Breakpoints.
- `c` (`ndarray`) – Coefficients of the polynomials. They are reshaped to a 3-D array with the last dimension representing the trailing dimensions of the original coefficient array.
- `axis` (*int*) – Interpolation axis.

See also:

### PPoly

piecewise polynomials in the power basis

## Notes

Properties of Bernstein polynomials are well documented in the literature, see for example<sup>123</sup>.

## References

## Examples

```
>>> from cupyx.scipy.interpolate import BPoly
>>> x = [0, 1]
>>> c = [[1], [2], [3]]
>>> bp = BPoly(c, x)
```

This creates a 2nd order polynomial

$$\begin{aligned} B(x) &= 1 \times b_{0,2}(x) + 2 \times b_{1,2}(x) + 3 \times b_{2,2}(x) \\ &= 1 \times (1-x)^2 + 2 \times 2x(1-x) + 3 \times x^2 \end{aligned}$$

## Methods

**\_\_call\_\_**(*x*, *nu*=0, *extrapolate*=None)

Evaluate the piecewise polynomial or its derivative.

### Parameters

- **x** (*array\_like*) – Points to evaluate the interpolant at.
- **nu** (*int*, *optional*) – Order of derivative to evaluate. Must be non-negative.
- **extrapolate** (*{bool, 'periodic', None}*, *optional*) – If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

### Return type

*array\_like*

## Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**antiderivative**(*nu*=1)

Construct a new piecewise polynomial representing the antiderivative.

### Parameters

**nu** (*int*, *optional*) – Order of antiderivative to evaluate. Default is 1, i.e., compute the first integral. If negative, the derivative is returned.

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Bernstein\\_polynomial](https://en.wikipedia.org/wiki/Bernstein_polynomial)

<sup>2</sup> Kenneth I. Joy, Bernstein polynomials, <http://www.idav.ucdavis.edu/education/CAGDNotes/Bernstein-Polynomials.pdf>

<sup>3</sup> E. H. Doha, A. H. Bhrawy, and M. A. Saker, Boundary Value Problems, vol 2011, article ID 829546, 10.1155/2011/829543.



**Returns**

**bp** – Piecewise polynomial of order  $k + nu$  representing the antiderivative of this polynomial.

**Return type**

*BPoly*

**Notes**

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to `False` for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given `x` interval is difficult.

**classmethod** `construct_fast(c, x, extrapolate=None, axis=0)`

Construct the piecewise polynomial without making checks. Takes the same parameters as the constructor. Input arguments `c` and `x` must be arrays of the correct shape and type. The `c` array can only be of dtypes float and complex, and `x` array must have dtype float.

**derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

**Parameters**

**nu** (*int*, *optional*) – Order of derivative to evaluate. Default is 1, i.e., compute the first derivative. If negative, the antiderivative is returned.

**Returns**

**bp** – Piecewise polynomial of order  $k - nu$  representing the derivative of this polynomial.

**Return type**

*BPoly*

**extend**(*c, x*)

Add additional breakpoints and coefficients to the polynomial.

**Parameters**

- **c** (*ndarray*, *size* ( $k, m, \dots$ )) – Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the `self.x` end points.
- **x** (*ndarray*, *size* ( $m,$ )) – Additional breakpoints. Must be sorted in the same order as `self.x` and either to the right or to the left of the current breakpoints.

**classmethod** `from_derivatives(xi, yi, orders=None, extrapolate=None)`

Construct a piecewise polynomial in the Bernstein basis, compatible with the specified values and derivatives at breakpoints.

**Parameters**

- **xi** (*array\_like*) – sorted 1-D array of `x`-coordinates
- **yi** (*array\_like* or *list of array\_likes*) – `yi[i][j]` is the  $j$ th derivative known at `xi[i]`
- **orders** (*None* or *int* or *array\_like of ints*. *Default: None.*) – Specifies the degree of local polynomials. If not `None`, some derivatives are ignored.
- **extrapolate** (*bool* or *'periodic'*, *optional*) – If `bool`, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If `'periodic'`, periodic extrapolation is used. Default is `True`.

## Notes

If  $k$  derivatives are specified at a breakpoint  $x$ , the constructed polynomial is exactly  $k$  times continuously differentiable at  $x$ , unless the `order` is provided explicitly. In the latter case, the smoothness of the polynomial at the breakpoint is controlled by the `order`.

Deduces the number of derivatives to match at each end from `order` and the number of derivatives available. If possible it uses the same number of derivatives from each end; if the number is odd it tries to take the extra one from `y2`. In any case if not enough derivatives are available at one end or another it draws enough to make up the total from the other end.

If the order is too high and not enough derivatives are available, an exception is raised.

## Examples

```
>>> from cupyx.scipy.interpolate import BPoly
>>> BPoly.from_derivatives([0, 1], [[1, 2], [3, 4]])
```

Creates a polynomial  $f(x)$  of degree 3, defined on  $[0, 1]$  such that  $f(0) = 1$ ,  $df/dx(0) = 2$ ,  $f(1) = 3$ ,  $df/dx(1) = 4$

```
>>> BPoly.from_derivatives([0, 1, 2], [[0, 1], [0], [2]])
```

Creates a piecewise polynomial  $f(x)$ , such that  $f(0) = f(1) = 0$ ,  $f(2) = 2$ , and  $df/dx(0) = 1$ . Based on the number of derivatives provided, the order of the local polynomials is 2 on  $[0, 1]$  and 1 on  $[1, 2]$ . Notice that no restriction is imposed on the derivatives at  $x = 1$  and  $x = 2$ .

Indeed, the explicit form of the polynomial is:

```
f(x) = | x * (1 - x),  0 <= x < 1
        | 2 * (x - 1), 1 <= x <= 2
```

So that  $f'(1-0) = -1$  and  $f'(1+0) = 2$

**classmethod** `from_power_basis(pp, extrapolate=None)`

Construct a piecewise polynomial in Bernstein basis from a power basis polynomial.

### Parameters

- **pp** (`BPoly`) – A piecewise polynomial in the power basis
- **extrapolate** (`bool` or `'periodic'`, *optional*) – If `bool`, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If `'periodic'`, periodic extrapolation is used. Default is `True`.

**integrate**(*a*, *b*, *extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

### Parameters

- **a** (`float`) – Lower integration bound
- **b** (`float`) – Upper integration bound
- **extrapolate** (`{bool, 'periodic', None}`, *optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If `'periodic'`, periodic extrapolation is used. If `None` (default), use `self.extrapolate`.

**Returns**

Definite integral of the piecewise polynomial over [a, b]

**Return type**

array\_like

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

**Attributes**

**c**

**x**

**extrapolate**

**axis**

**1-D Splines**

<code>BSpline(t, c, k[, extrapolate, axis])</code>	Univariate spline in the B-spline basis.
<code>make_interp_spline(x, y[, k, t, bc_type, ...])</code>	Compute the (coefficients of) interpolating B-spline.
<code>splder(tck[, n])</code>	Compute the spline representation of the derivative of a given spline
<code>splantider(tck[, n])</code>	Compute the spline for the antiderivative (integral) of a given spline.

**cupyx.scipy.interpolate.BSpline**

**class** cupyx.scipy.interpolate.**BSpline**(*t, c, k, extrapolate=True, axis=0*)

Univariate spline in the B-spline basis.

$$S(x) = \sum_{j=0}^{n-1} c_j B_{j,k;t}(x)$$

where  $B_{j,k;t}$  are B-spline basis functions of degree  $k$  and knots  $t$ .

### Parameters

- **t** (`ndarray`, *shape*  $(n+k+1,)$ ) – knots
- **c** (`ndarray`, *shape*  $(\geq n, \dots)$ ) – spline coefficients
- **k** (`int`) – B-spline degree
- **extrapolate** (`bool` or `'periodic'`, *optional*) – whether to extrapolate beyond the base interval,  $t[k] \dots t[n]$ , or to return nans. If True, extrapolates the first and last polynomial pieces of b-spline functions active on the base interval. If `'periodic'`, periodic extrapolation is used. Default is True.
- **axis** (`int`, *optional*) – Interpolation axis. Default is zero.

### Variables

- **t** (`ndarray`) – knot vector
- **c** (`ndarray`) – spline coefficients
- **k** (`int`) – spline degree
- **extrapolate** (`bool`) – If True, extrapolates the first and last polynomial pieces of b-spline functions active on the base interval.
- **axis** (`int`) – Interpolation axis.
- **tck** (`tuple`) – A read-only equivalent of `(self.t, self.c, self.k)`

### Notes

B-spline basis elements are defined via

$$B_{i,0}(x) = 1, \text{ if } t_i \leq x < t_{i+1}, \text{ otherwise } 0,$$
$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)$$

### Implementation details

- At least  $k+1$  coefficients are required for a spline of degree  $k$ , so that  $n \geq k+1$ . Additional coefficients,  $c[j]$  with  $j > n$ , are ignored.
- B-spline basis elements of degree  $k$  form a partition of unity on the *base interval*,  $t[k] \leq x \leq t[n]$ .
- Based on<sup>1</sup> and<sup>2</sup>

See also:

`scipy.interpolate.BSpline`

---

<sup>1</sup> Tom Lyche and Knut Morken, Spline methods, <http://www.uio.no/studier/emner/matnat/ifi/INF-MAT5340/v05/undervisningsmateriale/>

<sup>2</sup> Carl de Boor, A practical guide to splines, Springer, 2001.

## References

### Methods

**\_\_call\_\_**(*x*, *nu*=0, *extrapolate*=None)

Evaluate a spline function.

#### Parameters

- **x** (*array\_like*) – points to evaluate the spline at.
- **nu** (*int*, *optional*) – derivative to evaluate (default is 0).
- **extrapolate** (*bool* or *'periodic'*, *optional*) – whether to extrapolate based on the first and last intervals or return nans. If *'periodic'*, periodic extrapolation is used. Default is *self.extrapolate*.

#### Returns

**y** – Shape is determined by replacing the interpolation axis in the coefficient array with the shape of *x*.

#### Return type

*array\_like*

**antiderivative**(*nu*=1)

Return a B-spline representing the antiderivative.

#### Parameters

**nu** (*int*, *optional*) – Antiderivative order. Default is 1.

#### Returns

**b** – A new instance representing the antiderivative.

#### Return type

BSpline object

### Notes

If antiderivative is computed and *self.extrapolate*=*'periodic'*, it will be set to *False* for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given *x* interval is difficult.

See also:

*splder*, *splantider*

**classmethod basis\_element**(*t*, *extrapolate*=True)

Return a B-spline basis element  $B(x \mid t[0], \dots, t[k+1])$ .

#### Parameters

- **t** (*ndarray*, *shape* (*k*+2,)) – internal knots
- **extrapolate** (*bool* or *'periodic'*, *optional*) – whether to extrapolate beyond the base interval,  $t[0] \dots t[k+1]$ , or to return nans. If *'periodic'*, periodic extrapolation is used. Default is *True*.

#### Returns

**basis\_element** – A callable representing a B-spline basis element for the knot vector *t*.

#### Return type

callable

## Notes

The degree of the B-spline,  $k$ , is inferred from the length of  $t$  as  $\text{len}(t) - 2$ . The knot vector is constructed by appending and prepending  $k+1$  elements to internal knots  $t$ .

See also:

`scipy.interpolate.BSpline`

**classmethod** `construct_fast`( $t, c, k, \text{extrapolate}=\text{True}, \text{axis}=0$ )

Construct a spline without making checks. Accepts same parameters as the regular constructor. Input arrays  $t$  and  $c$  must of correct shape and dtype.

**derivative**( $nu=1$ )

Return a B-spline representing the derivative.

### Parameters

**nu** (*int*, *optional*) – Derivative order. Default is 1.

### Returns

**b** – A new instance representing the derivative.

### Return type

BSpline object

See also:

`splder`, `splantider`

**classmethod** `design_matrix`( $x, t, k, \text{extrapolate}=\text{False}$ )

Returns a design matrix as a CSR format sparse array.

### Parameters

- **x** (*array\_like*, *shape* ( $n,$ )) – Points to evaluate the spline at.
- **t** (*array\_like*, *shape* ( $nt,$ )) – Sorted 1D array of knots.
- **k** (*int*) – B-spline degree.
- **extrapolate** (*bool* or *'periodic'*, *optional*) – Whether to extrapolate based on the first and last intervals or raise an error. If *'periodic'*, periodic extrapolation is used. Default is False.

### Returns

**design\_matrix** – Sparse matrix in CSR format where each row contains all the basis elements of the input row (first row = basis elements of  $x[0]$ , ..., last row = basis elements  $x[-1]$ ).

### Return type

*csr\_matrix* object

## Notes

In each row of the design matrix all the basis elements are evaluated at the certain point (first row -  $x[0]$ , ..., last row -  $x[-1]$ ).  $nt$  is a length of the vector of knots: as far as there are  $nt - k - 1$  basis elements,  $nt$  should be not less than  $2 * k + 2$  to have at least  $k + 1$  basis element.

Out of bounds  $x$  raises a `ValueError`.

---

**Note:** This method returns a *csr\_matrix* instance as CuPy still does not have *csr\_array*.

---

See also:

`scipy.interpolate.BSpline`

**integrate**(*a*, *b*, *extrapolate=None*)

Compute a definite integral of the spline.

#### Parameters

- **a** (*float*) – Lower limit of integration.
- **b** (*float*) – Upper limit of integration.
- **extrapolate** (*bool* or *'periodic'*, *optional*) – whether to extrapolate beyond the base interval,  $t[k] \dots t[-k-1]$ , or take the spline to be zero outside of the base interval. If *'periodic'*, periodic extrapolation is used. If *None* (default), use *self.extrapolate*.

#### Returns

**I** – Definite integral of the spline over the interval  $[a, b]$ .

#### Return type

array\_like

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

#### Attributes

##### tck

Equivalent to (self.t, self.c, self.k) (read-only).

### cupyx.scipy.interpolate.make\_interp\_spline

`cupyx.scipy.interpolate.make_interp_spline`(*x*, *y*, *k=3*, *t=None*, *bc\_type=None*, *axis=0*, *check\_finite=True*)

Compute the (coefficients of) interpolating B-spline.

#### Parameters

- **x** (*array\_like*, *shape (n,)*) – Abscissas.
- **y** (*array\_like*, *shape (n, ...)*) – Ordinates.
- **k** (*int*, *optional*) – B-spline degree. Default is cubic,  $k = 3$ .

- **t** (*array\_like, shape (nt + k + 1,)*, *optional.*) – Knots. The number of knots needs to agree with the number of data points and the number of derivatives at the edges. Specifically,  $nt - n$  must equal  $\text{len}(\text{deriv\_l}) + \text{len}(\text{deriv\_r})$ .
- **bc\_type** (*2-tuple or None*) – Boundary conditions. Default is `None`, which means choosing the boundary conditions automatically. Otherwise, it must be a length-two tuple where the first element (`deriv_l`) sets the boundary conditions at `x[0]` and the second element (`deriv_r`) sets the boundary conditions at `x[-1]`. Each of these must be an iterable of pairs (`order, value`) which gives the values of derivatives of specified orders at the given edge of the interpolation interval. Alternatively, the following string aliases are recognized:
  - **"clamped": The first derivatives at the ends are zero. This is**  
equivalent to `bc_type=([(1, 0.0)], [(1, 0.0)])`.
  - **"natural":** The second derivatives at ends are zero. This is equivalent to `bc_type=([(2, 0.0)], [(2, 0.0)])`.
  - **"not-a-knot"** (default): The first and second segments are the same polynomial. This is equivalent to having `bc_type=None`.
  - **"periodic":** The values and the first  $k-1$  derivatives at the ends are equivalent.
- **axis** (*int, optional*) – Interpolation axis. Default is 0.
- **check\_finite** (*bool, optional*) – Whether to check that the input arrays contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is `True`.

**Returns****b****Return type**a BSpline object of the degree  $k$  and with knots `t`.**cupyx.scipy.interpolate.splder**`cupyx.scipy.interpolate.splder(tck, n=1)`

Compute the spline representation of the derivative of a given spline

**Parameters**

- **tck** (*tuple of (t, c, k)*) – Spline whose derivative to compute
- **n** (*int, optional*) – Order of derivative to evaluate. Default: 1

**Returns****tck\_der** – Spline of order  $k_2=k-n$  representing the derivative of the input spline.**Return type***tuple of (t2, c2, k2)*



## Notes

See also:

`scipy.interpolate.splder`

See also:

*splantider*, `splev`, `spalde`

## cupyx.scipy.interpolate.splantider

`cupyx.scipy.interpolate.splantider(tck, n=1)`

Compute the spline for the antiderivative (integral) of a given spline.

### Parameters

- **tck** (*tuple of (t, c, k)*) – Spline whose antiderivative to compute
- **n** (*int, optional*) – Order of antiderivative to evaluate. Default: 1

### Returns

**tck\_ader** – Spline of order  $k_2=k+n$  representing the antiderivative of the input spline.

### Return type

*tuple of (t2, c2, k2)*

See also:

*splder*, `splev`, `spalde`

## Notes

The *splder* function is the inverse operation of this function. Namely, `splder(splantider(tck))` is identical to *tck*, modulo rounding error.

See also:

`scipy.interpolate.splantider`

## Multivariate interpolation

Unstructured data:

---

<i>RBFInterpolator</i> (y, d[, neighbors, ...])	Radial basis function (RBF) interpolation in N dimensions.
---	--

---

**cupyx.scipy.interpolate.RBFInterpolator**

```
class cupyx.scipy.interpolate.RBFInterpolator(y, d, neighbors=None, smoothing=0.0,  
                                              kernel='thin_plate_spline', epsilon=None,  
                                              degree=None)
```

Radial basis function (RBF) interpolation in N dimensions.

**Parameters**

- **y** ((*P*, *N*) *array\_like*) – Data point coordinates.
- **d** ((*P*, ...) *array\_like*) – Data values at y.
- **neighbors** (*int*, *optional*) – If specified, the value of the interpolant at each evaluation point will be computed using only this many nearest data points. All the data points are used by default.
- **smoothing** (*float* or (*P*,) *array\_like*, *optional*) – Smoothing parameter. The interpolant perfectly fits the data when this is set to 0. For large values, the interpolant approaches a least squares fit of a polynomial with the specified degree. Default is 0.
- **kernel** (*str*, *optional*) – Type of RBF. This should be one of
  - 'linear' :  $-r$
  - 'thin\_plate\_spline' :  $r^{**2} * \log(r)$
  - 'cubic' :  $r^{**3}$
  - 'quintic' :  $-r^{**5}$
  - 'multiquadric' :  $-\sqrt{1 + r^{**2}}$
  - 'inverse\_multiquadric' :  $1/\sqrt{1 + r^{**2}}$
  - 'inverse\_quadratic' :  $1/(1 + r^{**2})$
  - 'gaussian' :  $\exp(-r^{**2})$Default is 'thin\_plate\_spline'.
- **epsilon** (*float*, *optional*) – Shape parameter that scales the input to the RBF. If *kernel* is 'linear', 'thin\_plate\_spline', 'cubic', or 'quintic', this defaults to 1 and can be ignored because it has the same effect as scaling the smoothing parameter. Otherwise, this must be specified.
- **degree** (*int*, *optional*) – Degree of the added polynomial. For some RBFs the interpolant may not be well-posed if the polynomial degree is too small. Those RBFs and their corresponding minimum degrees are
  - 'multiquadric' : 0
  - 'linear' : 0
  - 'thin\_plate\_spline' : 1
  - 'cubic' : 1
  - 'quintic' : 2

The default value is the minimum degree for *kernel* or 0 if there is no minimum degree. Set this to -1 for no added polynomial.

## Notes

An RBF is a scalar valued function in N-dimensional space whose value at  $x$  can be expressed in terms of  $r = \|x - c\|$ , where  $c$  is the center of the RBF.

An RBF interpolant for the vector of data values  $d$ , which are from locations  $y$ , is a linear combination of RBFs centered at  $y$  plus a polynomial with a specified degree. The RBF interpolant is written as

$$f(x) = K(x, y)a + P(x)b,$$

where  $K(x, y)$  is a matrix of RBFs with centers at  $y$  evaluated at the points  $x$ , and  $P(x)$  is a matrix of monomials, which span polynomials with the specified degree, evaluated at  $x$ . The coefficients  $a$  and  $b$  are the solution to the linear equations

$$(K(y, y) + \lambda I)a + P(y)b = d$$

and

$$P(y)^T a = 0,$$

where  $\lambda$  is a non-negative smoothing parameter that controls how well we want to fit the data. The data are fit exactly when the smoothing parameter is 0.

The above system is uniquely solvable if the following requirements are met:

- $P(y)$  must have full column rank.  $P(y)$  always has full column rank when *degree* is -1 or 0. When *degree* is 1,  $P(y)$  has full column rank if the data point locations are not all collinear (N=2), coplanar (N=3), etc.
- If *kernel* is 'multiquadric', 'linear', 'thin\_plate\_spline', 'cubic', or 'quintic', then *degree* must not be lower than the minimum value listed above.
- If *smoothing* is 0, then each data point location must be distinct.

When using an RBF that is not scale invariant ('multiquadric', 'inverse\_multiquadric', 'inverse\_quadratic', or 'gaussian'), an appropriate shape parameter must be chosen (e.g., through cross validation). Smaller values for the shape parameter correspond to wider RBFs. The problem can become ill-conditioned or singular when the shape parameter is too small.

The memory required to solve for the RBF interpolation coefficients increases quadratically with the number of data points, which can become impractical when interpolating more than about a thousand data points. To overcome memory limitations for large interpolation problems, the *neighbors* argument can be specified to compute an RBF interpolant for each evaluation point using only the nearest data points.

**See also:**

`scipy.interpolate.RBFInterpolator`

## Methods

`__call__(x)`

Evaluate the interpolant at  $x$ .

### Parameters

**x** ((Q, N) array\_like) – Evaluation point coordinates.

### Returns

Values of the interpolant at  $x$ .

### Return type

(Q, ...) *ndarray*

`__eq__(value, /)`  
Return self==value.

`__ne__(value, /)`  
Return self!=value.

`__lt__(value, /)`  
Return self<value.

`__le__(value, /)`  
Return self<=value.

`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

For data on a grid:

---

<code>interp</code> (points, values, xi[, method, ...])	Multidimensional interpolation on regular or rectilinear grids.
<code>RegularGridInterpolator</code> (points, values[, ...])	Interpolation on a regular or rectilinear grid in arbitrary dimensions.

---

## cupyx.scipy.interpolate.interpn

`cupyx.scipy.interpolate.interpn`(points, values, xi, method='linear', bounds\_error=True, fill\_value=nan)

Multidimensional interpolation on regular or rectilinear grids.

Strictly speaking, not all regular grids are supported - this function works on *rectilinear* grids, that is, a rectangular grid with even or uneven spacing.

### Parameters

- **points** (*tuple of cupy.ndarray of float, with shapes (m1, ), ..., (mn, )*) – The points defining the regular grid in n dimensions. The points in each dimension (i.e. every elements of the points tuple) must be strictly ascending or descending.
- **values** (*cupy.ndarray of shape (m1, ..., mn, ...)*) – The data on the regular grid in n dimensions. Complex data can be acceptable.
- **xi** (*cupy.ndarray of shape (... , ndim)*) – The coordinates to sample the gridded data at
- **method** (*str, optional*) – The method of interpolation to perform. Supported are “linear”, “nearest”, “slinear”, “cubic”, “quintic” and “pchip”.
- **bounds\_error** (*bool, optional*) – If True, when interpolated values are requested outside of the domain of the input data, a `ValueError` is raised. If False, then *fill\_value* is used.
- **fill\_value** (*number, optional*) – If provided, the value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated.

### Returns

**values\_x** – Interpolated values at *xi*. See notes for behaviour when `xi.ndim == 1`.

### Return type

*ndarray*, shape `xi.shape[:-1] + values.shape[ndim:]`

## Notes

In the case that `xi.ndim == 1` a new axis is inserted into the 0 position of the returned array, `values_x`, so its shape is instead `(1,) + values.shape[ndim:]`.

If the input data is such that input dimensions have incommensurate units and differ by many orders of magnitude, the interpolant may have numerical artifacts. Consider rescaling the data before interpolation.

## Examples

Evaluate a simple example function on the points of a regular 3-D grid:

```
>>> import cupy as cp
>>> from cupyx.scipy.interpolate import interpn
>>> def value_func_3d(x, y, z):
...     return 2 * x + 3 * y - z
>>> x = cp.linspace(0, 4, 5)
>>> y = cp.linspace(0, 5, 6)
>>> z = cp.linspace(0, 6, 7)
>>> points = (x, y, z)
>>> values = value_func_3d(*cp.meshgrid(*points, indexing='ij'))
```

Evaluate the interpolating function at a point

```
>>> point = cp.array([2.21, 3.12, 1.15])
>>> print(interpn(points, values, point))
[12.63]
```

See also:

### **RegularGridInterpolator**

interpolation on a regular or rectilinear grid in arbitrary dimensions (*interp* wraps this class).

### **cupyx.scipy.ndimage.map\_coordinates**

interpolation on grids with equal spacing (suitable for e.g., N-D image resampling)

## **cupyx.scipy.interpolate.RegularGridInterpolator**

```
class cupyx.scipy.interpolate.RegularGridInterpolator(points, values, method='linear',
                                                    bounds_error=True, fill_value=nan)
```

Interpolation on a regular or rectilinear grid in arbitrary dimensions.

The data must be defined on a rectilinear grid; that is, a rectangular grid with even or uneven spacing. Linear and nearest-neighbor interpolations are supported. After setting up the interpolator object, the interpolation method may be chosen at each evaluation.

### Parameters

- **points** (*tuple of ndarray of float, with shapes (m1, ), ..., (mn, )*) – The points defining the regular grid in *n* dimensions. The points in each dimension (i.e. every elements of the points tuple) must be strictly ascending or descending.
- **values** (*ndarray, shape (m1, ..., mn, ...)*) – The data on the regular grid in *n* dimensions. Complex data can be acceptable.

- **method** (*str*, *optional*) – The method of interpolation to perform. Supported are “linear”, “nearest”, “slinear”, “cubic”, “quintic” and “pchip”. This parameter will become the default for the object’s `__call__` method. Default is “linear”.
- **bounds\_error** (*bool*, *optional*) – If True, when interpolated values are requested outside of the domain of the input data, a `ValueError` is raised. If False, then *fill\_value* is used. Default is True.
- **fill\_value** (*float* or *None*, *optional*) – The value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated. Default is `cp.nan`.

## Notes

Contrary to `scipy’s LinearNDInterpolator` and `NearestNDInterpolator`, this class avoids expensive triangulation of the input data by taking advantage of the regular grid structure.

In other words, this class assumes that the data is defined on a *rectilinear* grid.

If the input data is such that dimensions have incommensurate units and differ by many orders of magnitude, the interpolant may have numerical artifacts. Consider rescaling the data before interpolating.

## Examples

### Evaluate a function on the points of a 3-D grid

As a first example, we evaluate a simple example function on the points of a 3-D grid:

```
>>> from cupyx.scipy.interpolate import RegularGridInterpolator
>>> import cupy as cp
>>> def f(x, y, z):
...     return 2 * x**3 + 3 * y**2 - z
>>> x = cp.linspace(1, 4, 11)
>>> y = cp.linspace(4, 7, 22)
>>> z = cp.linspace(7, 9, 33)
>>> xg, yg, zg = cp.meshgrid(x, y, z, indexing='ij', sparse=True)
>>> data = f(xg, yg, zg)
```

`data` is now a 3-D array with `data[i, j, k] = f(x[i], y[j], z[k])`. Next, define an interpolating function from this data:

```
>>> interp = RegularGridInterpolator((x, y, z), data)
```

Evaluate the interpolating function at the two points  $(x, y, z) = (2.1, 6.2, 8.3)$  and  $(3.3, 5.2, 7.1)$ :

```
>>> pts = cp.array([[2.1, 6.2, 8.3],
...                 [3.3, 5.2, 7.1]])
>>> interp(pts)
array([ 125.80469388, 146.30069388])
```

which is indeed a close approximation to

```
>>> f(2.1, 6.2, 8.3), f(3.3, 5.2, 7.1)
(125.54200000000002, 145.894)
```

### Interpolate and extrapolate a 2D dataset

As a second example, we interpolate and extrapolate a 2D data set:

```
>>> x, y = cp.array([-2, 0, 4]), cp.array([-2, 0, 2, 5])
>>> def ff(x, y):
...     return x**2 + y**2
```

```
>>> xg, yg = cp.meshgrid(x, y, indexing='ij')
>>> data = ff(xg, yg)
>>> interp = RegularGridInterpolator((x, y), data,
...                                 bounds_error=False, fill_value=None)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(projection='3d')
>>> ax.scatter(xg.ravel().get(), yg.ravel().get(), data.ravel().get(),
...           s=60, c='k', label='data')
```

Evaluate and plot the interpolator on a finer grid

```
>>> xx = cp.linspace(-4, 9, 31)
>>> yy = cp.linspace(-4, 9, 31)
>>> X, Y = cp.meshgrid(xx, yy, indexing='ij')
```

```
>>> # interpolator
>>> ax.plot_wireframe(X.get(), Y.get(), interp((X, Y)).get(),
...                  rstride=3, cstride=3, alpha=0.4, color='m',
...                  label='linear interp')
```

```
>>> # ground truth
>>> ax.plot_wireframe(X.get(), Y.get(), ff(X, Y).get(),
...                  rstride=3, cstride=3,
...                  alpha=0.4, label='ground truth')
>>> plt.legend()
>>> plt.show()
```

See also:

#### [`interp`](#)

a convenience function which wraps *RegularGridInterpolator*

#### [`scipy.ndimage.map\_coordinates`](#)

interpolation on grids with equal spacing (suitable for e.g., N-D image resampling)

## References

- [1] Python package *regulargrid* by Johannes Buchner, see <https://pypi.python.org/pypi/regulargrid/>
- [2] Wikipedia, “Trilinear interpolation”, [https://en.wikipedia.org/wiki/Trilinear\\_interpolation](https://en.wikipedia.org/wiki/Trilinear_interpolation)
- [3] Weiser, Alan, and Sergio E. Zarantonello. “A note on piecewise linear and multilinear table interpolation in many dimensions.” MATH. COMPUT. 50.181 (1988): 189-196. <https://www.ams.org/journals/mcom/1988-50-181/S0025-5718-1988-0917826-0/S0025-5718-1988-0917826-0.pdf>

## Methods

`__call__(xi, method=None)`

Interpolation at coordinates.

### Parameters

- **xi** (`cupy.ndarray` of shape `(..., ndim)`) – The coordinates to evaluate the interpolator at.
- **method** (`str`, *optional*) – The method of interpolation to perform. Supported are “linear” and “nearest”. Default is the method chosen when the interpolator was created.

### Returns

**values\_x** – Interpolated values at *xi*. See notes for behaviour when `xi.ndim == 1`.

### Return type

`cupy.ndarray`, shape `xi.shape[:-1] + values.shape[ndim:]`

## Notes

In the case that `xi.ndim == 1` a new axis is inserted into the 0 position of the returned array, `values_x`, so its shape is instead `(1,) + values.shape[ndim:]`.

## Examples

Here we define a nearest-neighbor interpolator of a simple function

```
>>> import cupy as cp
>>> x, y = cp.array([0, 1, 2]), cp.array([1, 3, 7])
>>> def f(x, y):
...     return x**2 + y**2
>>> data = f(*cp.meshgrid(x, y, indexing='ij', sparse=True))
>>> from cupyx.scipy.interpolate import RegularGridInterpolator
>>> interp = RegularGridInterpolator((x, y), data, method='nearest')
```

By construction, the interpolator uses the nearest-neighbor interpolation

```
>>> interp([[1.5, 1.3], [0.3, 4.5]])
array([2., 9.])
```

We can however evaluate the linear interpolant by overriding the *method* parameter



```
>>> interp([[1.5, 1.3], [0.3, 4.5]], method='linear')
array([ 4.7, 24.3])
```

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

Tensor product polynomials:

---

`NdPPoly(c, x[, extrapolate])`

Piecewise tensor product polynomial

---

### cupyx.scipy.interpolate.NdPPoly

**class** cupyx.scipy.interpolate.NdPPoly(*c*, *x*, *extrapolate*=None)

Piecewise tensor product polynomial

The value at point `xp = (x', y', z', ...)` is evaluated by first computing the interval indices *i* such that:

```
x[0][i[0]] <= x' < x[0][i[0]+1]
x[1][i[1]] <= y' < x[1][i[1]+1]
...
```

and then computing:

```
S = sum(c[k0-m0-1,...,kn-mn-1,i[0],...,i[n]]
        * (xp[0] - x[0][i[0]])**m0
        * ...
        * (xp[n] - x[n][i[n]])**mn
    for m0 in range(k[0]+1)
    ...
    for mn in range(k[n]+1))
```

where `k[j]` is the degree of the polynomial in dimension *j*. This representation is the piecewise multivariate power basis.

#### Parameters

- **c** (`ndarray`, shape `(k0, ..., kn, m0, ..., mn, ...)`) – Polynomial coefficients, with polynomial order *kj* and *mj+1* intervals for each dimension *j*.
- **x** (*ndim-tuple of ndarrays*, shapes `(mj+1,)`) – Polynomial breakpoints for each dimension. These must be sorted in increasing order.

- **extrapolate** (*bool*, *optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. Default: True.

#### Variables

- **x** (*tuple of ndarrays*) – Breakpoints.
- **c** (*ndarray*) – Coefficients of the polynomials.

See also:

#### *PPoly*

piecewise polynomials in 1D

#### Notes

High-order polynomials in the power basis can be numerically unstable.

#### Methods

**\_\_call\_\_**(*x*, *nu=None*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative

##### Parameters

- **x** (*array-like*) – Points to evaluate the interpolant at.
- **nu** (*tuple, optional*) – Orders of derivatives to evaluate. Each must be non-negative.
- **extrapolate** (*bool, optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

##### Returns

**y** – Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of **x**.

##### Return type

array-like

#### Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**antiderivative**(*nu*)

Construct a new piecewise polynomial representing the antiderivative. Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

##### Parameters

**nu** (*ndim-tuple of int*) – Order of derivatives to evaluate for each dimension. If negative, the derivative is returned.

##### Returns

**pp** – Piecewise polynomial of order  $k_2 = k + n$  representing the antiderivative of this polynomial.

**Return type***PPoly***Notes**

The antiderivative returned by this function is continuous and continuously differentiable to order  $n-1$ , up to floating point rounding error.

**classmethod `construct_fast`**(*c, x, extrapolate=None*)

Construct the piecewise polynomial without making checks.

Takes the same parameters as the constructor. Input arguments *c* and *x* must be arrays of the correct shape and type. The *c* array can only be of dtypes float and complex, and *x* array must have dtype float.

**derivative**(*nu*)

Construct a new piecewise polynomial representing the derivative.

**Parameters**

**nu** (*ndim-tuple of int*) – Order of derivatives to evaluate for each dimension. If negative, the antiderivative is returned.

**Returns**

**pp** – Piecewise polynomial of orders ( $k[0] - \text{nu}[0], \dots, k[n] - \text{nu}[n]$ ) representing the derivative of this polynomial.

**Return type***NdPPoly***Notes**

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals in each dimension are considered half-open,  $[a, b)$ , except for the last interval which is closed  $[a, b]$ .

**integrate**(*ranges, extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

**Parameters**

- **ranges** (*ndim-tuple of 2-tuples float*) – Sequence of lower and upper bounds for each dimension,  $[(a[0], b[0]), \dots, (a[\text{ndim}-1], b[\text{ndim}-1])]$
- **extrapolate** (*bool, optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

**Returns**

**ig** – Definite integral of the piecewise polynomial over  $[a[0], b[0]] \times \dots \times [a[\text{ndim}-1], b[\text{ndim}-1]]$

**Return type**

array\_like

**integrate\_1d**(*a, b, axis, extrapolate=None*)

Compute NdPPoly representation for one dimensional definite integral The result is a piecewise polynomial representing the integral:

$$p(y, z, \dots) = \int_a^b dx p(x, y, z, \dots)$$

where the dimension integrated over is specified with the *axis* parameter.

#### Parameters

- **a** (*float*) – Lower and upper bound for integration.
- **b** (*float*) – Lower and upper bound for integration.
- **axis** (*int*) – Dimension over which to compute the 1-D integrals
- **extrapolate** (*bool*, *optional*) – Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

#### Returns

**ig** – Definite integral of the piecewise polynomial over [a, b]. If the polynomial was 1D, an array is returned, otherwise, an NdPPoly object.

#### Return type

*NdPPoly* or array-like

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

## 5.4.4 Linear algebra (`cupyx.scipy.linalg`)

---

**Hint:** SciPy API Reference: Linear algebra (`scipy.linalg`)

---

### Basics

<code>solve_triangular(a, b[, trans, lower, ...])</code>	Solve the equation $ax = b$ for $x$ , assuming $a$ is a triangular matrix.
<code>tril(m[, k])</code>	Make a copy of a matrix with elements above the $k$ -th diagonal zeroed.
<code>triu(m[, k])</code>	Make a copy of a matrix with elements below the $k$ -th diagonal zeroed.

**cupyx.scipy.linalg.solve\_triangular**

`cupyx.scipy.linalg.solve_triangular(a, b, trans=0, lower=False, unit_diagonal=False, overwrite_b=False, check_finite=False)`

Solve the equation  $a x = b$  for  $x$ , assuming  $a$  is a triangular matrix.

**Parameters**

- **a** (`cupy.ndarray`) – The matrix with dimension  $(M, M)$ .
- **b** (`cupy.ndarray`) – The matrix with dimension  $(M, )$  or  $(M, N)$ .
- **lower** (`bool`) – Use only data contained in the lower triangle of  $a$ . Default is to use upper triangle.
- **trans** (`0, 1, 2, 'N', 'T' or 'C'`) – Type of system to solve:
  - `'0'` or `'N'` –  $ax = b$
  - `'1'` or `'T'` –  $a^T x = b$
  - `'2'` or `'C'` –  $a^H x = b$
- **unit\_diagonal** (`bool`) – If True, diagonal elements of  $a$  are assumed to be 1 and will not be referenced.
- **overwrite\_b** (`bool`) – Allow overwriting data in  $b$  (may enhance performance)
- **check\_finite** (`bool`) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns**

The matrix with dimension  $(M, )$  or  $(M, N)$ .

**Return type**

`cupy.ndarray`

See also:

`scipy.linalg.solve_triangular()`

**cupyx.scipy.linalg.tril**

`cupyx.scipy.linalg.tril(m, k=0)`

Make a copy of a matrix with elements above the  $k$ -th diagonal zeroed.

**Parameters**

- **m** (`cupy.ndarray`) – Matrix whose elements to return
- **k** (`int, optional`) – Diagonal above which to zero elements.  $k == 0$  is the main diagonal,  $k < 0$  subdiagonal and  $k > 0$  superdiagonal.

**Returns**

Return is the same shape and type as  $m$ .

**Return type**

`(cupy.ndarray)`

See also:

`scipy.linalg.tril()`

## cupyx.scipy.linalg.triu

cupyx.scipy.linalg.triu(*m*, *k*=0)

Make a copy of a matrix with elements below the *k*-th diagonal zeroed.

### Parameters

- **m** ([cupy.ndarray](#)) – Matrix whose elements to return
- **k** ([int](#), *optional*) – Diagonal above which to zero elements. *k* == 0 is the main diagonal, *k* < 0 subdiagonal and *k* > 0 superdiagonal.

### Returns

Return matrix with zeroed elements below the *k*th diagonal and has same shape and type as *m*.

### Return type

([cupy.ndarray](#))

See also:

[scipy.linalg.triu\(\)](#)

## Matrix Functions

---

[expm](#)(*a*)

Compute the matrix exponential.

---

## cupyx.scipy.linalg.expm

cupyx.scipy.linalg.expm(*a*)

Compute the matrix exponential.

### Parameters

**a** ([ndarray](#), 2D) –

### Return type

matrix exponential of *a*

## Notes

Uses (a simplified) version of Algorithm 2.3 of<sup>1</sup>: a [13 / 13] Pade approximant with scaling and squaring.

Simplifications:

- we always use a [13/13] approximate
- no matrix balancing

---

<sup>1</sup> N. Higham, SIAM J. MATRIX ANAL. APPL. Vol. 26(4), p. 1179 (2005) <https://doi.org/10.1137/04061101X>

## References

### Decompositions

<code>lu(a[, permute_l, overwrite_a, check_finite])</code>	LU decomposition.
<code>lu_factor(a[, overwrite_a, check_finite])</code>	LU decomposition.
<code>lu_solve(lu_and_piv, b[, trans, ...])</code>	Solve an equation system, $a * x = b$ , given the LU factorization of $a$

### cupyx.scipy.linalg.lu

`cupyx.scipy.linalg.lu(a, permute_l=False, overwrite_a=False, check_finite=True)`

LU decomposition.

Decomposes a given two-dimensional matrix into  $P @ L @ U$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular or trapezoidal matrix with unit diagonal, and  $U$  is a upper triangular or trapezoidal matrix.

#### Parameters

- **a** (`cupy.ndarray`) – The input matrix with dimension  $(M, N)$ .
- **permute\_l** (`bool`) – If True, perform the multiplication  $P @ L$ .
- **overwrite\_a** (`bool`) – Allow overwriting data in  $a$  (may enhance performance)
- **check\_finite** (`bool`) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

#### Returns

$(P, L, U)$  if `permute_l == False`, otherwise  $(PL, U)$ .  $P$  is a `cupy.ndarray` storing permutation matrix with dimension  $(M, M)$ .  $L$  is a `cupy.ndarray` storing lower triangular or trapezoidal matrix with unit diagonal with dimension  $(M, K)$  where  $K = \min(M, N)$ .  $U$  is a `cupy.ndarray` storing upper triangular or trapezoidal matrix with dimension  $(K, N)$ .  $PL$  is a `cupy.ndarray` storing permuted  $L$  matrix with dimension  $(M, K)$ .

#### Return type

tuple

See also:

`scipy.linalg.lu()`

### cupyx.scipy.linalg.lu\_factor

`cupyx.scipy.linalg.lu_factor(a, overwrite_a=False, check_finite=True)`

LU decomposition.

Decompose a given two-dimensional square matrix into  $P * L * U$ , where  $P$  is a permutation matrix,  $L$  lower-triangular with unit diagonal elements, and  $U$  upper-triangular matrix.

#### Parameters

- **a** (`cupy.ndarray`) – The input matrix with dimension  $(M, N)$
- **overwrite\_a** (`bool`) – Allow overwriting data in  $a$  (may enhance performance)

- **check\_finite** (*bool*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns**

(*lu*, *piv*) where *lu* is a [cupy.ndarray](#) storing U in its upper triangle, and L without unit diagonal elements in its lower triangle, and *piv* is a [cupy.ndarray](#) storing pivot indices representing permutation matrix P. For  $0 \leq i < \min(M, N)$ , row *i* of the matrix was interchanged with row *piv*[*i*]

**Return type**

[tuple](#)

See also:

[scipy.linalg.lu\\_factor\(\)](#)

**cupyx.scipy.linalg.lu\_solve**

`cupyx.scipy.linalg.lu_solve(lu_and_piv, b, trans=0, overwrite_b=False, check_finite=True)`

Solve an equation system,  $a * x = b$ , given the LU factorization of *a*

**Parameters**

- **lu\_and\_piv** (*tuple*) – LU factorization of matrix *a* ((*M*, *M*)) together with pivot indices.
- **b** ([cupy.ndarray](#)) – The matrix with dimension (*M*,) or (*M*, *N*).
- **trans** ({0, 1, 2}) – Type of system to solve:

trans	system
0	$a x = b$
1	$a^T x = b$
2	$a^H x = b$

- **overwrite\_b** (*bool*) – Allow overwriting data in *b* (may enhance performance)
- **check\_finite** (*bool*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

**Returns**

The matrix with dimension (*M*,) or (*M*, *N*).

**Return type**

[cupy.ndarray](#)

See also:

[scipy.linalg.lu\\_solve\(\)](#)



## Special Matrices

<code>block_diag(*arrs)</code>	Create a block diagonal matrix from provided arrays.
<code>circulant(c)</code>	Construct a circulant matrix.
<code>companion(a)</code>	Create a companion matrix.
<code>convolution_matrix(a, n[, mode])</code>	Construct a convolution matrix.
<code>dft(n[, scale])</code>	Discrete Fourier transform matrix.
<code>fiedler(a)</code>	Returns a symmetric Fiedler matrix
<code>fiedler_companion(a)</code>	Returns a Fiedler companion matrix
<code>hadamard(n[, dtype])</code>	Construct an Hadamard matrix.
<code>hankel(c[, r])</code>	Construct a Hankel matrix.
<code>helmert(n[, full])</code>	Create an Helmert matrix of order <code>n</code> .
<code>hilbert(n)</code>	Create a Hilbert matrix of order <code>n</code> .
<code>kron(a, b)</code>	Kronecker product.
<code>leslie(f, s)</code>	Create a Leslie matrix.
<code>toeplitz(c[, r])</code>	Construct a Toeplitz matrix.
<code>tri(N[, M, k, dtype])</code>	Construct (N, M) matrix filled with ones at and below the <code>k</code> -th diagonal.

### cupyx.scipy.linalg.block\_diag

`cupyx.scipy.linalg.block_diag(*arrs)`

Create a block diagonal matrix from provided arrays.

Given the inputs A, B, and C, the output will have these arrays arranged on the diagonal:

```
[A,  0,  0]
[0,  B,  0]
[0,  0,  C]
```

#### Parameters

- **A** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- **B** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- **C** (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.
- ... (`cupy.ndarray`) – Input arrays. A 1-D array of length `n` is treated as a 2-D array with shape `(1,n)`.

#### Returns

Array with A, B, C, ... on the diagonal. Output has the same dtype as A.

#### Return type

(`cupy.ndarray`)

See also:

`scipy.linalg.block_diag()`

**cupyx.scipy.linalg.circulant****cupyx.scipy.linalg.circulant**(*c*)

Construct a circulant matrix.

**Parameters****c** ([cupy.ndarray](#)) – 1-D array, the first column of the matrix.**Returns**A circulant matrix whose first column is *c*.**Return type***cupy.ndarray***See also:**[cupyx.scipy.linalg.toeplitz\(\)](#)**See also:**[cupyx.scipy.linalg.hankel\(\)](#)**See also:**[cupyx.scipy.linalg.solve\\_circulant\(\)](#)**See also:**[cupyx.scipy.linalg.fiedler\(\)](#)**See also:**[scipy.linalg.circulant\(\)](#)**cupyx.scipy.linalg.companion****cupyx.scipy.linalg.companion**(*a*)

Create a companion matrix.

Create the companion matrix associated with the polynomial whose coefficients are given in *a*.**Parameters****a** ([cupy.ndarray](#)) – 1-D array of polynomial coefficients. The length of *a* must be at least two, and *a*[0] must not be zero.**Returns**The first row of the output is  $-a[1:]/a[0]$ , and the first sub-diagonal is all ones. The data-type of the array is the same as the data-type of  $-a[1:]/a[0]$ .**Return type***(cupy.ndarray)***See also:**[cupyx.scipy.linalg.fiedler\\_companion\(\)](#)**See also:**[scipy.linalg.companion\(\)](#)

**cupyx.scipy.linalg.convolution\_matrix**

`cupyx.scipy.linalg.convolution_matrix(a, n, mode='full')`

Construct a convolution matrix.

Constructs the Toeplitz matrix representing one-dimensional convolution.

**Parameters**

- **a** (`cupy.ndarray`) – The 1-D array to convolve.
- **n** (`int`) – The number of columns in the resulting matrix. It gives the length of the input to be convolved with a. This is analogous to the length of v in `numpy.convolve(a, v)`.
- **mode** (`str`) – This must be one of ('full', 'valid', 'same'). This is analogous to mode in `numpy.convolve(v, a, mode)`.

**Returns**

The convolution matrix whose row count k depends on mode:

mode	k
'full'	$m + n - 1$
'same'	$\max(m, n)$
'valid'	$\max(m, n) - \min(m, n) + 1$

**Return type**

*cupy.ndarray*

See also:

`cupyx.scipy.linalg.toeplitz()`

See also:

`scipy.linalg.convolution_matrix()`

**cupyx.scipy.linalg.dft**

`cupyx.scipy.linalg.dft(n, scale=None)`

Discrete Fourier transform matrix.

Create the matrix that computes the discrete Fourier transform of a sequence. The nth primitive root of unity used to generate the matrix is  $\exp(-2\pi i/n)$ , where  $i = \sqrt{-1}$ .

**Parameters**

- **n** (`int`) – Size the matrix to create.
- **scale** (`str`, *optional*) – Must be None, 'sqrtn', or 'n'. If scale is 'sqrtn', the matrix is divided by  $\sqrt{n}$ . If scale is 'n', the matrix is divided by n. If scale is None (default), the matrix is not normalized, and the return value is simply the Vandermonde matrix of the roots of unity.

**Returns**

The DFT matrix.

**Return type**

(*cupy.ndarray*)

## Notes

When `scale` is `None`, multiplying a vector by the matrix returned by `dft` is mathematically equivalent to (but much less efficient than) the calculation performed by `scipy.fft.fft`.

See also:

`scipy.linalg.dft()`

## `cupyx.scipy.linalg.fiedler`

`cupyx.scipy.linalg.fiedler(a)`

Returns a symmetric Fiedler matrix

Given an sequence of numbers `a`, Fiedler matrices have the structure  $F[i, j] = \text{np.abs}(a[i] - a[j])$ , and hence zero diagonals and nonnegative entries. A Fiedler matrix has a dominant positive eigenvalue and other eigenvalues are negative. Although not valid generally, for certain inputs, the inverse and the determinant can be derived explicitly.

### Parameters

**a** (`cupy.ndarray`) – coefficient array

### Returns

the symmetric Fiedler matrix

### Return type

*cupy.ndarray*

See also:

`cupyx.scipy.linalg.circulant()`

See also:

`cupyx.scipy.linalg.toeplitz()`

See also:

`scipy.linalg.fiedler()`

## `cupyx.scipy.linalg.fiedler_companion`

`cupyx.scipy.linalg.fiedler_companion(a)`

Returns a Fiedler companion matrix

Given a polynomial coefficient array `a`, this function forms a pentadiagonal matrix with a special structure whose eigenvalues coincides with the roots of `a`.

### Parameters

**a** (`cupy.ndarray`) – 1-D array of polynomial coefficients in descending order with a nonzero leading coefficient. For  $N < 2$ , an empty array is returned.

### Returns

Resulting companion matrix

### Return type

*cupy.ndarray*

## Notes

Similar to `companion` the leading coefficient should be nonzero. In the case the leading coefficient is not 1, other coefficients are rescaled before the array generation. To avoid numerical issues, it is best to provide a monic polynomial.

See also:

`cupyx.scipy.linalg.companion()`

See also:

`scipy.linalg.fiedler_companion()`

## cupyx.scipy.linalg.hadamard

`cupyx.scipy.linalg.hadamard(n, dtype=<class 'int'>)`

Construct an Hadamard matrix.

Constructs an n-by-n Hadamard matrix, using Sylvester's construction. `n` must be a power of 2.

### Parameters

- `n` (*int*) – The order of the matrix. `n` must be a power of 2.
- `dtype` (*dtype, optional*) – The data type of the array to be constructed.

### Returns

The Hadamard matrix.

### Return type

*cupy.ndarray*

See also:

`scipy.linalg.hadamard()`

## cupyx.scipy.linalg.hankel

`cupyx.scipy.linalg.hankel(c, r=None)`

Construct a Hankel matrix.

The Hankel matrix has constant anti-diagonals, with `c` as its first column and `r` as its last row. If `r` is not given, then `r = zeros_like(c)` is assumed.

### Parameters

- `c` (*cupy.ndarray*) – First column of the matrix. Whatever the actual shape of `c`, it will be converted to a 1-D array.
- `r` (*cupy.ndarray, optional*) – Last row of the matrix. If `None`, `r = zeros_like(c)` is assumed. `r[0]` is ignored; the last row of the returned matrix is `[c[-1], r[1:]]`. Whatever the actual shape of `r`, it will be converted to a 1-D array.

### Returns

The Hankel matrix. Dtype is the same as `(c[0] + r[0]).dtype`.

### Return type

*cupy.ndarray*

See also:

`cupyx.scipy.linalg.toeplitz()`

See also:

`cupyx.scipy.linalg.circulant()`

See also:

`scipy.linalg.hankel()`

### **cupyx.scipy.linalg.helmert**

`cupyx.scipy.linalg.helmert(n, full=False)`

Create an Helmert matrix of order *n*.

This has applications in statistics, compositional or simplicial analysis, and in Aitchison geometry.

#### **Parameters**

- **n** (*int*) – The size of the array to create.
- **full** (*bool*, *optional*) – If True the (n, n) ndarray will be returned. Otherwise, the default, the submatrix that does not include the first row will be returned.

#### **Returns**

The Helmert matrix. The shape is (n, n) or (n-1, n) depending on the *full* argument.

#### **Return type**

*cupy.ndarray*

See also:

`scipy.linalg.helmert()`

### **cupyx.scipy.linalg.hilbert**

`cupyx.scipy.linalg.hilbert(n)`

Create a Hilbert matrix of order *n*.

Returns the *n* by *n* array with entries  $h[i, j] = 1 / (i + j + 1)$ .

#### **Parameters**

- **n** (*int*) – The size of the array to create.

#### **Returns**

The Hilbert matrix.

#### **Return type**

*cupy.ndarray*

See also:

`scipy.linalg.hilbert()`

**cupyx.scipy.linalg.kron**`cupyx.scipy.linalg.kron(a, b)`

Kronecker product.

**The result is the block matrix::**
$$a[0,0]*b \ a[0,1]*b \ \dots \ a[0,-1]*b \ a[1,0]*b \ a[1,1]*b \ \dots \ a[1,-1]*b \ \dots \ a[-1,0]*b \ a[-1,1]*b \ \dots \ a[-1,-1]*b$$
**Parameters**

- **a** (`cupy.ndarray`) – Input array
- **b** (`cupy.ndarray`) – Input array

**Returns**

Kronecker product of a and b.

**Return type***cupy.ndarray***See also:**`scipy.linalg.kron()`**cupyx.scipy.linalg.leslie**`cupyx.scipy.linalg.leslie(f, s)`

Create a Leslie matrix.

Given the length n array of fecundity coefficients *f* and the length n-1 array of survival coefficients *s*, return the associated Leslie matrix.

**Parameters**

- **f** (`cupy.ndarray`) – The “fecundity” coefficients.
- **s** (`cupy.ndarray`) – The “survival” coefficients, has to be 1-D. The length of *s* must be one less than the length of *f*, and it must be at least 1.

**Returns**

The array is zero except for the first row, which is *f*, and the first sub-diagonal, which is *s*. The data-type of the array will be the data-type of *f*[0]+*s*[0].

**Return type***cupy.ndarray***See also:**`scipy.linalg.leslie()`

## cupyx.scipy.linalg.toeplitz

`cupyx.scipy.linalg.toeplitz(c, r=None)`

Construct a Toeplitz matrix.

The Toeplitz matrix has constant diagonals, with `c` as its first column and `r` as its first row. If `r` is not given, `r == conjugate(c)` is assumed.

### Parameters

- **c** (`cupy.ndarray`) – First column of the matrix. Whatever the actual shape of `c`, it will be converted to a 1-D array.
- **r** (`cupy.ndarray`, *optional*) – First row of the matrix. If `None`, `r = conjugate(c)` is assumed; in this case, if `c[0]` is real, the result is a Hermitian matrix. `r[0]` is ignored; the first row of the returned matrix is `[c[0], r[1:]]`. Whatever the actual shape of `r`, it will be converted to a 1-D array.

### Returns

The Toeplitz matrix. Dtype is the same as `(c[0] + r[0]).dtype`.

### Return type

*cupy.ndarray*

See also:

`cupyx.scipy.linalg.circulant()`

See also:

`cupyx.scipy.linalg.hankel()`

See also:

`cupyx.scipy.linalg.solve_toeplitz()`

See also:

`cupyx.scipy.linalg.fiedler()`

See also:

`scipy.linalg.toeplitz()`

## cupyx.scipy.linalg.tri

`cupyx.scipy.linalg.tri(N, M=None, k=0, dtype=None)`

Construct (N, M) matrix filled with ones at and below the `k`-th diagonal. The matrix has `A[i, j] == 1` for `i <= j + k`.

### Parameters

- **N** (*int*) – The size of the first dimension of the matrix.
- **M** (*int*, *optional*) – The size of the second dimension of the matrix. If `M` is `None`, `M = N` is assumed.
- **k** (*int*, *optional*) – Number of subdiagonal below which matrix is filled with ones. `k = 0` is the main diagonal, `k < 0` subdiagonal and `k > 0` superdiagonal.
- **dtype** (*dtype*, *optional*) – Data type of the matrix.



**Returns**

Tri matrix.

**Return type***cupy.ndarray***See also:**`scipy.linalg.tri()`

## 5.4.5 Multidimensional image processing (`cupyx.scipy.ndimage`)

**Hint:** SciPy API Reference: Multidimensional image processing (`scipy.ndimage`)

### Filters

<i>convolve</i> (input, weights[, output, mode, ...])	Multi-dimensional convolution.
<i>convolve1d</i> (input, weights[, axis, output, ...])	One-dimensional convolution.
<i>correlate</i> (input, weights[, output, mode, ...])	Multi-dimensional correlate.
<i>correlate1d</i> (input, weights[, axis, output, ...])	One-dimensional correlate.
<i>gaussian_filter</i> (input, sigma[, order, ...])	Multi-dimensional Gaussian filter.
<i>gaussian_filter1d</i> (input, sigma[, axis, ...])	One-dimensional Gaussian filter along the given axis.
<i>gaussian_gradient_magnitude</i> (input, sigma[, ...])	Multi-dimensional gradient magnitude using Gaussian derivatives.
<i>gaussian_laplace</i> (input, sigma[, output, ...])	Multi-dimensional Laplace filter using Gaussian second derivatives.
<i>generic_filter</i> (input, function[, size, ...])	Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.
<i>generic_filter1d</i> (input, function, filter_size)	Compute a 1D filter along the given axis using the provided raw kernel.
<i>generic_gradient_magnitude</i> (input, derivative)	Multi-dimensional gradient magnitude filter using a provided derivative function.
<i>generic_laplace</i> (input, derivative2[, ...])	Multi-dimensional Laplace filter using a provided second derivative function.
<i>laplace</i> (input[, output, mode, cval])	Multi-dimensional Laplace filter based on approximate second derivatives.
<i>maximum_filter</i> (input[, size, footprint, ...])	Multi-dimensional maximum filter.
<i>maximum_filter1d</i> (input, size[, axis, ...])	Compute the maximum filter along a single axis.
<i>median_filter</i> (input[, size, footprint, ...])	Multi-dimensional median filter.
<i>minimum_filter</i> (input[, size, footprint, ...])	Multi-dimensional minimum filter.
<i>minimum_filter1d</i> (input, size[, axis, ...])	Compute the minimum filter along a single axis.
<i>percentile_filter</i> (input, percentile[, size, ...])	Multi-dimensional percentile filter.
<i>prewitt</i> (input[, axis, output, mode, cval])	Compute a Prewitt filter along the given axis.
<i>rank_filter</i> (input, rank[, size, footprint, ...])	Multi-dimensional rank filter.
<i>sobel</i> (input[, axis, output, mode, cval])	Compute a Sobel filter along the given axis.
<i>uniform_filter</i> (input[, size, output, mode, ...])	Multi-dimensional uniform filter.
<i>uniform_filter1d</i> (input, size[, axis, ...])	One-dimensional uniform filter along the given axis.

### cupyx.scipy.ndimage.convolve

cupyx.scipy.ndimage.**convolve**(input, weights, output=None, mode='reflect', cval=0.0, origin=0)

Multi-dimensional convolution.

The array is convolved with the given kernel.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **weights** (`cupy.ndarray`) – Array of weights, same number of dimensions as input
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (`scalar` or `tuple of scalar`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

#### Returns

The result of convolution.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.convolve()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.convolve1d

cupyx.scipy.ndimage.**convolve1d**(input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)

One-dimensional convolution.

The array is convolved with the given kernel.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **weights** (`cupy.ndarray`) – One-dimensional array of weights
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **origin** (*int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns**

The result of the 1D convolution.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.convolve1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.correlate**

`cupyx.scipy.ndimage.correlate(input, weights, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional correlate.

The array is correlated with the given kernel.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **weights** (*cupy.ndarray*) – Array of weights, same number of dimensions as input
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of correlate.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.correlate()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.correlate1d

cupyx.scipy.ndimage.**correlate1d**(input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)

One-dimensional correlate.

The array is correlated with the given kernel.

#### Parameters

- **input** ([cupy.ndarray](#)) – The input array.
- **weights** ([cupy.ndarray](#)) – One-dimensional array of weights
- **axis** ([int](#)) – The axis of input along which to calculate. Default is -1.
- **output** ([cupy.ndarray](#), *dtype* or *None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** ([int](#)) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

#### Returns

The result of the 1D correlation.

#### Return type

[cupy.ndarray](#)

See also:

[scipy.ndimage.correlate1d\(\)](#)

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.gaussian\_filter

cupyx.scipy.ndimage.**gaussian\_filter**(input, sigma, order=0, output=None, mode='reflect', cval=0.0, truncate=4.0)

Multi-dimensional Gaussian filter.

#### Parameters

- **input** ([cupy.ndarray](#)) – The input array.
- **sigma** (*scalar or sequence of scalar*) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **order** ([int](#) or *sequence of scalar*) – An order of 0, the default, corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian. A single value applies to all axes.
- **output** ([cupy.ndarray](#), *dtype* or *None*) – The array in which to place the output. Default is is same dtype as the input.

- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **truncate** (*float*) – Truncate the filter at this many standard deviations. Default is 4.0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.gaussian_filter()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.gaussian\_filter1d**

`cupyx.scipy.ndimage.gaussian_filter1d(input, sigma, axis=-1, order=0, output=None, mode='reflect', cval=0.0, truncate=4.0)`

One-dimensional Gaussian filter along the given axis.

The lines of the array along the given axis are filtered with a Gaussian filter of the given standard deviation.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel.
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **order** (*int*) – An order of 0, the default, corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **truncate** (*float*) – Truncate the filter at this many standard deviations. Default is 4.0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.gaussian_filter1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.gaussian\_gradient\_magnitude

cupyx.scipy.ndimage.gaussian\_gradient\_magnitude(*input*, *sigma*, *output=None*, *mode='reflect'*, *cval=0.0*, *\*\*kwargs*)

Multi-dimensional gradient magnitude using Gaussian derivatives.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (*scalar or sequence of scalar*) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **kwargs** (*dict*, *optional*) – dict of extra keyword arguments to pass `gaussian_filter()`.

#### Returns

The result of the filtering.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.gaussian_gradient_magnitude()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

### cupyx.scipy.ndimage.gaussian\_laplace

cupyx.scipy.ndimage.gaussian\_laplace(*input*, *sigma*, *output=None*, *mode='reflect'*, *cval=0.0*, *\*\*kwargs*)

Multi-dimensional Laplace filter using Gaussian second derivatives.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (*scalar or sequence of scalar*) – Standard deviations for each axis of Gaussian kernel. A single value applies to all axes.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.

- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **kwargs** (*dict*, *optional*) – dict of extra keyword arguments to pass `gaussian_filter()`.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.gaussian_laplace()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.generic\_filter**

`cupyx.scipy.ndimage.generic_filter(input, function, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.

Unlike the `scipy.ndimage` function, this does not support the `extra_arguments` or `extra_keywordsdict` arguments and has significant restrictions on the `function` provided.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **function** (*cupy.ReductionKernel* or *cupy.RawKernel*) – The kernel or function to apply to each region.
- **size** (*int* or *sequence of int*) – One of `size` or `footprint` must be provided. If `footprint` is given, `size` is ignored. Otherwise `footprint = cupy.ones(size)` with `size` automatically made to match the number of dimensions in `input`.
- **footprint** (*cupy.ndarray*) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (*cupy.ndarray*, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*scalar* or *tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The result of the filtering.

**Return type***cupy.ndarray*

---

**Note:** If the *function* is a [cupy.RawKernel](#) then it must be for a function that has the following signature. Unlike most functions, this should not utilize *blockDim/blockIdx/threadIdx*:

```
__global__ void func(double *buffer, int filter_size,
                    double *return_value)
```

---

If the *function* is a [cupy.ReductionKernel](#) then it must be for a kernel that takes 1 array input and produces 1 ‘scalar’ output.

---

**See also:**[scipy.ndimage.generic\\_filter\(\)](#)**cupyx.scipy.ndimage.generic\_filter1d**

`cupyx.scipy.ndimage.generic_filter1d(input, function, filter_size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

Compute a 1D filter along the given axis using the provided raw kernel.

Unlike the `scipy.ndimage` function, this does not support the `extra_arguments` or `extra_keywordsdict` arguments and has significant restrictions on the `function` provided.

**Parameters**

- **input** ([cupy.ndarray](#)) – The input array.
- **function** ([cupy.RawKernel](#)) – The kernel to apply along each axis.
- **filter\_size** ([int](#)) – Length of the filter.
- **axis** ([int](#)) – The axis of input along which to calculate. Default is -1.
- **output** ([cupy.ndarray](#), *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** ([int](#)) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns**

The result of the filtering.

**Return type***cupy.ndarray*

---

**Note:** The provided function (as a `RawKernel`) must have the following signature. Unlike most functions, this should not utilize *blockDim/blockIdx/threadIdx*:

```
__global__ void func(double *input_line, ptrdiff_t input_length,
                    double *output_line, ptrdiff_t output_length)
```



See also:

`scipy.ndimage.generic_filter1d()`

### `cupyx.scipy.ndimage.generic_gradient_magnitude`

`cupyx.scipy.ndimage.generic_gradient_magnitude(input, derivative, output=None, mode='reflect', cval=0.0, extra_arguments=(), extra_keywords=None)`

Multi-dimensional gradient magnitude filter using a provided derivative function.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **derivative** (*callable*) – Function or other callable with the following signature that is called once per axis:

```
derivative(input, axis, output, mode, cval,
          *extra_arguments, **extra_keywords)
```

where `input` and `output` are `cupy.ndarray`, `axis` is an `int` from 0 to the number of dimensions, and `mode`, `cval`, `extra_arguments`, `extra_keywords` are the values given to this function.

- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output. Default is same `dtype` as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **extra\_arguments** (*sequence*, *optional*) – Sequence of extra positional arguments to pass to `derivative2`.
- **extra\_keywords** (*dict*, *optional*) – dict of extra keyword arguments to pass to `derivative2`.

#### Returns

The result of the filtering.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.generic_gradient_magnitude()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## cupyx.scipy.ndimage.generic\_laplace

cupyx.scipy.ndimage.**generic\_laplace**(input, derivative2, output=None, mode='reflect', cval=0.0, extra\_arguments=(), extra\_keywords=None)

Multi-dimensional Laplace filter using a provided second derivative function.

### Parameters

- **input** ([cupy.ndarray](#)) – The input array.
- **derivative2** (*callable*) – Function or other callable with the following signature that is called once per axis:

```
derivative2(input, axis, output, mode, cval,
            *extra_arguments, **extra_keywords)
```

where `input` and `output` are `cupy.ndarray`, `axis` is an `int` from 0 to the number of dimensions, and `mode`, `cval`, `extra_arguments`, `extra_keywords` are the values given to this function.

- **output** ([cupy.ndarray](#), *dtype* or *None*) – The array in which to place the output. Default is same `dtype` as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **extra\_arguments** (*sequence, optional*) – Sequence of extra positional arguments to pass to `derivative2`.
- **extra\_keywords** (*dict, optional*) – dict of extra keyword arguments to pass to `derivative2`.

### Returns

The result of the filtering.

### Return type

[cupy.ndarray](#)

See also:

[scipy.ndimage.generic\\_laplace\(\)](#)

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## cupyx.scipy.ndimage.laplace

cupyx.scipy.ndimage.**laplace**(input, output=None, mode='reflect', cval=0.0)

Multi-dimensional Laplace filter based on approximate second derivatives.

### Parameters

- **input** ([cupy.ndarray](#)) – The input array.
- **output** ([cupy.ndarray](#), *dtype* or *None*) – The array in which to place the output. Default is same `dtype` as the input.

- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.laplace()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.maximum\_filter**

`cupyx.scipy.ndimage.maximum_filter(input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional maximum filter.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*int or sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise footprint = `cupy.ones(size)` with size automatically made to match the number of dimensions in input.
- **footprint** (*cupy.ndarray*) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*int or sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.maximum_filter()`

### cupyx.scipy.ndimage.maximum\_filter1d

```
cupyx.scipy.ndimage.maximum_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0,
                                     origin=0)
```

Compute the maximum filter along a single axis.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int`) – Length of the maximum filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

#### Returns

The result of the filtering.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.maximum_filter1d()`

### cupyx.scipy.ndimage.median\_filter

```
cupyx.scipy.ndimage.median_filter(input, size=None, footprint=None, output=None, mode='reflect',
                                  cval=0.0, origin=0)
```

Multi-dimensional median filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int` or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise footprint = `cupy.ones(size)` with size automatically made to match the number of dimensions in input.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **origin** (*int or sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.median_filter()`

**cupyx.scipy.ndimage.minimum\_filter**

`cupyx.scipy.ndimage.minimum_filter(input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional minimum filter.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*int or sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise `footprint = cupy.ones(size)` with size automatically made to match the number of dimensions in input.
- **footprint** (*cupy.ndarray*) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*int or sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.minimum_filter()`

### cupyx.scipy.ndimage.minimum\_filter1d

`cupyx.scipy.ndimage.minimum_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

Compute the minimum filter along a single axis.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int`) – Length of the minimum filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int`) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

#### Returns

The result of the filtering.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.minimum_filter1d()`

### cupyx.scipy.ndimage.percentile\_filter

`cupyx.scipy.ndimage.percentile_filter(input, percentile, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional percentile filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **percentile** (`scalar`) – The percentile of the element to get (from 0 to 100). Can be negative, thus -20 equals 80.
- **size** (`int` or *sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise footprint = `cupy.ones(size)` with size automatically made to match the number of dimensions in input.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **origin** (*int* or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.percentile_filter()`

**cupyx.scipy.ndimage.prewitt**

`cupyx.scipy.ndimage.prewitt(input, axis=-1, output=None, mode='reflect', cval=0.0)`

Compute a Prewitt filter along the given axis.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **output** (*cupy.ndarray*, *dtype* or *None*) – The array in which to place the output. Default is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.prewitt()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.rank\_filter**

`cupyx.scipy.ndimage.rank_filter(input, rank, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional rank filter.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **rank** (*int*) – The rank of the element to get. Can be negative to count from the largest value, e.g. -1 indicates the largest value.

- **size** (*int or sequence of int*) – One of size or footprint must be provided. If footprint is given, size is ignored. Otherwise `footprint = cupy.ones(size)` with size automatically made to match the number of dimensions in `input`.
- **footprint** (`cupy.ndarray`) – a boolean array which specifies which of the elements within this shape will get passed to the filter function.
- **output** (`cupy.ndarray`, *dtype or None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (*int or sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.rank_filter()`

### `cupyx.scipy.ndimage.sobel`

`cupyx.scipy.ndimage.sobel(input, axis=-1, output=None, mode='reflect', cval=0.0)`

Compute a Sobel filter along the given axis.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **axis** (*int*) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, *dtype or None*) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.sobel()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---



**cupyx.scipy.ndimage.uniform\_filter**

`cupyx.scipy.ndimage.uniform_filter(input, size=3, output=None, mode='reflect', cval=0.0, origin=0)`

Multi-dimensional uniform filter.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int` or *sequence of int*) – Lengths of the uniform filter for each dimension. A single value applies to all axes.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.
- **origin** (`int` or *sequence of int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of the filtering.

**Return type**

`cupy.ndarray`

See also:

`scipy.ndimage.uniform_filter()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

**cupyx.scipy.ndimage.uniform\_filter1d**

`cupyx.scipy.ndimage.uniform_filter1d(input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0)`

One-dimensional uniform filter along the given axis.

The lines of the array along the given axis are filtered with a uniform filter of the given size.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **size** (`int`) – Length of the uniform filter.
- **axis** (`int`) – The axis of input along which to calculate. Default is -1.
- **output** (`cupy.ndarray`, `dtype` or `None`) – The array in which to place the output. Default is is same dtype as the input.
- **mode** (`str`) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (`scalar`) – Value to fill past edges of input if mode is 'constant'. Default is 0.0.

- **origin** (*int*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default is 0.

**Returns**

The result of the filtering.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.uniform_filter1d()`

---

**Note:** When the output data type is integral (or when no output is provided and input is integral) the results may not perfectly match the results from SciPy due to floating-point rounding of intermediate results.

---

## Fourier filters

<i>fourier_ellipsoid</i> (input, size[, n, axis, output])	Multidimensional ellipsoid Fourier filter.
<i>fourier_gaussian</i> (input, sigma[, n, axis, output])	Multidimensional Gaussian shift filter.
<i>fourier_shift</i> (input, shift[, n, axis, output])	Multidimensional Fourier shift filter.
<i>fourier_uniform</i> (input, size[, n, axis, output])	Multidimensional uniform shift filter.

### cupyx.scipy.ndimage.fourier\_ellipsoid

`cupyx.scipy.ndimage.fourier_ellipsoid(input, size, n=-1, axis=-1, output=None)`

Multidimensional ellipsoid Fourier filter.

The array is multiplied with the fourier transform of a ellipsoid of given sizes.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*float or sequence of float*) – The size of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.
- **n** (*int, optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (*int, optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (*cupy.ndarray, optional*) – If given, the result of shifting the input is placed in this array.

**Returns**

The filtered output.

**Return type**

output (*cupy.ndarray*)

**cupyx.scipy.ndimage.fourier\_gaussian**

`cupyx.scipy.ndimage.fourier_gaussian(input, sigma, n=-1, axis=-1, output=None)`

Multidimensional Gaussian shift filter.

The array is multiplied with the Fourier transform of a (separable) Gaussian kernel.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **sigma** (`float` or *sequence of float*) – The sigma of the Gaussian kernel. If a float, *sigma* is the same for all axes. If a sequence, *sigma* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns**

The filtered output.

**Return type**

output (`cupy.ndarray`)

**cupyx.scipy.ndimage.fourier\_shift**

`cupyx.scipy.ndimage.fourier_shift(input, shift, n=-1, axis=-1, output=None)`

Multidimensional Fourier shift filter.

The array is multiplied with the Fourier transform of a shift operation.

**Parameters**

- **input** (`cupy.ndarray`) – The input array. This should be in the Fourier domain.
- **shift** (`float` or *sequence of float*) – The size of shift. If a float, *shift* is the same for all axes. If a sequence, *shift* has to contain one value for each axis.
- **n** (`int`, *optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (`int`, *optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

**Returns**

The shifted output (in the Fourier domain).

**Return type**

output (`cupy.ndarray`)

## cupyx.scipy.ndimage.fourier\_uniform

cupyx.scipy.ndimage.fourier\_uniform(*input*, *size*, *n=-1*, *axis=-1*, *output=None*)

Multidimensional uniform shift filter.

The array is multiplied with the Fourier transform of a box of given size.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*float or sequence of float*) – The sigma of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.
- **n** (*int, optional*) – If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.
- **axis** (*int, optional*) – The axis of the real transform (only used when *n* > -1).
- **output** (`cupy.ndarray`, *optional*) – If given, the result of shifting the input is placed in this array.

### Returns

The filtered output.

### Return type

output (`cupy.ndarray`)

## Interpolation

<code>affine_transform</code> ( <i>input</i> , <i>matrix</i> [, <i>offset</i> , ...])	Apply an affine transformation.
<code>map_coordinates</code> ( <i>input</i> , <i>coordinates</i> [, ...])	Map the input array to new coordinates by interpolation.
<code>rotate</code> ( <i>input</i> , <i>angle</i> [, <i>axes</i> , <i>reshape</i> , ...])	Rotate an array.
<code>shift</code> ( <i>input</i> , <i>shift</i> [, <i>output</i> , <i>order</i> , <i>mode</i> , ...])	Shift an array.
<code>spline_filter</code> ( <i>input</i> [, <i>order</i> , <i>output</i> , <i>mode</i> ])	Multidimensional spline filter.
<code>spline_filter1d</code> ( <i>input</i> [, <i>order</i> , <i>axis</i> , ...])	Calculate a 1-D spline filter along the given axis.
<code>zoom</code> ( <i>input</i> , <i>zoom</i> [, <i>output</i> , <i>order</i> , <i>mode</i> , ...])	Zoom an array.

## cupyx.scipy.ndimage.affine\_transform

cupyx.scipy.ndimage.affine\_transform(*input*, *matrix*, *offset=0.0*, *output\_shape=None*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*, \*, *texture\_memory=False*)

Apply an affine transformation.

Given an output image pixel index vector *o*, the pixel value is determined from the input image at position `cupy.dot(matrix, o) + offset`.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **matrix** (`cupy.ndarray`) – The inverse coordinate transformation matrix, mapping output coordinates to input coordinates. If *ndim* is the number of dimensions of *input*, the given matrix must have one of the following shapes:

- (`ndim`, `ndim`): the linear transformation matrix for each output coordinate.
- (`ndim`,): assume that the 2D transformation matrix is diagonal, with the diagonal specified by the given value.
- (`ndim + 1`, `ndim + 1`): assume that the transformation is specified using homogeneous coordinates. In this case, any value passed to `offset` is ignored.
- (`ndim`, `ndim + 1`): as above, but the bottom row of a homogeneous transformation matrix is always `[0, 0, ..., 1]`, and may be omitted.
- **offset** (*float or sequence*) – The offset into the array where the transform is applied. If a float, `offset` is the same for each axis. If a sequence, `offset` should contain one value for each axis.
- **output\_shape** (*tuple of ints*) – Shape tuple.
- **output** (*cupy.ndarray or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (*bool*) – Determines if the input array is prefiltered with `spline_filter` before interpolation. The default is True, which will create a temporary float64 array of filtered values if `order > 1`. If setting this to False, the output will be slightly blurred if `order > 1`, unless the input is prefiltered, i.e. it is the result of calling `spline_filter` on the original input.
- **texture\_memory** (*bool*) – If True, uses GPU texture memory. Supports only:
  - 2D and 3D float32 arrays as input
  - (`ndim + 1`, `ndim + 1`) homogeneous float32 transformation matrix
  - mode='constant' and mode='nearest'
  - **order=0 (nearest neighbor) and order=1 (linear interpolation)**
  - NVIDIA CUDA GPUs

**Returns**

The transformed input. If `output` is given as a parameter, None is returned.

**Return type**

*cupy.ndarray* or None

**See also:**

`scipy.ndimage.affine_transform()`

## cupyx.scipy.ndimage.map\_coordinates

`cupyx.scipy.ndimage.map_coordinates(input, coordinates, output=None, order=3, mode='constant', cval=0.0, prefilter=True)`

Map the input array to new coordinates by interpolation.

The array of coordinates is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

The shape of the output is derived from that of the coordinate array by dropping the first axis. The values of the array along the first axis are the coordinates in the input array at which the output value is found.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **coordinates** (`array_like`) – The coordinates at which **input** is evaluated.
- **output** (`cupy.ndarray` or `dtype`) – The array in which to place the output, or the dtype of the returned array.
- **order** (`int`) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (`str`) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencl').
- **cval** (`scalar`) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencl'. Default is 0.0
- **prefilter** (`bool`) – Determines if the input array is prefiltered with `spline_filter` before interpolation. The default is True, which will create a temporary float64 array of filtered values if `order > 1`. If setting this to False, the output will be slightly blurred if `order > 1`, unless the input is prefiltered, i.e. it is the result of calling `spline_filter` on the original input.

### Returns

The result of transforming the input. The shape of the output is derived from that of coordinates by dropping the first axis.

### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.map_coordinates()`

## cupyx.scipy.ndimage.rotate

`cupyx.scipy.ndimage.rotate(input, angle, axes=(1, 0), reshape=True, output=None, order=3, mode='constant', cval=0.0, prefilter=True)`

Rotate an array.

The array is rotated in the plane defined by the two axes given by the `axes` parameter using spline interpolation of the requested order.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **angle** (`float`) – The rotation angle in degrees.

- **axes** (*tuple of 2 ints*) – The two axes that define the plane of rotation. Default is the first two axes.
- **reshape** (*bool*) – If `reshape` is `True`, the output shape is adapted so that the input array is contained completely in the output. Default is `True`.
- **output** (*cupy.ndarray or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (*bool*) – Determines if the input array is prefiltered with `spline_filter` before interpolation. The default is `True`, which will create a temporary `float64` array of filtered values if `order > 1`. If setting this to `False`, the output will be slightly blurred if `order > 1`, unless the input is prefiltered, i.e. it is the result of calling `spline_filter` on the original input.

**Returns**

The rotated input.

**Return type**

*cupy.ndarray* or `None`

**See also:**

`scipy.ndimage.rotate()`

**cupyx.scipy.ndimage.shift**

`cupyx.scipy.ndimage.shift(input, shift, output=None, order=3, mode='constant', cval=0.0, prefilter=True)`

Shift an array.

The array is shifted using spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **shift** (*float or sequence*) – The shift along the axes. If a float, `shift` is the same for each axis. If a sequence, `shift` should contain one value for each axis.
- **output** (*cupy.ndarray or dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0

- **prefilter** (*bool*) – Determines if the input array is prefiltered with `spline_filter` before interpolation. The default is `True`, which will create a temporary `float64` array of filtered values if `order > 1`. If setting this to `False`, the output will be slightly blurred if `order > 1`, unless the input is prefiltered, i.e. it is the result of calling `spline_filter` on the original input.

**Returns**

The shifted input.

**Return type**

`cupy.ndarray` or `None`

**See also:**

`scipy.ndimage.shift()`

### `cupyx.scipy.ndimage.spline_filter`

`cupyx.scipy.ndimage.spline_filter(input, order=3, output=<class 'numpy.float64'>, mode='mirror')`

Multidimensional spline filter.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **output** (`cupy.ndarray` or *dtype*, *optional*) – The array in which to place the output, or the dtype of the returned array. Default is `numpy.float64`.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').

**Returns**

The result of prefiltering the input.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.spline_filter1d()`

### `cupyx.scipy.ndimage.spline_filter1d`

`cupyx.scipy.ndimage.spline_filter1d(input, order=3, axis=-1, output=<class 'numpy.float64'>, mode='mirror')`

Calculate a 1-D spline filter along the given axis.

The lines of the array along the given axis are filtered by a spline filter. The order of the spline must be  $\geq 2$  and  $\leq 5$ .

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **axis** (*int*) – The axis along which the spline filter is applied. Default is the last axis.



- **output** (`cupy.ndarray` or *dtype*, *optional*) – The array in which to place the output, or the dtype of the returned array. Default is `numpy.float64`.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').

**Returns**

The result of prefiltering the input.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.spline_filter1d()`

**cupyx.scipy.ndimage.zoom**

`cupyx.scipy.ndimage.zoom(input, zoom, output=None, order=3, mode='constant', cval=0.0, prefilter=True, *, grid_mode=False)`

Zoom an array.

The array is zoomed using spline interpolation of the requested order.

**Parameters**

- **input** (`cupy.ndarray`) – The input array.
- **zoom** (*float* or *sequence*) – The zoom factor along the axes. If a float, zoom is the same for each axis. If a sequence, zoom should contain one value for each axis.
- **output** (`cupy.ndarray` or *dtype*) – The array in which to place the output, or the dtype of the returned array.
- **order** (*int*) – The order of the spline interpolation, default is 3. Must be in the range 0-5.
- **mode** (*str*) – Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'mirror', 'reflect', 'wrap', 'grid-mirror', 'grid-wrap', 'grid-constant' or 'opencv').
- **cval** (*scalar*) – Value used for points outside the boundaries of the input if mode='constant' or mode='opencv'. Default is 0.0
- **prefilter** (*bool*) – Determines if the input array is prefiltered with `spline_filter` before interpolation. The default is True, which will create a temporary float64 array of filtered values if `order > 1`. If setting this to False, the output will be slightly blurred if `order > 1`, unless the input is prefiltered, i.e. it is the result of calling `spline_filter` on the original input.
- **grid\_mode** (*bool*, *optional*) – If False, the distance from the pixel centers is zoomed. Otherwise, the distance including the full pixel extent is used. For example, a 1d signal of length 5 is considered to have length 4 when `grid_mode` is False, but length 5 when `grid_mode` is True. See the following visual illustration:

pixel 1	pixel 2	pixel 3	pixel 4	pixel 5	
<----->					
vs.					
<----->					

The starting point of the arrow in the diagram above corresponds to coordinate location 0 in each mode.

**Returns**

The zoomed input.

**Return type**

*cupy.ndarray* or None

**See also:**

`scipy.ndimage.zoom()`

## Measurements

<i>center_of_mass</i> (input[, labels, index])	Calculate the center of mass of the values of an array at labels.
<i>extrema</i> (input[, labels, index])	Calculate the minimums and maximums of the values of an array at labels, along with their positions.
<i>histogram</i> (input, min, max, bins[, labels, index])	Calculate the histogram of the values of an array, optionally at labels.
<i>label</i> (input[, structure, output])	Labels features in an array.
<i>labeled_comprehension</i> (input, labels, index, ...)	Array resulting from applying func to each labeled region.
<i>maximum</i> (input[, labels, index])	Calculate the maximum of the values of an array over labeled regions.
<i>maximum_position</i> (input[, labels, index])	Find the positions of the maximums of the values of an array at labels.
<i>mean</i> (input[, labels, index])	Calculates the mean of the values of an n-D image array, optionally
<i>median</i> (input[, labels, index])	Calculate the median of the values of an array over labeled regions.
<i>minimum</i> (input[, labels, index])	Calculate the minimum of the values of an array over labeled regions.
<i>minimum_position</i> (input[, labels, index])	Find the positions of the minimums of the values of an array at labels.
<i>standard_deviation</i> (input[, labels, index])	Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.
<i>sum_labels</i> (input[, labels, index])	Calculates the sum of the values of an n-D image array, optionally
<i>value_indices</i> (arr, *[, ignore_value, ...])	Find indices of each distinct value in given array.
<i>variance</i> (input[, labels, index])	Calculates the variance of the values of an n-D image array, optionally at specified sub-regions.

**cupyx.scipy.ndimage.center\_of\_mass**

`cupyx.scipy.ndimage.center_of_mass(input, labels=None, index=None)`

Calculate the center of mass of the values of an array at labels.

**Parameters**

- **input** (`cupy.ndarray`) – Data from which to calculate center-of-mass. The masses can either be positive or negative.
- **labels** (`cupy.ndarray`, *optional*) – Labels for objects in *input*, as generated by *ndimage.label*. Only used with *index*. Dimensions must be the same as *input*.
- **index** (*int or sequence of ints*, *optional*) – Labels for which to calculate centers-of-mass. If not specified, all labels greater than zero are used. Only used with *labels*.

**Returns**

Coordinates of centers-of-mass.

**Return type**

*tuple or list of tuples*

See also:

`scipy.ndimage.center_of_mass()`

**cupyx.scipy.ndimage.extrema**

`cupyx.scipy.ndimage.extrema(input, labels=None, index=None)`

Calculate the minimums and maximums of the values of an array at labels, along with their positions.

**Parameters**

- **input** (`cupy.ndarray`) – N-D image data to process.
- **labels** (`cupy.ndarray`, *optional*) – Labels of features in input. If not None, must be same shape as *input*.
- **index** (*int or sequence of ints*, *optional*) – Labels to include in output. If None (default), all values where non-zero *labels* are used.

**Returns**

A tuple that contains the following values.

**minimums** (`cupy.ndarray`): Values of minimums in each feature.

**maximums** (`cupy.ndarray`): Values of maximums in each feature.

**min\_positions** (*tuple or list of tuples*): Each tuple gives the N-D coordinates of the corresponding minimum.

**max\_positions** (*tuple or list of tuples*): Each tuple gives the N-D coordinates of the corresponding maximum.

See also:

`scipy.ndimage.extrema()`

### cupyx.scipy.ndimage.histogram

cupyx.scipy.ndimage.**histogram**(*input*, *min*, *max*, *bins*, *labels=None*, *index=None*)

Calculate the histogram of the values of an array, optionally at labels.

Histogram calculates the frequency of values in an array within bins determined by *min*, *max*, and *bins*. The *labels* and *index* keywords can limit the scope of the histogram to specified sub-regions within the array.

#### Parameters

- **input** (`cupy.ndarray`) – Data for which to calculate histogram.
- **min** (`int`) – Minimum values of range of histogram bins.
- **max** (`int`) – Maximum values of range of histogram bins.
- **bins** (`int`) – Number of bins.
- **labels** (`cupy.ndarray`, *optional*) – Labels for objects in *input*. If not None, must be same shape as *input*.
- **index** (`int` or *sequence of ints*, *optional*) – Label or labels for which to calculate histogram. If None, all values where label is greater than zero are used.

#### Returns

Histogram counts.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.histogram()`

### cupyx.scipy.ndimage.label

cupyx.scipy.ndimage.**label**(*input*, *structure=None*, *output=None*)

Labels features in an array.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **structure** (*array\_like* or *None*) – A structuring element that defines feature connections. `structure`` must be centersymmetric. If None, structure is automatically generated with a squared connectivity equal to one.
- **output** (`cupy.ndarray`, *dtype* or *None*) – The array in which to place the output.

#### Returns

An integer array where each unique feature in `input`` has a unique label in the array.

`num_features` (int): Number of features found.

#### Return type

label (`cupy.ndarray`)

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.label()`

### `cupyx.scipy.ndimage.labeled_comprehension`

`cupyx.scipy.ndimage.labeled_comprehension(input, labels, index, func, out_dtype, default, pass_positions=False)`

Array resulting from applying `func` to each labeled region.

Roughly equivalent to `[func(input[labels == i]) for i in index]`.

Sequentially applies an arbitrary function (that works on array\_like input) to subsets of an N-D image array specified by `labels` and `index`. The option exists to provide the function with positional parameters as the second argument.

#### Parameters

- **input** (`cupy.ndarray`) – Data from which to select `labels` to process.
- **labels** (`cupy.ndarray` or `None`) – Labels to objects in `input`. If not `None`, array must be same shape as `input`. If `None`, `func` is applied to raveled `input`.
- **index** (`int`, sequence of `ints` or `None`) – Subset of `labels` to which to apply `func`. If a scalar, a single value is returned. If `None`, `func` is applied to all non-zero values of `labels`.
- **func** (`callable`) – Python function to apply to `labels` from `input`.
- **out\_dtype** (`dtype`) – Dtype to use for `result`.
- **default** (`int`, `float` or `None`) – Default return value when a element of `index` does not exist in `labels`.
- **pass\_positions** (`bool`, optional) – If True, pass linear indices to `func` as a second argument.

#### Returns

Result of applying `func` to each of `labels` to `input` in `index`.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.labeled_comprehension()`

### `cupyx.scipy.ndimage.maximum`

`cupyx.scipy.ndimage.maximum(input, labels=None, index=None)`

Calculate the maximum of the values of an array over labeled regions.

#### Parameters

- **input** (`cupy.ndarray`) – Array of values. For each region specified by `labels`, the maximal values of `input` over the region is computed.
- **labels** (`cupy.ndarray`, optional) – An array of integers marking different regions over which the maximum value of `input` is to be computed. `labels` must have the same shape as `input`. If `labels` is not specified, the maximum over the whole array is returned.

- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the maxima. If *index* is None, the maximum over all elements where *labels* is non-zero is returned.

**Returns**

Array of maxima of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a 0-dimensional `cupy.ndarray` is returned: the maximal value of *input* if *labels* is None, and the maximal value of elements where *labels* is greater than zero if *index* is None.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.ndimage.maximum()`

### `cupyx.scipy.ndimage.maximum_position`

`cupyx.scipy.ndimage.maximum_position(input, labels=None, index=None)`

Find the positions of the maximums of the values of an array at labels.

For each region specified by *labels*, the position of the maximum value of *input* within the region is returned.

**Parameters**

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the maximal values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the position of the maximum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first maximum over the whole array is returned.

The *labels* argument only works when *index* is specified.

- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for finding the location of the maxima. If *index* is None, the first maximum over all elements where *labels* is non-zero is returned.

The *index* argument only works when *labels* is specified.

**Returns**

Tuple of ints or list of tuples of ints that specify the location of maxima of *input* over the regions determined by *labels* and whose index is in *index*.

If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the first maximal value of *input*.

---

**Note:** When *input* has multiple identical maxima within a labeled region, the coordinates returned are not guaranteed to match those returned by SciPy.

---

**See also:**

`scipy.ndimage.maximum_position()`

**cupyx.scipy.ndimage.mean**

`cupyx.scipy.ndimage.mean(input, labels=None, index=None)`

Calculates the mean of the values of an n-D image array, optionally at specified sub-regions.

**Parameters**

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns**

mean of values, for each sub-region if *labels* and *index* are specified.

**Return type**

mean (`cupy.ndarray`)

See also:

`scipy.ndimage.mean()`

**cupyx.scipy.ndimage.median**

`cupyx.scipy.ndimage.median(input, labels=None, index=None)`

Calculate the median of the values of an array over labeled regions.

**Parameters**

- **input** (`cupy.ndarray`) – Array of values. For each region specified by *labels*, the median values of *input* over the region is computed.
- **labels** (`cupy.ndarray`, *optional*) – An array of integers marking different regions over which the median value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the median over the whole array is returned.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the medians. If *index* is `None`, the median over all elements where *labels* is non-zero is returned.

**Returns**

Array of medians of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a 0-dimensional `cupy.ndarray` is returned: the median value of *input* if *labels* is `None`, and the median value of elements where *labels* is greater than zero if *index* is `None`.

**Return type**

`cupy.ndarray`

See also:

`scipy.ndimage.median()`

## cupyx.scipy.ndimage.minimum

cupyx.scipy.ndimage.minimum(*input*, *labels=None*, *index=None*)

Calculate the minimum of the values of an array over labeled regions.

### Parameters

- **input** ([cupy.ndarray](#)) – Array of values. For each region specified by *labels*, the minimal values of *input* over the region is computed.
- **labels** ([cupy.ndarray](#), *optional*) – An array of integers marking different regions over which the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the minimum over the whole array is returned.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for computing the minima. If *index* is None, the minimum over all elements where *labels* is non-zero is returned.

### Returns

Array of minima of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a 0-dimensional [cupy.ndarray](#) is returned: the minimal value of *input* if *labels* is None, and the minimal value of elements where *labels* is greater than zero if *index* is None.

### Return type

[cupy.ndarray](#)

See also:

[scipy.ndimage.minimum\(\)](#)

## cupyx.scipy.ndimage.minimum\_position

cupyx.scipy.ndimage.minimum\_position(*input*, *labels=None*, *index=None*)

Find the positions of the minimums of the values of an array at labels.

For each region specified by *labels*, the position of the minimum value of *input* within the region is returned.

### Parameters

- **input** ([cupy.ndarray](#)) – Array of values. For each region specified by *labels*, the minimal values of *input* over the region is computed.
- **labels** ([cupy.ndarray](#), *optional*) – An array of integers marking different regions over which the position of the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first minimum over the whole array is returned.  
  
The *labels* argument only works when *index* is specified.
- **index** (*array\_like*, *optional*) – A list of region labels that are taken into account for finding the location of the minima. If *index* is None, the first minimum over all elements where *labels* is non-zero is returned.

The *index* argument only works when *labels* is specified.

### Returns

Tuple of ints or list of tuples of ints that specify the location of minima of *input* over the regions determined by *labels* and whose index is in *index*.



If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the first minimal value of *input*.

---

**Note:** When *input* has multiple identical minima within a labeled region, the coordinates returned are not guaranteed to match those returned by SciPy.

---

**See also:**

`scipy.ndimage.minimum_position()`

### cupyx.scipy.ndimage.standard\_deviation

`cupyx.scipy.ndimage.standard_deviation(input, labels=None, index=None)`

Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

#### Returns

standard deviation of values, for each sub-region if *labels* and *index* are specified.

#### Return type

standard\_deviation (`cupy.ndarray`)

**See also:**

`scipy.ndimage.standard_deviation()`

### cupyx.scipy.ndimage.sum\_labels

`cupyx.scipy.ndimage.sum_labels(input, labels=None, index=None)`

**Calculates the sum of the values of an n-D image array, optionally**  
at specified sub-regions.

#### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

#### Returns

sum of values, for each sub-region if *labels* and *index* are specified.

#### Return type

sum (`cupy.ndarray`)

See also:

`scipy.ndimage.sum_labels()`

### `cupyx.scipy.ndimage.value_indices`

`cupyx.scipy.ndimage.value_indices(arr, *, ignore_value=None, adaptive_index_dtype=False)`

Find indices of each distinct value in given array.

#### Parameters

- **arr** (`ndarray of ints`) – Array containing integer values.
- **ignore\_value** (`int, optional`) – This value will be ignored in searching the *arr* array. If not given, all values found will be included in output. Default is `None`.
- **adaptive\_index\_dtype** (`bool, optional`) – If `True`, instead of returning the default CuPy signed integer dtype, the smallest signed integer dtype capable of representing the image coordinate range will be used. This can substantially reduce memory usage and slightly reduce runtime. Note that this optional parameter is not available in the SciPy API.

#### Returns

**indices** – A Python dictionary of array indices for each distinct value. The dictionary is keyed by the distinct values, the entries are array index tuples covering all occurrences of the value within the array.

This dictionary can occupy significant memory, often several times the size of the input array. To help reduce memory overhead, the argument *adaptive\_index\_dtype* can be set to `True`.

#### Return type

dictionary

#### Notes

For a small array with few distinct values, one might use `numpy.unique()` to find all possible values, and `(arr == val)` to locate each value within that array. However, for large arrays, with many distinct values, this can become extremely inefficient, as locating each value would require a new search through the entire array. Using this function, there is essentially one search, with the indices saved for all distinct values.

This is useful when matching a categorical image (e.g. a segmentation or classification) to an associated image of other data, allowing any per-class statistic(s) to then be calculated. Provides a more flexible alternative to functions like `scipy.ndimage.mean()` and `scipy.ndimage.variance()`.

Some other closely related functionality, with different strengths and weaknesses, can also be found in `scipy.stats.binned_statistic()` and the `scikit-image` function `skimage.measure.regionprops()`.

Note for IDL users: this provides functionality equivalent to IDL's `REVERSE_INDICES` option (as per the IDL documentation for the `HISTOGRAM` function).

New in version 1.10.0.

See also:

`label`, `maximum`, `median`, `minimum_position`, `extrema`, `sum`, `mean`, `variance`, `standard_deviation`, `cupy.where`, `cupy.unique`

## Examples

```
>>> import cupy
>>> from cupyx.scipy import ndimage
>>> a = cupy.zeros((6, 6), dtype=int)
>>> a[2:4, 2:4] = 1
>>> a[4, 4] = 1
>>> a[:2, :3] = 2
>>> a[0, 5] = 3
>>> a
array([[2, 2, 2, 0, 0, 3],
       [2, 2, 2, 0, 0, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0]])
>>> val_indices = ndimage.value_indices(a)
```

The dictionary `val_indices` will have an entry for each distinct value in the input array.

```
>>> val_indices.keys()
dict_keys([0, 1, 2, 3])
```

The entry for each value is an index tuple, locating the elements with that value.

```
>>> ndx1 = val_indices[1]
>>> ndx1
(array([2, 2, 3, 3, 4]), array([2, 3, 2, 3, 4]))
```

This can be used to index into the original array, or any other array with the same shape.

```
>>> a[ndx1]
array([1, 1, 1, 1, 1])
```

If the zeros were to be ignored, then the resulting dictionary would no longer have an entry for zero.

```
>>> val_indices = ndimage.value_indices(a, ignore_value=0)
>>> val_indices.keys()
dict_keys([1, 2, 3])
```

## cupyx.scipy.ndimage.variance

`cupyx.scipy.ndimage.variance(input, labels=None, index=None)`

Calculates the variance of the values of an n-D image array, optionally at specified sub-regions.

### Parameters

- **input** (`cupy.ndarray`) – Nd-image data to process.
- **labels** (`cupy.ndarray` or `None`) – Labels defining sub-regions in *input*. If not `None`, must be same shape as *input*.
- **index** (`cupy.ndarray` or `None`) – *labels* to include in output. If `None` (default), all values where *labels* is non-zero are used.

**Returns**

Values of variance, for each sub-region if *labels* and *index* are specified.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.variance()`

## Morphology

<code>binary_closing</code> (input[, structure, ...])	Multidimensional binary closing with the given structuring element.
<code>binary_dilation</code> (input[, structure, ...])	Multidimensional binary dilation with the given structuring element.
<code>binary_erosion</code> (input[, structure, ...])	Multidimensional binary erosion with a given structuring element.
<code>binary_fill_holes</code> (input[, structure, ...])	Fill the holes in binary objects.
<code>binary_hit_or_miss</code> (input[, structure1, ...])	Multidimensional binary hit-or-miss transform.
<code>binary_opening</code> (input[, structure, ...])	Multidimensional binary opening with the given structuring element.
<code>binary_propagation</code> (input[, structure, mask, ...])	Multidimensional binary propagation with the given structuring element.
<code>black_tophat</code> (input[, size, footprint, ...])	Multidimensional black tophat filter.
<code>distance_transform_edt</code> (image[, sampling, ...])	Exact Euclidean distance transform.
<code>generate_binary_structure</code> (rank, connectivity)	Generate a binary structure for binary morphological operations.
<code>grey_closing</code> (input[, size, footprint, ...])	Calculates a multi-dimensional greyscale closing.
<code>grey_dilation</code> (input[, size, footprint, ...])	Calculates a greyscale dilation.
<code>grey_erosion</code> (input[, size, footprint, ...])	Calculates a greyscale erosion.
<code>grey_opening</code> (input[, size, footprint, ...])	Calculates a multi-dimensional greyscale opening.
<code>iterate_structure</code> (structure, iterations[, ...])	Iterate a structure by dilating it with itself.
<code>morphological_gradient</code> (input[, size, ...])	Multidimensional morphological gradient.
<code>morphological_laplace</code> (input[, size, ...])	Multidimensional morphological laplace.
<code>white_tophat</code> (input[, size, footprint, ...])	Multidimensional white tophat filter.

## cupyx.scipy.ndimage.binary\_closing

`cupyx.scipy.ndimage.binary_closing`(input, structure=None, iterations=1, output=None, origin=0, mask=None, border\_value=0, brute\_force=False)

Multidimensional binary closing with the given structuring element.

The *closing* of an input image by a structuring element is the *erosion* of the *dilation* of the image by the structuring element.

**Parameters**

- **input** (*cupy.ndarray*) – The input binary array to be closed. Non-zero (True) elements form the subset to be closed.
- **structure** (*cupy.ndarray*, *optional*) – The structuring element used for the closing. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).

- **iterations** (*int*, *optional*) – The closing is repeated *iterations* times (one, by default). If *iterations* is less than 1, the closing is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **output** (*cupy.ndarray*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **origin** (*int* or *tuple of ints*, *optional*) – Placement of the filter, by default 0.
- **mask** (*cupy.ndarray* or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **border\_value** (*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)
- **brute\_force** (*boolean*, *optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for closing, regardless of what happened in the previous iteration.

**Returns**

The result of binary closing.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_closing()`

### cupyx.scipy.ndimage.binary\_dilation

`cupyx.scipy.ndimage.binary_dilation(input, structure=None, iterations=1, mask=None, output=None, border_value=0, origin=0, brute_force=False)`

Multidimensional binary dilation with the given structuring element.

**Parameters**

- **input** (*cupy.ndarray*) – The input binary array\_like to be dilated. Non-zero (True) elements form the subset to be dilated.
- **structure** (*cupy.ndarray*, *optional*) – The structuring element used for the dilation. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).
- **iterations** (*int*, *optional*) – The dilation is repeated *iterations* times (one, by default). If *iterations* is less than 1, the dilation is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **mask** (*cupy.ndarray* or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **output** (*cupy.ndarray*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.

- **border\_value**(*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)
- **origin**(*int* or *tuple of ints*, *optional*) – Placement of the filter, by default 0.
- **brute\_force**(*boolean*, *optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for dilation, regardless of what happened in the previous iteration.

**Returns**

The result of binary dilation.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

See also:

*scipy.ndimage.binary\_dilation()*

## cupyx.scipy.ndimage.binary\_erosion

*cupyx.scipy.ndimage.binary\_erosion*(*input*, *structure=None*, *iterations=1*, *mask=None*, *output=None*, *border\_value=0*, *origin=0*, *brute\_force=False*)

Multidimensional binary erosion with a given structuring element.

Binary erosion is a mathematical morphology operation used for image processing.

**Parameters**

- **input** (*cupy.ndarray*) – The input binary array\_like to be eroded. Non-zero (True) elements form the subset to be eroded.
- **structure** (*cupy.ndarray*, *optional*) – The structuring element used for the erosion. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).
- **iterations** (*int*, *optional*) – The erosion is repeated *iterations* times (one, by default). If *iterations* is less than 1, the erosion is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **mask** (*cupy.ndarray* or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **output** (*cupy.ndarray*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **border\_value**(*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)
- **origin**(*int* or *tuple of ints*, *optional*) – Placement of the filter, by default 0.
- **brute\_force**(*boolean*, *optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (eroded) in the current iteration; if True all pixels are considered as candidates for erosion, regardless of what happened in the previous iteration.

**Returns**

The result of binary erosion.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

**See also:**

`scipy.ndimage.binary_erosion()`

**cupyx.scipy.ndimage.binary\_fill\_holes**

`cupyx.scipy.ndimage.binary_fill_holes(input, structure=None, output=None, origin=0)`

Fill the holes in binary objects.

**Parameters**

- **input** (*cupy.ndarray*) – N-D binary array with holes to be filled.
- **structure** (*cupy.ndarray*, *optional*) – Structuring element used in the computation; large-size elements make computations faster but may miss holes separated from the background by thin regions. The default element (with a square connectivity equal to one) yields the intuitive result where all holes in the input have been filled.
- **output** (*cupy.ndarray*, *dtype* or *None*, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **origin** (*int*, *tuple of ints*, *optional*) – Position of the structuring element.

**Returns**

Transformation of the initial image `input` where holes have been filled.

**Return type**

*cupy.ndarray*

**Warning:** This function may synchronize the device.

**See also:**

`scipy.ndimage.binary_fill_holes()`

**cupyx.scipy.ndimage.binary\_hit\_or\_miss**

`cupyx.scipy.ndimage.binary_hit_or_miss(input, structure1=None, structure2=None, output=None, origin1=0, origin2=None)`

Multidimensional binary hit-or-miss transform.

The hit-or-miss transform finds the locations of a given pattern inside the input image.

**Parameters**

- **input** (*cupy.ndarray*) – Binary image where a pattern is to be detected.

- **structure1** (`cupy.ndarray`, *optional*) – Part of the structuring element to be fitted to the foreground (non-zero elements) of `input`. If no value is provided, a structure of square connectivity 1 is chosen.
- **structure2** (`cupy.ndarray`, *optional*) – Second part of the structuring element that has to miss completely the foreground. If no value is provided, the complementary of `structure1` is taken.
- **output** (`cupy.ndarray`, *dtype or None, optional*) – Array of the same shape as `input`, into which the output is placed. By default, a new array is created.
- **origin1** (*int or tuple of ints, optional*) – Placement of the first part of the structuring element `structure1`, by default 0 for a centered structure.
- **origin2** (*int or tuple of ints or None, optional*) – Placement of the second part of the structuring element `structure2`, by default 0 for a centered structure. If a value is provided for `origin1` and not for `origin2`, then `origin2` is set to `origin1`.

**Returns**

Hit-or-miss transform of `input` with the given structuring element (`structure1`, `structure2`).

**Return type**

`cupy.ndarray`

**Warning:** This function may synchronize the device.

**See also:**

`scipy.ndimage.binary_hit_or_miss()`

### `cupyx.scipy.ndimage.binary_opening`

`cupyx.scipy.ndimage.binary_opening`(*input*, *structure=None*, *iterations=1*, *output=None*, *origin=0*, *mask=None*, *border\_value=0*, *brute\_force=False*)

Multidimensional binary opening with the given structuring element.

The *opening* of an input image by a structuring element is the *dilation* of the *erosion* of the image by the structuring element.

**Parameters**

- **input** (`cupy.ndarray`) – The input binary array to be opened. Non-zero (True) elements form the subset to be opened.
- **structure** (`cupy.ndarray`, *optional*) – The structuring element used for the opening. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one. (Default value = None).
- **iterations** (*int, optional*) – The opening is repeated `iterations` times (one, by default). If `iterations` is less than 1, the opening is repeated until the result does not change anymore. Only an integer of iterations is accepted.
- **output** (`cupy.ndarray`, *optional*) – Array of the same shape as `input`, into which the output is placed. By default, a new array is created.
- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.



- **mask** (`cupy.ndarray` or *None*, *optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration. (Default value = None)
- **border\_value** (*int* (cast to 0 or 1), *optional*) – Value at the border in the output array. (Default value = 0)
- **brute\_force** (*boolean*, *optional*) – Memory condition: if False, only the pixels whose value was changed in the last iteration are tracked as candidates to be updated (dilated) in the current iteration; if True all pixels are considered as candidates for opening, regardless of what happened in the previous iteration.

**Returns**

The result of binary opening.

**Return type**

`cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_opening()`

## cupyx.scipy.ndimage.binary\_propagation

`cupyx.scipy.ndimage.binary_propagation(input, structure=None, mask=None, output=None, border_value=0, origin=0)`

Multidimensional binary propagation with the given structuring element.

**Parameters**

- **input** (`cupy.ndarray`) – Binary image to be propagated inside mask.
- **structure** (`cupy.ndarray`, *optional*) – Structuring element used in the successive dilations. The output may depend on the structuring element, especially if mask has several connex components. If no structuring element is provided, an element is generated with a squared connectivity equal to one.
- **mask** (`cupy.ndarray`, *optional*) – Binary mask defining the region into which input is allowed to propagate.
- **output** (`cupy.ndarray`, *optional*) – Array of the same shape as input, into which the output is placed. By default, a new array is created.
- **border\_value** (*int*, *optional*) – Value at the border in the output array. The value is cast to 0 or 1.
- **origin** (*int* or *tuple of ints*, *optional*) – Placement of the filter.

**Returns**

Binary propagation of input inside mask.

**Return type**

`cupy.ndarray`

**Warning:** This function may synchronize the device.

See also:

`scipy.ndimage.binary_propagation()`

### `cupyx.scipy.ndimage.black_tophat`

`cupyx.scipy.ndimage.black_tophat(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional black tophat filter.

#### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the black tophat. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for the black tophat. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the black tophat. `structure` may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

#### Returns

Result of the filter of `input` with `structure`.

#### Return type

`cupy.ndarray`

See also:

`scipy.ndimage.black_tophat()`

### `cupyx.scipy.ndimage.distance_transform_edt`

`cupyx.scipy.ndimage.distance_transform_edt(image, sampling=None, return_distances=True, return_indices=False, distances=None, indices=None, *, block_params=None, float64_distances=True)`

Exact Euclidean distance transform.

This function calculates the distance transform of the *input*, by replacing each foreground (non-zero) element, with its shortest distance to the background (any zero-valued element).

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element to each foreground element is returned in a separate array.

### Parameters

- **image** (*array\_like*) – Input data to transform. Can be any type but will be converted into binary: 1 wherever image equates to True, 0 elsewhere.
- **sampling** (*float, or sequence of float, optional*) – Spacing of elements along each dimension. If a sequence, must be of length equal to the image rank; if a single number, this is used for all axes. If not specified, a grid spacing of unity is implied.
- **return\_distances** (*bool, optional*) – Whether to calculate the distance transform.
- **return\_indices** (*bool, optional*) – Whether to calculate the feature transform.
- **distances** (*cupy.ndarray, optional*) – An output array to store the calculated distance transform, instead of returning it. *return\_distances* must be True. It must be the same shape as *image*. Should have dtype `cp.float32` if *float64\_distances* is False, otherwise it should be `cp.float64`.
- **indices** (*cupy.ndarray, optional*) – An output array to store the calculated feature transform, instead of returning it. *return\_indices* must be True. Its shape must be `(image.ndim,) + image.shape`. Its dtype must be a signed or unsigned integer type of at least 16-bits in 2D or 32-bits in 3D.
- **block\_params** (*3-tuple of int*) – The m1, m2, m3 algorithm parameters as described in<sup>2</sup>. If None, suitable defaults will be chosen. Note: This parameter is specific to cuCIM and does not exist in SciPy.
- **float64\_distances** (*bool, optional*) – If True, use double precision in the distance computation (to match SciPy behavior). Otherwise, single precision will be used for efficiency. Note: This parameter is specific to cuCIM and does not exist in SciPy.

### Returns

- **distances** (*cupy.ndarray, optional*) – The calculated distance transform. Returned only when *return\_distances* is True and *distances* is not supplied. It will have the same shape as *image*. Will have dtype `cp.float64` if *float64\_distances* is True, otherwise it will have dtype `cp.float32`.
- **indices** (*ndarray, optional*) – The calculated feature transform. It has an image-shaped array for each dimension of the image. See example below. Returned only when *return\_indices* is True and *indices* is not supplied.

### Notes

The Euclidean distance transform gives values of the Euclidean distance.

$$y_i = \sqrt{\sum_i^n (x[i] - b[i])^2}$$

where  $b[i]$  is the background point (value 0) with the smallest Euclidean distance to input points  $x[i]$ , and  $n$  is the number of dimensions.

Note that the *indices* output may differ from the one given by `scipy.ndimage.distance_transform_edt()` in the case of input pixels that are equidistant from multiple background points.

<sup>2</sup> <https://www.comp.nus.edu.sg/~tants/pba.html>

The parallel banding algorithm implemented here was originally described in<sup>1</sup>. The kernels used here correspond to the revised PBA+ implementation that is described on the author's website<sup>2</sup>. The source code of the author's PBA+ implementation is available at<sup>3</sup>.

## References

## Examples

```
>>> import cupy as cp
>>> from cucim.core.operations import morphology
>>> a = cp.array([[0,1,1,1,1],
...               [0,0,1,1,1],
...               [0,1,1,1,1],
...               [0,1,1,1,0],
...               [0,1,1,0,0]])
>>> morphology.distance_transform_edt(a)
array([[ 0.    ,  1.    ,  1.4142,  2.2361,  3.    ],
       [ 0.    ,  0.    ,  1.    ,  2.    ,  2.    ],
       [ 0.    ,  1.    ,  1.4142,  1.4142,  1.    ],
       [ 0.    ,  1.    ,  1.4142,  1.    ,  0.    ],
       [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
```

With a sampling of 2 units along x, 1 along y:

```
>>> morphology.distance_transform_edt(a, sampling=[2,1])
array([[ 0.    ,  1.    ,  2.    ,  2.8284,  3.6056],
       [ 0.    ,  0.    ,  1.    ,  2.    ,  3.    ],
       [ 0.    ,  1.    ,  2.    ,  2.2361,  2.    ],
       [ 0.    ,  1.    ,  2.    ,  1.    ,  0.    ],
       [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
```

Asking for indices as well:

```
>>> edt, inds = morphology.distance_transform_edt(a, return_indices=True)
>>> inds
array([[0, 0, 1, 1, 3],
       [1, 1, 1, 1, 3],
       [2, 2, 1, 3, 3],
       [3, 3, 4, 4, 3],
       [4, 4, 4, 4, 4],
       [0, 0, 1, 1, 4],
       [0, 1, 1, 1, 4],
       [0, 0, 1, 4, 4],
       [0, 0, 3, 3, 4],
       [0, 0, 3, 3, 4]])
```

<sup>1</sup> Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. 2010. Parallel Banding Algorithm to compute exact distance transform with the GPU. In Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10). Association for Computing Machinery, New York, NY, USA, 83–90. DOI:<https://doi.org/10.1145/1730804.1730818>

<sup>3</sup> <https://github.com/orzzzjq/Parallel-Banding-Algorithm-plus>

## cupyx.scipy.ndimage.generate\_binary\_structure

cupyx.scipy.ndimage.generate\_binary\_structure(rank, connectivity)

Generate a binary structure for binary morphological operations.

### Parameters

- **rank** (*int*) – Number of dimensions of the array to which the structuring element will be applied, as returned by `np.ndim`.
- **connectivity** (*int*) – `connectivity` determines which elements of the output array belong to the structure, i.e., are considered as neighbors of the central element. Elements up to a squared distance of `connectivity` from the center are considered neighbors. `connectivity` may range from 1 (no diagonal elements are neighbors) to `rank` (all elements are neighbors).

### Returns

Structuring element which may be used for binary morphological operations, with `rank` dimensions and all dimensions equal to 3.

### Return type

*cupy.ndarray*

See also:

`scipy.ndimage.generate_binary_structure()`

## cupyx.scipy.ndimage.grey\_closing

cupyx.scipy.ndimage.grey\_closing(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)

Calculates a multi-dimensional greyscale closing.

### Parameters

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale closing. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale closing. Non-zero values give the set of neighbors of the center over which closing is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale closing. structure may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is `0.0`.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

### Returns

The result of greyscale closing.

**Return type***cupy.ndarray***See also:**`scipy.ndimage.grey_closing()`**cupyx.scipy.ndimage.grey\_dilation**

`cupyx.scipy.ndimage.grey_dilation(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a greyscale dilation.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale dilation. Optional if *footprint* or *structure* is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale dilation. Non-zero values give the set of neighbors of the center over which maximum is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale dilation. *structure* may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of greyscale dilation.

**Return type***cupy.ndarray***See also:**`scipy.ndimage.grey_dilation()`**cupyx.scipy.ndimage.grey\_erosion**

`cupyx.scipy.ndimage.grey_erosion(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a greyscale erosion.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale erosion. Optional if *footprint* or *structure* is provided.

- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale erosion. Non-zero values give the set of neighbors of the center over which minimum is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale erosion. structure may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of greyscale erosion.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.grey_erosion()`

**cupyx.scipy.ndimage.grey\_opening**

`cupyx.scipy.ndimage.grey_opening(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Calculates a multi-dimensional greyscale opening.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the greyscale opening. Optional if footprint or structure is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for greyscale opening. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the greyscale opening. structure may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

**Returns**

The result of greyscale opening.

**Return type***cupy.ndarray***See also:**`scipy.ndimage.grey_opening()`**cupyx.scipy.ndimage.iterate\_structure**`cupyx.scipy.ndimage.iterate_structure(structure, iterations, origin=None)`

Iterate a structure by dilating it with itself.

**Parameters**

- **structure** (*array\_like*) – Structuring element (an array of bools, for example), to be dilated with itself.
- **iterations** (*int*) – The number of dilations performed on the structure with itself.
- **origin** (*int or tuple of int, optional*) – If origin is None, only the iterated structure is returned. If not, a tuple of the iterated structure and the modified origin is returned.

**Returns**

A new structuring element obtained by dilating `structure` (`iterations - 1`) times with itself.

**Return type***cupy.ndarray***See also:**`scipy.ndimage.iterate_structure()`**cupyx.scipy.ndimage.morphological\_gradient**`cupyx.scipy.ndimage.morphological_gradient(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional morphological gradient.

The morphological gradient is calculated as the difference between a dilation and an erosion of the input with a given structuring element.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the morphological gradient. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for morphological gradient. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the morphological gradient. `structure` may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is `0.0`.



- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The morphological gradient of the input.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.morphological_gradient()`

**cupyx.scipy.ndimage.morphological\_laplace**

`cupyx.scipy.ndimage.morphological_laplace(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional morphological laplace.

**Parameters**

- **input** (*cupy.ndarray*) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the morphological laplace. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for morphological laplace. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the morphological laplace. structure may be a non-flat structuring element.
- **output** (*cupy.ndarray, dtype or None*) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

**Returns**

The morphological laplace of the input.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.ndimage.morphological_laplace()`

## cupyx.scipy.ndimage.white\_tophat

`cupyx.scipy.ndimage.white_tophat(input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0)`

Multidimensional white tophat filter.

### Parameters

- **input** (`cupy.ndarray`) – The input array.
- **size** (*tuple of ints*) – Shape of a flat and full structuring element used for the white tophat. Optional if `footprint` or `structure` is provided.
- **footprint** (*array of ints*) – Positions of non-infinite elements of a flat structuring element used for the white tophat. Non-zero values give the set of neighbors of the center over which opening is chosen.
- **structure** (*array of ints*) – Structuring element used for the white tophat. `structure` may be a non-flat structuring element.
- **output** (`cupy.ndarray`, *dtype* or `None`) – The array in which to place the output.
- **mode** (*str*) – The array borders are handled according to the given mode ('reflect', 'constant', 'nearest', 'mirror', 'wrap'). Default is 'reflect'.
- **cval** (*scalar*) – Value to fill past edges of input if mode is constant. Default is 0.0.
- **origin** (*scalar or tuple of scalar*) – The origin parameter controls the placement of the filter, relative to the center of the current element of the input. Default of 0 is equivalent to (0,)\*input.ndim.

### Returns

Result of the filter of input with structure.

### Return type

*cupy.ndarray*

See also:

`scipy.ndimage.white_tophat()`

## OpenCV mode

`cupyx.scipy.ndimage` supports additional mode, `opencv`. If it is given, the function performs like `cv2.warpAffine` or `cv2.resize`. Example:

```
import cupyx.scipy.ndimage
import cupy as cp
import cv2

im = cv2.imread('TODO') # pls fill in your image path

trans_mat = cp.eye(4)
trans_mat[0][0] = trans_mat[1][1] = 0.5

smaller_shape = (im.shape[0] // 2, im.shape[1] // 2, 3)
smaller = cp.zeros(smaller_shape) # preallocate memory for resized image
```

(continues on next page)

(continued from previous page)

```
cupyx.scipy.ndimage.affine_transform(im, trans_mat, output_shape=smaller_shape,
                                     output=smaller, mode='opencv')

cv2.imwrite('smaller.jpg', cp.asnumpy(smaller)) # smaller image saved locally
```

## 5.4.6 Signal processing (cupyx.scipy.signal)

**Hint:** SciPy API Reference: Signal processing (scipy.signal)

### Convolution

<code>convolve(in1, in2[, mode, method])</code>	Convolve two N-dimensional arrays.
<code>correlate(in1, in2[, mode, method])</code>	Cross-correlate two N-dimensional arrays.
<code>fftconvolve(in1, in2[, mode, axes])</code>	Convolve two N-dimensional arrays using FFT.
<code>oaconvolve(in1, in2[, mode, axes])</code>	Convolve two N-dimensional arrays using the overlap-add method.
<code>convolve2d(in1, in2[, mode, boundary, fillvalue])</code>	Convolve two 2-dimensional arrays.
<code>correlate2d(in1, in2[, mode, boundary, ...])</code>	Cross-correlate two 2-dimensional arrays.
<code>sepfir2d(input, hrow, hcol)</code>	Convolve with a 2-D separable FIR filter.
<code>choose_conv_method(in1, in2[, mode])</code>	Find the fastest convolution/correlation method.
<code>correlation_lags(in1_len, in2_len[, mode])</code>	Calculates the lag / displacement indices array for 1D cross-correlation.

### cupyx.scipy.signal.convolve

`cupyx.scipy.signal.convolve(in1, in2, mode='full', method='auto')`

Convolve two N-dimensional arrays.

Convolve `in1` and `in2`, with the output size determined by the `mode` argument.

#### Parameters

- **in1** (`cupy.ndarray`) – First input.
- **in2** (`cupy.ndarray`) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (`str`) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **method** (`str`) – Indicates which method to use for the computations:
  - 'direct': The convolution is determined directly from sums, the definition of convolution
  - 'fft': The Fourier Transform is used to perform the convolution by calling `fftconvolve`.

- 'auto': Automatically choose direct or FFT based on an estimate of which is faster for the arguments (default).

**Returns**

the result of convolution.

**Return type**

*cupy.ndarray*

**See also:**

*cupyx.scipy.signal.choose\_conv\_method()*

**See also:**

*cupyx.scipy.signal.correlation()*

**See also:**

*cupyx.scipy.signal.fftconvolve()*

**See also:**

*cupyx.scipy.signal.oaconvolve()*

**See also:**

*cupyx.scipy.ndimage.convolve()*

**See also:**

*scipy.signal.convolve()*

---

**Note:** By default, `convolve` and `correlate` use `method='auto'`, which calls `choose_conv_method` to choose the fastest method using pre-computed values. CuPy may not choose the same method to compute the convolution as SciPy does given the same inputs.

---

## cupyx.scipy.signal.correlate

`cupyx.scipy.signal.correlate(in1, in2, mode='full', method='auto')`

Cross-correlate two N-dimensional arrays.

Cross-correlate `in1` and `in2`, with the output size determined by the `mode` argument.

**Parameters**

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **method** (*str*) – Indicates which method to use for the computations:
  - 'direct': The convolution is determined directly from sums, the definition of convolution

- 'fft': The Fourier Transform is used to perform the convolution by calling `fftconvolve`.
- 'auto': Automatically choose direct or FFT based on an estimate of which is faster for the arguments (default).

**Returns**

the result of correlation.

**Return type**

*cupy.ndarray*

**See also:**

*cupyx.scipy.signal.choose\_conv\_method()*

**See also:**

*cupyx.scipy.signal.convolve()*

**See also:**

*cupyx.scipy.signal.fftconvolve()*

**See also:**

*cupyx.scipy.signal.oaconvolve()*

**See also:**

`cupyx.scipy.ndimage.correlation()`

**See also:**

`scipy.signal.correlation()`

---

**Note:** By default, `convolve` and `correlate` use `method='auto'`, which calls `choose_conv_method` to choose the fastest method using pre-computed values. CuPy may not choose the same method to compute the convolution as SciPy does given the same inputs.

---

## cupyx.scipy.signal.fftconvolve

`cupyx.scipy.signal.fftconvolve(in1, in2, mode='full', axes=None)`

Convolve two N-dimensional arrays using FFT.

Convolve `in1` and `in2` using the fast Fourier transform method, with the output size determined by the `mode` argument.

This is generally much faster than the 'direct' method of `convolve` for large arrays, but can be slower when only a few output values are needed, and can only output float arrays (int or object array inputs will be cast to float).

**Parameters**

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear cross-correlation (default)

- 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
- 'same': output is the same size as `in1`, centered with respect to the 'full' output
- **axes** (*scalar or tuple of scalar or None*) – Axes over which to compute the convolution. The default is over all axes.

**Returns**

the result of convolution

**Return type**

*cupy.ndarray*

**See also:**

*cupyx.scipy.signal.choose\_conv\_method()*

**See also:**

*cupyx.scipy.signal.correlation()*

**See also:**

*cupyx.scipy.signal.convolve()*

**See also:**

*cupyx.scipy.signal.oaconvolve()*

**See also:**

*cupyx.scipy.ndimage.convolve()*

**See also:**

*scipy.signal.correlation()*

## cupyx.scipy.signal.oaconvolve

`cupyx.scipy.signal.oaconvolve(in1, in2, mode='full', axes=None)`

Convolve two N-dimensional arrays using the overlap-add method.

Convolve `in1` and `in2` using the overlap-add method, with the output size determined by the `mode` argument. This is generally faster than `convolve` for large arrays, and generally faster than `fftconvolve` when one array is much larger than the other, but can be slower when only a few output values are needed or when the arrays are very similar in shape, and can only output float arrays (int or object array inputs will be cast to float).

**Parameters**

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear cross-correlation (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': output is the same size as `in1`, centered with respect to the 'full' output
- **axes** (*scalar or tuple of scalar or None*) – Axes over which to compute the convolution. The default is over all axes.

**Returns**

the result of convolution

**Return type**

*cupy.ndarray*

**See also:**

*cupyx.scipy.signal.convolve()*

**See also:**

*cupyx.scipy.signal.fftconvolve()*

**See also:**

*cupyx.scipy.ndimage.convolve()*

**See also:**

*scipy.signal.oaconvolve()*

**cupyx.scipy.signal.convolve2d**

`cupyx.scipy.signal.convolve2d(in1, in2, mode='full', boundary='fill', fillvalue=0)`

Convolve two 2-dimensional arrays.

Convolve `in1` and `in2` with output size determined by `mode`, and boundary conditions determined by `boundary` and `fillvalue`.

**Parameters**

- **in1** (*cupy.ndarray*) – First input.
- **in2** (*cupy.ndarray*) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (*str*) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **boundary** (*str*) – Indicates how to handle boundaries:
  - fill: pad input arrays with `fillvalue` (default)
  - wrap: circular boundary conditions
  - symm: symmetrical boundary conditions
- **fillvalue** (*scalar*) – Value to fill pad input arrays with. Default is 0.

**Returns**

A 2-dimensional array containing a subset of the discrete linear convolution of `in1` with `in2`.

**Return type**

*cupy.ndarray*

**See also:**

*cupyx.scipy.signal.convolve()*

See also:

`cupyx.scipy.signal.fftconvolve()`

See also:

`cupyx.scipy.signal.oaconvolve()`

See also:

`cupyx.scipy.signal.correlate2d()`

See also:

`cupyx.scipy.ndimage.convolve()`

See also:

`scipy.signal.convolve2d()`

### **cupyx.scipy.signal.correlate2d**

`cupyx.scipy.signal.correlate2d(in1, in2, mode='full', boundary='fill', fillvalue=0)`

Cross-correlate two 2-dimensional arrays.

Cross correlate `in1` and `in2` with output size determined by `mode`, and boundary conditions determined by `boundary` and `fillvalue`.

#### **Parameters**

- **in1** (`cupy.ndarray`) – First input.
- **in2** (`cupy.ndarray`) – Second input. Should have the same number of dimensions as `in1`.
- **mode** (`str`) – Indicates the size of the output:
  - 'full': output is the full discrete linear convolution (default)
  - 'valid': output consists only of those elements that do not rely on the zero-padding. Either `in1` or `in2` must be at least as large as the other in every dimension.
  - 'same': - output is the same size as `in1`, centered with respect to the 'full' output
- **boundary** (`str`) – Indicates how to handle boundaries:
  - fill: pad input arrays with `fillvalue` (default)
  - wrap: circular boundary conditions
  - symm: symmetrical boundary conditions
- **fillvalue** (`scalar`) – Value to fill pad input arrays with. Default is 0.

#### **Returns**

A 2-dimensional array containing a subset of the discrete linear cross-correlation of `in1` with `in2`.

#### **Return type**

`cupy.ndarray`

---

**Note:** When using "same" mode with even-length inputs, the outputs of `correlate` and `correlate2d` differ: There is a 1-index offset between them.

---



See also:

`cupyx.scipy.signal.correlate()`

See also:

`cupyx.scipy.signal.convolve2d()`

See also:

`cupyx.scipy.ndimage.correlate()`

See also:

`scipy.signal.correlate2d()`

### `cupyx.scipy.signal.sepfir2d`

`cupyx.scipy.signal.sepfir2d(input, hrow, hcol)`

Convolve with a 2-D separable FIR filter.

Convolve the rank-2 input array with the separable filter defined by the rank-1 arrays `hrow`, and `hcol`. Mirror symmetric boundary conditions are assumed. This function can be used to find an image given its B-spline representation.

The arguments `hrow` and `hcol` must be 1-dimensional and of odd length.

#### Parameters

- **input** (`cupy.ndarray`) – The input signal
- **hrow** (`cupy.ndarray`) – Row direction filter
- **hcol** (`cupy.ndarray`) – Column direction filter

#### Returns

The filtered signal

#### Return type

`cupy.ndarray`

See also:

`scipy.signal.sepfir2d()`

### `cupyx.scipy.signal.choose_conv_method`

`cupyx.scipy.signal.choose_conv_method(in1, in2, mode='full')`

Find the fastest convolution/correlation method.

#### Parameters

- **in1** (`cupy.ndarray`) – first input.
- **in2** (`cupy.ndarray`) – second input.
- **mode** (`str`, optional) – 'valid', 'same', 'full'.

#### Returns

A string indicating which convolution method is fastest, either 'direct' or 'fft'.

#### Return type

`str`

**Warning:** This function currently doesn't support measure option, nor multidimensional inputs. It does not guarantee the compatibility of the return value to SciPy's one.

See also:

`scipy.signal.choose_conv_method()`

## cupyx.scipy.signal.correlation\_lags

`cupyx.scipy.signal.correlation_lags(in1_len, in2_len, mode='full')`

Calculates the lag / displacement indices array for 1D cross-correlation.

### Parameters

- **in1\_len** (*int*) – First input size.
- **in2\_len** (*int*) – Second input size.
- **mode** (*str* {'full', 'valid', 'same'}, *optional*) – A string indicating the size of the output. See the documentation *correlate* for more information.

### Returns

**lags** – Returns an array containing cross-correlation lag/displacement indices. Indices can be indexed with the `np.argmax` of the correlation to return the lag/displacement.

### Return type

array

See also:

### *correlate*

Compute the N-dimensional cross-correlation.

`scipy.signal.correlation_lags`

## B-Splines

<code>gauss_spline(x, n)</code>	Gaussian approximation to B-spline basis function of order <i>n</i> .
<code>cspline1d(signal[, lamb])</code>	Compute cubic spline coefficients for rank-1 array.
<code>qspline1d(signal[, lamb])</code>	Compute quadratic spline coefficients for rank-1 array.
<code>cspline2d(signal[, lamb, precision])</code>	Coefficients for 2-D cubic (3rd order) B-spline.
<code>qspline2d(signal[, lamb, precision])</code>	Coefficients for 2-D quadratic (2nd order) B-spline.
<code>cspline1d_eval(cj, newx[, dx, x0])</code>	Evaluate a cubic spline at the new set of points.
<code>qspline1d_eval(cj, newx[, dx, x0])</code>	Evaluate a quadratic spline at the new set of points.
<code>spline_filter(lin[, lmbda])</code>	Smoothing spline (cubic) filtering of a rank-2 array.

**cupyx.scipy.signal.gauss\_spline**`cupyx.scipy.signal.gauss_spline(x, n)`Gaussian approximation to B-spline basis function of order  $n$ .**Parameters**

- **x** (*array\_like*) – a knot vector
- **n** (*int*) – The order of the spline. Must be nonnegative, i.e.  $n \geq 0$

**Returns****res** – B-spline basis function values approximated by a zero-mean Gaussian function.**Return type***ndarray***Notes**

The B-spline basis function can be approximated well by a zero-mean Gaussian function with standard-deviation equal to  $\sigma = (n + 1)/12$  for large  $n$  :

$$\frac{1}{\sqrt{2\pi}\sigma^2} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

See<sup>1,2</sup> for more information.**References****cupyx.scipy.signal.cspline1d**`cupyx.scipy.signal.cspline1d(signal, lamb=0.0)`

Compute cubic spline coefficients for rank-1 array.

Find the cubic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window [1.0, 4.0, 1.0]/6.0 .

**Parameters**

- **signal** (*ndarray*) – A rank-1 array representing samples of a signal.
- **lamb** (*float, optional*) – Smoothing coefficient, default is 0.0.

**Returns****c** – Cubic spline coefficients.**Return type***ndarray*

See also:

***cspline1d\_eval***

Evaluate a cubic spline at the new set of points.

<sup>1</sup> Bouma H., Vilanova A., Bescos J.O., ter Haar Romeny B.M., Gerritsen F.A. (2007) Fast and Accurate Gaussian Derivatives Based on B-Splines. In: Sgallari F., Murli A., Paragios N. (eds) Scale Space and Variational Methods in Computer Vision. SSVM 2007. Lecture Notes in Computer Science, vol 4485. Springer, Berlin, Heidelberg

<sup>2</sup> <http://folk.uio.no/inf3330/scripting/doc/python/SciPy/tutorial/old/node24.html>

## cupyx.scipy.signal.qspline1d

`cupyx.scipy.signal.qspline1d(signal, lamb=0.0)`

Compute quadratic spline coefficients for rank-1 array.

### Parameters

- **signal** (`ndarray`) – A rank-1 array representing samples of a signal.
- **lamb** (`float`, *optional*) – Smoothing coefficient (must be zero for now).

### Returns

**c** – Quadratic spline coefficients.

### Return type

*ndarray*

See also:

### [\*qspline1d\\_eval\*](#)

Evaluate a quadratic spline at the new set of points.

### Notes

Find the quadratic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window `[1.0, 6.0, 1.0]/ 8.0`.

## cupyx.scipy.signal.cspline2d

`cupyx.scipy.signal.cspline2d(signal, lamb=0.0, precision=-1.0)`

Coefficients for 2-D cubic (3rd order) B-spline.

Return the third-order B-spline coefficients over a regularly spaced input grid for the two-dimensional input image.

### Parameters

- **input** (`ndarray`) – The input signal.
- **lamb** (`float`) – Specifies the amount of smoothing in the transfer function.
- **precision** (`float`) – Specifies the precision for computing the infinite sum needed to apply mirror-symmetric boundary conditions.

### Returns

**output** – The filtered signal.

### Return type

*ndarray*

## cupyx.scipy.signal.qspline2d

`cupyx.scipy.signal.qspline2d(signal, lamb=0.0, precision=-1.0)`

Coefficients for 2-D quadratic (2nd order) B-spline.

Return the second-order B-spline coefficients over a regularly spaced input grid for the two-dimensional input image.

### Parameters

- **input** (`ndarray`) – The input signal.
- **lamb** (`float`) – Specifies the amount of smoothing in the transfer function.
- **precision** (`float`) – Specifies the precision for computing the infinite sum needed to apply mirror-symmetric boundary conditions.

### Returns

**output** – The filtered signal.

### Return type

`ndarray`

## cupyx.scipy.signal.cspline1d\_eval

`cupyx.scipy.signal.cspline1d_eval(cj, newx, dx=1.0, x0=0)`

Evaluate a cubic spline at the new set of points.

$dx$  is the old sample-spacing while  $x0$  was the old origin. In other-words the old-sample points (knot-points) for which the  $cj$  represent spline coefficients were at equally-spaced points of:

$oldx = x0 + j * dx \quad j=0 \dots N-1$ , with  $N=\text{len}(cj)$

Edges are handled using mirror-symmetric boundary conditions.

### Parameters

- **cj** (`ndarray`) – cubic spline coefficients
- **newx** (`ndarray`) – New set of points.
- **dx** (`float`, *optional*) – Old sample-spacing, the default value is 1.0.
- **x0** (`int`, *optional*) – Old origin, the default value is 0.

### Returns

**res** – Evaluated a cubic spline points.

### Return type

`ndarray`

See also:

### `cspline1d`

Compute cubic spline coefficients for rank-1 array.

### cupyx.scipy.signal.qspline1d\_eval

cupyx.scipy.signal.qspline1d\_eval(*cj*, *newx*, *dx*=1.0, *x0*=0)

Evaluate a quadratic spline at the new set of points.

#### Parameters

- **cj** (*ndarray*) – Quadratic spline coefficients
- **newx** (*ndarray*) – New set of points.
- **dx** (*float*, *optional*) – Old sample-spacing, the default value is 1.0.
- **x0** (*int*, *optional*) – Old origin, the default value is 0.

#### Returns

**res** – Evaluated a quadratic spline points.

#### Return type

*ndarray*

See also:

#### qspline1d

Compute quadratic spline coefficients for rank-1 array.

#### Notes

*dx* is the old sample-spacing while *x0* was the old origin. In other-words the old-sample points (knot-points) for which the *cj* represent spline coefficients were at equally-spaced points of:

$$\text{oldx} = \text{x0} + \text{j} * \text{dx} \quad \text{j} = 0 \dots \text{N}-1, \text{ with } \text{N} = \text{len}(\text{cj})$$

Edges are handled using mirror-symmetric boundary conditions.

### cupyx.scipy.signal.spline\_filter

cupyx.scipy.signal.spline\_filter(*Iin*, *lmbda*=5.0)

Smoothing spline (cubic) filtering of a rank-2 array.

Filter an input data set, *Iin*, using a (cubic) smoothing spline of fall-off *lmbda*.

#### Parameters

- **Iin** (*array\_like*) – input data set
- **lmbda** (*float*, *optional*) – spline smooching fall-off value, default is 5.0.

#### Returns

**res** – filtered input data

#### Return type

*ndarray*

## Filtering

<code>order_filter(a, domain, rank)</code>	Perform an order filter on an N-D array.
<code>medfilt(volume[, kernel_size])</code>	Perform a median filter on an N-dimensional array.
<code>medfilt2d(input[, kernel_size])</code>	Median filter a 2-dimensional array.
<code>wiener(im[, mysize, noise])</code>	Perform a Wiener filter on an N-dimensional array.
<code>symiirorder1(input, c0, z1[, precision])</code>	Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of first-order sections. The second section uses a reversed sequence. This implements a system with the following transfer function and mirror-symmetric boundary conditions::
<code>symiirorder2(input, r, omega[, precision])</code>	Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of second-order sections. The second section uses a reversed sequence. This implements the following transfer function::
<code>lfiltfilt(b, a, x[, axis, zi])</code>	Filter data along one-dimension with an IIR or FIR filter.
<code>lfiltic(b, a, y[, x])</code>	Construct initial conditions for lfilter given input and output vectors.
<code>lfiltfilt_zi(b, a)</code>	Construct initial conditions for lfilter for step response steady-state.
<code>filtfilt(b, a, x[, axis, padtype, padlen, ...])</code>	Apply a digital filter forward and backward to a signal.
<code>savgol_filter(x, window_length, polyorder[, ...])</code>	Apply a Savitzky-Golay filter to an array.
<code>deconvolve(signal, divisor)</code>	Deconvolves divisor out of signal using inverse filtering.
<code>sosfilt(sos, x[, axis, zi])</code>	Filter data along one dimension using cascaded second-order sections.
<code>sosfilt_zi(sos)</code>	Construct initial conditions for sosfilt for step response steady-state.
<code>sosfiltfilt(sos, x[, axis, padtype, padlen])</code>	A forward-backward digital filter using cascaded second-order sections.
<code>hilbert(x[, N, axis])</code>	Compute the analytic signal, using the Hilbert transform.
<code>hilbert2(x[, N])</code>	Compute the '2-D' analytic signal of $x$
<code>decimate(x, q[, n, ftype, axis, zero_phase])</code>	Downsample the signal after applying an anti-aliasing filter.
<code>detrend(data[, axis, type, bp, overwrite_data])</code>	Remove linear trend along axis from data.
<code>resample(x, num[, t, axis, window, domain])</code>	Resample $x$ to $num$ samples using Fourier method along the given axis.
<code>resample_poly(x, up, down[, axis, window, ...])</code>	Resample $x$ along the given axis using polyphase filtering.
<code>upfirdn(h, x[, up, down, axis, mode, cval])</code>	Upsample, FIR filter, and downsample.

### cupyx.scipy.signal.order\_filter

`cupyx.scipy.signal.order_filter(a, domain, rank)`

Perform an order filter on an N-D array.

Perform an order filter on the array in. The domain argument acts as a mask centered over each pixel. The non-zero elements of domain are used to select elements surrounding each input pixel which are placed in a list. The list is sorted, and the output for that pixel is the element corresponding to rank in the sorted list.

#### Parameters

- **a** (`cupy.ndarray`) – The N-dimensional input array.

- **domain** (`cupy.ndarray`) – A mask array with the same number of dimensions as *a*. Each dimension should have an odd number of elements.
- **rank** (`int`) – A non-negative integer which selects the element from the sorted list (0 corresponds to the smallest element).

**Returns**

The results of the order filter in an array with the same shape as *a*.

**Return type**

`cupy.ndarray`

**See also:**

`cupyx.scipy.ndimage.rank_filter()`

**See also:**

`scipy.signal.order_filter()`

### `cupyx.scipy.signal.medfilt`

`cupyx.scipy.signal.medfilt(volume, kernel_size=None)`

Perform a median filter on an N-dimensional array.

Apply a median filter to the input array using a local window-size given by *kernel\_size*. The array will automatically be zero-padded.

**Parameters**

- **volume** (`cupy.ndarray`) – An N-dimensional input array.
- **kernel\_size** (`int` or `list of ints`) – Gives the size of the median filter window in each dimension. Elements of *kernel\_size* should be odd. If *kernel\_size* is a scalar, then this scalar is used as the size in each dimension. Default size is 3 for each dimension.

**Returns**

An array the same size as input containing the median filtered result.

**Return type**

`cupy.ndarray`

**See also:**

`cupyx.scipy.ndimage.median_filter()`

**See also:**

`scipy.signal.medfilt()`

### `cupyx.scipy.signal.medfilt2d`

`cupyx.scipy.signal.medfilt2d(input, kernel_size=3)`

Median filter a 2-dimensional array.

Apply a median filter to the *input* array using a local window-size given by *kernel\_size* (must be odd). The array is zero-padded automatically.

**Parameters**

- **input** (`cupy.ndarray`) – A 2-dimensional input array.



- **kernel\_size** (*int of list of ints of length 2*) – Gives the size of the median filter window in each dimension. Elements of *kernel\_size* should be odd. If *kernel\_size* is a scalar, then this scalar is used as the size in each dimension. Default is a kernel of size (3, 3).

**Returns**

An array the same size as input containing the median filtered result.

**Return type**

*cupy.ndarray*

**See also:**

, ,

**cupyx.scipy.signal.wiener**

`cupyx.scipy.signal.wiener(im, mysize=None, noise=None)`

Perform a Wiener filter on an N-dimensional array.

Apply a Wiener filter to the N-dimensional array *im*.

**Parameters**

- **im** (*cupy.ndarray*) – An N-dimensional array.
- **mysize** (*int or cupy.ndarray, optional*) – A scalar or an N-length list giving the size of the Wiener filter window in each dimension. Elements of *mysize* should be odd. If *mysize* is a scalar, then this scalar is used as the size in each dimension.
- **noise** (*float, optional*) – The noise-power to use. If None, then noise is estimated as the average of the local variance of the input.

**Returns**

Wiener filtered result with the same shape as *im*.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.signal.wiener()`

**cupyx.scipy.signal.symiirorder1**

`cupyx.scipy.signal.symiirorder1(input, c0, z1, precision=-1.0)`

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of first-order sections. The second section uses a reversed sequence. This implements a system with the following transfer function and mirror-symmetric boundary conditions:

$$H(z) = \frac{c0}{(1-z1/z)(1-z1z)}$$

The resulting signal will have mirror symmetric boundary conditions as well.

**Parameters**

- **input** (*ndarray*) – The input signal.

- **c0** (*scalar*) – Parameters in the transfer function.
- **z1** (*scalar*) – Parameters in the transfer function.
- **precision** – Specifies the precision for calculating initial conditions of the recursive filter based on mirror-symmetric input.

**Returns**

**output** – The filtered signal.

**Return type**

*ndarray*

### cupyx.scipy.signal.symiirorder2

cupyx.scipy.signal.**symiirorder2**(*input*, *r*, *omega*, *precision=-1.0*)

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of second-order sections. The second section uses a reversed sequence. This implements the following transfer function:

$$H(z) = \frac{cs^2}{(1 - a2/z - a3/z^2)(1 - a2z - a3z^2)}$$

where:

```
a2 = 2 * r * cos(omega)
a3 = - r ** 2
cs = 1 - 2 * r * cos(omega) + r ** 2
```

**Parameters**

- **input** (*ndarray*) – The input signal.
- **r** (*float*) – Parameters in the transfer function.
- **omega** (*float*) – Parameters in the transfer function.
- **precision** (*float*) – Specifies the precision for calculating initial conditions of the recursive filter based on mirror-symmetric input.

**Returns**

**output** – The filtered signal.

**Return type**

*ndarray*

### cupyx.scipy.signal.lfilter

cupyx.scipy.signal.**lfilter**(*b*, *a*, *x*, *axis=-1*, *zi=None*)

Filter data along one-dimension with an IIR or FIR filter.

Filter a data sequence, *x*, using a digital filter. This works for many fundamental data types (including Object type). The filter is a direct form II transposed implementation of the standard difference equation (see Notes).

The function *sosfilt* (and filter design using *output='sos'*) should be preferred over *lfilter* for most filtering tasks, as second-order sections have fewer numerical problems.

**Parameters**

- **b** (*array\_like*) – The numerator coefficient vector in a 1-D sequence.
- **a** (*array\_like*) – The denominator coefficient vector in a 1-D sequence. If `a[0]` is not 1, then both *a* and *b* are normalized by `a[0]`.
- **x** (*array\_like*) – An N-dimensional input array.
- **axis** (*int*, *optional*) – The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.
- **zi** (*array\_like*, *optional*) – Initial conditions for the filter delays. It is a vector (or array of vectors for an N-dimensional input) of length `len(b) + len(a) - 2`. The first `len(b)` numbers correspond to the last elements of the previous input and the last `len(a)` to the last elements of the previous output. If *zi* is None or is not given then initial rest is assumed. See *lfiltic* for more information.

**Note:** This argument differs from dimensions from the SciPy implementation! However, as long as they are chained from the same library, the output result will be the same. Please make sure to use the *zi* from CuPy calls and not from SciPy. This due to the parallel nature of this implementation as opposed to the serial one in SciPy.

#### Returns

- **y** (*array*) – The output of the digital filter.
- **zf** (*array*, *optional*) – If *zi* is None, this is not returned, otherwise, *zf* holds the final filter delay values.

#### See also:

##### *lfiltic*

Construct initial conditions for *lfilter*.

##### *lfilter\_zi*

Compute initial state (steady state of step response) for *lfilter*.

##### *filtfilt*

A forward-backward filter, to obtain a filter with zero phase.

##### *savgol\_filter*

A Savitzky-Golay filter.

##### *sosfilt*

Filter data using cascaded second-order sections.

##### *sosfiltfilt*

A forward-backward filter using second-order sections.

#### Notes

The filter function is implemented as a direct II transposed structure. This means that the filter implements:

$$\begin{aligned} a[0]*y[n] = & b[0]*x[n] + b[1]*x[n-1] + \dots + b[M]*x[n-M] \\ & - a[1]*y[n-1] - \dots - a[N]*y[n-N] \end{aligned}$$

where *M* is the degree of the numerator, *N* is the degree of the denominator, *n* is the sample number and *L* denotes the length of the input. It is implemented by computing first the FIR part and then computing the IIR part from it:

```

a[0] * y = r(f(x, b), a)
f(x, b)[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M]
r(y, a)[n] = - a[1]*y[n-1] - ... - a[N]*y[n-N]

```

The IIR result is computed in parallel by first dividing the input signal into chunks ( $g_i$ ) of size  $m$ . For each chunk, the IIR recurrence equation is applied to each chunk (in parallel). Then the chunks are merged based on the last  $N$  values of the last chunk:

```

nc = L/m
x = [g_0, g_1, ..., g_nc]

g_i = [x[i * m], ..., x[i * m + m - 1]]
p_i = r(g_i, a)

o_i = r(p_i, c(p_{i-1}))    if i > 1,
                           otherwise
y = [o_0, o_1, ..., o_nc]

```

where  $c$  denotes a function that takes a chunk, slices the last  $N$  values and adjust them using a correction factor table computed using the  $(1, 2, \dots, N)$ -fibonacci sequence. For more information see<sup>1</sup>.

The rational transfer function describing this filter in the  $z$ -transform domain is:

$$Y(z) = \frac{b[0] + b[1]z^{-1} + \dots + b[M]z^{-M}}{a[0] + a[1]z^{-1} + \dots + a[N]z^{-N}} X(z)$$

## References

### cupyx.scipy.signal.lfiltic

`cupyx.scipy.signal.lfiltic(b, a, y, x=None)`

Construct initial conditions for `lfilt` given input and output vectors.

Given a linear filter  $(b, a)$  and initial conditions on the output  $y$  and the input  $x$ , return the initial conditions on the state vector  $z_i$  which is used by `lfilt` to generate the output given the input.

#### Parameters

- **b** (*array\_like*) – Linear filter term.
- **a** (*array\_like*) – Linear filter term.
- **y** (*array\_like*) – Initial conditions. If  $N = \text{len}(a) - 1$ , then  $y = \{y[-1], y[-2], \dots, y[-N]\}$ . If  $y$  is too short, it is padded with zeros.
- **x** (*array\_like, optional*) – Initial conditions. If  $M = \text{len}(b) - 1$ , then  $x = \{x[-1], x[-2], \dots, x[-M]\}$ . If  $x$  is not given, its initial conditions are assumed zero. If  $x$  is too short, it is padded with zeros.

<sup>1</sup> Sepideh Maleki and Martin Burtcher. 2018. Automatic Hierarchical Parallelization of Linear Recurrences. SIGPLAN Not. 53, 2 (February 2018), 128-138. 10.1145/3173162.3173168

- **axis** (*int*, *optional*) – The axis to take the initial conditions from, if *x* and *y* are *n*-dimensional

**Returns**

**zi** – The state vector  $z_i = \{z_0[-1], z_1[-1], \dots, z_{K-1}[-1]\}$ , where  $K = M + N$ .

**Return type**

*ndarray*

**See also:**

*lfilter*, *lfilter\_zi*

**cupyx.scipy.signal.lfilter\_zi**

`cupyx.scipy.signal.lfilter_zi(b, a)`

Construct initial conditions for *lfilter* for step response steady-state.

Compute an initial state *zi* for the *lfilter* function that corresponds to the steady state of the step response.

A typical use of this function is to set the initial state so that the output of the filter starts at the same value as the first element of the signal to be filtered.

**Parameters**

- **b** (*array\_like* (1-D)) – The IIR filter coefficients. See *lfilter* for more information.
- **a** (*array\_like* (1-D)) – The IIR filter coefficients. See *lfilter* for more information.

**Returns**

**zi** – The initial state for the filter.

**Return type**

1-D *ndarray*

**See also:**

*lfilter*, *lfiltic*, *filtfilt*

**cupyx.scipy.signal.filtfilt**

`cupyx.scipy.signal.filtfilt(b, a, x, axis=-1, padtype='odd', padlen=None, method='pad', irlen=None)`

Apply a digital filter forward and backward to a signal.

This function applies a linear digital filter twice, once forward and once backwards. The combined filter has zero phase and a filter order twice that of the original.

The function provides options for handling the edges of the signal.

The function *sosfiltfilt* (and filter design using `output='sos'`) should be preferred over *filtfilt* for most filtering tasks, as second-order sections have fewer numerical problems.

**Parameters**

- **b** ((*N*,) *array\_like*) – The numerator coefficient vector of the filter.
- **a** ((*N*,) *array\_like*) – The denominator coefficient vector of the filter. If *a*[0] is not 1, then both *a* and *b* are normalized by *a*[0].
- **x** (*array\_like*) – The array of data to be filtered.
- **axis** (*int*, *optional*) – The axis of *x* to which the filter is applied. Default is -1.

- **padtype** (*str* or *None*, *optional*) – Must be ‘odd’, ‘even’, ‘constant’, or *None*. This determines the type of extension to use for the padded signal to which the filter is applied. If *padtype* is *None*, no padding is used. The default is ‘odd’.
- **padlen** (*int* or *None*, *optional*) – The number of elements by which to extend *x* at both ends of *axis* before applying the filter. This value must be less than `x.shape[axis] - 1`. `padlen=0` implies no padding. The default value is `3 * max(len(a), len(b))`.
- **method** (*str*, *optional*) – Determines the method for handling the edges of the signal, either “pad” or “gust”. When *method* is “pad”, the signal is padded; the type of padding is determined by *padtype* and *padlen*, and *irlen* is ignored. When *method* is “gust”, Gustafsson’s method is used, and *padtype* and *padlen* are ignored.
- **irlen** (*int* or *None*, *optional*) – When *method* is “gust”, *irlen* specifies the length of the impulse response of the filter. If *irlen* is *None*, no part of the impulse response is ignored. For a long signal, specifying *irlen* can significantly improve the performance of the filter.

**Returns**

*y* – The filtered output with the same shape as *x*.

**Return type**

*ndarray*

See also:

*sosfiltfilt*, *lfilter\_zi*, *lfilter*, *lfiltic*, *savgol\_filter*, *sosfilt*

**Notes**

When *method* is “pad”, the function pads the data along the given axis in one of three ways: odd, even or constant. The odd and even extensions have the corresponding symmetry about the end point of the data. The constant extension extends the data with the values at the end points. On both the forward and backward passes, the initial condition of the filter is found by using *lfilter\_zi* and scaling it by the end point of the extended data.

When *method* is “gust”, Gustafsson’s method<sup>1</sup> is used. Initial conditions are chosen for the forward and backward passes so that the forward-backward filter gives the same result as the backward-forward filter.

**References****cupyx.scipy.signal.savgol\_filter**

`cupyx.scipy.signal.savgol_filter(x, window_length, polyorder, deriv=0, delta=1.0, axis=-1, mode='interp', cval=0.0)`

Apply a Savitzky-Golay filter to an array.

This is a 1-D filter. If *x* has dimension greater than 1, *axis* determines the axis along which the filter is applied.

**Parameters**

- **x** (*array\_like*) – The data to be filtered. If *x* is not a single or double precision floating point array, it will be converted to type `numpy.float64` before filtering.
- **window\_length** (*int*) – The length of the filter window (i.e., the number of coefficients). If *mode* is ‘interp’, *window\_length* must be less than or equal to the size of *x*.
- **polyorder** (*int*) – The order of the polynomial used to fit the samples. *polyorder* must be less than *window\_length*.

---

<sup>1</sup> F. Gustafsson, “Determining the initial states in forward-backward filtering”, Transactions on Signal Processing, Vol. 46, pp. 988-992, 1996.

- **deriv** (*int*, *optional*) – The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.
- **delta** (*float*, *optional*) – The spacing of the samples to which the filter will be applied. This is only used if `deriv > 0`. Default is 1.0.
- **axis** (*int*, *optional*) – The axis of the array *x* along which the filter is to be applied. Default is -1.
- **mode** (*str*, *optional*) – Must be ‘mirror’, ‘constant’, ‘nearest’, ‘wrap’ or ‘interp’. This determines the type of extension to use for the padded signal to which the filter is applied. When *mode* is ‘constant’, the padding value is given by *cval*. See the Notes for more details on ‘mirror’, ‘constant’, ‘wrap’, and ‘nearest’. When the ‘interp’ mode is selected (the default), no extension is used. Instead, a degree *polyorder* polynomial is fit to the last *window\_length* values of the edges, and this polynomial is used to evaluate the last *window\_length*//2 output values.
- **cval** (*scalar*, *optional*) – Value to fill past the edges of the input if *mode* is ‘constant’. Default is 0.0.

**Returns**

*y* – The filtered data.

**Return type**

ndarray, same shape as *x*

**See also:**

[\*savgol\\_coeffs\*](#)

**Notes**

Details on the *mode* options:

**‘mirror’:**

Repeats the values at the edges in reverse order. The value closest to the edge is not included.

**‘nearest’:**

The extension contains the nearest input value.

**‘constant’:**

The extension contains the value given by the *cval* argument.

**‘wrap’:**

The extension contains the values from the other end of the array.

For example, if the input is [1, 2, 3, 4, 5, 6, 7, 8], and *window\_length* is 7, the following shows the extended data for the various *mode* options (assuming *cval* is 0):

mode	Ext			Input								Ext		
'mirror'	4	3	2	1	2	3	4	5	6	7	8	7	6	5
'nearest'	1	1	1	1	2	3	4	5	6	7	8	8	8	8
'constant'	0	0	0	1	2	3	4	5	6	7	8	0	0	0
'wrap'	6	7	8	1	2	3	4	5	6	7	8	1	2	3

New in version 0.14.0.

## Examples

```
>>> import numpy as np
>>> from scipy.signal import savgol_filter
>>> np.set_printoptions(precision=2) # For compact display.
>>> x = np.array([2, 2, 5, 2, 1, 0, 1, 4, 9])
```

Filter with a window length of 5 and a degree 2 polynomial. Use the defaults for all other parameters.

```
>>> savgol_filter(x, 5, 2)
array([1.66, 3.17, 3.54, 2.86, 0.66, 0.17, 1. , 4. , 9. ])
```

Note that the last five values in `x` are samples of a parabola, so when `mode='interp'` (the default) is used with `polyorder=2`, the last three values are unchanged. Compare that to, for example, `mode='nearest'`:

```
>>> savgol_filter(x, 5, 2, mode='nearest')
array([1.74, 3.03, 3.54, 2.86, 0.66, 0.17, 1. , 4.6 , 7.97])
```

## cupyx.scipy.signal.deconvolve

`cupyx.scipy.signal.deconvolve(signal, divisor)`

Deconvolves divisor out of signal using inverse filtering.

Returns the quotient and remainder such that `signal = convolve(divisor, quotient) + remainder`

### Parameters

- **signal** ((*N*,) *array\_like*) – Signal data, typically a recorded signal
- **divisor** ((*N*,) *array\_like*) – Divisor data, typically an impulse response or filter that was applied to the original signal

### Returns

- **quotient** (*ndarray*) – Quotient, typically the recovered original signal
- **remainder** (*ndarray*) – Remainder

See also:

### cupy.polydiv

performs polynomial division (same operation, but also accepts `poly1d` objects)

## Examples

Deconvolve a signal that's been filtered:

```
>>> from cupyx.scipy import signal
>>> original = [0, 1, 0, 0, 1, 1, 0, 0]
>>> impulse_response = [2, 1]
>>> recorded = signal.convolve(impulse_response, original)
>>> recorded
array([0, 2, 1, 0, 2, 3, 1, 0, 0])
>>> recovered, remainder = signal.deconvolve(recorded, impulse_response)
>>> recovered
array([ 0.,  1.,  0.,  0.,  1.,  1.,  0.,  0.])
```



## cupyx.scipy.signal.sosfilt

`cupyx.scipy.signal.sosfilt(sos, x, axis=-1, zi=None)`

Filter data along one dimension using cascaded second-order sections.

Filter a data sequence, *x*, using a digital IIR filter defined by *sos*.

### Parameters

- **sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (*n\_sections*, 6). Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.
- **x** (*array\_like*) – An N-dimensional input array.
- **axis** (*int*, *optional*) – The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.
- **zi** (*array\_like*, *optional*) – Initial conditions for the cascaded filter delays. It is a (at least 2D) vector of shape (*n\_sections*, ..., 4, ...), where ..., 4, ... denotes the shape of *x*, but with *x.shape[axis]* replaced by 4. If *zi* is None or is not given then initial rest (i.e. all zeros) is assumed. Note that these initial conditions are *not* the same as the initial conditions given by *lfiltic* or *lfilt\_zi*.

### Returns

- **y** (*ndarray*) – The output of the digital filter.
- **zf** (*ndarray*, *optional*) – If *zi* is None, this is not returned, otherwise, *zf* holds the final filter delay values.

See also:

[\*zpk2sos\*](#), [\*sos2zpk\*](#), [\*sosfilt\\_zi\*](#), [\*sosfiltfilt\*](#), [\*sosfreqz\*](#)

## cupyx.scipy.signal.sosfilt\_zi

`cupyx.scipy.signal.sosfilt_zi(sos)`

Construct initial conditions for *sosfilt* for step response steady-state.

Compute an initial state *zi* for the *sosfilt* function that corresponds to the steady state of the step response.

A typical use of this function is to set the initial state so that the output of the filter starts at the same value as the first element of the signal to be filtered.

### Parameters

- **sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (*n\_sections*, 6). See *sosfilt* for the SOS filter format specification.

### Returns

- **zi** – Initial conditions suitable for use with *sosfilt*, shape (*n\_sections*, 4).

### Return type

*ndarray*

See also:

[\*sosfilt\*](#), [\*zpk2sos\*](#)

## cupyx.scipy.signal.sosfiltfilt

`cupyx.scipy.signal.sosfiltfilt(sos, x, axis=-1, padtype='odd', padlen=None)`

A forward-backward digital filter using cascaded second-order sections.

See *filtfilt* for more complete information about this method.

### Parameters

- **sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (n\_sections, 6). Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.
- **x** (*array\_like*) – The array of data to be filtered.
- **axis** (*int*, *optional*) – The axis of *x* to which the filter is applied. Default is -1.
- **padtype** (*str* or *None*, *optional*) – Must be 'odd', 'even', 'constant', or None. This determines the type of extension to use for the padded signal to which the filter is applied. If *padtype* is None, no padding is used. The default is 'odd'.
- **padlen** (*int* or *None*, *optional*) – The number of elements by which to extend *x* at both ends of *axis* before applying the filter. This value must be less than `x.shape[axis] - 1`. `padlen=0` implies no padding. The default value is:

```
3 * (2 * len(sos) + 1 - min((sos[:, 2] == 0).sum(),
                             (sos[:, 5] == 0).sum()))
```

The extra subtraction at the end attempts to compensate for poles and zeros at the origin (e.g. for odd-order filters) to yield equivalent estimates of *padlen* to those of *filtfilt* for second-order section filters built with *scipy.signal* functions.

### Returns

**y** – The filtered output with the same shape as *x*.

### Return type

*ndarray*

See also:

*filtfilt*, *sosfilt*, *sosfilt\_zi*, *sosfreqz*

## cupyx.scipy.signal.hilbert

`cupyx.scipy.signal.hilbert(x, N=None, axis=-1)`

Compute the analytic signal, using the Hilbert transform.

The transformation is done along the last axis by default.

### Parameters

- **x** (*ndarray*) – Signal data. Must be real.
- **N** (*int*, *optional*) – Number of Fourier components. Default: `x.shape[axis]`
- **axis** (*int*, *optional*) – Axis along which to do the transformation. Default: -1.

### Returns

**xa** – Analytic signal of *x*, of each 1-D array along *axis*

**Return type***ndarray***Notes**

The analytic signal  $x_a(t)$  of signal  $x(t)$  is:

$$x_a = F^{-1}(F(x)2U) = x + iy$$

where  $F$  is the Fourier transform,  $U$  the unit step function, and  $y$  the Hilbert transform of  $x$ .<sup>1</sup>

In other words, the negative half of the frequency spectrum is zeroed out, turning the real-valued signal into a complex signal. The Hilbert transformed signal can be obtained from `np.imag(hilbert(x))`, and the original signal from `np.real(hilbert(x))`.

**References****See also:**`scipy.signal.hilbert`**cupyx.scipy.signal.hilbert2**

`cupyx.scipy.signal.hilbert2(x, N=None)`

Compute the ‘2-D’ analytic signal of  $x$

**Parameters**

- **$x$  (ndarray)** – 2-D signal data.
- **$N$  (int or tuple of two ints, optional)** – Number of Fourier components. Default is `x.shape`

**Returns**

**$xa$**  – Analytic signal of  $x$  taken along axes (0,1).

**Return type***ndarray***See also:**`scipy.signal.hilbert2`**cupyx.scipy.signal.decimate**

`cupyx.scipy.signal.decimate(x, q, n=None, ftype='iir', axis=-1, zero_phase=True)`

Downsample the signal after applying an anti-aliasing filter.

By default, an order 8 Chebyshev type I filter is used. A 30 point FIR filter with Hamming window is used if `ftype` is ‘fir’.

**Parameters**

- **$x$  (array\_like)** – The signal to be downsampled, as an N-dimensional array.

<sup>1</sup> Wikipedia, “Analytic signal”. [https://en.wikipedia.org/wiki/Analytic\\_signal](https://en.wikipedia.org/wiki/Analytic_signal)

- **q** (*int*) – The downsampling factor. When using IIR downsampling, it is recommended to call *decimate* multiple times for downsampling factors higher than 13.
- **n** (*int*, *optional*) – The order of the filter (1 less than the length for ‘fir’). Defaults to 8 for ‘iir’ and 20 times the downsampling factor for ‘fir’.
- **ftype** (str {‘iir’, ‘fir’} or *dlti* instance, *optional*) – If ‘iir’ or ‘fir’, specifies the type of lowpass filter. If an instance of an *dlti* object, uses that object to filter before downsampling.
- **axis** (*int*, *optional*) – The axis along which to decimate.
- **zero\_phase** (*bool*, *optional*) – Prevent phase shift by filtering with *filtfilt* instead of *lfilter* when using an IIR filter, and shifting the outputs back by the filter’s group delay when using an FIR filter. The default value of True is recommended, since a phase shift is generally not desired.

**Returns**

**y** – The down-sampled signal.

**Return type**

*ndarray*

See also:

*resample*

Resample up or down using the FFT method.

*resample\_poly*

Resample using polyphase filtering and an FIR filter.

## cupyx.scipy.signal.detrend

`cupyx.scipy.signal.detrend(data, axis=-1, type='linear', bp=0, overwrite_data=False)`

Remove linear trend along axis from data.

**Parameters**

- **data** (*array\_like*) – The input data.
- **axis** (*int*, *optional*) – The axis along which to detrend the data. By default this is the last axis (-1).
- **type** ({‘linear’, ‘constant’}, *optional*) – The type of detrending. If **type** == ‘linear’ (default), the result of a linear least-squares fit to *data* is subtracted from *data*. If **type** == ‘constant’, only the mean of *data* is subtracted.
- **bp** (*array\_like of ints*, *optional*) – A sequence of break points. If given, an individual linear fit is performed for each part of *data* between two break points. Break points are specified as indices into *data*. This parameter only has an effect when **type** == ‘linear’.
- **overwrite\_data** (*bool*, *optional*) – If True, perform in place detrending and avoid a copy. Default is False

**Returns**

**ret** – The detrended input data.

**Return type**

*ndarray*

See also:

`scipy.signal.detrend`

## cupyx.scipy.signal.resample

`cupyx.scipy.signal.resample(x, num, t=None, axis=0, window=None, domain='time')`

Resample  $x$  to  $num$  samples using Fourier method along the given axis.

The resampled signal starts at the same value as  $x$  but is sampled with a spacing of  $\text{len}(x) / num * (\text{spacing of } x)$ . Because a Fourier method is used, the signal is assumed to be periodic.

### Parameters

- **x** (*array\_like*) – The data to be resampled.
- **num** (*int*) – The number of samples in the resampled signal.
- **t** (*array\_like, optional*) – If  $t$  is given, it is assumed to be the sample positions associated with the signal data in  $x$ .
- **axis** (*int, optional*) – The axis of  $x$  that is resampled. Default is 0.
- **window** (*array\_like, callable, string, float, or tuple, optional*) – Specifies the window applied to the signal in the Fourier domain. See below for details.
- **domain** (*string, optional*) – A string indicating the domain of the input  $x$ :

#### **time**

Consider the input  $x$  as time-domain. (Default)

#### **freq**

Consider the input  $x$  as frequency-domain.

### Returns

Either the resampled array, or, if  $t$  was given, a tuple containing the resampled array and the corresponding resampled positions.

### Return type

resampled\_x or (resampled\_x, resampled\_t)

See also:

#### *decimate*

Downsample the signal after applying an FIR or IIR filter.

#### *resample\_poly*

Resample using polyphase filtering and an FIR filter.

### Notes

The argument *window* controls a Fourier-domain window that tapers the Fourier spectrum before zero-padding to alleviate ringing in the resampled values for sampled signals you didn't intend to be interpreted as band-limited.

If *window* is a function, then it is called with a vector of inputs indicating the frequency bins (i.e. `fft-freq(x.shape[axis])`).

If *window* is an array of the same length as  $x.\text{shape}[\text{axis}]$  it is assumed to be the window to be applied directly in the Fourier domain (with dc and low-frequency first).

For any other type of *window*, the function `cusignal.get_window` is called to generate the window.

The first sample of the returned vector is the same as the first sample of the input vector. The spacing between samples is changed from  $dx$  to  $dx * \text{len}(x) / num$ .

If  $t$  is not None, then it represents the old sample positions, and the new sample positions will be returned as well as the new samples.

As noted, *resample* uses FFT transformations, which can be very slow if the number of input or output samples is large and prime; see *scipy.fftpack.fft*.

## Examples

Note that the end of the resampled data rises to meet the first sample of the next cycle:

```
>>> import cupy as cp
>>> import cupyx.scipy.signal import resample
```

```
>>> x = cupy.linspace(0, 10, 20, endpoint=False)
>>> y = cupy.cos(-x**2/6.0)
>>> f = resample(y, 100)
>>> xnew = cupy.linspace(0, 10, 100, endpoint=False)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(cupy.asnumpy(x), cupy.asnumpy(y), 'go-', cupy.asnumpy(xnew),
↳      cupy.asnumpy(f), '.-', 10, cupy.asnumpy(y[0]), 'ro')
>>> plt.legend(['data', 'resampled'], loc='best')
>>> plt.show()
```

## cupyx.scipy.signal.resample\_poly

`cupyx.scipy.signal.resample_poly(x, up, down, axis=0, window=('kaiser', 5.0), padtype='constant', cval=None)`

Resample  $x$  along the given axis using polyphase filtering.

The signal  $x$  is upsampled by the factor  $up$ , a zero-phase low-pass FIR filter is applied, and then it is downsampled by the factor  $down$ . The resulting sample rate is  $up / down$  times the original sample rate. Values beyond the boundary of the signal are assumed to be zero during the filtering step.

### Parameters

- **x** (*array\_like*) – The data to be resampled.
- **up** (*int*) – The upsampling factor.
- **down** (*int*) – The downsampling factor.
- **axis** (*int*, *optional*) – The axis of  $x$  that is resampled. Default is 0.
- **window** (*string*, *tuple*, or *array\_like*, *optional*) – Desired window to use to design the low-pass filter, or the FIR filter coefficients to employ. See below for details.
- **padtype** (*string*, *optional*) – *constant*, *line*, *mean*, *median*, *maximum*, *minimum* or any of the other signal extension modes supported by *cupyx.scipy.signal.upfirdn*. Changes assumptions on values beyond the boundary. If *constant*, assumed to be *cval* (default zero). If *line* assumed to continue a linear trend defined by the first and last points. *mean*, *median*, *maximum* and *minimum* work as in *cupy.pad* and assume that the values beyond the boundary are the mean, median, maximum or minimum respectively of the array along the axis.
- **cval** (*float*, *optional*) – Value to use if *padtype*='constant'. Default is zero.

**Returns****resampled\_x** – The resampled array.**Return type**

array

**See also:*****decimate***

Downsample the signal after applying an FIR or IIR filter.

***resample***

Resample up or down using the FFT method.

**Notes**

This polyphase method will likely be faster than the Fourier method in *cusignal.resample* when the number of samples is large and prime, or when the number of samples is large and *up* and *down* share a large greatest common denominator. The length of the FIR filter used will depend on  $\max(\text{up}, \text{down}) // \text{gcd}(\text{up}, \text{down})$ , and the number of operations during polyphase filtering will depend on the filter length and *down* (see *cusignal.upfirdn* for details).

The argument *window* specifies the FIR low-pass filter design.

If *window* is an array\_like it is assumed to be the FIR filter coefficients. Note that the FIR filter is applied after the upsampling step, so it should be designed to operate on a signal at a sampling frequency higher than the original by a factor of  $\text{up} // \text{gcd}(\text{up}, \text{down})$ . This function's output will be centered with respect to this array, so it is best to pass a symmetric filter with an odd number of samples if, as is usually the case, a zero-phase filter is desired.

For any other type of *window*, the functions *cusignal.get\_window* and *cusignal.firwin* are called to generate the appropriate filter coefficients.

The first sample of the returned vector is the same as the first sample of the input vector. The spacing between samples is changed from *dx* to  $\text{dx} * \text{down} / \text{float}(\text{up})$ .

**Examples**

Note that the end of the resampled data rises to meet the first sample of the next cycle for the FFT method, and gets closer to zero for the polyphase method:

```
>>> import cupy
>>> import cupyx.scipy.signal import resample, resample_poly
```

```
>>> x = cupy.linspace(0, 10, 20, endpoint=False)
>>> y = cupy.cos(-x**2/6.0)
>>> f_fft = resample(y, 100)
>>> f_poly = resample_poly(y, 100, 20)
>>> xnew = cupy.linspace(0, 10, 100, endpoint=False)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(cupy.asnumpy(xnew), cupy.asnumpy(f_fft), 'b.-',
            ↪ asnumpy(xnew), cupy.asnumpy(f_poly), 'r.-')
>>> plt.plot(cupy.asnumpy(x), cupy.asnumpy(y), 'ko-')
>>> plt.plot(10, cupy.asnumpy(y[0]), 'bo', 10, 0., 'ro') # boundaries
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend(['resample', 'resamp_poly', 'data'], loc='best')
>>> plt.show()
```

## cupyx.scipy.signal.upfirdn

`cupyx.scipy.signal.upfirdn(h, x, up=1, down=1, axis=-1, mode=None, cval=0)`

Upsample, FIR filter, and downsample.

### Parameters

- **h** (*array\_like*) – 1-dimensional FIR (finite-impulse response) filter coefficients.
- **x** (*array\_like*) – Input signal array.
- **up** (*int*, *optional*) – Upsampling rate. Default is 1.
- **down** (*int*, *optional*) – Downsampling rate. Default is 1.
- **axis** (*int*, *optional*) – The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.
- **mode** (*str*, *optional*) – This parameter is not implemented.
- **cval** (*float*, *optional*) – This parameter is not implemented.

### Returns

**y** – The output signal array. Dimensions will be the same as *x* except for along *axis*, which will change size according to the *h*, *up*, and *down* parameters.

### Return type

*ndarray*

## Notes

The algorithm is an implementation of the block diagram shown on page 129 of the Vaidyanathan text<sup>1</sup> (Figure 4.3-8d).

The direct approach of upsampling by factor of P with zero insertion, FIR filtering of length N, and downsampling by factor of Q is  $O(N*Q)$  per output sample. The polyphase implementation used here is  $O(N/P)$ .

### See also:

`scipy.signal.upfirdn`

## References

---

<sup>1</sup> P. P. Vaidyanathan, Multirate Systems and Filter Banks, Prentice Hall, 1993.



## Filter design

<code>bilinear(b, a[, fs])</code>	Return a digital IIR filter from an analog one using a bilinear transform.
<code>bilinear_zpk(z, p, k, fs)</code>	Return a digital IIR filter from an analog one using a bilinear transform.
<code>findfreqs(num, den, N[, kind])</code>	Find array of frequencies for computing the response of an analog filter.
<code>freqs(b, a[, worN, plot])</code>	Compute frequency response of analog filter.
<code>freqs_zpk(z, p, k[, worN])</code>	Compute frequency response of analog filter.
<code>freqz(b[, a, worN, whole, plot, fs, ...])</code>	Compute the frequency response of a digital filter.
<code>freqz_zpk(z, p, k[, worN, whole, fs])</code>	Compute the frequency response of a digital filter in ZPK form.
<code>sosfreqz(sos[, worN, whole, fs])</code>	Compute the frequency response of a digital filter in SOS format.
<code>firwin(numtaps, cutoff[, width, window, ...])</code>	FIR filter design using the window method.
<code>firwin2(numtaps, freq, gain[, nfreqs, ...])</code>	FIR filter design using the window method.
<code>firls(numtaps, bands, desired[, weight, fs])</code>	FIR filter design using least-squares error minimization.
<code>minimum_phase(h[, method, n_fft])</code>	Convert a linear-phase FIR filter to minimum phase
<code>savgol_coeffs(window_length, polyorder[, ...])</code>	Compute the coefficients for a 1-D Savitzky-Golay FIR filter.
<code>gammatone(freq, ftype[, order, numtaps, fs])</code>	Gammatone filter design.
<code>group_delay(system[, w, whole, fs])</code>	Compute the group delay of a digital filter.
<code>iirdesign(wp, ws, gpass, gstop[, analog, ...])</code>	Complete IIR digital and analog filter design.
<code>iirfilter(N, Wn[, rp, rs, btype, analog, ...])</code>	IIR digital and analog filter design given order and critical points.
<code>kaiser_atten(numtaps, width)</code>	Compute the attenuation of a Kaiser FIR filter.
<code>kaiser_beta(a)</code>	Compute the Kaiser parameter <i>beta</i> , given the attenuation <i>a</i> .
<code>kaiserord(ripple, width)</code>	Determine the filter window parameters for the Kaiser window method.
<code>unique_roots(p[, tol, rtype])</code>	Determine unique roots and their multiplicities from a list of roots.
<code>residue(b, a[, tol, rtype])</code>	Compute partial-fraction expansion of $b(s) / a(s)$ .
<code>residuez(b, a[, tol, rtype])</code>	Compute partial-fraction expansion of $b(z) / a(z)$ .
<code>invres(r, p, k[, tol, rtype])</code>	Compute $b(s)$ and $a(s)$ from partial fraction expansion.
<code>invresz(r, p, k[, tol, rtype])</code>	Compute $b(z)$ and $a(z)$ from partial fraction expansion.
<code>BadCoefficients</code>	Warning about badly conditioned filter coefficients

## cupyx.scipy.signal.bilinear

`cupyx.scipy.signal.bilinear(b, a, fs=1.0)`

Return a digital IIR filter from an analog one using a bilinear transform.

Transform a set of poles and zeros from the analog s-plane to the digital z-plane using Tustin's method, which substitutes  $2*fs*(z-1) / (z+1)$  for  $s$ , maintaining the shape of the frequency response.

### Parameters

- **b** (*array\_like*) – Numerator of the analog filter transfer function.
- **a** (*array\_like*) – Denominator of the analog filter transfer function.

- **fs** (*float*) – Sample rate, as ordinary frequency (e.g., hertz). No prewarping is done in this function.

#### Returns

- **b** (*ndarray*) – Numerator of the transformed digital filter transfer function.
- **a** (*ndarray*) – Denominator of the transformed digital filter transfer function.

See also:

[\*lp2lp\*](#), [\*lp2hp\*](#), [\*lp2bp\*](#), [\*lp2bs\*](#), [\*bilinear\\_zpk\*](#), [`scipy.signal.bilinear`](#)

### **cupyx.scipy.signal.bilinear\_zpk**

`cupyx.scipy.signal.bilinear_zpk(z, p, k, fs)`

Return a digital IIR filter from an analog one using a bilinear transform.

Transform a set of poles and zeros from the analog s-plane to the digital z-plane using Tustin's method, which substitutes  $2*fs*(z-1) / (z+1)$  for s, maintaining the shape of the frequency response.

#### Parameters

- **z** (*array\_like*) – Zeros of the analog filter transfer function.
- **p** (*array\_like*) – Poles of the analog filter transfer function.
- **k** (*float*) – System gain of the analog filter transfer function.
- **fs** (*float*) – Sample rate, as ordinary frequency (e.g., hertz). No prewarping is done in this function.

#### Returns

- **z** (*ndarray*) – Zeros of the transformed digital filter transfer function.
- **p** (*ndarray*) – Poles of the transformed digital filter transfer function.
- **k** (*float*) – System gain of the transformed digital filter.

See also:

[\*lp2lp\\_zpk\*](#), [\*lp2hp\\_zpk\*](#), [\*lp2bp\\_zpk\*](#), [\*lp2bs\\_zpk\*](#), [\*bilinear\*](#), [`scipy.signal.bilinear\_zpk`](#)

### **cupyx.scipy.signal.findfreqs**

`cupyx.scipy.signal.findfreqs(num, den, N, kind='ba')`

Find array of frequencies for computing the response of an analog filter.

#### Parameters

- **num** (*array\_like*, 1-D) – The polynomial coefficients of the numerator and denominator of the transfer function of the filter or LTI system, where the coefficients are ordered from highest to lowest degree. Or, the roots of the transfer function numerator and denominator (i.e., zeroes and poles).
- **den** (*array\_like*, 1-D) – The polynomial coefficients of the numerator and denominator of the transfer function of the filter or LTI system, where the coefficients are ordered from highest to lowest degree. Or, the roots of the transfer function numerator and denominator (i.e., zeroes and poles).
- **N** (*int*) – The length of the array to be computed.

- **kind** (*str* {'ba', 'zp'}, *optional*) – Specifies whether the numerator and denominator are specified by their polynomial coefficients ('ba'), or their roots ('zp').

**Returns**

**w** – A 1-D array of frequencies, logarithmically spaced.

**Return type**

(N,) *ndarray*

**Warning:** This function may synchronize the device.

**See also:**

`scipy.signal.find_freqs`

**Examples**

Find a set of nine frequencies that span the “interesting part” of the frequency response for the filter with the transfer function

$$H(s) = s / (s^2 + 8s + 25)$$

```
>>> from scipy import signal
>>> signal.findfreqs([1, 0], [1, 8, 25], N=9)
array([ 1.00000000e-02,  3.16227766e-02,  1.00000000e-01,
        3.16227766e-01,  1.00000000e+00,  3.16227766e+00,
        1.00000000e+01,  3.16227766e+01,  1.00000000e+02])
```

**cupyx.scipy.signal.freqs**

`cupyx.scipy.signal.freqs(b, a, worN=200, plot=None)`

Compute frequency response of analog filter.

Given the M-order numerator *b* and N-order denominator *a* of an analog filter, compute its frequency response:

$$H(w) = \frac{b[0]*(jw)**M + b[1]*(jw)**(M-1) + \dots + b[M]}{a[0]*(jw)**N + a[1]*(jw)**(N-1) + \dots + a[N]}$$

**Parameters**

- **b** (*array\_like*) – Numerator of a linear filter.
- **a** (*array\_like*) – Denominator of a linear filter.
- **worN** (*{None, int, array\_like}*, *optional*) – If *None*, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, then compute at that many frequencies. Otherwise, compute the response at the angular frequencies (e.g., rad/s) given in *worN*.
- **plot** (*callable*, *optional*) – A callable that takes two arguments. If given, the return parameters *w* and *h* are passed to plot. Useful for plotting the frequency response inside *freqs*.

**Returns**

- **w** (*ndarray*) – The angular frequencies at which *h* was computed.
- **h** (*ndarray*) – The frequency response.

See also:

`scipy.signal.freqs`

`freqz`

Compute the frequency response of a digital filter.

### `cupyx.scipy.signal.freqs_zpk`

`cupyx.scipy.signal.freqs_zpk(z, p, k, worN=200)`

Compute frequency response of analog filter.

Given the zeros *z*, poles *p*, and gain *k* of a filter, compute its frequency response:

$$H(w) = k * \frac{(jw - z[0]) * (jw - z[1]) * \dots * (jw - z[-1])}{(jw - p[0]) * (jw - p[1]) * \dots * (jw - p[-1])}$$

#### Parameters

- **z** (*array\_like*) – Zeroes of a linear filter
- **p** (*array\_like*) – Poles of a linear filter
- **k** (*scalar*) – Gain of a linear filter
- **worN** (*{None, int, array\_like}*, *optional*) – If *None*, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, then compute at that many frequencies. Otherwise, compute the response at the angular frequencies (e.g., rad/s) given in *worN*.

#### Returns

- **w** (*ndarray*) – The angular frequencies at which *h* was computed.
- **h** (*ndarray*) – The frequency response.

See also:

`scipy.signal.freqs_zpk`

### `cupyx.scipy.signal.freqz`

`cupyx.scipy.signal.freqz(b, a=1, worN=512, whole=False, plot=None, fs=6.283185307179586, include_nyquist=False)`

Compute the frequency response of a digital filter.

Given the M-order numerator *b* and N-order denominator *a* of a digital filter, compute its frequency response:

$$H(e^{jw}) = \frac{B(e^{jw})}{A(e^{jw})} = \frac{b[0] + b[1]e^{-jw} + \dots + b[M]e^{-jwM}}{a[0] + a[1]e^{-jw} + \dots + a[N]e^{-jwN}}$$

### Parameters

- **b** (*array\_like*) – Numerator of a linear filter. If *b* has dimension greater than 1, it is assumed that the coefficients are stored in the first dimension, and *b.shape[1:]*, *a.shape[1:]*, and the shape of the frequencies array must be compatible for broadcasting.
- **a** (*array\_like*) – Denominator of a linear filter. If *b* has dimension greater than 1, it is assumed that the coefficients are stored in the first dimension, and *b.shape[1:]*, *a.shape[1:]*, and the shape of the frequencies array must be compatible for broadcasting.
- **worN** (*{None, int, array\_like}*, *optional*) – If a single integer, then compute at that many frequencies (default is *N=512*). This is a convenient alternative to:

```
cupy.linspace(0, fs if whole else fs/2, N,
              endpoint=include_nyquist)
```

Using a number that is fast for FFT computations can result in faster computations (see Notes).

If an *array\_like*, compute the response at the frequencies given. These are in the same units as *fs*.

- **whole** (*bool*, *optional*) – Normally, frequencies are computed from 0 to the Nyquist frequency, *fs/2* (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to *fs*. Ignored if *worN* is *array\_like*.
- **plot** (*callable*) – A callable that takes two arguments. If given, the return parameters *w* and *h* are passed to *plot*. Useful for plotting the frequency response inside *freqz*.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system. Defaults to  $2\pi$  radians/sample (so *w* is from 0 to  $\pi$ ).
- **include\_nyquist** (*bool*, *optional*) – If *whole* is False and *worN* is an integer, setting *include\_nyquist* to True will include the last frequency (Nyquist frequency) and is otherwise ignored.

### Returns

- **w** (*ndarray*) – The frequencies at which *h* was computed, in the same units as *fs*. By default, *w* is normalized to the range  $[0, \pi)$  (radians/sample).
- **h** (*ndarray*) – The frequency response, as complex numbers.

See also:

[\*freqz\\_zpk\*](#), [\*sosfreqz\*](#), [\*scipy.signal.freqz\*](#)

### Notes

Using Matplotlib's `matplotlib.pyplot.plot()` function as the callable for *plot* produces unexpected results, as this plots the real part of the complex transfer function, not the magnitude. Try `lambda w, h: plot(w, cupy.abs(h))`.

A direct computation via (R)FFT is used to compute the frequency response when the following conditions are met:

1. An integer value is given for *worN*.
2. *worN* is fast to compute via FFT (i.e., `next_fast_len(worN) < scipy.fft.next_fast_len` equals *worN*).
3. The denominator coefficients are a single value (`a.shape[0] == 1`).

4. `worN` is at least as long as the numerator coefficients (`worN >= b.shape[0]`).

5. If `b.ndim > 1`, then `b.shape[-1] == 1`.

For long FIR filters, the FFT approach can have lower error and be much faster than the equivalent direct polynomial calculation.

## cupyx.scipy.signal.freqz\_zpk

`cupyx.scipy.signal.freqz_zpk(z, p, k, worN=512, whole=False, fs=6.283185307179586)`

Compute the frequency response of a digital filter in ZPK form.

Given the Zeros, Poles and Gain of a digital filter, compute its frequency response:

$$H(z) = k \prod_i (z - Z[i]) / \prod_j (z - P[j])$$

where  $k$  is the *gain*,  $Z$  are the *zeros* and  $P$  are the *poles*.

### Parameters

- **z** (*array\_like*) – Zeroes of a linear filter
- **p** (*array\_like*) – Poles of a linear filter
- **k** (*scalar*) – Gain of a linear filter
- **worN** (*{None, int, array\_like}, optional*) – If a single integer, then compute at that many frequencies (default is  $N=512$ ).

If an *array\_like*, compute the response at the frequencies given. These are in the same units as *fs*.

- **whole** (*bool, optional*) – Normally, frequencies are computed from 0 to the Nyquist frequency,  $fs/2$  (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to  $fs$ . Ignored if *w* is *array\_like*.
- **fs** (*float, optional*) – The sampling frequency of the digital system. Defaults to  $2\pi$  radians/sample (so *w* is from 0 to  $\pi$ ).

### Returns

- **w** (*ndarray*) – The frequencies at which *h* was computed, in the same units as *fs*. By default, *w* is normalized to the range  $[0, \pi)$  (radians/sample).
- **h** (*ndarray*) – The frequency response, as complex numbers.

See also:

### *freqs*

Compute the frequency response of an analog filter in TF form

### *freqs\_zpk*

Compute the frequency response of an analog filter in ZPK form

### *freqz*

Compute the frequency response of a digital filter in TF form

`scipy.signal.freqz_zpk`

**cupyx.scipy.signal.sosfreqz**

`cupyx.scipy.signal.sosfreqz(sos, worN=512, whole=False, fs=6.283185307179586)`

Compute the frequency response of a digital filter in SOS format.

Given *sos*, an array with shape (n, 6) of second order sections of a digital filter, compute the frequency response of the system function:

$$H(z) = \frac{B_0(z)}{A_0(z)} * \frac{B_1(z)}{A_1(z)} * \dots * \frac{B_{n-1}(z)}{A_{n-1}(z)}$$

for  $z = \exp(j\omega)$ , where  $B\{k\}(z)$  and  $A\{k\}(z)$  are numerator and denominator of the transfer function of the k-th second order section.

**Parameters**

- **sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (n\_sections, 6). Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.
- **worN** (*{None, int, array\_like}*, *optional*) – If a single integer, then compute at that many frequencies (default is N=512). Using a number that is fast for FFT computations can result in faster computations (see Notes of *freqz*).

If an *array\_like*, compute the response at the frequencies given (must be 1-D). These are in the same units as *fs*.

- **whole** (*bool*, *optional*) – Normally, frequencies are computed from 0 to the Nyquist frequency,  $fs/2$  (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to *fs*.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system. Defaults to  $2\pi$  radians/sample (so  $\omega$  is from 0 to  $\pi$ ).

New in version 1.2.0.

**Returns**

- **w** (*ndarray*) – The frequencies at which *h* was computed, in the same units as *fs*. By default, *w* is normalized to the range [0,  $\pi$ ) (radians/sample).
- **h** (*ndarray*) – The frequency response, as complex numbers.

See also:

[\*freqz\*](#), [\*sosfilt\*](#), [`scipy.signal.sosfreqz`](#)

**cupyx.scipy.signal.firwin**

`cupyx.scipy.signal.firwin(numtaps, cutoff, width=None, window='hamming', pass_zero=True, scale=True, fs=2)`

FIR filter design using the window method.

This function computes the coefficients of a finite impulse response filter. The filter will have linear phase; it will be Type I if *numtaps* is odd and Type II if *numtaps* is even.

Type II filters always have zero response at the Nyquist frequency, so a `ValueError` exception is raised if *firwin* is called with *numtaps* even and having a passband whose right end is at the Nyquist frequency.

### Parameters

- **numtaps** (*int*) – Length of the filter (number of coefficients, i.e. the filter order + 1). *numtaps* must be odd if a passband includes the Nyquist frequency.
- **cutoff** (*float or 1D array\_like*) – Cutoff frequency of filter (expressed in the same units as *fs*) OR an array of cutoff frequencies (that is, band edges). In the latter case, the frequencies in *cutoff* should be positive and monotonically increasing between 0 and *fs*/2. The values 0 and *fs*/2 must not be included in *cutoff*.
- **width** (*float or None, optional*) – If *width* is not *None*, then assume it is the approximate width of the transition region (expressed in the same units as *fs*) for use in Kaiser FIR filter design. In this case, the *window* argument is ignored.
- **window** (*string or tuple of string and parameter values, optional*) – Desired window to use. See *cusignal.get\_window* for a list of windows and required parameters.
- **pass\_zero** (*{True, False, 'bandpass', 'lowpass', 'highpass', 'bandstop'},*) – optional If *True*, the gain at the frequency 0 (i.e. the “DC gain”) is 1. If *False*, the DC gain is 0. Can also be a string argument for the desired filter type (equivalent to *btype* in IIR design functions).
- **scale** (*bool, optional*) – Set to *True* to scale the coefficients so that the frequency response is exactly unity at a certain frequency. That frequency is either:
  - 0 (DC) if the first passband starts at 0 (i.e. *pass\_zero* is *True*)
  - *fs*/2 (the Nyquist frequency) if the first passband ends at *fs*/2 (i.e the filter is a single band highpass filter); center of first passband otherwise
- **fs** (*float, optional*) – The sampling frequency of the signal. Each frequency in *cutoff* must be between 0 and *fs*/2. Default is 2.

### Returns

**h** – Coefficients of length *numtaps* FIR filter.

### Return type

(*numtaps*,) *ndarray*

### Raises

**ValueError** – If any value in *cutoff* is less than or equal to 0 or greater than or equal to *fs*/2, if the values in *cutoff* are not strictly monotonically increasing, or if *numtaps* is even but a passband includes the Nyquist frequency.

### See also:

*firwin2*, *firls*, *minimum\_phase*, *remez*

### Examples

Low-pass from 0 to *f*:

```
>>> import cusignal
>>> numtaps = 3
>>> f = 0.1
>>> cusignal.firwin(numtaps, f)
array([ 0.06799017,  0.86401967,  0.06799017])
```

Use a specific window function:



```
>>> cusignal.firwin(numtaps, f, window='nutall')
array([ 3.56607041e-04,  9.99286786e-01,  3.56607041e-04])
```

High-pass ('stop' from 0 to f):

```
>>> cusignal.firwin(numtaps, f, pass_zero=False)
array([-0.00859313,  0.98281375, -0.00859313])
```

Band-pass:

```
>>> f1, f2 = 0.1, 0.2
>>> cusignal.firwin(numtaps, [f1, f2], pass_zero=False)
array([ 0.06301614,  0.88770441,  0.06301614])
```

Band-stop:

```
>>> cusignal.firwin(numtaps, [f1, f2])
array([-0.00801395,  1.0160279 , -0.00801395])
```

Multi-band (passbands are [0, f1], [f2, f3] and [f4, 1]):

```
>>> f3, f4 = 0.3, 0.4
>>> cusignal.firwin(numtaps, [f1, f2, f3, f4])
array([-0.01376344,  1.02752689, -0.01376344])
```

Multi-band (passbands are [f1, f2] and [f3, f4]):

```
>>> cusignal.firwin(numtaps, [f1, f2, f3, f4], pass_zero=False)
array([ 0.04890915,  0.91284326,  0.04890915])
```

## cupyx.scipy.signal.firwin2

`cupyx.scipy.signal.firwin2(numtaps, freq, gain, nfreqs=None, window='hamming', nyq=None, antisymmetric=False, fs=2.0)`

FIR filter design using the window method.

From the given frequencies *freq* and corresponding gains *gain*, this function constructs an FIR filter with linear phase and (approximately) the given frequency response.

### Parameters

- **numtaps** (*int*) – The number of taps in the FIR filter. *numtaps* must be less than *nfreqs*.
- **freq** (*array\_like*, 1-D) – The frequency sampling points. Typically 0.0 to 1.0 with 1.0 being Nyquist. The Nyquist frequency is half *fs*. The values in *freq* must be nondecreasing. A value can be repeated once to implement a discontinuity. The first value in *freq* must be 0, and the last value must be *fs*/2. Values 0 and *fs*/2 must not be repeated.
- **gain** (*array\_like*) – The filter gains at the frequency sampling points. Certain constraints to gain values, depending on the filter type, are applied, see Notes for details.
- **nfreqs** (*int*, *optional*) – The size of the interpolation mesh used to construct the filter. For most efficient behavior, this should be a power of 2 plus 1 (e.g, 129, 257, etc). The default is one more than the smallest power of 2 that is not less than *numtaps*. *nfreqs* must be greater than *numtaps*.

- **window** (*string or (string, float) or float, or None, optional*) – Window function to use. Default is “hamming”. See `scipy.signal.get_window` for the complete list of possible values. If `None`, no window function is applied.
- **antisymmetric** (*bool, optional*) – Whether resulting impulse response is symmetric/antisymmetric. See Notes for more details.
- **fs** (*float, optional*) – The sampling frequency of the signal. Each frequency in *cutoff* must be between 0 and  $fs/2$ . Default is 2.

**Returns**

**taps** – The filter coefficients of the FIR filter, as a 1-D array of length *numtaps*.

**Return type**

*ndarray*

**See also:**

`scipy.signal.firwin2`, `firls`, `firwin`, `minimum_phase`, `remez`

**Notes**

From the given set of frequencies and gains, the desired response is constructed in the frequency domain. The inverse FFT is applied to the desired response to create the associated convolution kernel, and the first *numtaps* coefficients of this kernel, scaled by *window*, are returned. The FIR filter will have linear phase. The type of filter is determined by the value of ‘numtaps’ and *antisymmetric* flag. There are four possible combinations:

- odd *numtaps*, *antisymmetric* is False, type I filter is produced
- even *numtaps*, *antisymmetric* is False, type II filter is produced
- odd *numtaps*, *antisymmetric* is True, type III filter is produced
- even *numtaps*, *antisymmetric* is True, type IV filter is produced

Magnitude response of all but type I filters are subjects to following constraints:

- type II – zero at the Nyquist frequency
- type III – zero at zero and Nyquist frequencies
- type IV – zero at zero frequency

**cupyx.scipy.signal.firls**

`cupyx.scipy.signal.firls(numtaps, bands, desired, weight=None, fs=2)`

FIR filter design using least-squares error minimization.

Calculate the filter coefficients for the linear-phase finite impulse response (FIR) filter which has the best approximation to the desired frequency response described by *bands* and *desired* in the least squares sense (i.e., the integral of the weighted mean-squared error within the specified bands is minimized).

**Parameters**

- **numtaps** (*int*) – The number of taps in the FIR filter. *numtaps* must be odd.
- **bands** (*array\_like*) – A monotonic nondecreasing sequence containing the band edges in Hz. All elements must be non-negative and less than or equal to the Nyquist frequency given by  $fs/2$ . The bands are specified as frequency pairs, thus, if using a 1D array, its length must be even, e.g., `cupy.array([0, 1, 2, 3, 4, 5])`. Alternatively, the bands can be specified as an  $n \times 2$  sized 2D array, where  $n$  is the number of bands, e.g., `cupy.array([[0, 1], [2, 3], [4, 5]])`.

All elements of *bands* must be monotonically nondecreasing, have width > 0, and must not overlap. (This is not checked by the routine).

- **desired** (*array\_like*) – A sequence the same size as *bands* containing the desired gain at the start and end point of each band. All elements must be non-negative (this is not checked by the routine).
- **weight** (*array\_like*, *optional*) – A relative weighting to give to each band region when solving the least squares problem. *weight* has to be half the size of *bands*. All elements must be non-negative (this is not checked by the routine).
- **fs** (*float*, *optional*) – The sampling frequency of the signal. Each frequency in *bands* must be between 0 and  $fs/2$  (inclusive). Default is 2.

#### Returns

**coeffs** – Coefficients of the optimal (in a least squares sense) FIR filter.

#### Return type

*ndarray*

#### See also:

*firwin*, *firwin2*, *minimum\_phase*, *remez*, *scipy.signal.firls*

### cupyx.scipy.signal.minimum\_phase

`cupyx.scipy.signal.minimum_phase(h, method='homomorphic', n_fft=None)`

Convert a linear-phase FIR filter to minimum phase

#### Parameters

- **h** (*array*) – Linear-phase FIR filter coefficients.
- **method** (*{'hilbert', 'homomorphic'}*) – The method to use:
  - 'homomorphic' (default)**  
This method<sup>45</sup> works best with filters with an odd number of taps, and the resulting minimum phase filter will have a magnitude response that approximates the square root of the original filter's magnitude response.
  - 'hilbert'**  
This method<sup>1</sup> is designed to be used with equiripple filters (e.g., from *remez*) with unity or zero gain regions.
- **n\_fft** (*int*) – The number of points to use for the FFT. Should be at least a few times larger than the signal length (see Notes).

#### Returns

**h\_minimum** – The minimum-phase version of the filter, with length  $(\text{length}(h) + 1) // 2$ .

#### Return type

*array*

#### See also:

*scipy.signal.minimum\_phase*

<sup>4</sup> J. S. Lim, Advanced Topics in Signal Processing. Englewood Cliffs, N.J.: Prentice Hall, 1988.

<sup>5</sup> A. V. Oppenheim, R. W. Schaffer, and J. R. Buck, "Discrete-Time Signal Processing," 2nd edition. Upper Saddle River, N.J.: Prentice Hall, 1999.

<sup>1</sup> N. Damera-Venkata and B. L. Evans, "Optimal design of real and complex minimum phase digital FIR filters," Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on, Phoenix, AZ, 1999, pp. 1145-1148 vol.3. DOI:10.1109/ICASSP.1999.756179

## Notes

Both the Hilbert<sup>Page 495, 1</sup> or homomorphic<sup>Page 495, 4Page 495, 5</sup> methods require selection of an FFT length to estimate the complex cepstrum of the filter.

In the case of the Hilbert method, the deviation from the ideal spectrum `epsilon` is related to the number of stopband zeros `n_stop` and FFT length `n_fft` as:

$$\text{epsilon} = 2. * \text{n\_stop} / \text{n\_fft}$$

For example, with 100 stopband zeros and a FFT length of 2048, `epsilon = 0.0976`. If we conservatively assume that the number of stopband zeros is one less than the filter length, we can take the FFT length to be the next power of 2 that satisfies `epsilon=0.01` as:

$$\text{n\_fft} = 2 ** \text{int}(\text{np.ceil}(\text{np.log2}(2 * (\text{len}(h) - 1) / 0.01)))$$

This gives reasonable results for both the Hilbert and homomorphic methods, and gives the value used when `n_fft=None`.

Alternative implementations exist for creating minimum-phase filters, including zero inversion<sup>2</sup> and spectral factorization<sup>3Page 495, 4Page 495, 5</sup>. For more information, see:

<http://dspguru.com/dsp/howtos/how-to-design-minimum-phase-fir-filters>

## References

### cupyx.scipy.signal.savgol\_coeffs

`cupyx.scipy.signal.savgol_coeffs(window_length, polyorder, deriv=0, delta=1.0, pos=None, use='conv')`

Compute the coefficients for a 1-D Savitzky-Golay FIR filter.

#### Parameters

- **window\_length** (*int*) – The length of the filter window (i.e., the number of coefficients).
- **polyorder** (*int*) – The order of the polynomial used to fit the samples. *polyorder* must be less than *window\_length*.
- **deriv** (*int*, *optional*) – The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.
- **delta** (*float*, *optional*) – The spacing of the samples to which the filter will be applied. This is only used if *deriv* > 0.
- **pos** (*int* or *None*, *optional*) – If *pos* is not *None*, it specifies evaluation position within the window. The default is the middle of the window.
- **use** (*str*, *optional*) – Either ‘conv’ or ‘dot’. This argument chooses the order of the coefficients. The default is ‘conv’, which means that the coefficients are ordered to be used in a convolution. With *use*=‘dot’, the order is reversed, so the filter is applied by dotting the coefficients with the data set.

#### Returns

**coeffs** – The filter coefficients.

<sup>2</sup> X. Chen and T. W. Parks, “Design of optimal minimum phase FIR filters by direct factorization,” *Signal Processing*, vol. 10, no. 4, pp. 369-383, Jun. 1986.

<sup>3</sup> T. Saramaki, “Finite Impulse Response Filter Design,” in *Handbook for Digital Signal Processing*, chapter 4, New York: Wiley-Interscience, 1993.

**Return type**

1-D ndarray

**See also:**`scipy.signal.savgol_coeffs`, `savgol_filter`**References**

A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. *Analytical Chemistry*, 1964, 36 (8), pp 1627-1639. Jianwen Luo, Kui Ying, and Jing Bai. 2005. Savitzky-Golay smoothing and differentiation filter for even number data. *Signal Process.* 85, 7 (July 2005), 1429-1434.

**Examples**

```
>>> import numpy as np
>>> from scipy.signal import savgol_coeffs
>>> savgol_coeffs(5, 2)
array([-0.08571429,  0.34285714,  0.48571429,  0.34285714, -0.08571429])
>>> savgol_coeffs(5, 2, deriv=1)
array([ 2.00000000e-01,  1.00000000e-01,  2.07548111e-16, -1.00000000e-01,
        -2.00000000e-01])
```

Note that use='dot' simply reverses the coefficients.

```
>>> savgol_coeffs(5, 2, pos=3)
array([ 0.25714286,  0.37142857,  0.34285714,  0.17142857, -0.14285714])
>>> savgol_coeffs(5, 2, pos=3, use='dot')
array([-0.14285714,  0.17142857,  0.34285714,  0.37142857,  0.25714286])
>>> savgol_coeffs(4, 2, pos=3, deriv=1, use='dot')
array([0.45, -0.85, -0.65,  1.05])
```

`x` contains data from the parabola  $x = t^2$ , sampled at  $t = -1, 0, 1, 2, 3$ . `c` holds the coefficients that will compute the derivative at the last position. When dotted with `x` the result should be 6.

```
>>> x = np.array([1, 0, 1, 4, 9])
>>> c = savgol_coeffs(5, 2, pos=4, deriv=1, use='dot')
>>> c.dot(x)
6.0
```

**cupyx.scipy.signal.gammatone**

`cupyx.scipy.signal.gammatone(freq, ftype, order=None, numtaps=None, fs=None)`

Gammatone filter design.

This function computes the coefficients of an FIR or IIR gammatone digital filter<sup>1</sup>.

**Parameters**

- **freq** (*float*) – Center frequency of the filter (expressed in the same units as *fs*).

<sup>1</sup> Slaney, Malcolm, “An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank”, Apple Computer Technical Report 35, 1993, pp.3-8, 34-39.

- **ftype** (`{'fir', 'iir'}`) – The type of filter the function generates. If ‘fir’, the function will generate an Nth order FIR gammatone filter. If ‘iir’, the function will generate an 8th order digital IIR filter, modeled as as 4th order gammatone filter.
- **order** (`int`, *optional*) – The order of the filter. Only used when `ftype='fir'`. Default is 4 to model the human auditory system. Must be between 0 and 24.
- **numtaps** (`int`, *optional*) – Length of the filter. Only used when `ftype='fir'`. Default is `fs*0.015` if `fs` is greater than 1000, 15 if `fs` is less than or equal to 1000.
- **fs** (`float`, *optional*) – The sampling frequency of the signal. `freq` must be between 0 and `fs/2`. Default is 2.

**Returns**

**b, a** – Numerator (b) and denominator (a) polynomials of the filter.

**Return type**

*ndarray, ndarray*

**Raises**

**ValueError** – If `freq` is less than or equal to 0 or greater than or equal to `fs/2`, if `ftype` is not ‘fir’ or ‘iir’, if `order` is less than or equal to 0 or greater than 24 when `ftype='fir'`

**See also:**

*[firwin](#), [iirfilter](#)*

**References****cupyx.scipy.signal.group\_delay**

`cupyx.scipy.signal.group_delay(system, w=512, whole=False, fs=6.283185307179586)`

Compute the group delay of a digital filter.

The group delay measures by how many samples amplitude envelopes of various spectral components of a signal are delayed by a filter. It is formally defined as the derivative of continuous (unwrapped) phase:

$$D(w) = - \frac{d}{dw} \arg H(e^{jw})$$

**Parameters**

- **system** (*tuple of array\_like* (`b, a`)) – Numerator and denominator coefficients of a filter transfer function.
- **w** (`{None, int, array_like}`, *optional*) – If a single integer, then compute at that many frequencies (default is `N=512`).  
If an `array_like`, compute the delay at the frequencies given. These are in the same units as `fs`.
- **whole** (`bool`, *optional*) – Normally, frequencies are computed from 0 to the Nyquist frequency, `fs/2` (upper-half of unit-circle). If `whole` is `True`, compute frequencies from 0 to `fs`. Ignored if `w` is `array_like`.
- **fs** (`float`, *optional*) – The sampling frequency of the digital system. Defaults to `2*pi` radians/sample (so `w` is from 0 to `pi`).

**Returns**

- **w** (*ndarray*) – The frequencies at which group delay was computed, in the same units as *fs*. By default, *w* is normalized to the range  $[0, \pi)$  (radians/sample).
- **gd** (*ndarray*) – The group delay.

See also:

*freqz*

Frequency response of a digital filter

## Notes

The similar function in MATLAB is called *grpdelay*.

If the transfer function  $H(z)$  has zeros or poles on the unit circle, the group delay at corresponding frequencies is undefined. When such a case arises the warning is raised and the group delay is set to 0 at those frequencies.

For the details of numerical computation of the group delay refer to<sup>1</sup>.

## References

### cupyx.scipy.signal.iirdesign

`cupyx.scipy.signal.iirdesign(wp, ws, gpass, gstop, analog=False, ftype='ellip', output='ba', fs=None)`

Complete IIR digital and analog filter design.

Given passband and stopband frequencies and gains, construct an analog or digital IIR filter of minimum order for a given basic type. Return the output in numerator, denominator ('ba'), pole-zero ('zpk') or second order sections ('sos') form.

#### Parameters

- **wp** (*float or array like, shape (2,)*) – Passband and stopband edge frequencies. Possible values are scalars (for lowpass and highpass filters) or ranges (for bandpass and bandstop filters). For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s). Note, that for bandpass and bandstop filters passband must lie strictly inside stopband or vice versa.

- **ws** (*float or array like, shape (2,)*) – Passband and stopband edge frequencies. Possible values are scalars (for lowpass and highpass filters) or ranges (for bandpass and bandstop filters). For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3

<sup>1</sup> Richard G. Lyons, "Understanding Digital Signal Processing, 3rd edition", p. 830.

- Highpass:  $w_p = 0.3$ ,  $w_s = 0.2$
- Bandpass:  $w_p = [0.2, 0.5]$ ,  $w_s = [0.1, 0.6]$
- Bandstop:  $w_p = [0.1, 0.6]$ ,  $w_s = [0.2, 0.5]$

For analog filters,  $w_p$  and  $w_s$  are angular frequencies (e.g., rad/s). Note, that for bandpass and bandstop filters passband must lie strictly inside stopband or vice versa.

- **gpass** (*float*) – The maximum loss in the passband (dB).
- **gstop** (*float*) – The minimum attenuation in the stopband (dB).
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **ftype** (*str*, *optional*) – The type of IIR filter to design:
  - Butterworth : ‘butter’
  - Chebyshev I : ‘cheby1’
  - Chebyshev II : ‘cheby2’
  - Cauer/elliptic: ‘ellip’
- **output** (*{‘ba’, ‘zpk’, ‘sos’}*, *optional*) – Filter form of the output:
  - second-order sections (recommended): ‘sos’
  - numerator/denominator (default) : ‘ba’
  - pole-zero : ‘zpk’

In general the second-order sections (‘sos’) form is recommended because inferring the coefficients for the numerator/denominator form (‘ba’) suffers from numerical instabilities. For reasons of backward compatibility the default form is the numerator/denominator form (‘ba’), where the ‘b’ and the ‘a’ in ‘ba’ refer to the commonly used names of the coefficients used.

Note: Using the second-order sections form (‘sos’) is sometimes associated with additional computational costs: for data-intense use cases it is therefore recommended to also investigate the numerator/denominator form (‘ba’).

- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

New in version 1.2.0.

### Returns

- **b, a** (*ndarray*, *ndarray*) – Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.
- **z, p, k** (*ndarray*, *ndarray*, *float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if `output='sos'`.

See also:

`scipy.signal.iirdesign`

`butter`

Filter design using order and critical points

`cheby1`, `cheby2`, `ellip`, `bessel`



**buttord**

Find order and critical points from passband and stopband spec

*cheb1ord*, *cheb2ord*, *ellipord*

**iirfilter**

General filter design using order and critical frequencies

**cupyx.scipy.signal.iirfilter**

```
cupyx.scipy.signal.iirfilter(N, Wn, rp=None, rs=None, btype='band', analog=False, ftype='butter',
                             output='ba', fs=None)
```

IIR digital and analog filter design given order and critical points.

Design an Nth-order digital or analog filter and return the filter coefficients.

**Parameters**

- **N** (*int*) – The order of the filter.
- **Wn** (*array\_like*) – A scalar or length-2 sequence giving the critical frequencies.  
 For digital filters, *Wn* are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*Wn* is thus in half-cycles / sample.)  
 For analog filters, *Wn* is an angular frequency (e.g., rad/s).  
 When *Wn* is a length-2 sequence, *Wn*[0] must be less than *Wn*[1].
- **rp** (*float*, *optional*) – For Chebyshev and elliptic filters, provides the maximum ripple in the passband. (dB)
- **rs** (*float*, *optional*) – For Chebyshev and elliptic filters, provides the minimum attenuation in the stop band. (dB)
- **btype** ({'bandpass', 'lowpass', 'highpass', 'bandstop'}, *optional*) – The type of filter. Default is 'bandpass'.
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **ftype** (*str*, *optional*) – The type of IIR filter to design:
  - Butterworth : 'butter'
  - Chebyshev I : 'cheby1'
  - Chebyshev II : 'cheby2'
  - Cauer/elliptic: 'ellip'
  - Bessel/Thomson: 'bessel'
- **output** ({'ba', 'zpk', 'sos'}, *optional*) – Filter form of the output:
  - second-order sections (recommended): 'sos'
  - numerator/denominator (default) : 'ba'
  - pole-zero : 'zpk'

In general the second-order sections ('sos') form is recommended because inferring the coefficients for the numerator/denominator form ('ba') suffers from numerical instabilities. For reasons of backward compatibility the default form is the numerator/denominator form ('ba'), where the 'b' and the 'a' in 'ba' refer to the commonly used names of the coefficients used.

Note: Using the second-order sections form ('sos') is sometimes associated with additional computational costs: for data-intense use cases it is therefore recommended to also investigate the numerator/denominator form ('ba').

- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

#### Returns

- **b, a** (*ndarray*, *ndarray*) – Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.
- **z, p, k** (*ndarray*, *ndarray*, *float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if `output='sos'`.

See also:

#### *butter*

Filter design using order and critical points

*cheby1*, *cheby2*, *ellip*, *bessel*

#### *buttord*

Find order and critical points from passband and stopband spec

*cheb1ord*, *cheb2ord*, *ellipord*

#### *iirdesign*

General filter design using passband and stopband spec

`scipy.signal.iirfilter`

## cupyx.scipy.signal.kaiser\_atten

`cupyx.scipy.signal.kaiser_atten(numtaps, width)`

Compute the attenuation of a Kaiser FIR filter.

Given the number of taps  $N$  and the transition width  $width$ , compute the attenuation  $a$  in dB, given by Kaiser's formula:

$$a = 2.285 * (N - 1) * \pi * width + 7.95$$

#### Parameters

- **numtaps** (*int*) – The number of taps in the FIR filter.
- **width** (*float*) – The desired width of the transition region between passband and stopband (or, in general, at any discontinuity) for the filter, expressed as a fraction of the Nyquist frequency.

#### Returns

**a** – The attenuation of the ripple, in dB.

**Return type**

float

**See also:**`scipy.signal.kaiser_atten`**cupyx.scipy.signal.kaiser\_beta**`cupyx.scipy.signal.kaiser_beta(a)`Compute the Kaiser parameter *beta*, given the attenuation *a*.**Parameters****a** (*float*) – The desired attenuation in the stopband and maximum ripple in the passband, in dB. This should be a *positive* number.**Returns****beta** – The *beta* parameter to be used in the formula for a Kaiser window.**Return type**

float

**References**

Oppenheim, Schaffer, “Discrete-Time Signal Processing”, p.475-476.

**See also:**`scipy.signal.kaiser_beta`**cupyx.scipy.signal.kaiserord**`cupyx.scipy.signal.kaiserord(ripple, width)`

Determine the filter window parameters for the Kaiser window method.

The parameters returned by this function are generally used to create a finite impulse response filter using the window method, with either *firwin* or *firwin2*.**Parameters**

- **ripple** (*float*) – Upper bound for the deviation (in dB) of the magnitude of the filter’s frequency response from that of the desired filter (not including frequencies in any transition intervals). That is, if *w* is the frequency expressed as a fraction of the Nyquist frequency, *A(w)* is the actual frequency response of the filter and *D(w)* is the desired frequency response, the design requirement is that:

$$| \text{abs}(A(w) - D(w)) | < 10^{**(-\text{ripple}/20)}$$

for  $0 \leq w \leq 1$  and *w* not in a transition interval.

- **width** (*float*) – Width of transition region, normalized so that 1 corresponds to pi radians / sample. That is, the frequency is expressed as a fraction of the Nyquist frequency.

**Returns**

- **numtaps** (*int*) – The length of the Kaiser window.

- **beta** (*float*) – The beta parameter for the Kaiser window.

See also:

`scipy.signal.kaiserord`

### cupyx.scipy.signal.unique\_roots

`cupyx.scipy.signal.unique_roots(p, tol=0.001, rtype='min')`

Determine unique roots and their multiplicities from a list of roots.

#### Parameters

- **p** (*array\_like*) – The list of roots.
- **tol** (*float*, *optional*) – The tolerance for two roots to be considered equal in terms of the distance between them. Default is 1e-3. Refer to Notes about the details on roots grouping.
- **rtype** (*{'max', 'maximum', 'min', 'minimum', 'avg', 'mean'}, optional*) – How to determine the returned root if multiple roots are within *tol* of each other.
  - 'max', 'maximum': pick the maximum of those roots
  - 'min', 'minimum': pick the minimum of those roots
  - 'avg', 'mean': take the average of those roots

When finding minimum or maximum among complex roots they are compared first by the real part and then by the imaginary part.

#### Returns

- **unique** (*ndarray*) – The list of unique roots.
- **multiplicity** (*ndarray*) – The multiplicity of each root.

See also:

`scipy.signal.unique_roots`

### Notes

If we have 3 roots *a*, *b* and *c*, such that *a* is close to *b* and *b* is close to *c* (distance is less than *tol*), then it doesn't necessarily mean that *a* is close to *c*. It means that roots grouping is not unique. In this function we use “greedy” grouping going through the roots in the order they are given in the input *p*.

This utility function is not specific to roots but can be used for any sequence of values for which uniqueness and multiplicity has to be determined. For a more general routine, see `numpy.unique`.

### cupyx.scipy.signal.residue

`cupyx.scipy.signal.residue(b, a, tol=0.001, rtype='avg')`

Compute partial-fraction expansion of  $b(s) / a(s)$ .

If *M* is the degree of numerator *b* and *N* the degree of denominator *a*:

$$H(s) = \frac{b(s)}{a(s)} = \frac{b[0] s^{**}(M) + b[1] s^{**}(M-1) + \dots + b[M]}{a[0] s^{**}(N) + a[1] s^{**}(N-1) + \dots + a[N]}$$

then the partial-fraction expansion  $H(s)$  is defined as:

$$= \frac{r[0]}{(s-p[0])} + \frac{r[1]}{(s-p[1])} + \dots + \frac{r[-1]}{(s-p[-1])} + k(s)$$

If there are any repeated roots (closer together than *tol*), then  $H(s)$  has terms like:

$$\frac{r[i]}{(s-p[i])} + \frac{r[i+1]}{(s-p[i])**2} + \dots + \frac{r[i+n-1]}{(s-p[i])**n}$$

This function is used for polynomials in positive powers of  $s$  or  $z$ , such as analog filters or digital filters in controls engineering. For negative powers of  $z$  (typical for digital filters in DSP), use *residuez*.

See Notes for details about the algorithm.

### Parameters

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **tol** (*float, optional*) – The tolerance for two roots to be considered equal in terms of the distance between them. Default is 1e-3. See *unique\_roots* for further details.
- **rtype** (*{'avg', 'min', 'max'}, optional*) – Method for computing a root to represent a group of identical roots. Default is 'avg'. See *unique\_roots* for further details.

### Returns

- **r** (*ndarray*) – Residues corresponding to the poles. For repeated poles, the residues are ordered to correspond to ascending by power fractions.
- **p** (*ndarray*) – Poles ordered by magnitude in ascending order.
- **k** (*ndarray*) – Coefficients of the direct polynomial term.

**Warning:** This function may synchronize the device.

See also:

`scipy.signal.residue`, `invres`, `residuez`, `numpy.poly`, `unique_roots`

### Notes

The “deflation through subtraction” algorithm is used for computations — method 6 in<sup>1</sup>.

The form of partial fraction expansion depends on poles multiplicity in the exact mathematical sense. However there is no way to exactly determine multiplicity of roots of a polynomial in numerical computing. Thus you should think of the result of *residue* with given *tol* as partial fraction expansion computed for the denominator composed of the computed poles with empirically determined multiplicity. The choice of *tol* can drastically change the result if there are close poles.

<sup>1</sup> J. F. Mahoney, B. D. Sivazlian, “Partial fractions expansion: a review of computational methodology and efficiency”, Journal of Computational and Applied Mathematics, Vol. 9, 1983.

## References

### cupyx.scipy.signal.residuez

`cupyx.scipy.signal.residuez(b, a, tol=0.001, rtype='avg')`

Compute partial-fraction expansion of  $b(z) / a(z)$ .

If  $M$  is the degree of numerator  $b$  and  $N$  the degree of denominator  $a$ :

$$H(z) = \frac{b(z)}{a(z)} = \frac{b[0] + b[1] z^{**(-1)} + \dots + b[M] z^{**(-M)}}{a[0] + a[1] z^{**(-1)} + \dots + a[N] z^{**(-N)}}$$

then the partial-fraction expansion  $H(z)$  is defined as:

$$= \frac{r[0]}{(1-p[0]z^{**(-1)})} + \dots + \frac{r[-1]}{(1-p[-1]z^{**(-1)})} + k[0] + k[1]z^{**(-1)} \dots$$

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

$$\frac{r[i]}{(1-p[i]z^{**(-1)})} + \frac{r[i+1]}{(1-p[i]z^{**(-1)})^{**2}} + \dots + \frac{r[i+n-1]}{(1-p[i]z^{**(-1)})^{**n}}$$

This function is used for polynomials in negative powers of  $z$ , such as digital filters in DSP. For positive powers, use *residue*.

See Notes of *residue* for details about the algorithm.

#### Parameters

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **tol** (*float*, *optional*) – The tolerance for two roots to be considered equal in terms of the distance between them. Default is 1e-3. See *unique\_roots* for further details.
- **rtype** (*{'avg', 'min', 'max'}*, *optional*) – Method for computing a root to represent a group of identical roots. Default is 'avg'. See *unique\_roots* for further details.

#### Returns

- **r** (*ndarray*) – Residues corresponding to the poles. For repeated poles, the residues are ordered to correspond to ascending by power fractions.
- **p** (*ndarray*) – Poles ordered by magnitude in ascending order.
- **k** (*ndarray*) – Coefficients of the direct polynomial term.

**Warning:** This function may synchronize the device.

See also:

`scipy.signal.residuez`, `invresz`, `residue`, `unique_roots`

**cupyx.scipy.signal.invres**

`cupyx.scipy.signal.invres(r, p, k, tol=0.001, rtype='avg')`

Compute  $b(s)$  and  $a(s)$  from partial fraction expansion.

If  $M$  is the degree of numerator  $b$  and  $N$  the degree of denominator  $a$ :

$$H(s) = \frac{b(s)}{a(s)} = \frac{b[0] s^{**}(M) + b[1] s^{**}(M-1) + \dots + b[M]}{a[0] s^{**}(N) + a[1] s^{**}(N-1) + \dots + a[N]}$$

then the partial-fraction expansion  $H(s)$  is defined as:

$$= \frac{r[0]}{(s-p[0])} + \frac{r[1]}{(s-p[1])} + \dots + \frac{r[-1]}{(s-p[-1])} + k(s)$$

If there are any repeated roots (closer together than *tol*), then  $H(s)$  has terms like:

$$\frac{r[i]}{(s-p[i])} + \frac{r[i+1]}{(s-p[i])**2} + \dots + \frac{r[i+n-1]}{(s-p[i])**n}$$

This function is used for polynomials in positive powers of  $s$  or  $z$ , such as analog filters or digital filters in controls engineering. For negative powers of  $z$  (typical for digital filters in DSP), use *invresz*.

**Parameters**

- **r** (*array\_like*) – Residues corresponding to the poles. For repeated poles, the residues must be ordered to correspond to ascending by power fractions.
- **p** (*array\_like*) – Poles. Equal poles must be adjacent.
- **k** (*array\_like*) – Coefficients of the direct polynomial term.
- **tol** (*float, optional*) – The tolerance for two roots to be considered equal in terms of the distance between them. Default is 1e-3. See *unique\_roots* for further details.
- **rtype** (*{'avg', 'min', 'max'}, optional*) – Method for computing a root to represent a group of identical roots. Default is 'avg'. See *unique\_roots* for further details.

**Returns**

- **b** (*ndarray*) – Numerator polynomial coefficients.
- **a** (*ndarray*) – Denominator polynomial coefficients.

See also:

`scipy.signal.invres`, `residue`, `invresz`, `unique_roots`

**cupyx.scipy.signal.invresz**

`cupyx.scipy.signal.invresz(r, p, k, tol=0.001, rtype='avg')`

Compute  $b(z)$  and  $a(z)$  from partial fraction expansion.

If  $M$  is the degree of numerator  $b$  and  $N$  the degree of denominator  $a$ :

$$H(z) = \frac{b(z)}{a(z)} = \frac{b[0] + b[1] z^{**}(-1) + \dots + b[M] z^{**}(-M)}{a[0] + a[1] z^{**}(-1) + \dots + a[N] z^{**}(-N)}$$

then the partial-fraction expansion  $H(z)$  is defined as:

$$= \frac{r[0]}{(1-p[0]z^{**}(-1))} + \dots + \frac{r[-1]}{(1-p[-1]z^{**}(-1))} + k[0] + k[1]z^{**}(-1) \dots$$

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

$$\frac{r[i]}{(1-p[i]z^{**}(-1))} + \frac{r[i+1]}{(1-p[i]z^{**}(-1))^{**2}} + \dots + \frac{r[i+n-1]}{(1-p[i]z^{**}(-1))^{**n}}$$

This function is used for polynomials in negative powers of  $z$ , such as digital filters in DSP. For positive powers, use *invres*.

**Parameters**

- **r** (*array\_like*) – Residues corresponding to the poles. For repeated poles, the residues must be ordered to correspond to ascending by power fractions.
- **p** (*array\_like*) – Poles. Equal poles must be adjacent.
- **k** (*array\_like*) – Coefficients of the direct polynomial term.
- **tol** (*float, optional*) – The tolerance for two roots to be considered equal in terms of the distance between them. Default is 1e-3. See *unique\_roots* for further details.
- **rtype** (*{'avg', 'min', 'max'}, optional*) – Method for computing a root to represent a group of identical roots. Default is 'avg'. See *unique\_roots* for further details.

**Returns**

- **b** (*ndarray*) – Numerator polynomial coefficients.
- **a** (*ndarray*) – Denominator polynomial coefficients.

See also:

`scipy.signal.invresz`, `residuez`, `unique_roots`, `invres`



**cupyx.scipy.signal.BadCoefficients****exception** cupyx.scipy.signal.BadCoefficients

Warning about badly conditioned filter coefficients

**Matlab-style IIR filter design**

<i>butter</i> (N, Wn[, btype, analog, output, fs])	Butterworth digital and analog filter design.
<i>buttord</i> (wp, ws, gpass, gstop[, analog, fs])	Butterworth filter order selection.
<i>ellip</i> (N, rp, rs, Wn[, btype, analog, output, fs])	Elliptic (Cauer) digital and analog filter design.
<i>ellipord</i> (wp, ws, gpass, gstop[, analog, fs])	Elliptic (Cauer) filter order selection.
<i>cheby1</i> (N, rp, Wn[, btype, analog, output, fs])	Chebyshev type I digital and analog filter design.
<i>cheb1ord</i> (wp, ws, gpass, gstop[, analog, fs])	Chebyshev type I filter order selection.
<i>cheby2</i> (N, rs, Wn[, btype, analog, output, fs])	Chebyshev type II digital and analog filter design.
<i>cheb2ord</i> (wp, ws, gpass, gstop[, analog, fs])	Chebyshev type II filter order selection.
<i>iircomb</i> (w0, Q[, ftype, fs, pass_zero])	Design IIR notching or peaking digital comb filter.
<i>iirnotch</i> (w0, Q[, fs])	Design second-order IIR notch digital filter.
<i>iirpeak</i> (w0, Q[, fs])	Design second-order IIR peak (resonant) digital filter.

**cupyx.scipy.signal.butter**

cupyx.scipy.signal.butter(N, Wn, btype='low', analog=False, output='ba', fs=None)

Butterworth digital and analog filter design.

Design an Nth-order digital or analog Butterworth filter and return the filter coefficients.

**Parameters**

- **N** (*int*) – The order of the filter. For ‘bandpass’ and ‘bandstop’ filters, the resulting order of the final second-order sections (‘sos’) matrix is 2\*N, with *N* the number of biquad sections of the desired system.
- **Wn** (*array\_like*) – The critical frequency or frequencies. For lowpass and highpass filters, Wn is a scalar; for bandpass and bandstop filters, Wn is a length-2 sequence.  
  
For a Butterworth filter, this is the point at which the gain drops to 1/sqrt(2) that of the passband (the “-3 dB point”).  
  
For digital filters, if *fs* is not specified, *Wn* units are normalized from 0 to 1, where 1 is the Nyquist frequency (*Wn* is thus in half cycles / sample and defined as 2\*critical frequencies / *fs*). If *fs* is specified, *Wn* is in the same units as *fs*.  
  
For analog filters, *Wn* is an angular frequency (e.g. rad/s).
- **btype** ({‘lowpass’, ‘highpass’, ‘bandpass’, ‘bandstop’}, *optional*) – The type of filter. Default is ‘lowpass’.
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **output** ({‘ba’, ‘zpk’, ‘sos’}, *optional*) – Type of output: numerator/denominator (‘ba’), pole-zero (‘zpk’), or second-order sections (‘sos’). Default is ‘ba’ for backwards compatibility, but ‘sos’ should be used for general-purpose filtering.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

### Returns

- **b, a** (*ndarray, ndarray*) – Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.
- **z, p, k** (*ndarray, ndarray, float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if `output='sos'`.

See also:

[`buttord`](#), [`buttap`](#), [`iirfilter`](#), [`scipy.signal.butter`](#)

### Notes

The Butterworth filter has maximally flat frequency response in the passband.

If the transfer function form `[b, a]` is requested, numerical problems can occur since the conversion between roots and the polynomial coefficients is a numerically sensitive operation, even for  $N \geq 4$ . It is recommended to work with the SOS representation.

**Warning:** Designing high-order and narrowband IIR filters in TF form can result in unstable or incorrect filtering due to floating point numerical precision issues. Consider inspecting output filter characteristics `freqz` or designing the filters with second-order sections via `output='sos'`.

## cupyx.scipy.signal.buttord

`cupyx.scipy.signal.buttord(wp, ws, gpass, gstop, analog=False, fs=None)`

Butterworth filter order selection.

Return the order of the lowest order digital or analog Butterworth filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

### Parameters

- **wp** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **ws** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3

- Highpass:  $w_p = 0.3$ ,  $w_s = 0.2$
- Bandpass:  $w_p = [0.2, 0.5]$ ,  $w_s = [0.1, 0.6]$
- Bandstop:  $w_p = [0.1, 0.6]$ ,  $w_s = [0.2, 0.5]$

For analog filters,  $w_p$  and  $w_s$  are angular frequencies (e.g., rad/s).

- **gpass** (*float*) – The maximum loss in the passband (dB).
- **gstop** (*float*) – The minimum attenuation in the stopband (dB).
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

New in version 1.2.0.

#### Returns

- **ord** (*int*) – The lowest order for a Butterworth filter which meets specs.
- **wn** (*ndarray or float*) – The Butterworth natural frequency (i.e. the “3dB frequency”). Should be used with *butter* to give filter results. If *fs* is specified, this is in the same units, and *fs* must also be passed to *butter*.

#### See also:

`scipy.signal.buttord`

#### *butter*

Filter design using order and critical points

#### *cheb1ord*

Find order and critical points from passband and stopband spec

*cheb2ord*, *ellipord*

#### *iirfilter*

General filter design using order and critical frequencies

#### *iirdesign*

General filter design using passband and stopband spec

### cupyx.scipy.signal.ellip

`cupyx.scipy.signal.ellip(N, rp, rs, Wn, btype='low', analog=False, output='ba', fs=None)`

Elliptic (Cauer) digital and analog filter design.

Design an Nth-order digital or analog elliptic filter and return the filter coefficients.

#### Parameters

- **N** (*int*) – The order of the filter.
- **rp** (*float*) – The maximum ripple allowed below unity gain in the passband. Specified in decibels, as a positive number.
- **rs** (*float*) – The minimum attenuation required in the stop band. Specified in decibels, as a positive number.

- **Wn** (*array\_like*) – A scalar or length-2 sequence giving the critical frequencies. For elliptic filters, this is the point in the transition band at which the gain first drops below *-rp*.

For digital filters, *Wn* are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*Wn* is thus in half-cycles / sample.)

For analog filters, *Wn* is an angular frequency (e.g., rad/s).

- **btype** (*{'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional*) – The type of filter. Default is 'lowpass'.
- **analog** (*bool, optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **output** (*{'ba', 'zpk', 'sos'}, optional*) – Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba' for backwards compatibility, but 'sos' should be used for general-purpose filtering.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

#### Returns

- **b, a** (*ndarray, ndarray*) – Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if *output*='ba'.
- **z, p, k** (*ndarray, ndarray, float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if *output*='zpk'.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if *output*='sos'.

See also:

[ellipord](#), [ellipap](#), [iirfilter](#), [scipy.signal.ellip](#)

#### Notes

Also known as Cauer or Zolotarev filters, the elliptical filter maximizes the rate of transition between the frequency response's passband and stopband, at the expense of ripple in both, and increased ringing in the step response.

As *rp* approaches 0, the elliptical filter becomes a Chebyshev type II filter (*cheby2*). As *rs* approaches 0, it becomes a Chebyshev type I filter (*cheby1*). As both approach 0, it becomes a Butterworth filter (*butter*).

The equiripple passband has *N* maxima or minima (for example, a 5th-order filter has 3 maxima and 2 minima). Consequently, the DC gain is unity for odd-order filters, or -*rp* dB for even-order filters.

### cupyx.scipy.signal.ellipord

`cupyx.scipy.signal.ellipord(wp, ws, gpass, gstop, analog=False, fs=None)`

Elliptic (Cauer) filter order selection.

Return the order of the lowest order digital or analog elliptic filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

#### Parameters

- **wp** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **ws** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **gpass** (*float*) – The maximum loss in the passband (dB).
- **gstop** (*float*) – The minimum attenuation in the stopband (dB).
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

#### Returns

- **ord** (*int*) – The lowest order for an Elliptic (Cauer) filter that meets specs.
- **wn** (*ndarray or float*) – The Chebyshev natural frequency (the “3dB frequency”) for use with *ellip* to give filter results. If *fs* is specified, this is in the same units, and *fs* must also be passed to *ellip*.

#### See also:

`scipy.signal.ellipord`

`ellip`

Filter design using order and critical points

`buttord`

Find order and critical points from passband and stopband spec

`cheb1ord`, `cheb2ord`

`iirfilter`

General filter design using order and critical frequencies

`iirdesign`

General filter design using passband and stopband spec

## cupyx.scipy.signal.cheby1

`cupyx.scipy.signal.cheby1(N, rp, Wn, btype='low', analog=False, output='ba', fs=None)`

Chebyshev type I digital and analog filter design.

Design an Nth-order digital or analog Chebyshev type I filter and return the filter coefficients.

### Parameters

- **N** (*int*) – The order of the filter.
- **rp** (*float*) – The maximum ripple allowed below unity gain in the passband. Specified in decibels, as a positive number.
- **Wn** (*array\_like*) – A scalar or length-2 sequence giving the critical frequencies. For Type I filters, this is the point in the transition band at which the gain first drops below  $-rp$ .  
  
For digital filters,  $Wn$  are in the same units as  $fs$ . By default,  $fs$  is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. ( $Wn$  is thus in half-cycles / sample.)  
  
For analog filters,  $Wn$  is an angular frequency (e.g., rad/s).
- **btype** (*{'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional*) – The type of filter. Default is 'lowpass'.
- **analog** (*bool, optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **output** (*{'ba', 'zpk', 'sos'}, optional*) – Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba' for backwards compatibility, but 'sos' should be used for general-purpose filtering.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

### Returns

- **b, a** (*ndarray, ndarray*) – Numerator ( $b$ ) and denominator ( $a$ ) polynomials of the IIR filter. Only returned if `output='ba'`.
- **z, p, k** (*ndarray, ndarray, float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if `output='sos'`.

### See also:

[`cheblord`](#), [`cheblap`](#), [`iirfilter`](#), [`scipy.signal.cheby1`](#)

### Notes

The Chebyshev type I filter maximizes the rate of cutoff between the frequency response's passband and stopband, at the expense of ripple in the passband and increased ringing in the step response.

Type I filters roll off faster than Type II ([`cheby2`](#)), but Type II filters do not have any ripple in the passband.

The equiripple passband has  $N$  maxima or minima (for example, a 5th-order filter has 3 maxima and 2 minima). Consequently, the DC gain is unity for odd-order filters, or  $-rp$  dB for even-order filters.

**cupyx.scipy.signal.cheb1ord**

`cupyx.scipy.signal.cheb1ord(wp, ws, gpass, gstop, analog=False, fs=None)`

Chebyshev type I filter order selection.

Return the order of the lowest order digital or analog Chebyshev Type I filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

**Parameters**

- **wp** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **ws** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **gpass** (*float*) – The maximum loss in the passband (dB).
- **gstop** (*float*) – The minimum attenuation in the stopband (dB).
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

**Returns**

- **ord** (*int*) – The lowest order for a Chebyshev type I filter that meets specs.
- **wn** (*ndarray or float*) – The Chebyshev natural frequency (the “3dB frequency”) for use with *cheby1* to give filter results. If *fs* is specified, this is in the same units, and *fs* must also be passed to *cheby1*.

See also:

[`scipy.signal.cheb1ord`](#)

[`cheby1`](#)

Filter design using order and critical points

**buttord**

Find order and critical points from passband and stopband spec

**cheb2ord, ellipord****iirfilter**

General filter design using order and critical frequencies

**iirdesign**

General filter design using passband and stopband spec

**cupyx.scipy.signal.cheby2**

`cupyx.scipy.signal.cheby2(N, rs, Wn, btype='low', analog=False, output='ba', fs=None)`

Chebyshev type II digital and analog filter design.

Design an Nth-order digital or analog Chebyshev type II filter and return the filter coefficients.

**Parameters**

- **N** (*int*) – The order of the filter.
- **rs** (*float*) – The minimum attenuation required in the stop band. Specified in decibels, as a positive number.
- **Wn** (*array\_like*) – A scalar or length-2 sequence giving the critical frequencies. For Type II filters, this is the point in the transition band at which the gain first reaches  $-rs$ .  
  
For digital filters,  $Wn$  are in the same units as  $fs$ . By default,  $fs$  is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. ( $Wn$  is thus in half-cycles / sample.)  
  
For analog filters,  $Wn$  is an angular frequency (e.g., rad/s).
- **btype** (*{'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional*) – The type of filter. Default is 'lowpass'.
- **analog** (*bool, optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **output** (*{'ba', 'zpk', 'sos'}, optional*) – Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba' for backwards compatibility, but 'sos' should be used for general-purpose filtering.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

**Returns**

- **b, a** (*ndarray, ndarray*) – Numerator ( $b$ ) and denominator ( $a$ ) polynomials of the IIR filter. Only returned if `output='ba'`.
- **z, p, k** (*ndarray, ndarray, float*) – Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.
- **sos** (*ndarray*) – Second-order sections representation of the IIR filter. Only returned if `output='sos'`.

See also:

[`cheb2ord`](#), [`cheb2ap`](#), [`iirfilter`](#), [`scipy.signal.cheby2`](#)



## Notes

The Chebyshev type II filter maximizes the rate of cutoff between the frequency response's passband and stopband, at the expense of ripple in the stopband and increased ringing in the step response.

Type II filters do not roll off as fast as Type I (*cheby1*).

## cupyx.scipy.signal.cheb2ord

`cupyx.scipy.signal.cheb2ord(wp, ws, gpass, gstop, analog=False, fs=None)`

Chebyshev type II filter order selection.

Return the order of the lowest order digital or analog Chebyshev Type II filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

### Parameters

- **wp** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **ws** (*float*) – Passband and stopband edge frequencies.

For digital filters, these are in the same units as *fs*. By default, *fs* is 2 half-cycles/sample, so these are normalized from 0 to 1, where 1 is the Nyquist frequency. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: *wp* = 0.2, *ws* = 0.3
- Highpass: *wp* = 0.3, *ws* = 0.2
- Bandpass: *wp* = [0.2, 0.5], *ws* = [0.1, 0.6]
- Bandstop: *wp* = [0.1, 0.6], *ws* = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g., rad/s).

- **gpass** (*float*) – The maximum loss in the passband (dB).
- **gstop** (*float*) – The minimum attenuation in the stopband (dB).
- **analog** (*bool*, *optional*) – When True, return an analog filter, otherwise a digital filter is returned.
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

### Returns

- **ord** (*int*) – The lowest order for a Chebyshev type II filter that meets specs.
- **wn** (*ndarray or float*) – The Chebyshev natural frequency (the “3dB frequency”) for use with *cheby2* to give filter results. If *fs* is specified, this is in the same units, and *fs* must also be passed to *cheby2*.

See also:

`scipy.signal.cheb2ord`

**`cheby2`**

Filter design using order and critical points

**`buttord`**

Find order and critical points from passband and stopband spec

`cheb1ord`, `ellipord`

**`iirfilter`**

General filter design using order and critical frequencies

**`iirdesign`**

General filter design using passband and stopband spec

## **cupyx.scipy.signal.iircomb**

`cupyx.scipy.signal.iircomb(w0, Q, ftype='notch', fs=2.0, *, pass_zero=False)`

Design IIR notching or peaking digital comb filter.

A notching comb filter consists of regularly-spaced band-stop filters with a narrow bandwidth (high quality factor). Each rejects a narrow frequency band and leaves the rest of the spectrum little changed.

A peaking comb filter consists of regularly-spaced band-pass filters with a narrow bandwidth (high quality factor). Each rejects components outside a narrow frequency band.

### **Parameters**

- **`w0`** (*float*) – The fundamental frequency of the comb filter (the spacing between its peaks). This must evenly divide the sampling frequency. If `fs` is specified, this is in the same units as `fs`. By default, it is a normalized scalar that must satisfy  $0 < w0 < 1$ , with  $w0 = 1$  corresponding to half of the sampling frequency.
- **`Q`** (*float*) – Quality factor. Dimensionless parameter that characterizes notch filter -3 dB bandwidth `bw` relative to its center frequency,  $Q = w0/bw$ .
- **`ftype`** (`{'notch', 'peak'}`) – The type of comb filter generated by the function. If 'notch', then the `Q` factor applies to the notches. If 'peak', then the `Q` factor applies to the peaks. Default is 'notch'.
- **`fs`** (*float*, *optional*) – The sampling frequency of the signal. Default is 2.0.
- **`pass_zero`** (*bool*, *optional*) – If False (default), the notches (nulls) of the filter are centered on frequencies  $[0, w0, 2*w0, \dots]$ , and the peaks are centered on the midpoints  $[w0/2, 3*w0/2, 5*w0/2, \dots]$ . If True, the peaks are centered on  $[0, w0, 2*w0, \dots]$  (passing zero frequency) and vice versa.

### **Returns**

**`b`**, **`a`** – Numerator (`b`) and denominator (`a`) polynomials of the IIR filter.

### **Return type**

*ndarray*, *ndarray*

### **Raises**

**`ValueError`** – If `w0` is less than or equal to 0 or greater than or equal to `fs/2`, if `fs` is not divisible by `w0`, if `ftype` is not 'notch' or 'peak'

See also:

`scipy.signal.iircomb`, `iirnotch`, `iirpeak`

## Notes

The TF implementation of the comb filter is numerically stable even at higher orders due to the use of a single repeated pole, which won't suffer from precision loss.

## References

Sophocles J. Orfanidis, “Introduction To Signal Processing”,  
Prentice-Hall, 1996, ch. 11, “Digital Filter Design”

## cupyx.scipy.signal.iirnotch

`cupyx.scipy.signal.iirnotch(w0, Q, fs=2.0)`

Design second-order IIR notch digital filter.

A notch filter is a band-stop filter with a narrow bandwidth (high quality factor). It rejects a narrow frequency band and leaves the rest of the spectrum little changed.

### Parameters

- **w0** (*float*) – Frequency to remove from a signal. If *fs* is specified, this is in the same units as *fs*. By default, it is a normalized scalar that must satisfy  $0 < w0 < 1$ , with  $w0 = 1$  corresponding to half of the sampling frequency.
- **Q** (*float*) – Quality factor. Dimensionless parameter that characterizes notch filter -3 dB bandwidth *bw* relative to its center frequency,  $Q = w0/bw$ .
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

### Returns

**b, a** – Numerator (b) and denominator (a) polynomials of the IIR filter.

### Return type

*ndarray, ndarray*

See also:

`scipy.signal.iirnotch`

## References

Sophocles J. Orfanidis, “Introduction To Signal Processing”,  
Prentice-Hall, 1996

**cupyx.scipy.signal.iirpeak**

`cupyx.scipy.signal.iirpeak(w0, Q, fs=2.0)`

Design second-order IIR peak (resonant) digital filter.

A peak filter is a band-pass filter with a narrow bandwidth (high quality factor). It rejects components outside a narrow frequency band.

**Parameters**

- **w0** (*float*) – Frequency to be retained in a signal. If *fs* is specified, this is in the same units as *fs*. By default, it is a normalized scalar that must satisfy  $0 < w0 < 1$ , with  $w0 = 1$  corresponding to half of the sampling frequency.
- **Q** (*float*) – Quality factor. Dimensionless parameter that characterizes peak filter -3 dB bandwidth *bw* relative to its center frequency,  $Q = w0/bw$ .
- **fs** (*float*, *optional*) – The sampling frequency of the digital system.

**Returns**

**b, a** – Numerator (b) and denominator (a) polynomials of the IIR filter.

**Return type**

*ndarray, ndarray*

**See also:**

`scipy.signal.iirpeak`

**References**

Sophocles J. Orfanidis, “Introduction To Signal Processing”,  
Prentice-Hall, 1996

## Low-level filter design functions

<code>abcd_normalize([A, B, C, D])</code>	Check state-space matrices and ensure they are 2-D.
<code>band_stop_obj(wp, ind, passb, stopb, gpass, ...)</code>	Band Stop Objective Function for order minimization.
<code>buttap(N)</code>	Return (z,p,k) for analog prototype of Nth-order Butterworth filter.
<code>cheblap(N, rp)</code>	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
<code>cheb2ap(N, rs)</code>	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
<code>ellipap(N, rp, rs)</code>	Return (z,p,k) of Nth-order elliptic analog lowpass filter.
<code>lp2bp(b, a[, wo, bw])</code>	Transform a lowpass filter prototype to a bandpass filter.
<code>lp2bp_zpk(z, p, k[, wo, bw])</code>	Transform a lowpass filter prototype to a bandpass filter.
<code>lp2bs(b, a[, wo, bw])</code>	Transform a lowpass filter prototype to a bandstop filter.
<code>lp2bs_zpk(z, p, k[, wo, bw])</code>	Transform a lowpass filter prototype to a bandstop filter.
<code>lp2hp(b, a[, wo])</code>	Transform a lowpass filter prototype to a highpass filter.
<code>lp2hp_zpk(z, p, k[, wo])</code>	Transform a lowpass filter prototype to a highpass filter.
<code>lp2lp(b, a[, wo])</code>	Transform a lowpass filter prototype to a different frequency.
<code>lp2lp_zpk(z, p, k[, wo])</code>	Transform a lowpass filter prototype to a different frequency.
<code>normalize(b, a)</code>	Normalize numerator/denominator of a continuous-time transfer function.

### cupyx.scipy.signal.abcd\_normalize

`cupyx.scipy.signal.abcd_normalize(A=None, B=None, C=None, D=None)`

Check state-space matrices and ensure they are 2-D.

If enough information on the system is provided, that is, enough properly-shaped arrays are passed to the function, the missing ones are built from this information, ensuring the correct number of rows and columns. Otherwise a `ValueError` is raised.

#### Parameters

- **A** (*array\_like, optional*) – State-space matrices. All of them are `None` (missing) by default. See `ss2tf` for format.
- **B** (*array\_like, optional*) – State-space matrices. All of them are `None` (missing) by default. See `ss2tf` for format.
- **C** (*array\_like, optional*) – State-space matrices. All of them are `None` (missing) by default. See `ss2tf` for format.
- **D** (*array\_like, optional*) – State-space matrices. All of them are `None` (missing) by default. See `ss2tf` for format.

#### Returns

**A, B, C, D** – Properly shaped state-space matrices.

#### Return type

array

#### Raises

**ValueError** – If not enough information on the system was provided.

### cupyx.scipy.signal.band\_stop\_obj

`cupyx.scipy.signal.band_stop_obj(wp, ind, passb, stopb, gpass, gstop, type)`

Band Stop Objective Function for order minimization.

Returns the non-integer order for an analog band stop filter.

#### Parameters

- **wp** (*scalar*) – Edge of passband *passb*.
- **ind** (*int*, {0, 1}) – Index specifying which *passb* edge to vary (0 or 1).
- **passb** (*ndarray*) – Two element sequence of fixed passband edges.
- **stopb** (*ndarray*) – Two element sequence of fixed stopband edges.
- **gstop** (*float*) – Amount of attenuation in stopband in dB.
- **gpass** (*float*) – Amount of ripple in the passband in dB.
- **type** ({'butter', 'cheby', 'ellip'}) – Type of filter.

#### Returns

**n** – Filter order (possibly non-integer).

#### Return type

scalar

See also:

`scipy.signal.band_stop_obj`

### cupyx.scipy.signal.buttap

`cupyx.scipy.signal.buttap(N)`

Return (z,p,k) for analog prototype of Nth-order Butterworth filter.

The filter will have an angular (e.g., rad/s) cutoff frequency of 1.

See also:

#### *butter*

Filter design function using this prototype

`scipy.signal.buttap`

### cupyx.scipy.signal.cheb1ap

`cupyx.scipy.signal.cheb1ap(N, rp)`

Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.

The returned filter prototype has *rp* decibels of ripple in the passband.

The filter's angular (e.g. rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first drops below  $-rp$ .

See also:

***cheby1***

Filter design function using this prototype

**cupyx.scipy.signal.cheb2ap**

`cupyx.scipy.signal.cheb2ap(N, rs)`

Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.

The returned filter prototype has *rs* decibels of ripple in the stopband.

The filter's angular (e.g. rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first reaches -*rs*.

**See also:**

***cheby2***

Filter design function using this prototype

**cupyx.scipy.signal.ellipap**

`cupyx.scipy.signal.ellipap(N, rp, rs)`

Return (z,p,k) of Nth-order elliptic analog lowpass filter.

The filter is a normalized prototype that has *rp* decibels of ripple in the passband and a stopband *rs* decibels down.

The filter's angular (e.g., rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first drops below -*rp*.

**See also:**

***ellip***

Filter design function using this prototype

`scipy.signal.elliap`

**cupyx.scipy.signal.lp2bp**

`cupyx.scipy.signal.lp2bp(b, a, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandpass filter.

Return an analog band-pass filter with center frequency *wo* and bandwidth *bw* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

**Parameters**

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **wo** (*float*) – Desired passband center, as angular frequency (e.g., rad/s). Defaults to no change.
- **bw** (*float*) – Desired passband width, as angular frequency (e.g., rad/s). Defaults to 1.

**Returns**

- **b** (*array\_like*) – Numerator polynomial coefficients of the transformed band-pass filter.
- **a** (*array\_like*) – Denominator polynomial coefficients of the transformed band-pass filter.

See also:

[`lp2lp`](#), [`lp2hp`](#), [`lp2bs`](#), [`bilinear`](#), [`lp2bp\_zpk`](#), [`scipy.signal.lp2bp`](#)

## Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{s^2 + \omega_0^2}{s \cdot BW}$$

This is the “wideband” transformation, producing a passband with geometric (log frequency) symmetry about  $\omega_0$ .

## `cupyx.scipy.signal.lp2bp_zpk`

`cupyx.scipy.signal.lp2bp_zpk(z, p, k, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandpass filter.

Return an analog band-pass filter with center frequency  $\omega_0$  and bandwidth  $BW$  from an analog low-pass filter prototype with unity cutoff frequency, using zeros, poles, and gain (‘zpk’) representation.

### Parameters

- **z** (*array\_like*) – Zeros of the analog filter transfer function.
- **p** (*array\_like*) – Poles of the analog filter transfer function.
- **k** (*float*) – System gain of the analog filter transfer function.
- **wo** (*float*) – Desired passband center, as angular frequency (e.g., rad/s). Defaults to no change.
- **bw** (*float*) – Desired passband width, as angular frequency (e.g., rad/s). Defaults to 1.

### Returns

- **z** (*ndarray*) – Zeros of the transformed band-pass filter transfer function.
- **p** (*ndarray*) – Poles of the transformed band-pass filter transfer function.
- **k** (*float*) – System gain of the transformed band-pass filter.

See also:

[`lp2lp\_zpk`](#), [`lp2hp\_zpk`](#), [`lp2bs\_zpk`](#), [`bilinear`](#), [`lp2bp`](#), [`scipy.signal.lp2bp\_zpk`](#)

## Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{s^2 + \omega_0^2}{s \cdot BW}$$

This is the “wideband” transformation, producing a passband with geometric (log frequency) symmetry about  $\omega_0$ .



## cupyx.scipy.signal.lp2bs

`cupyx.scipy.signal.lp2bs(b, a, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandstop filter.

Return an analog band-stop filter with center frequency *wo* and bandwidth *bw* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

### Parameters

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **wo** (*float*) – Desired stopband center, as angular frequency (e.g., rad/s). Defaults to no change.
- **bw** (*float*) – Desired stopband width, as angular frequency (e.g., rad/s). Defaults to 1.

### Returns

- **b** (*array\_like*) – Numerator polynomial coefficients of the transformed band-stop filter.
- **a** (*array\_like*) – Denominator polynomial coefficients of the transformed band-stop filter.

See also:

[lp2lp](#), [lp2hp](#), [lp2bp](#), [bilinear](#), [lp2bs\\_zpk](#), [scipy.signal.lp2bs](#)

### Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{s \cdot BW}{s^2 + \omega_0^2}$$

This is the “wideband” transformation, producing a stopband with geometric (log frequency) symmetry about *wo*.

## cupyx.scipy.signal.lp2bs\_zpk

`cupyx.scipy.signal.lp2bs_zpk(z, p, k, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandstop filter.

Return an analog band-stop filter with center frequency *wo* and stopband width *bw* from an analog low-pass filter prototype with unity cutoff frequency, using zeros, poles, and gain ('zpk') representation.

### Parameters

- **z** (*array\_like*) – Zeros of the analog filter transfer function.
- **p** (*array\_like*) – Poles of the analog filter transfer function.
- **k** (*float*) – System gain of the analog filter transfer function.
- **wo** (*float*) – Desired stopband center, as angular frequency (e.g., rad/s). Defaults to no change.
- **bw** (*float*) – Desired stopband width, as angular frequency (e.g., rad/s). Defaults to 1.

### Returns

- **z** (*ndarray*) – Zeros of the transformed band-stop filter transfer function.
- **p** (*ndarray*) – Poles of the transformed band-stop filter transfer function.
- **k** (*float*) – System gain of the transformed band-stop filter.

See also:

[lp2lp\\_zpk](#), [lp2hp\\_zpk](#), [lp2bp\\_zpk](#), [bilinear](#), [lp2bs](#), [scipy.signal.lp2bs\\_zpk](#)

## Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{s \cdot BW}{s^2 + \omega_0^2}$$

This is the “wideband” transformation, producing a stopband with geometric (log frequency) symmetry about  $\omega_0$ .

## cupyx.scipy.signal.lp2hp

`cupyx.scipy.signal.lp2hp(b, a, wo=1.0)`

Transform a lowpass filter prototype to a highpass filter.

Return an analog high-pass filter with cutoff frequency  $\omega_0$  from an analog low-pass filter prototype with unity cutoff frequency, in transfer function (‘ba’) representation.

### Parameters

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **wo** (*float*) – Desired cutoff, as angular frequency (e.g., rad/s). Defaults to no change.

### Returns

- **b** (*array\_like*) – Numerator polynomial coefficients of the transformed high-pass filter.
- **a** (*array\_like*) – Denominator polynomial coefficients of the transformed high-pass filter.

See also:

[lp2lp](#), [lp2bp](#), [lp2bs](#), [bilinear](#), [lp2hp\\_zpk](#), [scipy.signal.lp2hp](#)

## Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{\omega_0}{s}$$

This maintains symmetry of the lowpass and highpass responses on a logarithmic scale.

**cupyx.scipy.signal.lp2hp\_zpk**

`cupyx.scipy.signal.lp2hp_zpk(z, p, k, wo=1.0)`

Transform a lowpass filter prototype to a highpass filter.

Return an analog high-pass filter with cutoff frequency *wo* from an analog low-pass filter prototype with unity cutoff frequency, using zeros, poles, and gain ('zpk') representation.

**Parameters**

- **z** (*array\_like*) – Zeros of the analog filter transfer function.
- **p** (*array\_like*) – Poles of the analog filter transfer function.
- **k** (*float*) – System gain of the analog filter transfer function.
- **wo** (*float*) – Desired cutoff, as angular frequency (e.g., rad/s). Defaults to no change.

**Returns**

- **z** (*ndarray*) – Zeros of the transformed high-pass filter transfer function.
- **p** (*ndarray*) – Poles of the transformed high-pass filter transfer function.
- **k** (*float*) – System gain of the transformed high-pass filter.

See also:

[\*lp2lp\\_zpk\*](#), [\*lp2bp\\_zpk\*](#), [\*lp2bs\\_zpk\*](#), [\*bilinear\*](#), [\*lp2hp\*](#), [`scipy.signal.lp2hp\_zpk`](#)

**Notes**

This is derived from the s-plane substitution

$$s \rightarrow \frac{\omega_0}{s}$$

This maintains symmetry of the lowpass and highpass responses on a logarithmic scale.

**cupyx.scipy.signal.lp2lp**

`cupyx.scipy.signal.lp2lp(b, a, wo=1.0)`

Transform a lowpass filter prototype to a different frequency.

Return an analog low-pass filter with cutoff frequency *wo* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

**Parameters**

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **wo** (*float*) – Desired cutoff, as angular frequency (e.g. rad/s). Defaults to no change.

**Returns**

- **b** (*array\_like*) – Numerator polynomial coefficients of the transformed low-pass filter.
- **a** (*array\_like*) – Denominator polynomial coefficients of the transformed low-pass filter.

See also:

[\*lp2hp\*](#), [\*lp2bp\*](#), [\*lp2bs\*](#), [\*bilinear\*](#), [\*lp2lp\\_zpk\*](#), [`scipy.signal.lp2lp`](#)

## Notes

This is derived from the s-plane substitution

$$s \rightarrow \frac{s}{\omega_0}$$

## cupyx.scipy.signal.lp2lp\_zpk

`cupyx.scipy.signal.lp2lp_zpk(z, p, k, wo=1.0)`

Transform a lowpass filter prototype to a different frequency.

Return an analog low-pass filter with cutoff frequency *wo* from an analog low-pass filter prototype with unity cutoff frequency, using zeros, poles, and gain ('zpk') representation.

### Parameters

- **z** (*array\_like*) – Zeros of the analog filter transfer function.
- **p** (*array\_like*) – Poles of the analog filter transfer function.
- **k** (*float*) – System gain of the analog filter transfer function.
- **wo** (*float*) – Desired cutoff, as angular frequency (e.g., rad/s). Defaults to no change.

### Returns

- **z** (*ndarray*) – Zeros of the transformed low-pass filter transfer function.
- **p** (*ndarray*) – Poles of the transformed low-pass filter transfer function.
- **k** (*float*) – System gain of the transformed low-pass filter.

See also:

[\*lp2hp\\_zpk\*](#), [\*lp2bp\\_zpk\*](#), [\*lp2bs\\_zpk\*](#), [\*bilinear\*](#), [\*lp2lp\*](#), [`scipy.signal.lp2lp\_zpk`](#)

## cupyx.scipy.signal.normalize

`cupyx.scipy.signal.normalize(b, a)`

Normalize numerator/denominator of a continuous-time transfer function.

If values of *b* are too close to 0, they are removed. In that case, a `BadCoefficients` warning is emitted.

### Parameters

- **b** (*array\_like*) – Numerator of the transfer function. Can be a 2-D array to normalize multiple transfer functions.
- **a** (*array\_like*) – Denominator of the transfer function. At most 1-D.

### Returns

- **num** (*array*) – The numerator of the normalized transfer function. At least a 1-D array. A 2-D array if the input *num* is a 2-D array.
- **den** (*1-D array*) – The denominator of the normalized transfer function.

## Notes

Coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.,  $s^2 + 3s + 5$  would be represented as `[1, 3, 5]`).

See also:

`scipy.signal.normalize`

## LTI representations

<code>zpk2tf(z, p, k)</code>	Return polynomial transfer function representation from zeros and poles
<code>zpk2sos(z, p, k[, pairing, analog])</code>	Return second-order sections from zeros, poles, and gain of a system
<code>zpk2ss(z, p, k)</code>	Zero-pole-gain representation to state-space representation
<code>tf2zpk(b, a)</code>	Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.
<code>tf2sos(b, a[, pairing, analog])</code>	Return second-order sections from transfer function representation
<code>tf2ss(num, den)</code>	Transfer function to state-space representation.
<code>ss2tf(A, B, C, D[, input])</code>	State-space to transfer function.
<code>ss2zpk(A, B, C, D[, input])</code>	State-space representation to zero-pole-gain representation.
<code>sos2tf(sos)</code>	Return a single transfer function from a series of second-order sections
<code>sos2zpk(sos)</code>	Return zeros, poles, and gain of a series of second-order sections
<code>cont2discrete(system, dt[, method, alpha])</code>	Transform a continuous to a discrete state-space system.
<code>place_poles(A, B, poles[, method, rtol, maxiter])</code>	Compute K such that eigenvalues (A - dot(B, K))=poles.

## cupyx.scipy.signal.zpk2tf

`cupyx.scipy.signal.zpk2tf(z, p, k)`

Return polynomial transfer function representation from zeros and poles

### Parameters

- **z** (*array\_like*) – Zeros of the transfer function.
- **p** (*array\_like*) – Poles of the transfer function.
- **k** (*float*) – System gain.

### Returns

- **b** (*ndarray*) – Numerator polynomial coefficients.
- **a** (*ndarray*) – Denominator polynomial coefficients.

See also:

`scipy.signal.zpk2tf`

### cupyx.scipy.signal.zpk2sos

`cupyx.scipy.signal.zpk2sos(z, p, k, pairing=None, *, analog=False)`

Return second-order sections from zeros, poles, and gain of a system

#### Parameters

- **z** (*array\_like*) – Zeros of the transfer function.
- **p** (*array\_like*) – Poles of the transfer function.
- **k** (*float*) – System gain.
- **pairing** (*{None, 'nearest', 'keep\_odd', 'minimal'}, optional*) – The method to use to combine pairs of poles and zeros into sections. If *analog* is *False* and *pairing* is *None*, *pairing* is set to 'nearest'; if *analog* is *True*, *pairing* must be 'minimal', and is set to that if it is *None*.
- **analog** (*bool, optional*) – If *True*, system is analog, otherwise discrete.

#### Returns

**sos** – Array of second-order filter coefficients, with shape (*n\_sections*, 6). See *sosfilt* for the SOS filter format specification.

#### Return type

*ndarray*

#### See also:

*sosfilt*, *scipy.signal.zpk2sos*

### cupyx.scipy.signal.zpk2ss

`cupyx.scipy.signal.zpk2ss(z, p, k)`

Zero-pole-gain representation to state-space representation

#### Parameters

- **z** (*sequence*) – Zeros and poles.
- **p** (*sequence*) – Zeros and poles.
- **k** (*float*) – System gain.

#### Returns

**A, B, C, D** – State space representation of the system, in controller canonical form.

#### Return type

*ndarray*

#### See also:

*scipy.signal.zpk2ss*

**cupyx.scipy.signal.tf2zpk**`cupyx.scipy.signal.tf2zpk(b, a)`

Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.

**Parameters**

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.

**Returns**

- **z** (*ndarray*) – Zeros of the transfer function.
- **p** (*ndarray*) – Poles of the transfer function.
- **k** (*float*) – System gain.

**Warning:** This function may synchronize the device.

**See also:**

`scipy.signal.tf2zpk`

**Notes**

If some values of  $b$  are too close to 0, they are removed. In that case, a `BadCoefficients` warning is emitted.

The  $b$  and  $a$  arrays are interpreted as coefficients for positive, descending powers of the transfer function variable. So the inputs  $b = [b_0, b_1, \dots, b_M]$  and  $a = [a_0, a_1, \dots, a_N]$  can represent an analog filter of the form:

$$H(s) = \frac{b_0 s^M + b_1 s^{(M-1)} + \dots + b_M}{a_0 s^N + a_1 s^{(N-1)} + \dots + a_N}$$

or a discrete-time filter of the form:

$$H(z) = \frac{b_0 z^M + b_1 z^{(M-1)} + \dots + b_M}{a_0 z^N + a_1 z^{(N-1)} + \dots + a_N}$$

This “positive powers” form is found more commonly in controls engineering. If  $M$  and  $N$  are equal (which is true for all filters generated by the bilinear transform), then this happens to be equivalent to the “negative powers” discrete-time form preferred in DSP:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \dots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \dots + a_N z^{-N}}$$

Although this is true for common filters, remember that this is not true in the general case. If  $M$  and  $N$  are not equal, the discrete-time transfer function coefficients must first be converted to the “positive powers” form before finding the poles and zeros.

### cupyx.scipy.signal.tf2sos

`cupyx.scipy.signal.tf2sos(b, a, pairing=None, *, analog=False)`

Return second-order sections from transfer function representation

#### Parameters

- **b** (*array\_like*) – Numerator polynomial coefficients.
- **a** (*array\_like*) – Denominator polynomial coefficients.
- **pairing** (*{None, 'nearest', 'keep\_odd', 'minimal'}, optional*) – The method to use to combine pairs of poles and zeros into sections. See *zpk2sos* for information and restrictions on *pairing* and *analog* arguments.
- **analog** (*bool, optional*) – If True, system is analog, otherwise discrete.

#### Returns

**sos** – Array of second-order filter coefficients, with shape (n\_sections, 6). See *sosfilt* for the SOS filter format specification.

#### Return type

*ndarray*

See also:

`scipy.signal.tf2sos`

#### Notes

It is generally discouraged to convert from TF to SOS format, since doing so usually will not improve numerical precision errors. Instead, consider designing filters in ZPK format and converting directly to SOS. TF is converted to SOS by first converting to ZPK format, then converting ZPK to SOS.

### cupyx.scipy.signal.tf2ss

`cupyx.scipy.signal.tf2ss(num, den)`

Transfer function to state-space representation.

#### Parameters

- **num** (*array\_like*) – Sequences representing the coefficients of the numerator and denominator polynomials, in order of descending degree. The denominator needs to be at least as long as the numerator.
- **den** (*array\_like*) – Sequences representing the coefficients of the numerator and denominator polynomials, in order of descending degree. The denominator needs to be at least as long as the numerator.

#### Returns

**A, B, C, D** – State space representation of the system, in controller canonical form.

#### Return type

*ndarray*

See also:

`scipy.signal.tf2ss`



**cupyx.scipy.signal.ss2tf**

`cupyx.scipy.signal.ss2tf(A, B, C, D, input=0)`

State-space to transfer function.

A, B, C, D defines a linear state-space system with  $p$  inputs,  $q$  outputs, and  $n$  state variables.

**Parameters**

- **A** (*array\_like*) – State (or system) matrix of shape  $(n, n)$
- **B** (*array\_like*) – Input matrix of shape  $(n, p)$
- **C** (*array\_like*) – Output matrix of shape  $(q, n)$
- **D** (*array\_like*) – Feedthrough (or feedforward) matrix of shape  $(q, p)$
- **input** (*int*, *optional*) – For multiple-input systems, the index of the input to use.

**Returns**

- **num** (*2-D ndarray*) – Numerator(s) of the resulting transfer function(s). *num* has one row for each of the system's outputs. Each row is a sequence representation of the numerator polynomial.
- **den** (*1-D ndarray*) – Denominator of the resulting transfer function(s). *den* is a sequence representation of the denominator polynomial.

**Warning:** This function may synchronize the device.

See also:

`scipy.signal.ss2tf`

**cupyx.scipy.signal.ss2zpk**

`cupyx.scipy.signal.ss2zpk(A, B, C, D, input=0)`

State-space representation to zero-pole-gain representation.

A, B, C, D defines a linear state-space system with  $p$  inputs,  $q$  outputs, and  $n$  state variables.

**Parameters**

- **A** (*array\_like*) – State (or system) matrix of shape  $(n, n)$
- **B** (*array\_like*) – Input matrix of shape  $(n, p)$
- **C** (*array\_like*) – Output matrix of shape  $(q, n)$
- **D** (*array\_like*) – Feedthrough (or feedforward) matrix of shape  $(q, p)$
- **input** (*int*, *optional*) – For multiple-input systems, the index of the input to use.

**Returns**

- **z, p** (*sequence*) – Zeros and poles.
- **k** (*float*) – System gain.

See also:

`scipy.signal.ss2zpk`

### cupyx.scipy.signal.sos2tf

cupyx.scipy.signal.**sos2tf**(*sos*)

Return a single transfer function from a series of second-order sections

**Parameters**

**sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (n\_sections, 6). See *sosfilt* for the SOS filter format specification.

**Returns**

- **b** (*ndarray*) – Numerator polynomial coefficients.
- **a** (*ndarray*) – Denominator polynomial coefficients.

See also:

`scipy.signal.sos2tf`

### cupyx.scipy.signal.sos2zpk

cupyx.scipy.signal.**sos2zpk**(*sos*)

Return zeros, poles, and gain of a series of second-order sections

**Parameters**

**sos** (*array\_like*) – Array of second-order filter coefficients, must have shape (n\_sections, 6). See *sosfilt* for the SOS filter format specification.

**Returns**

- **z** (*ndarray*) – Zeros of the transfer function.
- **p** (*ndarray*) – Poles of the transfer function.
- **k** (*float*) – System gain.

#### Notes

The number of zeros and poles returned will be `n_sections * 2` even if some of these are (effectively) zero.

See also:

`scipy.signal.sos2zpk`

### cupyx.scipy.signal.cont2discrete

cupyx.scipy.signal.**cont2discrete**(*system*, *dt*, *method*='zoh', *alpha*=None)

Transform a continuous to a discrete state-space system.

**Parameters**

- **system** (a tuple describing the system or an instance of *lti*) – The following gives the number of elements in the tuple and the interpretation:
  - 1: (instance of *lti*)
  - 2: (num, den)
  - 3: (zeros, poles, gain)

- 4: (A, B, C, D)
- **dt** (*float*) – The discretization time step.
- **method** (*str*, *optional*) – Which method to use:
  - gbt: generalized bilinear transformation
  - bilinear: Tustin’s approximation (“gbt” with alpha=0.5)
  - euler: Euler (or forward differencing) method (“gbt” with alpha=0)
  - backward\_diff: Backwards differencing (“gbt” with alpha=1.0)
  - zoh: zero-order hold (default)
  - foh: first-order hold (*versionadded: 1.3.0*)
  - impulse: equivalent impulse response (*versionadded: 1.3.0*)
- **alpha** (*float within [0, 1]*, *optional*) – The generalized bilinear transformation weighting parameter, which should only be specified with method=”gbt”, and is ignored otherwise

**Returns**

**sysd** – Based on the input type, the output will be of the form

- (num, den, dt) for transfer function input
- (zeros, poles, gain, dt) for zeros-poles-gain input
- (A, B, C, D, dt) for state-space system input

**Return type**

tuple containing the discrete system

**Notes**

By default, the routine uses a Zero-Order Hold (zoh) method to perform the transformation. Alternatively, a generalized bilinear transformation may be used, which includes the common Tustin’s bilinear approximation, an Euler’s method technique, or a backwards differencing technique.

**See also:**

`scipy.signal.cont2discrete`

**cupyx.scipy.signal.place\_poles**

`cupyx.scipy.signal.place_poles(A, B, poles, method='YT', rtol=0.001, maxiter=30)`

Compute K such that eigenvalues (A - dot(B, K))=poles.

K is the gain matrix such as the plant described by the linear system  $AX+BU$  will have its closed-loop poles, i.e the eigenvalues  $A - B*K$ , as close as possible to those asked for in poles.

SISO, MISO and MIMO systems are supported.

**Parameters**

- **A** (*ndarray*) – State-space representation of linear system  $AX + BU$ .
- **B** (*ndarray*) – State-space representation of linear system  $AX + BU$ .

- **poles** (*array\_like*) – Desired real poles and/or complex conjugates poles. Complex poles are only supported with `method="YT"` (default).
- **method** (`{'YT', 'KNV0'}`, *optional*) – Which method to choose to find the gain matrix K. One of:
  - 'YT': Yang Tits
  - 'KNV0': Kautsky, Nichols, Van Dooren update method 0See References and Notes for details on the algorithms.
- **rtol** (*float*, *optional*) – After each iteration the determinant of the eigenvectors of  $A - B*K$  is compared to its previous value, when the relative error between these two values becomes lower than *rtol* the algorithm stops. Default is 1e-3.
- **maxiter** (*int*, *optional*) – Maximum number of iterations to compute the gain matrix. Default is 30.

### Returns

**full\_state\_feedback** –

**full\_state\_feedback** is composed of:

**gain\_matrix**

[1-D ndarray] The closed loop matrix K such as the eigenvalues of  $A-BK$  are as close as possible to the requested poles.

**computed\_poles**

[1-D ndarray] The poles corresponding to  $A-BK$  sorted as first the real poles in increasing order, then the complex conjugates in lexicographic order.

**requested\_poles**

[1-D ndarray] The poles the algorithm was asked to place sorted as above, they may differ from what was achieved.

**X**

[2-D ndarray] The transfer matrix such as  $X * \text{diag}(\text{poles}) = (A - B*K)*X$  (see Notes)

**rtol**

[float] The relative tolerance achieved on  $\det(X)$  (see Notes). *rtol* will be NaN if it is possible to solve the system  $\text{diag}(\text{poles}) = (A - B*K)$ , or 0 when the optimization algorithms can't do anything i.e when  $B.\text{shape}[1] == 1$ .

**nb\_iter**

[int] The number of iterations performed before converging. *nb\_iter* will be NaN if it is possible to solve the system  $\text{diag}(\text{poles}) = (A - B*K)$ , or 0 when the optimization algorithms can't do anything i.e when  $B.\text{shape}[1] == 1$ .

### Return type

Bunch object

## Notes

The Tits and Yang (YT),<sup>2</sup> paper is an update of the original Kautsky et al. (KNV) paper<sup>1</sup>. KNV relies on rank-1 updates to find the transfer matrix  $X$  such that  $X * \text{diag}(\text{poles}) = (A - B*K)*X$ , whereas YT uses rank-2 updates. This yields on average more robust solutions (see [Page 537, 2](#) pp 21-22), furthermore the YT algorithm supports complex poles whereas KNV does not in its original version. Only update method 0 proposed by KNV has been implemented here, hence the name 'KNV0'.

KNV extended to complex poles is used in Matlab's `place` function, YT is distributed under a non-free licence by Slicot under the name `robpole`. It is unclear and undocumented how KNV0 has been extended to complex poles (Tits and Yang claim on page 14 of their paper that their method can not be used to extend KNV to complex poles), therefore only YT supports them in this implementation.

As the solution to the problem of pole placement is not unique for MIMO systems, both methods start with a tentative transfer matrix which is altered in various way to increase its determinant. Both methods have been proven to converge to a stable solution, however depending on the way the initial transfer matrix is chosen they will converge to different solutions and therefore there is absolutely no guarantee that using 'KNV0' will yield results similar to Matlab's or any other implementation of these algorithms.

Using the default method 'YT' should be fine in most cases; 'KNV0' is only provided because it is needed by 'YT' in some specific cases. Furthermore 'YT' gives on average more robust results than 'KNV0' when `abs(det(X))` is used as a robustness indicator.

<sup>2</sup> is available as a technical report on the following URL: <https://hdl.handle.net/1903/5598>

See also:

`scipy.signal.place_poles`

## References

### Continuous-time linear systems

<code>lti(*system)</code>	Continuous-time linear time invariant system base class.
<code>StateSpace(*system, **kwargs)</code>	Linear Time Invariant system in state-space form.
<code>TransferFunction(*system, **kwargs)</code>	Linear Time Invariant system class in transfer function form.
<code>ZerosPolesGain(*system, **kwargs)</code>	Linear Time Invariant system class in zeros, poles, gain form.
<code>lsim(system, U, T[, X0, interp])</code>	Simulate output of a continuous-time linear system.
<code>impulse(system[, X0, T, N])</code>	Impulse response of continuous-time system.
<code>step(system[, X0, T, N])</code>	Step response of continuous-time system.
<code>freqresp(system[, w, n])</code>	Calculate the frequency response of a continuous-time system.
<code>bode(system[, w, n])</code>	Calculate Bode magnitude and phase data of a continuous-time system.

<sup>2</sup> A.L. Tits and Y. Yang, "Globally convergent algorithms for robust pole assignment by state feedback", IEEE Transactions on Automatic Control, Vol. 41, pp. 1432-1452, 1996.

<sup>1</sup> J. Kautsky, N.K. Nichols and P. van Dooren, "Robust pole assignment in linear state feedback", International Journal of Control, Vol. 41 pp. 1129-1155, 1985.

## cupyx.scipy.signal.lti

**class** cupyx.scipy.signal.lti(\*system)

Continuous-time linear time invariant system base class.

### Parameters

**\*system** (*arguments*) – The *lti* class can be instantiated with either 2, 3 or 4 arguments. The following gives the number of arguments and the corresponding continuous-time subclass that is created:

- 2: *TransferFunction*: (numerator, denominator)
- 3: *ZerosPolesGain*: (zeros, poles, gain)
- 4: *StateSpace*: (A, B, C, D)

Each argument can be an array or a sequence.

See also:

[scipy.signal.lti](#), [ZerosPolesGain](#), [StateSpace](#), [TransferFunction](#), [dlti](#)

### Notes

*lti* instances do not exist directly. Instead, *lti* creates an instance of one of its subclasses: *StateSpace*, *TransferFunction* or *ZerosPolesGain*.

If (numerator, denominator) is passed in for *\*system*, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.,  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

Changing the value of properties that are not directly part of the current system representation (such as the *zeros* of a *StateSpace* system) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the zeros, poles or gain.

### Methods

**bode**(*w=None, n=100*)

Calculate Bode magnitude and phase data of a continuous-time system.

Returns a 3-tuple containing arrays of frequencies [rad/s], magnitude [dB] and phase [deg]. See *bode* for details.

**freqresp**(*w=None, n=10000*)

Calculate the frequency response of a continuous-time system.

Returns a 2-tuple containing arrays of frequencies [rad/s] and complex magnitude. See *freqresp* for details.

**impulse**(*X0=None, T=None, N=None*)

Return the impulse response of a continuous-time system. See *impulse* for details.

**output**(*U, T, X0=None*)

Return the response of a continuous-time system to input *U*. See *lsim* for details.

**step**(*X0=None, T=None, N=None*)

Return the step response of a continuous-time system. See *step* for details.

**to\_discrete**(*dt, method='zoh', alpha=None*)

Return a discretized version of the current system.

Parameters: See *cont2discrete* for details.

**Returns**

**sys**

**Return type**

instance of *dlti*

**\_\_eq\_\_**(*value, /*)

Return self==value.

**\_\_ne\_\_**(*value, /*)

Return self!=value.

**\_\_lt\_\_**(*value, /*)

Return self<value.

**\_\_le\_\_**(*value, /*)

Return self<=value.

**\_\_gt\_\_**(*value, /*)

Return self>value.

**\_\_ge\_\_**(*value, /*)

Return self>=value.

## Attributes

**dt**

Return the sampling time of the system, *None* for *lti* systems.

**poles**

Poles of the system.

**zeros**

Zeros of the system.

## cupyx.scipy.signal.StateSpace

**class** cupyx.scipy.signal.**StateSpace**(\**system*, \*\**kwargs*)

Linear Time Invariant system in state-space form.

Represents the system as the continuous-time, first order differential equation  $\dot{x} = Ax + Bu$  or the discrete-time difference equation  $x[k + 1] = Ax[k] + Bu[k]$ . *StateSpace* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

**Parameters**

- **\*system** (*arguments*) – The *StateSpace* class can be instantiated with 1 or 4 arguments. The following gives the number of input arguments and their interpretation:
  - 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
  - 4: array\_like: (A, B, C, D)

- **dt** (*float*, *optional*) – Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, `dt=0.1`.

See also:

`scipy.signal.StateSpace`, `TransferFunction`, `ZerosPolesGain`, `lti`, `dlti`, `ss2zpk`, `ss2tf`, `zpk2sos`

## Notes

Changing the value of properties that are not part of the *StateSpace* system representation (such as *zeros* or *poles*) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the zeros, poles or gain.

## Methods

**to\_ss()**

Return a copy of the current *StateSpace* system.

**Returns**

**sys** – The current system (copy)

**Return type**

instance of *StateSpace*

**to\_tf(\*\*kwargs)**

Convert system representation to *TransferFunction*.

**Parameters**

**kwargs** (*dict*, *optional*) – Additional keywords passed to *ss2zpk*

**Returns**

**sys** – Transfer function of the current system

**Return type**

instance of *TransferFunction*

**to\_zpk(\*\*kwargs)**

Convert system representation to *ZerosPolesGain*.

**Parameters**

**kwargs** (*dict*, *optional*) – Additional keywords passed to *ss2zpk*

**Returns**

**sys** – Zeros, poles, gain representation of the current system

**Return type**

instance of *ZerosPolesGain*

**\_\_eq\_\_(value, /)**

Return self==value.

**\_\_ne\_\_(value, /)**

Return self!=value.

**\_\_lt\_\_(value, /)**

Return self<value.

**\_\_le\_\_(value, /)**

Return self<=value.



`__gt__(value, /)`  
Return self>value.

`__ge__(value, /)`  
Return self>=value.

### Attributes

**A**  
State matrix of the *StateSpace* system.

**B**  
Input matrix of the *StateSpace* system.

**C**  
Output matrix of the *StateSpace* system.

**D**  
Feedthrough matrix of the *StateSpace* system.

**dt**  
Return the sampling time of the system, *None* for *lti* systems.

**poles**  
Poles of the system.

**zeros**  
Zeros of the system.

## cupyx.scipy.signal.TransferFunction

**class** cupyx.scipy.signal.**TransferFunction**(\*system, \*\*kwargs)

Linear Time Invariant system class in transfer function form.

Represents the system as the continuous-time transfer function  $H(s) = \sum_{i=0}^N b[N-i]s^i / \sum_{j=0}^M a[M-j]s^j$  or the discrete-time transfer function  $H(z) = \sum_{i=0}^N b[N-i]z^i / \sum_{j=0}^M a[M-j]z^j$ , where  $b$  are elements of the numerator *num*,  $a$  are elements of the denominator *den*, and  $N == \text{len}(b) - 1$ ,  $M == \text{len}(a) - 1$ . *TransferFunction* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

### Parameters

- **\*system** (*arguments*) – The *TransferFunction* class can be instantiated with 1 or 2 arguments. The following gives the number of input arguments and their interpretation:
  - 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
  - 2: array\_like: (numerator, denominator)
- **dt** (*float*, *optional*) – Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, `dt=0.1`.

See also:

`scipy.signal.TransferFunction`, `ZerosPolesGain`, `StateSpace`, `lti`, `dlti`, `tf2ss`, `tf2zpk`, `tf2sos`

## Notes

Changing the value of properties that are not part of the *TransferFunction* system representation (such as the *A*, *B*, *C*, *D* state-space matrices) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_ss()` before accessing/changing the *A*, *B*, *C*, *D* system matrices.

If (numerator, denominator) is passed in for *\*system*, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  or  $z^2 + 3z + 5$  would be represented as `[1, 3, 5]`)

## Methods

### `to_ss()`

Convert system representation to *StateSpace*.

#### Returns

**sys** – State space model of the current system

#### Return type

instance of *StateSpace*

### `to_tf()`

Return a copy of the current *TransferFunction* system.

#### Returns

**sys** – The current system (copy)

#### Return type

instance of *TransferFunction*

### `to_zpk()`

Convert system representation to *ZerosPolesGain*.

#### Returns

**sys** – Zeros, poles, gain representation of the current system

#### Return type

instance of *ZerosPolesGain*

### `__eq__(value, /)`

Return `self==value`.

### `__ne__(value, /)`

Return `self!=value`.

### `__lt__(value, /)`

Return `self<value`.

### `__le__(value, /)`

Return `self<=value`.

### `__gt__(value, /)`

Return `self>value`.

### `__ge__(value, /)`

Return `self>=value`.

## Attributes

### den

Denominator of the *TransferFunction* system.

### dt

Return the sampling time of the system, *None* for *lti* systems.

### num

Numerator of the *TransferFunction* system.

### poles

Poles of the system.

### zeros

Zeros of the system.

## cupyx.scipy.signal.ZerosPolesGain

**class** cupyx.scipy.signal.ZerosPolesGain(\*system, \*\*kwargs)

Linear Time Invariant system class in zeros, poles, gain form.

Represents the system as the continuous- or discrete-time transfer function  $H(s) = k \prod_i (s - z[i]) / \prod_j (s - p[j])$ , where  $k$  is the *gain*,  $z$  are the *zeros* and  $p$  are the *poles*. *ZerosPolesGain* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

### Parameters

- **\*system** (*arguments*) – The *ZerosPolesGain* class can be instantiated with 1 or 3 arguments. The following gives the number of input arguments and their interpretation:
  - 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
  - 3: *array\_like*: (zeros, poles, gain)
- **dt** (*float*, *optional*) – Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, `dt=0.1`.

### See also:

[scipy.signal.ZerosPolesGain](#), [TransferFunction](#), [StateSpace](#), [lti](#), [dlti](#), [zpk2ss](#), [zpk2tf](#), [zpk2sos](#)

## Notes

Changing the value of properties that are not part of the *ZerosPolesGain* system representation (such as the  $A$ ,  $B$ ,  $C$ ,  $D$  state-space matrices) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_ss()` before accessing/changing the  $A$ ,  $B$ ,  $C$ ,  $D$  system matrices.

## Methods

### `to_ss()`

Convert system representation to *StateSpace*.

#### Returns

**sys** – State space model of the current system

#### Return type

instance of *StateSpace*

### `to_tf()`

Convert system representation to *TransferFunction*.

#### Returns

**sys** – Transfer function of the current system

#### Return type

instance of *TransferFunction*

### `to_zpk()`

Return a copy of the current ‘ZerosPolesGain’ system.

#### Returns

**sys** – The current system (copy)

#### Return type

instance of *ZerosPolesGain*

### `__eq__(value, /)`

Return self==value.

### `__ne__(value, /)`

Return self!=value.

### `__lt__(value, /)`

Return self<value.

### `__le__(value, /)`

Return self<=value.

### `__gt__(value, /)`

Return self>value.

### `__ge__(value, /)`

Return self>=value.

## Attributes

### `dt`

Return the sampling time of the system, *None* for *lti* systems.

### `gain`

Gain of the *ZerosPolesGain* system.

### `poles`

Poles of the *ZerosPolesGain* system.

**zeros**

Zeros of the *ZerosPolesGain* system.

**cupyx.scipy.signal.lsim**

`cupyx.scipy.signal.lsim(system, U, T, X0=None, interp=True)`

Simulate output of a continuous-time linear system.

**Parameters**

- **system** (*an instance of the LTI class or a tuple describing the system.*) – The following gives the number of elements in the tuple and the interpretation:
  - 1: (instance of *lti*)
  - 2: (num, den)
  - 3: (zeros, poles, gain)
  - 4: (A, B, C, D)
- **U** (*array\_like*) – An input array describing the input at each time *T* (interpolation is assumed between given times). If there are multiple inputs, then each column of the rank-2 array represents an input. If *U* = 0 or *None*, a zero input is used.
- **T** (*array\_like*) – The time steps at which the input is defined and at which the output is desired. Must be nonnegative, increasing, and equally spaced
- **X0** (*array\_like, optional*) – The initial conditions on the state vector (zero by default).
- **interp** (*bool, optional*) – Whether to use linear (*True*, the default) or zero-order-hold (*False*) interpolation for the input array.

**Returns**

- **T** (*1D ndarray*) – Time values for the output.
- **yout** (*1D ndarray*) – System response.
- **xout** (*ndarray*) – Time evolution of the state vector.

**Notes**

If (num, den) is passed in for **system**, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

**See also:**

`scipy.signal.lsim`

## cupyx.scipy.signal.impulse

`cupyx.scipy.signal.impulse(system, X0=None, T=None, N=None)`

Impulse response of continuous-time system.

### Parameters

- **system** (an instance of the LTI class or a tuple of array\_like) – describing the system. The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *lti*)
  - 2 (num, den)
  - 3 (zeros, poles, gain)
  - 4 (A, B, C, D)
- **X0** (array\_like, optional) – Initial state-vector. Defaults to zero.
- **T** (array\_like, optional) – Time points. Computed if not given.
- **N** (int, optional) – The number of time points to compute (if *T* is not given).

### Returns

- **T** (ndarray) – A 1-D array of time points.
- **yout** (ndarray) – A 1-D array containing the impulse response of the system (except for singularities at zero).

### Notes

If (num, den) is passed in for **system**, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

See also:

`scipy.signal.impulse`

## cupyx.scipy.signal.step

`cupyx.scipy.signal.step(system, X0=None, T=None, N=None)`

Step response of continuous-time system.

### Parameters

- **system** (an instance of the LTI class or a tuple of array\_like) – describing the system. The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *lti*)
  - 2 (num, den)
  - 3 (zeros, poles, gain)
  - 4 (A, B, C, D)
- **X0** (array\_like, optional) – Initial state-vector (default is zero).

- **T** (*array\_like, optional*) – Time points (computed if not given).
- **N** (*int, optional*) – Number of time points to compute if *T* is not given.

**Returns**

- **T** (*1D ndarray*) – Output time points.
- **yout** (*1D ndarray*) – Step response of system.

**Notes**

If (num, den) is passed in for **system**, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

**See also:**

`scipy.signal.step`

**cupyx.scipy.signal.freqresp**

`cupyx.scipy.signal.freqresp(system, w=None, n=10000)`

Calculate the frequency response of a continuous-time system.

**Parameters**

- **system** (an instance of the *lti* class or a tuple describing the system.) – The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *lti*)
  - 2 (num, den)
  - 3 (zeros, poles, gain)
  - 4 (A, B, C, D)
- **w** (*array\_like, optional*) – Array of frequencies (in rad/s). Magnitude and phase data is calculated for every value in this array. If not given, a reasonable set will be calculated.
- **n** (*int, optional*) – Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

**Returns**

- **w** (*1D ndarray*) – Frequency array [rad/s]
- **H** (*1D ndarray*) – Array of complex magnitude values

**Notes**

If (num, den) is passed in for **system**, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

**See also:**

`scipy.signal.freqresp`

## cupyx.scipy.signal.bode

`cupyx.scipy.signal.bode(system, w=None, n=100)`

Calculate Bode magnitude and phase data of a continuous-time system.

### Parameters

- **system** (an instance of the LTI class or a tuple describing the system.) – The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *lti*)
  - 2 (num, den)
  - 3 (zeros, poles, gain)
  - 4 (A, B, C, D)
- **w** (*array\_like, optional*) – Array of frequencies (in rad/s). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.
- **n** (*int, optional*) – Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

### Returns

- **w** (*1D ndarray*) – Frequency array [rad/s]
- **mag** (*1D ndarray*) – Magnitude array [dB]
- **phase** (*1D ndarray*) – Phase array [deg]

See also:

`scipy.signal.bode`

### Notes

If (num, den) is passed in for **system**, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $s^2 + 3s + 5$  would be represented as [1, 3, 5]).

## Discrete-time linear systems

<code>dlti(*system, **kwargs)</code>	Discrete-time linear time invariant system base class.
<code>StateSpace(*system, **kwargs)</code>	Linear Time Invariant system in state-space form.
<code>TransferFunction(*system, **kwargs)</code>	Linear Time Invariant system class in transfer function form.
<code>ZerosPolesGain(*system, **kwargs)</code>	Linear Time Invariant system class in zeros, poles, gain form.
<code>dlsim(system, u[, t, x0])</code>	Simulate output of a discrete-time linear system.
<code>dimpulse(system[, x0, t, n])</code>	Impulse response of discrete-time system.
<code>dstep(system[, x0, t, n])</code>	Step response of discrete-time system.
<code>dfreqresp(system[, w, n, whole])</code>	Calculate the frequency response of a discrete-time system.
<code>dbode(system[, w, n])</code>	Calculate Bode magnitude and phase data of a discrete-time system.



**cupyx.scipy.signal.dlti**

**class** cupyx.scipy.signal.dlti(\*system, \*\*kwargs)

Discrete-time linear time invariant system base class.

**Parameters**

- **\*system** (*arguments*) – The *dlti* class can be instantiated with either 2, 3 or 4 arguments. The following gives the number of arguments and the corresponding discrete-time subclass that is created:
  - 2: *TransferFunction*: (numerator, denominator)
  - 3: *ZerosPolesGain*: (zeros, poles, gain)
  - 4: *StateSpace*: (A, B, C, D)
 Each argument can be an array or a sequence.
- **dt** (*float*, *optional*) – Sampling time [s] of the discrete-time systems. Defaults to True (unspecified sampling time). Must be specified as a keyword argument, for example, `dt=0.1`.

See also:

`scipy.signal.dlti`, `ZerosPolesGain`, `StateSpace`, `TransferFunction`, `lti`

**Notes**

*dlti* instances do not exist directly. Instead, *dlti* creates an instance of one of its subclasses: *StateSpace*, *TransferFunction* or *ZerosPolesGain*.

Changing the value of properties that are not directly part of the current system representation (such as the *zeros* of a *StateSpace* system) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the *zeros*, *poles* or *gain*.

If (numerator, denominator) is passed in for *\*system*, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.,  $z^2 + 3z + 5$  would be represented as `[1, 3, 5]`).

**Methods**

**bode**(*w=None*, *n=100*)

Calculate Bode magnitude and phase data of a discrete-time system.

Returns a 3-tuple containing arrays of frequencies [rad/s], magnitude [dB] and phase [deg]. See *dbode* for details.

**freqresp**(*w=None*, *n=10000*, *whole=False*)

Calculate the frequency response of a discrete-time system.

Returns a 2-tuple containing arrays of frequencies [rad/s] and complex magnitude. See *dfreqresp* for details.

**impulse**(*x0=None*, *t=None*, *n=None*)

Return the impulse response of the discrete-time *dlti* system. See *dimpulse* for details.

**output**(*u*, *t*, *x0=None*)

Return the response of the discrete-time system to input *u*. See *dlsim* for details.

**step**(*x0=None, t=None, n=None*)

Return the step response of the discrete-time *dlti* system. See *dstep* for details.

**\_\_eq\_\_**(*value, /*)

Return self==value.

**\_\_ne\_\_**(*value, /*)

Return self!=value.

**\_\_lt\_\_**(*value, /*)

Return self<value.

**\_\_le\_\_**(*value, /*)

Return self<=value.

**\_\_gt\_\_**(*value, /*)

Return self>value.

**\_\_ge\_\_**(*value, /*)

Return self>=value.

## Attributes

**dt**

Return the sampling time of the system.

**poles**

Poles of the system.

**zeros**

Zeros of the system.

## cupyx.scipy.signal.dlsim

**cupyx.scipy.signal.dlsim**(*system, u, t=None, x0=None*)

Simulate output of a discrete-time linear system.

### Parameters

- **system** (tuple of array\_like or instance of *dlti*) – A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:
  - 1: (instance of *dlti*)
  - 3: (num, den, dt)
  - 4: (zeros, poles, gain, dt)
  - 5: (A, B, C, D, dt)
- **u** (*array\_like*) – An input array describing the input at each time *t* (interpolation is assumed between given times). If there are multiple inputs, then each column of the rank-2 array represents an input.
- **t** (*array\_like, optional*) – The time steps at which the input is defined. If *t* is given, it must be the same length as *u*, and the final value in *t* determines the number of steps returned in the output.

- **x0** (*array\_like, optional*) – The initial conditions on the state vector (zero by default).

#### Returns

- **tout** (*ndarray*) – Time values for the output, as a 1-D array.
- **yout** (*ndarray*) – System response, as a 1-D array.
- **xout** (*ndarray, optional*) – Time-evolution of the state-vector. Only generated if the input is a *StateSpace* system.

#### See also:

`scipy.signal.dlsim`, `lsim`, `dstep`, `dimpulse`, `cont2discrete`

### cupyx.scipy.signal.dimpulse

`cupyx.scipy.signal.dimpulse(system, x0=None, t=None, n=None)`

Impulse response of discrete-time system.

#### Parameters

- **system** (tuple of *array\_like* or instance of *dlti*) – A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:
  - 1: (instance of *dlti*)
  - 3: (num, den, dt)
  - 4: (zeros, poles, gain, dt)
  - 5: (A, B, C, D, dt)
- **x0** (*array\_like, optional*) – Initial state-vector. Defaults to zero.
- **t** (*array\_like, optional*) – Time points. Computed if not given.
- **n** (*int, optional*) – The number of time points to compute (if *t* is not given).

#### Returns

- **tout** (*ndarray*) – Time values for the output, as a 1-D array.
- **yout** (*tuple of ndarray*) – Impulse response of system. Each element of the tuple represents the output of the system based on an impulse in each input.

#### See also:

`scipy.signal.dimpulse`, `impulse`, `dstep`, `dlsim`, `cont2discrete`

### cupyx.scipy.signal.dstep

`cupyx.scipy.signal.dstep(system, x0=None, t=None, n=None)`

Step response of discrete-time system.

#### Parameters

- **system** (*tuple of array\_like*) – A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:
  - 1: (instance of *dlti*)
  - 3: (num, den, dt)

- 4: (zeros, poles, gain, dt)
- 5: (A, B, C, D, dt)
- **x0** (*array\_like, optional*) – Initial state-vector. Defaults to zero.
- **t** (*array\_like, optional*) – Time points. Computed if not given.
- **n** (*int, optional*) – The number of time points to compute (if *t* is not given).

**Returns**

- **tout** (*ndarray*) – Output time points, as a 1-D array.
- **yout** (*tuple of ndarray*) – Step response of system. Each element of the tuple represents the output of the system based on a step response to each input.

**See also:**

`scipy.signal.dlstep`, [`step`](#), [`dimpulse`](#), [`dlsim`](#), [`cont2discrete`](#)

**cupyx.scipy.signal.dfreqresp**

`cupyx.scipy.signal.dfreqresp(system, w=None, n=10000, whole=False)`

Calculate the frequency response of a discrete-time system.

**Parameters**

- **system** (an instance of the *dlti* class or a tuple describing the system.) – The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *dlti*)
  - 2 (numerator, denominator, dt)
  - 3 (zeros, poles, gain, dt)
  - 4 (A, B, C, D, dt)
- **w** (*array\_like, optional*) – Array of frequencies (in radians/sample). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.
- **n** (*int, optional*) – Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.
- **whole** (*bool, optional*) – Normally, if ‘w’ is not given, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to 2\*pi radians/sample.

**Returns**

- **w** (*1D ndarray*) – Frequency array [radians/sample]
- **H** (*1D ndarray*) – Array of complex magnitude values

**See also:**

`scipy.signal.dfeqresp`

## Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $z^2 + 3z + 5$  would be represented as `[1, 3, 5]`).

## cupyx.scipy.signal.dbode

`cupyx.scipy.signal.dbode(system, w=None, n=100)`

Calculate Bode magnitude and phase data of a discrete-time system.

### Parameters

- **system** (an instance of the LTI class or a tuple describing the system.) – The following gives the number of elements in the tuple and the interpretation:
  - 1 (instance of *dlti*)
  - 2 (num, den, dt)
  - 3 (zeros, poles, gain, dt)
  - 4 (A, B, C, D, dt)
- **w** (*array\_like, optional*) – Array of frequencies (in radians/sample). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.
- **n** (*int, optional*) – Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

### Returns

- **w** (*1D ndarray*) – Frequency array [rad/time\_unit]
- **mag** (*1D ndarray*) – Magnitude array [dB]
- **phase** (*1D ndarray*) – Phase array [deg]

See also:

`scipy.signal.dbode`

## Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g.  $z^2 + 3z + 5$  would be represented as `[1, 3, 5]`).

## Waveforms

<code>chirp(t, f0, t1, f1[, method, phi, vertex_zero])</code>	Frequency-swept cosine generator.
<code>gausspulse(t[, fc, bw, bwr, tpr, retquad, ...])</code>	Return a Gaussian modulated sinusoid:
<code>max_len_seq(nbits[, state, length, taps])</code>	Maximum length sequence (MLS) generator.
<code>sawtooth(t[, width])</code>	Return a periodic sawtooth or triangle waveform.
<code>square(t[, duty])</code>	Return a periodic square-wave waveform.
<code>unit_impulse(shape[, idx, dtype])</code>	Unit impulse signal (discrete delta function) or unit basis vector.

## cupyx.scipy.signal.chirp

`cupyx.scipy.signal.chirp(t, f0, t1, f1, method='linear', phi=0, vertex_zero=True)`

Frequency-swept cosine generator.

In the following, ‘Hz’ should be interpreted as ‘cycles per unit’; there is no requirement here that the unit is one second. The important distinction is that the units of rotation are cycles, not radians. Likewise,  $t$  could be a measurement of space instead of time.

### Parameters

- **t** (*array\_like*) – Times at which to evaluate the waveform.
- **f0** (*float*) – Frequency (e.g. Hz) at time  $t=0$ .
- **t1** (*float*) – Time at which  $f1$  is specified.
- **f1** (*float*) – Frequency (e.g. Hz) of the waveform at time  $t1$ .
- **method** (*{'linear', 'quadratic', 'logarithmic', 'hyperbolic'}, optional*) – Kind of frequency sweep. If not given, *linear* is assumed. See Notes below for more details.
- **phi** (*float, optional*) – Phase offset, in degrees. Default is 0.
- **vertex\_zero** (*bool, optional*) – This parameter is only used when *method* is ‘quadratic’. It determines whether the vertex of the parabola that is the graph of the frequency is at  $t=0$  or  $t=t1$ .

### Returns

**y** – A numpy array containing the signal evaluated at  $t$  with the requested time-varying frequency. More precisely, the function returns  $\cos(\text{phase} + (\pi/180)*\text{phi})$  where *phase* is the integral (from 0 to  $t$ ) of  $2*\pi*f(t)$ .  $f(t)$  is defined below.

### Return type

*ndarray*

## Examples

The following will be used in the examples:

```
>>> from cupyx.scipy.signal import chirp, spectrogram
>>> import matplotlib.pyplot as plt
>>> import cupy as cp
```

For the first example, we’ll plot the waveform for a linear chirp from 6 Hz to 1 Hz over 10 seconds:

```
>>> t = cupy.linspace(0, 10, 5001)
>>> w = chirp(t, f0=6, f1=1, t1=10, method='linear')
>>> plt.plot(cupy.asnumpy(t), cupy.asnumpy(w))
>>> plt.title("Linear Chirp, f(0)=6, f(10)=1")
>>> plt.xlabel('t (sec)')
>>> plt.show()
```

For the remaining examples, we’ll use higher frequency ranges, and demonstrate the result using *cupyx.scipy.signal.spectrogram*. We’ll use a 10 second interval sampled at 8000 Hz.

```
>>> fs = 8000
>>> T = 10
>>> t = cupy.linspace(0, T, T*fs, endpoint=False)
```

Quadratic chirp from 1500 Hz to 250 Hz over 10 seconds (vertex of the parabolic curve of the frequency is at  $t=0$ ):

```
>>> w = chirp(t, f0=1500, f1=250, t1=10, method='quadratic')
>>> ff, tt, Sxx = spectrogram(w, fs=fs, noverlap=256, nperseg=512,
...                             nfft=2048)
>>> plt.pcolormesh(cupy.asnumpy(tt), cupy.asnumpy(ff[:513]),
...                 cupy.asnumpy(Sxx[:513]), cmap='gray_r')
>>> plt.title('Quadratic Chirp, f(0)=1500, f(10)=250')
>>> plt.xlabel('t (sec)')
>>> plt.ylabel('Frequency (Hz)')
>>> plt.grid()
>>> plt.show()
```

## cupyx.scipy.signal.gausspulse

`cupyx.scipy.signal.gausspulse(t, fc=1000, bw=0.5, bwr=-6, tpr=-60, retquad=False, retenv=False)`

Return a Gaussian modulated sinusoid:

$$\exp(-a \ t^2) \ \exp(1j*2*\pi*fc*t).$$

If *retquad* is True, then return the real and imaginary parts (in-phase and quadrature). If *retenv* is True, then return the envelope (unmodulated signal). Otherwise, return the real part of the modulated sinusoid.

### Parameters

- **t** (*ndarray* or the string 'cutoff') – Input array.
- **fc** (*int*, optional) – Center frequency (e.g. Hz). Default is 1000.
- **bw** (*float*, optional) – Fractional bandwidth in frequency domain of pulse (e.g. Hz). Default is 0.5.
- **bwr** (*float*, optional) – Reference level at which fractional bandwidth is calculated (dB). Default is -6.
- **tpr** (*float*, optional) – If *t* is 'cutoff', then the function returns the cutoff time for when the pulse amplitude falls below *tpr* (in dB). Default is -60.
- **retquad** (*bool*, optional) – If True, return the quadrature (imaginary) as well as the real part of the signal. Default is False.
- **retenv** (*bool*, optional) – If True, return the envelope of the signal. Default is False.

### Returns

- **yI** (*ndarray*) – Real part of signal. Always returned.
- **yQ** (*ndarray*) – Imaginary part of signal. Only returned if *retquad* is True.
- **yenv** (*ndarray*) – Envelope of signal. Only returned if *retenv* is True.

See also:

[`cupyx.scipy.signal.morlet`](#)

## Examples

Plot real component, imaginary component, and envelope for a 5 Hz pulse, sampled at 100 Hz for 2 seconds:

```
>>> import cupyx.scipy.signal
>>> import cupy as cp
>>> import matplotlib.pyplot as plt
>>> t = cupy.linspace(-1, 1, 2 * 100, endpoint=False)
>>> i, q, e = cupyx.scipy.signal.gausspulse(t, fc=5, retquad=True, retenv=True)
>>> plt.plot(cupy.asnumpy(t), cupy.asnumpy(i), cupy.asnumpy(t), cupy.asnumpy(q),
            cupy.asnumpy(t), cupy.asnumpy(e), '--')
```

## cupyx.scipy.signal.max\_len\_seq

cupyx.scipy.signal.max\_len\_seq(*nbits*, *state=None*, *length=None*, *taps=None*)

Maximum length sequence (MLS) generator.

### Parameters

- **nbits** (*int*) – Number of bits to use. Length of the resulting sequence will be  $(2^{**nbits}) - 1$ . Note that generating long sequences (e.g., greater than `nbits == 16`) can take a long time.
- **state** (*array\_like*, *optional*) – If array, must be of length `nbits`, and will be cast to binary (bool) representation. If `None`, a seed of ones will be used, producing a repeatable representation. If `state` is all zeros, an error is raised as this is invalid. Default: `None`.
- **length** (*int*, *optional*) – Number of samples to compute. If `None`, the entire length  $(2^{**nbits}) - 1$  is computed.
- **taps** (*array\_like*, *optional*) – Polynomial taps to use (e.g., `[7, 6, 1]` for an 8-bit sequence). If `None`, taps will be automatically selected (for up to `nbits == 32`).

### Returns

- **seq** (*array*) – Resulting MLS sequence of 0's and 1's.
- **state** (*array*) – The final state of the shift register.

## Notes

The algorithm for MLS generation is generically described in:

[https://en.wikipedia.org/wiki/Maximum\\_length\\_sequence](https://en.wikipedia.org/wiki/Maximum_length_sequence)

The default values for taps are specifically taken from the first option listed for each value of `nbits` in:

[https://web.archive.org/web/20181001062252/http://www.newwaveinstruments.com/resources/articles/m\\_sequence\\_linear\\_feedback\\_shift\\_register\\_lfsr.htm](https://web.archive.org/web/20181001062252/http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm)



## cupyx.scipy.signal.sawtooth

`cupyx.scipy.signal.sawtooth(t, width=1.0)`

Return a periodic sawtooth or triangle waveform.

The sawtooth waveform has a period  $2\pi$ , rises from -1 to 1 on the interval 0 to  $\text{width} \cdot 2\pi$ , then drops from 1 to -1 on the interval  $\text{width} \cdot 2\pi$  to  $2\pi$ . *width* must be in the interval [0, 1].

Note that this is not band-limited. It produces an infinite number of harmonics, which are aliased back and forth across the frequency spectrum.

### Parameters

- **t** (*array\_like*) – Time.
- **width** (*array\_like, optional*) – Width of the rising ramp as a proportion of the total cycle. Default is 1, producing a rising ramp, while 0 produces a falling ramp. *width* = 0.5 produces a triangle wave. If an array, causes wave shape to change over time, and must be the same length as *t*.

### Returns

**y** – Output array containing the sawtooth waveform.

### Return type

*ndarray*

## Examples

A 5 Hz waveform sampled at 500 Hz for 1 second:

```
>>> from cupyx.scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(0, 1, 500)
>>> plt.plot(t, signal.sawtooth(2 * np.pi * 5 * t))
```

## cupyx.scipy.signal.square

`cupyx.scipy.signal.square(t, duty=0.5)`

Return a periodic square-wave waveform.

The square wave has a period  $2\pi$ , has value +1 from 0 to  $2\pi \cdot \text{duty}$  and -1 from  $2\pi \cdot \text{duty}$  to  $2\pi$ . *duty* must be in the interval [0,1].

Note that this is not band-limited. It produces an infinite number of harmonics, which are aliased back and forth across the frequency spectrum.

### Parameters

- **t** (*array\_like*) – The input time array.
- **duty** (*array\_like, optional*) – Duty cycle. Default is 0.5 (50% duty cycle). If an array, causes wave shape to change over time, and must be the same length as *t*.

### Returns

**y** – Output array containing the square waveform.

### Return type

*ndarray*

## Examples

A 5 Hz waveform sampled at 500 Hz for 1 second:

```
>>> import cupyx.scipy.signal
>>> import cupy as cp
>>> import matplotlib.pyplot as plt
>>> t = cupy.linspace(0, 1, 500, endpoint=False)
>>> plt.plot(cupy.asnumpy(t), cupy.asnumpy(cupyx.scipy.signal.square(2 * cupy.pi *
↪ 5 * t)))
>>> plt.ylim(-2, 2)
```

A pulse-width modulated sine wave:

```
>>> plt.figure()
>>> sig = cupy.sin(2 * cupy.pi * t)
>>> pwm = cupyx.scipy.signal.square(2 * cupy.pi * 30 * t, duty=(sig + 1)/2)
>>> plt.subplot(2, 1, 1)
>>> plt.plot(cupy.asnumpy(t), cupy.asnumpy(sig))
>>> plt.subplot(2, 1, 2)
>>> plt.plot(cupy.asnumpy(t), cupy.asnumpy(pwm))
>>> plt.ylim(-1.5, 1.5)
```

## cupyx.scipy.signal.unit\_impulse

`cupyx.scipy.signal.unit_impulse(shape, idx=None, dtype=<class 'float'>)`

Unit impulse signal (discrete delta function) or unit basis vector.

### Parameters

- **shape** (*int* or *tuple of int*) – Number of samples in the output (1-D), or a tuple that represents the shape of the output (N-D).
- **idx** (*None* or *int* or *tuple of int* or *'mid'*, *optional*) – Index at which the value is 1. If *None*, defaults to the 0th element. If *idx='mid'*, the impulse will be centered at `shape // 2` in all dimensions. If an *int*, the impulse will be at *idx* in all dimensions.
- **dtype** (*data-type*, *optional*) – The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

### Returns

*y* – Output array containing an impulse signal.

### Return type

*ndarray*

## Notes

The 1D case is also known as the Kronecker delta.

## Examples

An impulse at the 0th element ( $\delta[n]$ ):

```
>>> import cupyx.scipy.signal
>>> import cupy as cp
>>> cupyx.scipy.signal.unit_impulse(8)
array([ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

Impulse offset by 2 samples ( $\delta[n - 2]$ ):

```
>>> cupyx.scipy.signal.unit_impulse(7, 2)
array([ 0.,  0.,  1.,  0.,  0.,  0.,  0.])
```

2-dimensional impulse, centered:

```
>>> cupyx.scipy.signal.unit_impulse((3, 3), 'mid')
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  0.]])
```

Impulse at (2, 2), using broadcasting:

```
>>> cupyx.scipy.signal.unit_impulse((4, 4), 2)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

## Window functions

For window functions, see the `cupyx.scipy.signal.windows` namespace.

In the `cupyx.scipy.signal` namespace, there is a convenience function to obtain these windows by name:

---

<code>get_window(window, Nx[, fftbins])</code>	Return a window of a given length and type.
--	---

---

### cupyx.scipy.signal.get\_window

`cupyx.scipy.signal.get_window(window, Nx, fftbins=True)`

Return a window of a given length and type.

#### Parameters

- **window** (*string*, *float*, or *tuple*) – The type of window to create. See below for more details.
- **Nx** (*int*) – The number of samples in the window.

- **fftbins** (*bool*, *optional*) – If True (default), create a “periodic” window, ready to use with *fftshift* and be multiplied by the result of an FFT (see also *fftpack.fftfreq*). If False, create a “symmetric” window, for use in filter design.

**Returns**

**get\_window** – Returns a window of length  $Nx$  and type *window*

**Return type**

*ndarray*

**Notes**

Window types:

- *boxcar()*
- *triang()*
- *blackman()*
- *hamming()*
- *hann()*
- *bartlett()*
- *flattop()*
- *parzen()*
- *bohman()*
- *blackmanharris()*
- *nuttall()*
- *barthann()*
- *kaiser()* (needs beta)
- *gaussian()* (needs standard deviation)
- *general\_gaussian()* (needs power, width)
- *chebwin()* (needs attenuation)
- *exponential()* (needs decay scale)
- *tukey()* (needs taper fraction)

If the window requires no parameters, then *window* can be a string.

If the window requires parameters, then *window* must be a tuple with the first argument the string name of the window, and the next arguments the needed parameters.

If *window* is a floating point number, it is interpreted as the beta parameter of the *kaiser()* window.

Each of the window types listed above is also the name of a function that can be called directly to create a window of that type.

## Examples

```
>>> import cupyx.scipy.signal.windows
>>> cupyx.scipy.signal.windows.get_window('triang', 7)
array([ 0.125,  0.375,  0.625,  0.875,  0.875,  0.625,  0.375])
>>> cupyx.scipy.signal.windows.get_window(('kaiser', 4.0), 9)
array([0.08848053, 0.32578323, 0.63343178, 0.89640418, 1.,
       0.89640418, 0.63343178, 0.32578323, 0.08848053])
>>> cupyx.scipy.signal.windows.get_window(4.0, 9)
array([0.08848053, 0.32578323, 0.63343178, 0.89640418, 1.,
       0.89640418, 0.63343178, 0.32578323, 0.08848053])
```

## Wavelets

<code>morlet(M[, w, s, complete])</code>	Complex Morlet wavelet.
<code>qmf(hk)</code>	Return high-pass qmf filter from low-pass
<code>ricker(points, a)</code>	Return a Ricker wavelet, also known as the "Mexican hat wavelet".
<code>morlet2(M, s[, w])</code>	Complex Morlet wavelet, designed to work with <code>cwt</code> . Returns the complete version of morlet wavelet, normalised according to <code>s::</code> .
<code>cwt(data, wavelet, widths)</code>	Continuous wavelet transform.

### cupyx.scipy.signal.morlet

`cupyx.scipy.signal.morlet(M, w=5.0, s=1.0, complete=True)`

Complex Morlet wavelet.

#### Parameters

- **M** (*int*) – Length of the wavelet.
- **w** (*float*, *optional*) – Omega0. Default is 5
- **s** (*float*, *optional*) – Scaling factor, windowed from  $-s*2*\pi$  to  $+s*2*\pi$ . Default is 1.
- **complete** (*bool*, *optional*) – Whether to use the complete or the standard version.

#### Returns

**morlet**

#### Return type

(M,) *ndarray*

#### See also:

`cupyx.scipy.signal.gausspulse`

## Notes

The standard version:

```
pi**-0.25 * exp(1j*w*x) * exp(-0.5*(x**2))
```

This commonly used wavelet is often referred to simply as the Morlet wavelet. Note that this simplified version can cause admissibility problems at low values of  $w$ .

The complete version:

```
pi**-0.25 * (exp(1j*w*x) - exp(-0.5*(w**2))) * exp(-0.5*(x**2))
```

This version has a correction term to improve admissibility. For  $w$  greater than 5, the correction term is negligible.

Note that the energy of the return wavelet is not normalised according to  $s$ .

The fundamental frequency of this wavelet in Hz is given by  $f = 2*s*w*r / M$  where  $r$  is the sampling rate.

Note: This function was created before `cwt` and is not compatible with it.

## cupyx.scipy.signal.qmf

`cupyx.scipy.signal.qmf(hk)`

Return high-pass qmf filter from low-pass

### Parameters

**hk** (*array\_like*) – Coefficients of high-pass filter.

## cupyx.scipy.signal.ricker

`cupyx.scipy.signal.ricker(points, a)`

Return a Ricker wavelet, also known as the “Mexican hat wavelet”.

It models the function:

$$A (1 - x^2/a^2) \exp(-x^2/2 a^2),$$

where  $A = 2/\sqrt{3a}\pi^{1/4}$ .

### Parameters

- **points** (*int*) – Number of points in *vector*. Will be centered around 0.
- **a** (*scalar*) – Width parameter of the wavelet.

### Returns

**vector** – Array of length *points* in shape of ricker curve.

### Return type

(N,) *ndarray*

## Examples

```
>>> import cupyx.scipy.signal
>>> import cupy as cp
>>> import matplotlib.pyplot as plt
```

```
>>> points = 100
>>> a = 4.0
>>> vec2 = cupyx.scipy.signal.ricker(points, a)
>>> print(len(vec2))
100
>>> plt.plot(cupy.asnumpy(vec2))
>>> plt.show()
```

## cupyx.scipy.signal.morlet2

cupyx.scipy.signal.morlet2(*M*, *s*, *w*=5)

Complex Morlet wavelet, designed to work with *cwt*. Returns the complete version of morlet wavelet, normalised according to *s*:

$$\exp(1j \cdot w \cdot x / s) * \exp(-0.5 \cdot (x/s)**2) * \pi ** (-0.25) * \sqrt{1/s}$$

### Parameters

- **M** (*int*) – Length of the wavelet.
- **s** (*float*) – Width parameter of the wavelet.
- **w** (*float*, *optional*) – Omega0. Default is 5

### Returns

morlet

### Return type

(M,) *ndarray*

### See also:

#### *morlet*

Implementation of Morlet wavelet, incompatible with *cwt*

## Notes

This function was designed to work with *cwt*. Because *morlet2* returns an array of complex numbers, the *dtype* argument of *cwt* should be set to *complex128* for best results.

Note the difference in implementation with *morlet*. The fundamental frequency of this wavelet in Hz is given by:

$$f = w \cdot f_s / (2 \cdot s \cdot \pi)$$

where *f<sub>s</sub>* is the sampling rate and *s* is the wavelet width parameter. Similarly we can get the wavelet width parameter at *f*:

```
s = w*fs / (2*f*np.pi)
```

## Examples

```
>>> from cupyx.scipy import signal
>>> import matplotlib.pyplot as plt
>>> M = 100
>>> s = 4.0
>>> w = 2.0
>>> wavelet = signal.morlet2(M, s, w)
>>> plt.plot(abs(wavelet))
>>> plt.show()
```

This example shows basic use of *morlet2* with *cwt* in time-frequency analysis:

```
>>> from cupyx.scipy import signal
>>> import matplotlib.pyplot as plt
>>> t, dt = np.linspace(0, 1, 200, retstep=True)
>>> fs = 1/dt
>>> w = 6.
>>> sig = np.cos(2*np.pi*(50 + 10*t)*t) + np.sin(40*np.pi*t)
>>> freq = np.linspace(1, fs/2, 100)
>>> widths = w*fs / (2*freq*np.pi)
>>> cwtm = signal.cwt(sig, signal.morlet2, widths, w=w)
>>> plt.pcolormesh(t, freq, np.abs(cwtm),
>>>                 cmap='viridis', shading='gouraud')
>>> plt.show()
```

## cupyx.scipy.signal.cwt

`cupyx.scipy.signal.cwt(data, wavelet, widths)`

Continuous wavelet transform.

Performs a continuous wavelet transform on *data*, using the *wavelet* function. A CWT performs a convolution with *data* using the *wavelet* function, which is characterized by a width parameter and length parameter.

### Parameters

- **data** ((*N*,) *ndarray*) – data on which to perform the transform.
- **wavelet** (*function*) – Wavelet function, which should take 2 arguments. The first argument is the number of points that the returned vector will have (`len(wavelet(length,width)) == length`). The second is a width parameter, defining the size of the wavelet (e.g. standard deviation of a gaussian). See *ricker*, which satisfies these requirements.
- **widths** ((*M*,) *sequence*) – Widths to use for transform.

### Returns

**cwt** – Will have shape of (`len(widths)`, `len(data)`).

### Return type

(*M*, *N*) *ndarray*



## Notes

```
length = min(10 * width[ii], len(data))
cwt[ii,:] = cupyx.scipy.signal.convolve(data, wavelet(length,
                                                    width[ii]), mode='same')
```

## Examples

```
>>> import cupyx.scipy.signal
>>> import cupy as cp
>>> import matplotlib.pyplot as plt
>>> t = cupy.linspace(-1, 1, 200, endpoint=False)
>>> sig = cupy.cos(2 * cupy.pi * 7 * t) + cupyx.scipy.signal.gausspulse(t - 0.4,
↪fc=2)
>>> widths = cupy.arange(1, 31)
>>> cwtmatr = cupyx.scipy.signal.cwt(sig, cupyx.scipy.signal.ricker, widths)
>>> plt.imshow(abs(cupy.asnumpy(cwtmatr)), extent=[-1, 1, 31, 1],
               cmap='PRGn', aspect='auto', vmax=abs(cwtmatr).max(),
               vmin=-abs(cwtmatr).max())
>>> plt.show()
```

## Peak finding

<code>argrelmin(data[, axis, order, mode])</code>	Calculate the relative minima of <i>data</i> .
<code>argrelmax(data[, axis, order, mode])</code>	Calculate the relative maxima of <i>data</i> .
<code>argrelextrema(data, comparator[, axis, ...])</code>	Calculate the relative extrema of <i>data</i> .
<code>find_peaks(x[, height, threshold, distance, ...])</code>	Find peaks inside a signal based on peak properties.
<code>peak_prominences(x, peaks[, wlen])</code>	Calculate the prominence of each peak in a signal.
<code>peak_widths(x, peaks[, rel_height, ...])</code>	Calculate the width of each peak in a signal.

### cupyx.scipy.signal.argrelmin

`cupyx.scipy.signal.argrelmin(data, axis=0, order=1, mode='clip')`

Calculate the relative minima of *data*.

#### Parameters

- **data** (`ndarray`) – Array in which to find the relative minima.
- **axis** (`int`, *optional*) – Axis over which to select from *data*. Default is 0.
- **order** (`int`, *optional*) – How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.
- **mode** (`str`, *optional*) – How the edges of the vector are treated. Available options are ‘wrap’ (wrap around) or ‘clip’ (treat overflow as the same as the last (or first) element). Default ‘clip’. See `cupy.take`.

#### Returns

**extrema** – Indices of the minima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

**Return type**

tuple of ndarrays

**See also:**[\*argrextrema\*](#), [\*argrelmax\*](#), [\*find\\_peaks\*](#)**Notes**

This function uses *argrextrema* with `cupy.less` as comparator. Therefore it requires a strict inequality on both sides of a value to consider it a minimum. This means flat minima (more than one sample wide) are not detected. In case of one-dimensional *data* *find\_peaks* can be used to detect all local minima, including flat ones, by calling it with negated *data*.

**Examples**

```
>>> from cupyx.scipy.signal import argrelmin
>>> import cupy
>>> x = cupy.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelmin(x)
(array([1, 5]),)
>>> y = cupy.array([[1, 2, 1, 2],
...                 [2, 2, 0, 0],
...                 [5, 3, 4, 4]])
>>> argrelmin(y, axis=1)
(array([0, 2]), array([2, 1]))
```

**cupyx.scipy.signal.argrelmax**`cupyx.scipy.signal.argrelmax(data, axis=0, order=1, mode='clip')`Calculate the relative maxima of *data*.**Parameters**

- **data** (ndarray) – Array in which to find the relative maxima.
- **axis** (int, optional) – Axis over which to select from *data*. Default is 0.
- **order** (int, optional) – How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.
- **mode** (str, optional) – How the edges of the vector are treated. Available options are ‘wrap’ (wrap around) or ‘clip’ (treat overflow as the same as the last (or first) element). Default ‘clip’. See `cupy.take`.

**Returns**

**extrema** – Indices of the maxima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

**Return type**

tuple of ndarrays

**See also:**[\*argrextrema\*](#), [\*argrelmin\*](#), [\*find\\_peaks\*](#)

## Notes

This function uses *argrextrema* with `cupy.greater` as comparator. Therefore it requires a strict inequality on both sides of a value to consider it a maximum. This means flat maxima (more than one sample wide) are not detected. In case of one-dimensional *data* *find\_peaks* can be used to detect all local maxima, including flat ones.

## Examples

```
>>> from cupyx.scipy.signal import argrelmax
>>> import cupy
>>> x = cupy.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelmax(x)
(array([3, 6]),)
>>> y = cupy.array([[1, 2, 1, 2],
...                 [2, 2, 0, 0],
...                 [5, 3, 4, 4]])
...
>>> argrelmax(y, axis=1)
(array([0]), array([1]))
```

## cupyx.scipy.signal.argrextrema

`cupyx.scipy.signal.argrextrema(data, comparator, axis=0, order=1, mode='clip')`

Calculate the relative extrema of *data*.

### Parameters

- **data** (`ndarray`) – Array in which to find the relative extrema.
- **comparator** (`callable`) – Function to use to compare two data points. Should take two arrays as arguments.
- **axis** (`int`, *optional*) – Axis over which to select from *data*. Default is 0.
- **order** (`int`, *optional*) – How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.
- **mode** (`str`, *optional*) – How the edges of the vector are treated. Available options are ‘wrap’ (wrap around) or ‘clip’ (treat overflow as the same as the last (or first) element). Default ‘clip’. See `cupy.take`.

### Returns

**extrema** – Indices of the maxima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

### Return type

tuple of ndarrays

See also:

*argrelmin*, *argrelmax*

## Examples

```
>>> from cupyx.scipy.signal import argrelextrema
>>> import cupy
>>> x = cupy.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelextrema(x, cupy.greater)
(array([3, 6]),)
>>> y = cupy.array([[1, 2, 1, 2],
...                 [2, 2, 0, 0],
...                 [5, 3, 4, 4]])
...
>>> argrelextrema(y, cupy.less, axis=1)
(array([0, 2]), array([2, 1]))
```

## cupyx.scipy.signal.find\_peaks

`cupyx.scipy.signal.find_peaks(x, height=None, threshold=None, distance=None, prominence=None, width=None, wlen=None, rel_height=0.5, plateau_size=None)`

Find peaks inside a signal based on peak properties.

This function takes a 1-D array and finds all local maxima by simple comparison of neighboring values. Optionally, a subset of these peaks can be selected by specifying conditions for a peak's properties.

### Parameters

- **x** (*sequence*) – A signal with peaks.
- **height** (*number or ndarray or sequence, optional*) – Required height of peaks. Either a number, `None`, an array matching `x` or a 2-element sequence of the former. The first element is always interpreted as the minimal and the second, if supplied, as the maximal required height.
- **threshold** (*number or ndarray or sequence, optional*) – Required threshold of peaks, the vertical distance to its neighboring samples. Either a number, `None`, an array matching `x` or a 2-element sequence of the former. The first element is always interpreted as the minimal and the second, if supplied, as the maximal required threshold.
- **distance** (*number, optional*) – Required minimal horizontal distance ( $\geq 1$ ) in samples between neighbouring peaks. Smaller peaks are removed first until the condition is fulfilled for all remaining peaks.
- **prominence** (*number or ndarray or sequence, optional*) – Required prominence of peaks. Either a number, `None`, an array matching `x` or a 2-element sequence of the former. The first element is always interpreted as the minimal and the second, if supplied, as the maximal required prominence.
- **width** (*number or ndarray or sequence, optional*) – Required width of peaks in samples. Either a number, `None`, an array matching `x` or a 2-element sequence of the former. The first element is always interpreted as the minimal and the second, if supplied, as the maximal required width.
- **wlen** (*int, optional*) – Used for calculation of the peaks prominences, thus it is only used if one of the arguments `prominence` or `width` is given. See argument `wlen` in `peak_prominences` for a full description of its effects.

- **rel\_height** (*float*, *optional*) – Used for calculation of the peaks width, thus it is only used if *width* is given. See argument *rel\_height* in *peak\_widths* for a full description of its effects.
- **plateau\_size** (*number or ndarray or sequence*, *optional*) – Required size of the flat top of peaks in samples. Either a number, *None*, an array matching *x* or a 2-element sequence of the former. The first element is always interpreted as the minimal and the second, if supplied as the maximal required plateau size.

New in version 1.2.0.

### Returns

- **peaks** (*ndarray*) – Indices of peaks in *x* that satisfy all given conditions.
- **properties** (*dict*) – A dictionary containing properties of the returned peaks which were calculated as intermediate results during evaluation of the specified conditions:
  - **'peak\_heights'**  
If *height* is given, the height of each peak in *x*.
  - **'left\_thresholds', 'right\_thresholds'**  
If *threshold* is given, these keys contain a peaks vertical distance to its neighbouring samples.
  - **'prominences', 'right\_bases', 'left\_bases'**  
If *prominence* is given, these keys are accessible. See *peak\_prominences* for a description of their content.
  - **'width\_heights', 'left\_ips', 'right\_ips'**  
If *width* is given, these keys are accessible. See *peak\_widths* for a description of their content.
  - **'plateau\_sizes', 'left\_edges', 'right\_edges'**  
If *plateau\_size* is given, these keys are accessible and contain the indices of a peak's edges (edges are still part of the plateau) and the calculated plateau sizes.

To calculate and return properties without excluding peaks, provide the open interval (*None*, *None*) as a value to the appropriate argument (excluding *distance*).

### Warns

**PeakPropertyWarning** – Raised if a peak's properties have unexpected values (see *peak\_prominences* and *peak\_widths*).

**Warning:** This function may return unexpected results for data containing NaNs. To avoid this, NaNs should either be removed or replaced.

See also:

#### **find\_peaks\_cwt**

Find peaks using the wavelet transformation.

#### **peak\_prominences**

Directly calculate the prominence of peaks.

#### **peak\_widths**

Directly calculate the width of peaks.

## Notes

In the context of this function, a peak or local maximum is defined as any sample whose two direct neighbours have a smaller amplitude. For flat peaks (more than one sample of equal amplitude wide) the index of the middle sample is returned (rounded down in case the number of samples is even). For noisy signals the peak locations can be off because the noise might change the position of local maxima. In those cases consider smoothing the signal before searching for peaks or use other peak finding and fitting methods (like *find\_peaks\_cwt*).

Some additional comments on specifying conditions:

- Almost all conditions (excluding *distance*) can be given as half-open or closed intervals, e.g., 1 or (1, None) defines the half-open interval  $[1, \infty]$  while (None, 1) defines the interval  $[-\infty, 1]$ . The open interval (None, None) can be specified as well, which returns the matching properties without exclusion of peaks.
- The border is always included in the interval used to select valid peaks.
- For several conditions the interval borders can be specified with arrays matching *x* in shape which enables dynamic constraints based on the sample position.
- The conditions are evaluated in the following order: *plateau\_size*, *height*, *threshold*, *distance*, *prominence*, *width*. In most cases this order is the fastest one because faster operations are applied first to reduce the number of peaks that need to be evaluated later.
- While indices in *peaks* are guaranteed to be at least *distance* samples apart, edges of flat peaks may be closer than the allowed *distance*.
- Use *wlen* to reduce the time it takes to evaluate the conditions for *prominence* or *width* if *x* is large or has many local maxima (see *peak\_prominences*).

## cupyx.scipy.signal.peak\_prominences

`cupyx.scipy.signal.peak_prominences(x, peaks, wlen=None)`

Calculate the prominence of each peak in a signal.

The prominence of a peak measures how much a peak stands out from the surrounding baseline of the signal and is defined as the vertical distance between the peak and its lowest contour line.

### Parameters

- **x** (*sequence*) – A signal with peaks.
- **peaks** (*sequence*) – Indices of peaks in *x*.
- **wlen** (*int*, *optional*) – A window length in samples that optionally limits the evaluated area for each peak to a subset of *x*. The peak is always placed in the middle of the window therefore the given length is rounded up to the next odd integer. This parameter can speed up the calculation (see Notes).

### Returns

- **prominences** (*ndarray*) – The calculated prominences for each peak in *peaks*.
- **left\_bases**, **right\_bases** (*ndarray*) – The peaks' bases as indices in *x* to the left and right of each peak. The higher base of each pair is a peak's lowest contour line.

### Raises

**ValueError** – If a value in *peaks* is an invalid index for *x*.

### Warns

**PeakPropertyWarning** – For indices in *peaks* that don't point to valid local maxima in *x*, the

returned prominence will be 0 and this warning is raised. This also happens if *wlen* is smaller than the plateau size of a peak.

**Warning:** This function may return unexpected results for data containing NaNs. To avoid this, NaNs should either be removed or replaced.

See also:

[\*find\\_peaks\*](#)

Find peaks inside a signal based on peak properties.

[\*peak\\_widths\*](#)

Calculate the width of peaks.

## Notes

Strategy to compute a peak's prominence:

1. Extend a horizontal line from the current peak to the left and right until the line either reaches the window border (see *wlen*) or intersects the signal again at the slope of a higher peak. An intersection with a peak of the same height is ignored.
2. On each side find the minimal signal value within the interval defined above. These points are the peak's bases.
3. The higher one of the two bases marks the peak's lowest contour line. The prominence can then be calculated as the vertical difference between the peaks height itself and its lowest contour line.

Searching for the peak's bases can be slow for large *x* with periodic behavior because large chunks or even the full signal need to be evaluated for the first algorithmic step. This evaluation area can be limited with the parameter *wlen* which restricts the algorithm to a window around the current peak and can shorten the calculation time if the window length is short in relation to *x*. However, this may stop the algorithm from finding the true global contour line if the peak's true bases are outside this window. Instead, a higher contour line is found within the restricted window leading to a smaller calculated prominence. In practice, this is only relevant for the highest set of peaks in *x*. This behavior may even be used intentionally to calculate "local" prominences.

## cupyx.scipy.signal.peak\_widths

`cupyx.scipy.signal.peak_widths(x, peaks, rel_height=0.5, prominence_data=None, wlen=None)`

Calculate the width of each peak in a signal.

This function calculates the width of a peak in samples at a relative distance to the peak's height and prominence.

### Parameters

- **x** (*sequence*) – A signal with peaks.
- **peaks** (*sequence*) – Indices of peaks in *x*.
- **rel\_height** (*float*, *optional*) – Chooses the relative height at which the peak width is measured as a percentage of its prominence. 1.0 calculates the width of the peak at its lowest contour line while 0.5 evaluates at half the prominence height. Must be at least 0. See notes for further explanation.

- **prominence\_data** (*tuple*, *optional*) – A tuple of three arrays matching the output of *peak\_prominences* when called with the same arguments *x* and *peaks*. This data are calculated internally if not provided.
- **wlen** (*int*, *optional*) – A window length in samples passed to *peak\_prominences* as an optional argument for internal calculation of *prominence\_data*. This argument is ignored if *prominence\_data* is given.

#### Returns

- **widths** (*ndarray*) – The widths for each peak in samples.
- **width\_heights** (*ndarray*) – The height of the contour lines at which the *widths* were evaluated.
- **left\_ips, right\_ips** (*ndarray*) – Interpolated positions of left and right intersection points of a horizontal line at the respective evaluation height.

#### Raises

**ValueError** – If *prominence\_data* is supplied but doesn't satisfy the condition  $0 \leq \text{left\_base} \leq \text{peak} \leq \text{right\_base} < x.\text{shape}[0]$  for each peak, has the wrong dtype, is not C-contiguous or does not have the same shape.

#### Warns

**PeakPropertyWarning** – Raised if any calculated width is 0. This may stem from the supplied *prominence\_data* or if *rel\_height* is set to 0.

**Warning:** This function may return unexpected results for data containing NaNs. To avoid this, NaNs should either be removed or replaced.

See also:

[\*find\\_peaks\*](#)

Find peaks inside a signal based on peak properties.

[\*peak\\_prominences\*](#)

Calculate the prominence of peaks.

#### Notes

The basic algorithm to calculate a peak's width is as follows:

- Calculate the evaluation height  $h_{eval}$  with the formula  $h_{eval} = h_{Peak} - P \cdot R$ , where  $h_{Peak}$  is the height of the peak itself,  $P$  is the peak's prominence and  $R$  a positive ratio specified with the argument *rel\_height*.
- Draw a horizontal line at the evaluation height to both sides, starting at the peak's current vertical position until the lines either intersect a slope, the signal border or cross the vertical position of the peak's base (see *peak\_prominences* for an definition). For the first case, intersection with the signal, the true intersection point is estimated with linear interpolation.
- Calculate the width as the horizontal distance between the chosen endpoints on both sides. As a consequence of this the maximal possible width for each peak is the horizontal distance between its bases.

As shown above to calculate a peak's width its prominence and bases must be known. You can supply these yourself with the argument *prominence\_data*. Otherwise, they are internally calculated (see *peak\_prominences*).



## Spectral analysis

<code>periodogram(x[, fs, window, nfft, detrend, ...])</code>	Estimate power spectral density using a periodogram.
<code>welch(x[, fs, window, nperseg, noverlap, ...])</code>	Estimate power spectral density using Welch's method.
<code>csd(x, y[, fs, window, nperseg, noverlap, ...])</code>	Estimate the cross power spectral density, $P_{xy}$ , using Welch's method.
<code>coherence(x, y[, fs, window, nperseg, ...])</code>	Estimate the magnitude squared coherence estimate, $C_{xy}$ , of discrete-time signals $X$ and $Y$ using Welch's method.
<code>spectrogram(x[, fs, window, nperseg, ...])</code>	Compute a spectrogram with consecutive Fourier transforms.
<code>lombscargle(x, y, freqs)</code>	Computes the Lomb-Scargle periodogram.
<code>vectorstrength(events, period)</code>	Determine the vector strength of the events corresponding to the given period.
<code>stft(x[, fs, window, nperseg, noverlap, ...])</code>	Compute the Short Time Fourier Transform (STFT).
<code>istft(Zxx[, fs, window, nperseg, noverlap, ...])</code>	Perform the inverse Short Time Fourier transform (iSTFT).
<code>check_COLA(window, nperseg, noverlap[, tol])</code>	Check whether the Constant OverLap Add (COLA) constraint is met.
<code>check_NOLA(window, nperseg, noverlap[, tol])</code>	Check whether the Nonzero Overlap Add (NOLA) constraint is met.

### cupyx.scipy.signal.periodogram

`cupyx.scipy.signal.periodogram(x, fs=1.0, window='boxcar', nfft=None, detrend='constant', return_onesided=True, scaling='density', axis=-1)`

Estimate power spectral density using a periodogram.

#### Parameters

- **x** (*array\_like*) – Time series of measurement values
- **fs** (*float, optional*) – Sampling frequency of the  $x$  time series. Defaults to 1.0.
- **window** (*str or tuple or array\_like, optional*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to ‘boxcar’.
- **nfft** (*int, optional*) – Length of the FFT used. If *None* the length of  $x$  will be used.
- **detrend** (*str or function or False, optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is *False*, no detrending is done. Defaults to ‘constant’.
- **return\_onesided** (*bool, optional*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Defaults to *True*, but for complex data, a two-sided spectrum is always returned.
- **scaling** (*{ 'density', 'spectrum' }, optional*) – Selects between computing the power spectral density (‘density’) where  $P_{xx}$  has units of  $V^2/Hz$  and computing the power spectrum (‘spectrum’) where  $P_{xx}$  has units of  $V^2$ , if  $x$  is measured in  $V$  and  $fs$  is measured in  $Hz$ . Defaults to ‘density’

- **axis** (*int*, *optional*) – Axis along which the periodogram is computed; the default is over the last axis (i.e. `axis=-1`).

#### Returns

- **f** (*ndarray*) – Array of sample frequencies.
- **Pxx** (*ndarray*) – Power spectral density or power spectrum of *x*.

See also:

#### *welch*

Estimate power spectral density using Welch’s method

#### *lombscargle*

Lomb-Scargle periodogram for unevenly sampled data

### cupyx.scipy.signal.welch

```
cupyx.scipy.signal.welch(x, fs=1.0, window='hann', nperseg=None, noverlap=None, nfft=None,
                        detrend='constant', return_onesided=True, scaling='density', axis=-1,
                        average='mean')
```

Estimate power spectral density using Welch’s method.

Welch’s method<sup>1</sup> computes an estimate of the power spectral density by dividing the data into overlapping segments, computing a modified periodogram for each segment and averaging the periodograms.

#### Parameters

- **x** (*array\_like*) – Time series of measurement values
- **fs** (*float*, *optional*) – Sampling frequency of the *x* time series. Defaults to 1.0.
- **window** (*str* or *tuple* or *array\_like*, *optional*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window.
- **nperseg** (*int*, *optional*) – Length of each segment. Defaults to *None*, but if *window* is *str* or *tuple*, is set to 256, and if *window* is *array\_like*, is set to the length of the window.
- **noverlap** (*int*, *optional*) – Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*.
- **nfft** (*int*, *optional*) – Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.
- **detrend** (*str* or *function* or *False*, *optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is *False*, no detrending is done. Defaults to ‘constant’.
- **return\_onesided** (*bool*, *optional*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Defaults to *True*, but for complex data, a two-sided spectrum is always returned.

---

<sup>1</sup> P. Welch, “The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”, IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

- **scaling** ({ 'density', 'spectrum' }, *optional*) – Selects between computing the power spectral density ('density') where  $P_{xx}$  has units of  $V^2/Hz$  and computing the power spectrum ('spectrum') where  $P_{xx}$  has units of  $V^2$ , if  $x$  is measured in V and  $fs$  is measured in Hz. Defaults to 'density'
- **axis** (*int*, *optional*) – Axis along which the periodogram is computed; the default is over the last axis (i.e. `axis=-1`).
- **average** ({ 'mean', 'median' }, *optional*) – Method to use when averaging periodograms. Defaults to 'mean'.

#### Returns

- **f** (*ndarray*) – Array of sample frequencies.
- **Pxx** (*ndarray*) – Power spectral density or power spectrum of  $x$ .

See also:

#### *periodogram*

Simple, optionally modified periodogram

#### *lombscargle*

Lomb-Scargle periodogram for unevenly sampled data

#### Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default Hann window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap.

If *noverlap* is 0, this method is equivalent to Bartlett's method<sup>2</sup>.

#### References

#### `cupyx.scipy.signal.csd`

`cupyx.scipy.signal.csd(x, y, fs=1.0, window='hann', nperseg=None, noverlap=None, nfft=None, detrend='constant', return_onesided=True, scaling='density', axis=-1, average='mean')`

Estimate the cross power spectral density,  $P_{xy}$ , using Welch's method.

#### Parameters

- **x** (*array\_like*) – Time series of measurement values
- **y** (*array\_like*) – Time series of measurement values
- **fs** (*float*, *optional*) – Sampling frequency of the  $x$  and  $y$  time series. Defaults to 1.0.
- **window** (*str* or *tuple* or *array\_like*, *optional*) – Desired window to use. If *window* is a string or tuple, it is passed to `get_window` to generate the window values, which are DFT-even by default. See `get_window` for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window.

<sup>2</sup> M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika, vol. 37, pp. 1-16, 1950.

- **nperseg** (*int*, *optional*) – Length of each segment. Defaults to *None*, but if *window* is *str* or *tuple*, is set to 256, and if *window* is *array\_like*, is set to the length of the window.
- **noverlap** (*int*, *optional*) – Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*.
- **nfft** (*int*, *optional*) – Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.
- **detrend** (*str* or *function* or *False*, *optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is *False*, no detrending is done. Defaults to ‘constant’.
- **return\_onesided** (*bool*, *optional*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Defaults to *True*, but for complex data, a two-sided spectrum is always returned.
- **scaling** ({ *'density'*, *'spectrum'* }, *optional*) – Selects between computing the cross spectral density (‘density’) where  $P_{xy}$  has units of  $V^2/Hz$  and computing the cross spectrum (‘spectrum’) where  $P_{xy}$  has units of  $V^2$ , if  $x$  and  $y$  are measured in  $V$  and  $fs$  is measured in  $Hz$ . Defaults to ‘density’
- **axis** (*int*, *optional*) – Axis along which the CSD is computed for both inputs; the default is over the last axis (i.e. `axis=-1`).
- **average** ({ *'mean'*, *'median'* }, *optional*) – Method to use when averaging periodograms. Defaults to ‘mean’.

#### Returns

- **f** (*ndarray*) – Array of sample frequencies.
- **Pxy** (*ndarray*) – Cross spectral density or cross power spectrum of  $x, y$ .

See also:

#### *periodogram*

Simple, optionally modified periodogram

#### *lombscargle*

Lomb-Scargle periodogram for unevenly sampled data

#### *welch*

Power spectral density by Welch’s method. [Equivalent to `csd(x,x)`]

#### *coherence*

Magnitude squared coherence by Welch’s method.

#### Notes

By convention,  $P_{xy}$  is computed with the conjugate FFT of  $X$  multiplied by the FFT of  $Y$ .

If the input series differ in length, the shorter series will be zero-padded to match.

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default Hann window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap.

**cupyx.scipy.signal.coherence**

`cupyx.scipy.signal.coherence(x, y, fs=1.0, window='hann', nperseg=None, noverlap=None, nfft=None, detrend='constant', axis=-1)`

Estimate the magnitude squared coherence estimate,  $C_{xy}$ , of discrete-time signals  $X$  and  $Y$  using Welch's method.

$C_{xy} = \text{abs}(P_{xy})^{**2} / (P_{xx} * P_{yy})$ , where  $P_{xx}$  and  $P_{yy}$  are power spectral density estimates of  $X$  and  $Y$ , and  $P_{xy}$  is the cross spectral density estimate of  $X$  and  $Y$ .

**Parameters**

- **x** (*array\_like*) – Time series of measurement values
- **y** (*array\_like*) – Time series of measurement values
- **fs** (*float, optional*) – Sampling frequency of the  $x$  and  $y$  time series. Defaults to 1.0.
- **window** (*str or tuple or array\_like, optional*) – Desired window to use. If *window* is a string or tuple, it is passed to `get_window` to generate the window values, which are DFT-even by default. See `get_window` for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be `nperseg`. Defaults to a Hann window.
- **nperseg** (*int, optional*) – Length of each segment. Defaults to `None`, but if *window* is *str* or *tuple*, is set to 256, and if *window* is *array\_like*, is set to the length of the window.
- **noverlap** (*int, optional*) – Number of points to overlap between segments. If `None`, `noverlap = nperseg // 2`. Defaults to `None`.
- **nfft** (*int, optional*) – Length of the FFT used, if a zero padded FFT is desired. If `None`, the FFT length is `nperseg`. Defaults to `None`.
- **detrend** (*str or function or False, optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is `False`, no detrending is done. Defaults to 'constant'.
- **axis** (*int, optional*) – Axis along which the coherence is computed for both inputs; the default is over the last axis (i.e. `axis=-1`).

**Returns**

- **f** (*ndarray*) – Array of sample frequencies.
- **Cxy** (*ndarray*) – Magnitude squared coherence of  $x$  and  $y$ .

See also:

***periodogram***

Simple, optionally modified periodogram

***lombscargle***

Lomb-Scargle periodogram for unevenly sampled data

***welch***

Power spectral density by Welch's method.

***csd***

Cross spectral density by Welch's method.

## Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default Hann window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap. See<sup>1</sup> and<sup>2</sup> for more information.

## References

## Examples

```
>>> import cupy as cp
>>> from cupyx.scipy.signal import butter, lfilter, coherence
>>> import matplotlib.pyplot as plt
```

Generate two test signals with some common features.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 20
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = cupy.arange(N) / fs
>>> b, a = butter(2, 0.25, 'low')
>>> x = cupy.random.normal(
...     scale=cupy.sqrt(noise_power), size=time.shape)
>>> y = lfilter(b, a, x)
>>> x += amp * cupy.sin(2*cupy.pi*freq*time)
>>> y += cupy.random.normal(
...     scale=0.1*cupy.sqrt(noise_power), size=time.shape)
```

Compute and plot the coherence.

```
>>> f, Cxy = coherence(x, y, fs, nperseg=1024)
>>> plt.semilogy(cupy.asnumpy(f), cupy.asnumpy(Cxy))
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('Coherence')
>>> plt.show()
```

## cupyx.scipy.signal.spectrogram

`cupyx.scipy.signal.spectrogram`(*x*, *fs*=1.0, *window*=('tukey', 0.25), *nperseg*=None, *noverlap*=None, *nfft*=None, *detrend*='constant', *return\_onesided*=True, *scaling*='density', *axis*=-1, *mode*='psd')

Compute a spectrogram with consecutive Fourier transforms.

Spectrograms can be used as a way of visualizing the change of a nonstationary signal's frequency content over time.

---

<sup>1</sup> P. Welch, "The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms", IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

<sup>2</sup> Stoica, Petre, and Randolph Moses, "Spectral Analysis of Signals" Prentice Hall, 2005

### Parameters

- **x** (*array\_like*) – Time series of measurement values
- **fs** (*float, optional*) – Sampling frequency of the *x* time series. Defaults to 1.0.
- **window** (*str or tuple or array\_like, optional*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to a Tukey window with shape parameter of 0.25.
- **nperseg** (*int, optional*) – Length of each segment. Defaults to *None*, but if *window* is *str* or *tuple*, is set to 256, and if *window* is *array\_like*, is set to the length of the window.
- **noverlap** (*int, optional*) – Number of points to overlap between segments. If *None*, *noverlap* = *nperseg* // 8. Defaults to *None*.
- **nfft** (*int, optional*) – Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.
- **detrend** (*str or function or False, optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is *False*, no detrending is done. Defaults to 'constant'.
- **return\_onesided** (*bool, optional*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Defaults to *True*, but for complex data, a two-sided spectrum is always returned.
- **scaling** (*{ 'density', 'spectrum' }, optional*) – Selects between computing the power spectral density ('density') where *Sxx* has units of  $V^2/Hz$  and computing the power spectrum ('spectrum') where *Sxx* has units of  $V^2$ , if *x* is measured in *V* and *fs* is measured in *Hz*. Defaults to 'density'.
- **axis** (*int, optional*) – Axis along which the spectrogram is computed; the default is over the last axis (i.e. *axis=-1*).
- **mode** (*str, optional*) – Defines what kind of return values are expected. Options are ['psd', 'complex', 'magnitude', 'angle', 'phase']. 'complex' is equivalent to the output of *stft* with no padding or boundary extension. 'magnitude' returns the absolute magnitude of the STFT. 'angle' and 'phase' return the complex angle of the STFT, with and without unwrapping, respectively.

### Returns

- **f** (*ndarray*) – Array of sample frequencies.
- **t** (*ndarray*) – Array of segment times.
- **Sxx** (*ndarray*) – Spectrogram of *x*. By default, the last axis of *Sxx* corresponds to the segment times.

See also:

#### *periodogram*

Simple, optionally modified periodogram

#### *lombscargle*

Lomb-Scargle periodogram for unevenly sampled data

#### *welch*

Power spectral density by Welch's method.

### *csd*

Cross spectral density by Welch's method.

### Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. In contrast to welch's method, where the entire data stream is averaged over, one may wish to use a smaller overlap (or perhaps none at all) when computing a spectrogram, to maintain some statistical independence between individual segments. It is for this reason that the default window is a Tukey window with 1/8th of a window's length overlap at each end. See<sup>1</sup> for more information.

### References

### Examples

```
>>> import cupy
>>> from cupyx.scipy.signal import spectrogram
>>> import matplotlib.pyplot as plt
```

Generate a test signal, a 2 Vrms sine wave whose frequency is slowly modulated around 3kHz, corrupted by white noise of exponentially decreasing magnitude sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2 * cupy.sqrt(2)
>>> noise_power = 0.01 * fs / 2
>>> time = cupy.arange(N) / float(fs)
>>> mod = 500 * cupy.cos(2 * cupy.pi * 0.25 * time)
>>> carrier = amp * cupy.sin(2 * cupy.pi * 3e3 * time + mod)
>>> noise = cupy.random.normal(
...     scale=cupy.sqrt(noise_power), size=time.shape)
>>> noise *= cupy.exp(-time/5)
>>> x = carrier + noise
```

Compute and plot the spectrogram.

```
>>> f, t, Sxx = spectrogram(x, fs)
>>> plt.pcolormesh(cupy.asnumpy(t), cupy.asnumpy(f), cupy.asnumpy(Sxx))
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```

Note, if using output that is not one sided, then use the following:

```
>>> f, t, Sxx = spectrogram(x, fs, return_onesided=False)
>>> plt.pcolormesh(cupy.asnumpy(t), cupy.fft.fftshift(f), cupy.fft.
...     fftshift(Sxx, axes=0))
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```

---

<sup>1</sup> Oppenheim, Alan V., Ronald W. Schaffer, John R. Buck "Discrete-Time Signal Processing", Prentice Hall, 1999.



## cupyx.scipy.signal.lombscargle

`cupyx.scipy.signal.lombscargle(x, y, freqs)`

Computes the Lomb-Scargle periodogram.

The Lomb-Scargle periodogram was developed by Lomb<sup>1</sup> and further extended by Scargle<sup>2</sup> to find, and test the significance of weak periodic signals with uneven temporal sampling.

When *normalize* is False (default) the computed periodogram is unnormalized, it takes the value  $(A^{**2}) * N/4$  for a harmonic signal with amplitude A for sufficiently large N.

When *normalize* is True the computed periodogram is normalized by the residuals of the data around a constant reference model (at zero).

Input arrays should be one-dimensional and will be cast to float64.

### Parameters

- **x** (*array\_like*) – Sample times.
- **y** (*array\_like*) – Measurement values.
- **freqs** (*array\_like*) – Angular frequencies for output periodogram.
- **precenter** (*bool, optional*) – Pre-center amplitudes by subtracting the mean.
- **normalize** (*bool, optional*) – Compute normalized periodogram.

### Returns

**pgram** – Lomb-Scargle periodogram.

### Return type

*array\_like*

### Raises

**ValueError** – If the input arrays *x* and *y* do not have the same shape.

## Notes

This subroutine calculates the periodogram using a slightly modified algorithm due to Townsend<sup>3</sup> which allows the periodogram to be calculated using only a single pass through the input arrays for each frequency. The algorithm running time scales roughly as  $O(x * \text{freqs})$  or  $O(N^2)$  for a large number of samples and frequencies.

## References

See also:

### *istft*

Inverse Short Time Fourier Transform

### *check\_COLA*

Check whether the Constant OverLap Add (COLA) constraint is met

### *welch*

Power spectral density by Welch's method

<sup>1</sup> N.R. Lomb “Least-squares frequency analysis of unequally spaced data”, Astrophysics and Space Science, vol 39, pp. 447-462, 1976

<sup>2</sup> J.D. Scargle “Studies in astronomical time series analysis. II - Statistical aspects of spectral analysis of unevenly spaced data”, The Astrophysical Journal, vol 263, pp. 835-853, 1982

<sup>3</sup> R.H.D. Townsend, “Fast calculation of the Lomb-Scargle periodogram using graphics processing units.”, The Astrophysical Journal Supplement Series, vol 191, pp. 247-253, 2010

**spectrogram**

Spectrogram by Welch's method

**csd**

Cross spectral density by Welch's method

**cupyx.scipy.signal.vectorstrength**

`cupyx.scipy.signal.vectorstrength(events, period)`

Determine the vector strength of the events corresponding to the given period.

The vector strength is a measure of phase synchrony, how well the timing of the events is synchronized to a single period of a periodic signal.

If multiple periods are used, calculate the vector strength of each. This is called the “resonating vector strength”.

**Parameters**

- **events** (*1D array\_like*) – An array of time points containing the timing of the events.
- **period** (*float or array\_like*) – The period of the signal that the events should synchronize to. The period is in the same units as *events*. It can also be an array of periods, in which case the outputs are arrays of the same length.

**Returns**

- **strength** (*float or 1D array*) – The strength of the synchronization. 1.0 is perfect synchronization and 0.0 is no synchronization. If *period* is an array, this is also an array with each element containing the vector strength at the corresponding period.
- **phase** (*float or array*) – The phase that the events are most strongly synchronized to in radians. If *period* is an array, this is also an array with each element containing the phase for the corresponding period.

**Notes**

See<sup>1</sup>, <sup>2</sup> and<sup>3</sup> for more information.

**References****cupyx.scipy.signal.stft**

`cupyx.scipy.signal.stft(x, fs=1.0, window='hann', nperseg=256, noverlap=None, nfft=None, detrend=False, return_onesided=True, boundary='zeros', padded=True, axis=-1, scaling='spectrum')`

Compute the Short Time Fourier Transform (STFT).

STFTs can be used as a way of quantifying the change of a nonstationary signal's frequency and phase content over time.

**Parameters**

---

<sup>1</sup> van Hemmen, JL, Longtin, A, and Vollmayr, AN. Testing resonating vector strength: Auditory system, electric fish, and noise. *Chaos* 21, 047508 (2011).

<sup>2</sup> van Hemmen, JL. Vector strength after Goldberg, Brown, and von Mises: biological and mathematical perspectives. *Biol Cybern.* 2013 Aug;107(4):385-96.

<sup>3</sup> van Hemmen, JL and Vollmayr, AN. Resonating vector strength: what happens when we vary the “probing” frequency while keeping the spike times fixed. *Biol Cybern.* 2013 Aug;107(4):491-94.

- **x** (*array\_like*) – Time series of measurement values
- **fs** (*float, optional*) – Sampling frequency of the *x* time series. Defaults to 1.0.
- **window** (*str or tuple or array\_like, optional*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window.
- **nperseg** (*int, optional*) – Length of each segment. Defaults to 256.
- **noverlap** (*int, optional*) – Number of points to overlap between segments. If *None*, *noverlap* = *nperseg* // 2. Defaults to *None*. When specified, the COLA constraint must be met (see Notes below).
- **nfft** (*int, optional*) – Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.
- **detrend** (*str or function or False, optional*) – Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes a segment and returns a detrended segment. If *detrend* is *False*, no detrending is done. Defaults to *False*.
- **return\_onesided** (*bool, optional*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Defaults to *True*, but for complex data, a two-sided spectrum is always returned.
- **boundary** (*str or None, optional*) – Specifies whether the input signal is extended at both ends, and how to generate the new values, in order to center the first windowed segment on the first input point. This has the benefit of enabling reconstruction of the first input point when the employed window function starts at zero. Valid options are ['even', 'odd', 'constant', 'zeros', *None*]. Defaults to 'zeros', for zero padding extension. I.e. [1, 2, 3, 4] is extended to [0, 1, 2, 3, 4, 0] for *nperseg*=3.
- **padded** (*bool, optional*) – Specifies whether the input signal is zero-padded at the end to make the signal fit exactly into an integer number of window segments, so that all of the signal is included in the output. Defaults to *True*. Padding occurs after boundary extension, if *boundary* is not *None*, and *padded* is *True*, as is the default.
- **axis** (*int, optional*) – Axis along which the STFT is computed; the default is over the last axis (i.e. *axis*=-1).
- **scaling** ({'spectrum', 'psd'}) – The default 'spectrum' scaling allows each frequency line of *Zxx* to be interpreted as a magnitude spectrum. The 'psd' option scales each line to a power spectral density - it allows to calculate the signal's energy by numerically integrating over  $\text{abs}(Z_{xx})^2$ .

#### Returns

- **f** (*ndarray*) – Array of sample frequencies.
- **t** (*ndarray*) – Array of segment times.
- **Zxx** (*ndarray*) – STFT of *x*. By default, the last axis of *Zxx* corresponds to the segment times.

See also:

[\*welch\*](#)

Power spectral density by Welch's method.

**spectrogram**

Spectrogram by Welch's method.

**csd**

Cross spectral density by Welch's method.

**lombscargle**

Lomb-Scargle periodogram for unevenly sampled data

**Notes**

In order to enable inversion of an STFT via the inverse STFT in *istft*, the signal windowing must obey the constraint of “Nonzero Overlap Add” (NOLA), and the input signal must have complete windowing coverage (i.e.  $(x.shape[axis] - nperseg) \% (nperseg - noverlap) == 0$ ). The *padded* argument may be used to accomplish this.

Given a time-domain signal  $x[n]$ , a window  $w[n]$ , and a hop size  $H = nperseg - noverlap$ , the windowed frame at time index  $t$  is given by

$$x_t[n] = x[n]w[n - tH]$$

The overlap-add (OLA) reconstruction equation is given by

$$x[n] = \frac{\sum_t x_t[n]w[n - tH]}{\sum_t w^2[n - tH]}$$

The NOLA constraint ensures that every normalization term that appears in the denominator of the OLA reconstruction equation is nonzero. Whether a choice of *window*, *nperseg*, and *noverlap* satisfy this constraint can be tested with *check\_NOLA*.

See<sup>1,2</sup> for more information.

**References****Examples**

```
>>> import cupy
>>> import cupyx.scipy.signal import stft
>>> import matplotlib.pyplot as plt
```

Generate a test signal, a 2 Vrms sine wave whose frequency is slowly modulated around 3kHz, corrupted by white noise of exponentially decreasing magnitude sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2 * cupy.sqrt(2)
>>> noise_power = 0.01 * fs / 2
>>> time = cupy.arange(N) / float(fs)
>>> mod = 500 * cupy.cos(2 * cupy.pi * 0.25 * time)
>>> carrier = amp * cupy.sin(2 * cupy.pi * 3e3 * time + mod)
>>> noise = cupy.random.normal(scale=cupy.sqrt(noise_power),
...                             size=time.shape)
>>> noise *= cupy.exp(-time/5)
>>> x = carrier + noise
```

---

<sup>1</sup> Oppenheim, Alan V., Ronald W. Schaffer, John R. Buck “Discrete-Time Signal Processing”, Prentice Hall, 1999.

<sup>2</sup> Daniel W. Griffin, Jae S. Lim “Signal Estimation from Modified Short-Time Fourier Transform”, IEEE 1984, 10.1109/TASSP.1984.1164317

Compute and plot the STFT's magnitude.

```
>>> f, t, Zxx = stft(x, fs, nperseg=1000)
>>> plt.pcolormesh(cupy.asnumpy(t), cupy.asnumpy(f),
...               cupy.asnumpy(cupy.abs(Zxx)), vmin=0, vmax=amp)
>>> plt.title('STFT Magnitude')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```

## cupyx.scipy.signal.istft

`cupyx.scipy.signal.istft(Zxx, fs=1.0, window='hann', nperseg=None, noverlap=None, nfft=None, input_onesided=True, boundary=True, time_axis=-1, freq_axis=-2, scaling='spectrum')`

Perform the inverse Short Time Fourier transform (iSTFT).

### Parameters

- **Zxx** (*array\_like*) – STFT of the signal to be reconstructed. If a purely real array is passed, it will be cast to a complex data type.
- **fs** (*float, optional*) – Sampling frequency of the time series. Defaults to 1.0.
- **window** (*str or tuple or array\_like, optional*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window. Must match the window used to generate the STFT for faithful inversion.
- **nperseg** (*int, optional*) – Number of data points corresponding to each STFT segment. This parameter must be specified if the number of data points per segment is odd, or if the STFT was padded via *nfft* > *nperseg*. If *None*, the value depends on the shape of *Zxx* and *input\_onesided*. If *input\_onesided* is *True*, *nperseg*=2\*(*Zxx*.shape[freq\_axis] - 1). Otherwise, *nperseg*=*Zxx*.shape[freq\_axis]. Defaults to *None*.
- **noverlap** (*int, optional*) – Number of points to overlap between segments. If *None*, half of the segment length. Defaults to *None*. When specified, the COLA constraint must be met (see Notes below), and should match the parameter used to generate the STFT. Defaults to *None*.
- **nfft** (*int, optional*) – Number of FFT points corresponding to each STFT segment. This parameter must be specified if the STFT was padded via *nfft* > *nperseg*. If *None*, the default values are the same as for *nperseg*, detailed above, with one exception: if *input\_onesided* is *True* and *nperseg*==2\**Zxx*.shape[freq\_axis] - 1, *nfft* also takes on that value. This case allows the proper inversion of an odd-length unpadded STFT using *nfft*=*None*. Defaults to *None*.
- **input\_onesided** (*bool, optional*) – If *True*, interpret the input array as one-sided FFTs, such as is returned by *stft* with *return\_onesided*=*True* and *numpy.fft.rfft*. If *False*, interpret the input as a two-sided FFT. Defaults to *True*.
- **boundary** (*bool, optional*) – Specifies whether the input signal was extended at its boundaries by supplying a non-*None* boundary argument to *stft*. Defaults to *True*.

- **time\_axis** (*int*, *optional*) – Where the time segments of the STFT is located; the default is the last axis (i.e. `axis=-1`).
- **freq\_axis** (*int*, *optional*) – Where the frequency axis of the STFT is located; the default is the penultimate axis (i.e. `axis=-2`).
- **scaling** (`{'spectrum', 'psd'}`) – The default ‘spectrum’ scaling allows each frequency line of `Zxx` to be interpreted as a magnitude spectrum. The ‘psd’ option scales each line to a power spectral density - it allows to calculate the signal’s energy by numerically integrating over `abs(Zxx)**2`.

#### Returns

- **t** (*ndarray*) – Array of output data times.
- **x** (*ndarray*) – iSTFT of `Zxx`.

See also:

#### *stft*

Short Time Fourier Transform

#### *check\_COLA*

Check whether the Constant OverLap Add (COLA) constraint is met

#### *check\_NOLA*

Check whether the Nonzero Overlap Add (NOLA) constraint is met

#### Notes

In order to enable inversion of an STFT via the inverse STFT with *istft*, the signal windowing must obey the constraint of “nonzero overlap add” (NOLA):

$$\sum_t w^2[n - tH] \neq 0$$

This ensures that the normalization factors that appear in the denominator of the overlap-add reconstruction equation

$$x[n] = \frac{\sum_t x_t[n]w[n - tH]}{\sum_t w^2[n - tH]}$$

are not zero. The NOLA constraint can be checked with the *check\_NOLA* function.

An STFT which has been modified (via masking or otherwise) is not guaranteed to correspond to a exactly realizable signal. This function implements the iSTFT via the least-squares estimation algorithm detailed in<sup>2</sup>, which produces a signal that minimizes the mean squared error between the STFT of the returned signal and the modified STFT.

See<sup>1</sup>, [Page 586](#), <sup>2</sup> for more information.

---

<sup>2</sup> Daniel W. Griffin, Jae S. Lim “Signal Estimation from Modified Short-Time Fourier Transform”, IEEE 1984, 10.1109/TASSP.1984.1164317

<sup>1</sup> Oppenheim, Alan V., Ronald W. Schaffer, John R. Buck “Discrete-Time Signal Processing”, Prentice Hall, 1999.

## References

### Examples

```
>>> import cupy
>>> from cupyx.scipy.signal import stft, istft
>>> import matplotlib.pyplot as plt
```

Generate a test signal, a 2 Vrms sine wave at 50Hz corrupted by 0.001 V\*\*2/Hz of white noise sampled at 1024 Hz.

```
>>> fs = 1024
>>> N = 10*fs
>>> nperseg = 512
>>> amp = 2 * np.sqrt(2)
>>> noise_power = 0.001 * fs / 2
>>> time = cupy.arange(N) / float(fs)
>>> carrier = amp * cupy.sin(2*cupy.pi*50*time)
>>> noise = cupy.random.normal(scale=cupy.sqrt(noise_power),
...                           size=time.shape)
>>> x = carrier + noise
```

Compute the STFT, and plot its magnitude

```
>>> f, t, Zxx = cusignal.stft(x, fs=fs, nperseg=nperseg)
>>> f = cupy.asnumpy(f)
>>> t = cupy.asnumpy(t)
>>> Zxx = cupy.asnumpy(Zxx)
>>> plt.figure()
>>> plt.pcolormesh(t, f, np.abs(Zxx), vmin=0, vmax=amp, shading='gouraud')
>>> plt.ylim([f[1], f[-1]])
>>> plt.title('STFT Magnitude')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.yscale('log')
>>> plt.show()
```

Zero the components that are 10% or less of the carrier magnitude, then convert back to a time series via inverse STFT

```
>>> Zxx = cupy.where(cupy.abs(Zxx) >= amp/10, Zxx, 0)
>>> _, xrec = cusignal.istft(Zxx, fs)
>>> xrec = cupy.asnumpy(xrec)
>>> x = cupy.asnumpy(x)
>>> time = cupy.asnumpy(time)
>>> carrier = cupy.asnumpy(carrier)
```

Compare the cleaned signal with the original and true carrier signals.

```
>>> plt.figure()
>>> plt.plot(time, x, time, xrec, time, carrier)
>>> plt.xlim([2, 2.1])*+
>>> plt.xlabel('Time [sec]')
>>> plt.ylabel('Signal')
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend(['Carrier + Noise', 'Filtered via STFT', 'True Carrier'])
>>> plt.show()
```

Note that the cleaned signal does not start as abruptly as the original, since some of the coefficients of the transient were also removed:

```
>>> plt.figure()
>>> plt.plot(time, x, time, xrec, time, carrier)
>>> plt.xlim([0, 0.1])
>>> plt.xlabel('Time [sec]')
>>> plt.ylabel('Signal')
>>> plt.legend(['Carrier + Noise', 'Filtered via STFT', 'True Carrier'])
>>> plt.show()
```

## cupyx.scipy.signal.check\_COLA

cupyx.scipy.signal.**check\_COLA**(*window*, *nperseg*, *noverlap*, *tol=1e-10*)

Check whether the Constant OverLap Add (COLA) constraint is met.

### Parameters

- **window** (*str* or *tuple* or *array\_like*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*.
- **nperseg** (*int*) – Length of each segment.
- **noverlap** (*int*) – Number of points to overlap between segments.
- **tol** (*float*, *optional*) – The allowed variance of a bin’s weighted sum from the median bin sum.

### Returns

**verdict** – *True* if chosen combination satisfies COLA within *tol*, *False* otherwise

### Return type

*bool*

See also:

### *check\_NOLA*

Check whether the Nonzero Overlap Add (NOLA) constraint is met

### *stft*

Short Time Fourier Transform

### *istft*

Inverse Short Time Fourier Transform



## Notes

In order to enable inversion of an STFT via the inverse STFT in *istft*, it is sufficient that the signal windowing obeys the constraint of “Constant OverLap Add” (COLA). This ensures that every point in the input data is equally weighted, thereby avoiding aliasing and allowing full reconstruction.

### Some examples of windows that satisfy COLA:

- Rectangular window at overlap of 0, 1/2, 2/3, 3/4, ...
- Bartlett window at overlap of 1/2, 3/4, 5/6, ...
- Hann window at 1/2, 2/3, 3/4, ...
- Any Blackman family window at 2/3 overlap
- Any window with `noverlap = nperseg-1`

A very comprehensive list of other windows may be found in<sup>2</sup>, wherein the COLA condition is satisfied when the “Amplitude Flatness” is unity. See<sup>1</sup> for more information.

## References

### cupyx.scipy.signal.check\_NOLA

`cupyx.scipy.signal.check_NOLA(window, nperseg, noverlap, tol=1e-10)`

Check whether the Nonzero Overlap Add (NOLA) constraint is met.

#### Parameters

- **window** (*str* or *tuple* or *array\_like*) – Desired window to use. If *window* is a string or tuple, it is passed to *get\_window* to generate the window values, which are DFT-even by default. See *get\_window* for a list of windows and required parameters. If *window* is *array\_like* it will be used directly as the window and its length must be *nperseg*.
- **nperseg** (*int*) – Length of each segment.
- **noverlap** (*int*) – Number of points to overlap between segments.
- **tol** (*float*, *optional*) – The allowed variance of a bin’s weighted sum from the median bin sum.

#### Returns

**verdict** – *True* if chosen combination satisfies the NOLA constraint within *tol*, *False* otherwise

#### Return type

*bool*

#### See also:

#### *check\_COLA*

Check whether the Constant OverLap Add (COLA) constraint is met

#### *stft*

Short Time Fourier Transform

#### *istft*

Inverse Short Time Fourier Transform

<sup>2</sup> G. Heinzel, A. Ruediger and R. Schilling, “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows”, 2002, <http://hdl.handle.net/11858/00-001M-0000-0013-557A-5>

<sup>1</sup> Julius O. Smith III, “Spectral Audio Signal Processing”, W3K Publishing, 2011, ISBN 978-0-9745607-3-1.

## Notes

In order to enable inversion of an STFT via the inverse STFT in *istft*, the signal windowing must obey the constraint of “nonzero overlap add” (NOLA):

$$\sum_t w^2[n - tH] \neq 0$$

for all  $n$ , where  $w$  is the window function,  $t$  is the frame index, and  $H$  is the hop size ( $H = nperseg - noverlap$ ).

This ensures that the normalization factors in the denominator of the overlap-add inversion equation are not zero. Only very pathological windows will fail the NOLA constraint.

See<sup>1,2</sup> for more information.

## References

### Chirp Z-transform and Zoom FFT

<code>czt(x[, m, w, a, axis])</code>	Compute the frequency response around a spiral in the Z plane.
<code>zoom_fft(x, fn[, m, fs, endpoint, axis])</code>	Compute the DFT of $x$ only for frequencies in range $fn$ .
<code>CZT(n[, m, w, a])</code>	Create a callable chirp z-transform function.
<code>ZoomFFT(n, fn[, m, fs, endpoint])</code>	Create a callable zoom FFT transform function.
<code>czt_points(m[, w, a])</code>	Return the points at which the chirp z-transform is computed.

### cupyx.scipy.signal.czt

`cupyx.scipy.signal.czt(x, m=None, w=None, a=1 + 0j, *, axis=-1)`

Compute the frequency response around a spiral in the Z plane.

#### Parameters

- **x** (*array*) – The signal to transform.
- **m** (*int, optional*) – The number of output points desired. Default is the length of the input data.
- **w** (*complex, optional*) – The ratio between points in each step. This must be precise or the accumulated error will degrade the tail of the output sequence. Defaults to equally spaced points around the entire unit circle.
- **a** (*complex, optional*) – The starting point in the complex plane. Default is  $1+0j$ .
- **axis** (*int, optional*) – Axis over which to compute the FFT. If not given, the last axis is used.

#### Returns

**out** – An array of the same dimensions as  $x$ , but with the length of the transformed axis set to  $m$ .

#### Return type

*ndarray*

<sup>1</sup> Julius O. Smith III, “Spectral Audio Signal Processing”, W3K Publishing, 2011, ISBN 978-0-9745607-3-1.

<sup>2</sup> G. Heinzel, A. Ruediger and R. Schilling, “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows”, 2002, <http://hdl.handle.net/11858/00-001M-0000-0013-557A-5>

See also:

*CZT*

Class that creates a callable chirp z-transform function.

*zoom\_fft*

Convenience function for partial FFT calculations.

`scipy.signal.czt`

## Notes

The defaults are chosen such that `signal.czt(x)` is equivalent to `fft.fft(x)` and, if `m > len(x)`, that `signal.czt(x, m)` is equivalent to `fft.fft(x, m)`.

If the transform needs to be repeated, use *CZT* to construct a specialized transform function which can be reused without recomputing constants.

An example application is in system identification, repeatedly evaluating small slices of the z-transform of a system, around where a pole is expected to exist, to refine the estimate of the pole's true location.<sup>1</sup>

## References

### `cupyx.scipy.signal.zoom_fft`

`cupyx.scipy.signal.zoom_fft(x, fn, m=None, *, fs=2, endpoint=False, axis=-1)`

Compute the DFT of *x* only for frequencies in range *fn*.

#### Parameters

- **x** (*array*) – The signal to transform.
- **fn** (*array\_like*) – A length-2 sequence [*f1*, *f2*] giving the frequency range, or a scalar, for which the range [0, *fn*] is assumed.
- **m** (*int*, *optional*) – The number of points to evaluate. The default is the length of *x*.
- **fs** (*float*, *optional*) – The sampling frequency. If *fs*=10 represented 10 kHz, for example, then *f1* and *f2* would also be given in kHz. The default sampling frequency is 2, so *f1* and *f2* should be in the range [0, 1] to keep the transform below the Nyquist frequency.
- **endpoint** (*bool*, *optional*) – If True, *f2* is the last sample. Otherwise, it is not included. Default is False.
- **axis** (*int*, *optional*) – Axis over which to compute the FFT. If not given, the last axis is used.

#### Returns

**out** – The transformed signal. The Fourier transform will be calculated at the points *f1*, *f1*+*df*, *f1*+2*df*, ..., *f2*, where *df*=(*f2*-*f1*)/*m*.

#### Return type

*ndarray*

See also:

<sup>1</sup> Steve Alan Shilling, "A study of the chirp z-transform and its applications", pg 20 (1970) <https://krex.k-state.edu/dspace/bitstream/handle/2097/7844/LD2668R41972S43.pdf>

**ZoomFFT**

Class that creates a callable partial FFT function.

```
scipy.signal.zoom_fft
```

**Notes**

The defaults are chosen such that `signal.zoom_fft(x, 2)` is equivalent to `fft.fft(x)` and, if `m > len(x)`, that `signal.zoom_fft(x, 2, m)` is equivalent to `fft.fft(x, m)`.

To graph the magnitude of the resulting transform, use:

```
plot(linspace(f1, f2, m, endpoint=False),  
     abs(zoom_fft(x, [f1, f2], m)))
```

If the transform needs to be repeated, use *ZoomFFT* to construct a specialized transform function which can be reused without recomputing constants.

**cupyx.scipy.signal.CZT**

```
class cupyx.scipy.signal.CZT(n, m=None, w=None, a=1 + 0j)
```

Create a callable chirp z-transform function.

Transform to compute the frequency response around a spiral. Objects of this class are callables which can compute the chirp z-transform on their inputs. This object precalculates the constant chirps used in the given transform.

**Parameters**

- **n** (*int*) – The size of the signal.
- **m** (*int*, *optional*) – The number of output points desired. Default is *n*.
- **w** (*complex*, *optional*) – The ratio between points in each step. This must be precise or the accumulated error will degrade the tail of the output sequence. Defaults to equally spaced points around the entire unit circle.
- **a** (*complex*, *optional*) – The starting point in the complex plane. Default is  $1+0j$ .

**Returns**

**f** – Callable object `f(x, axis=-1)` for computing the chirp z-transform on *x*.

**Return type**

*CZT*

See also:

**czt**

Convenience function for quickly calculating CZT.

**ZoomFFT**

Class that creates a callable partial FFT function.

```
scipy.signal.CZT
```

## Notes

The defaults are chosen such that  $f(x)$  is equivalent to `fft.fft(x)` and, if  $m > \text{len}(x)$ , that  $f(x, m)$  is equivalent to `fft.fft(x, m)`.

If  $w$  does not lie on the unit circle, then the transform will be around a spiral with exponentially-increasing radius. Regardless, angle will increase linearly.

For transforms that do lie on the unit circle, accuracy is better when using *ZoomFFT*, since any numerical error in  $w$  is accumulated for long data lengths, drifting away from the unit circle.

The chirp z-transform can be faster than an equivalent FFT with zero padding. Try it with your own array sizes to see.

However, the chirp z-transform is considerably less precise than the equivalent zero-padded FFT.

As this CZT is implemented using the Bluestein algorithm<sup>1</sup>, it can compute large prime-length Fourier transforms in  $O(N \log N)$  time, rather than the  $O(N^2)$  time required by the direct DFT calculation. (*scipy.fft* also uses Bluestein's algorithm<sup>2</sup>.)

(The name “chirp z-transform” comes from the use of a chirp in the Bluestein algorithm<sup>2</sup>. It does not decompose signals into chirps, like other transforms with “chirp” in the name.)

## References

## Methods

`__call__(x, *, axis=-1)`

Calculate the chirp z-transform of a signal.

### Parameters

- **x** (*array*) – The signal to transform.
- **axis** (*int*, *optional*) – Axis over which to compute the FFT. If not given, the last axis is used.

### Returns

**out** – An array of the same dimensions as *x*, but with the length of the transformed axis set to *m*.

### Return type

*ndarray*

`points()`

Return the points at which the chirp z-transform is computed.

`__eq__(value, /)`

Return `self==value`.

`__ne__(value, /)`

Return `self!=value`.

`__lt__(value, /)`

Return `self<value`.

<sup>1</sup> Leo I. Bluestein, “A linear filtering approach to the computation of the discrete Fourier transform,” Northeast Electronics Research and Engineering Meeting Record 10, 218-219 (1968).

<sup>2</sup> Rabiner, Schafer, and Rader, “The chirp z-transform algorithm and its application,” Bell Syst. Tech. J. 48, 1249-1292 (1969).

`__le__(value, /)`  
Return `self <= value`.

`__gt__(value, /)`  
Return `self > value`.

`__ge__(value, /)`  
Return `self >= value`.

## `cupyx.scipy.signal.ZoomFFT`

**class** `cupyx.scipy.signal.ZoomFFT(n, fn, m=None, *, fs=2, endpoint=False)`

Create a callable zoom FFT transform function.

This is a specialization of the chirp z-transform (CZT) for a set of equally-spaced frequencies around the unit circle, used to calculate a section of the FFT more efficiently than calculating the entire FFT and truncating.<sup>1</sup>

### Parameters

- **n** (*int*) – The size of the signal.
- **fn** (*array\_like*) – A length-2 sequence  $[f1, f2]$  giving the frequency range, or a scalar, for which the range  $[0, fn]$  is assumed.
- **m** (*int, optional*) – The number of points to evaluate. Default is *n*.
- **fs** (*float, optional*) – The sampling frequency. If `fs=10` represented 10 kHz, for example, then *f1* and *f2* would also be given in kHz. The default sampling frequency is 2, so *f1* and *f2* should be in the range  $[0, 1]$  to keep the transform below the Nyquist frequency.
- **endpoint** (*bool, optional*) – If True, *f2* is the last sample. Otherwise, it is not included. Default is False.

### Returns

**f** – Callable object `f(x, axis=-1)` for computing the zoom FFT on *x*.

### Return type

*ZoomFFT*

See also:

### `zoom_fft`

Convenience function for calculating a zoom FFT.

### `scipy.signal.ZoomFFT`

---

<sup>1</sup> Steve Alan Shilling, “A study of the chirp z-transform and its applications”, pg 29 (1970) <https://krex.k-state.edu/dspace/bitstream/handle/2097/7844/LD2668R41972S43.pdf>

## Notes

The defaults are chosen such that  $f(x, 2)$  is equivalent to `fft.fft(x)` and, if  $m > \text{len}(x)$ , that  $f(x, 2, m)$  is equivalent to `fft.fft(x, m)`.

Sampling frequency is  $1/\text{dt}$ , the time step between samples in the signal  $x$ . The unit circle corresponds to frequencies from 0 up to the sampling frequency. The default sampling frequency of 2 means that  $f1, f2$  values up to the Nyquist frequency are in the range  $[0, 1)$ . For  $f1, f2$  values expressed in radians, a sampling frequency of  $2\pi$  should be used.

Remember that a zoom FFT can only interpolate the points of the existing FFT. It cannot help to resolve two separate nearby frequencies. Frequency resolution can only be increased by increasing acquisition time.

These functions are implemented using Bluestein's algorithm (as is `scipy.fft`).<sup>2</sup>

## References

### Methods

`__call__(x, *, axis=-1)`

Calculate the chirp z-transform of a signal.

#### Parameters

- **x** (*array*) – The signal to transform.
- **axis** (*int*, *optional*) – Axis over which to compute the FFT. If not given, the last axis is used.

#### Returns

**out** – An array of the same dimensions as  $x$ , but with the length of the transformed axis set to  $m$ .

#### Return type

*ndarray*

`points()`

Return the points at which the chirp z-transform is computed.

`__eq__(value, /)`

Return `self==value`.

`__ne__(value, /)`

Return `self!=value`.

`__lt__(value, /)`

Return `self<value`.

`__le__(value, /)`

Return `self<=value`.

`__gt__(value, /)`

Return `self>value`.

`__ge__(value, /)`

Return `self>=value`.

<sup>2</sup> Leo I. Bluestein, "A linear filtering approach to the computation of the discrete Fourier transform," Northeast Electronics Research and Engineering Meeting Record 10, 218-219 (1968).

## cupyx.scipy.signal.czt\_points

`cupyx.scipy.signal.czt_points(m, w=None, a=1 + 0j)`

Return the points at which the chirp z-transform is computed.

### Parameters

- **m** (*int*) – The number of points desired.
- **w** (*complex*, *optional*) – The ratio between points in each step. Defaults to equally spaced points around the entire unit circle.
- **a** (*complex*, *optional*) – The starting point in the complex plane. Default is 1+0j.

### Returns

**out** – The points in the Z plane at which *CZT* samples the z-transform, when called with arguments *m*, *w*, and *a*, as complex numbers.

### Return type

*ndarray*

See also:

### *CZT*

Class that creates a callable chirp z-transform function.

### *czt*

Convenience function for quickly calculating CZT.

`scipy.signal.czt_points`

## 5.4.7 Signal processing windows (`cupyx.scipy.signal.windows`)

The suite of window functions for filtering and spectral estimation.

---

**Hint:** SciPy API Reference: Signal processing windows (`scipy.signal.windows`)

---



<code>get_window(window, Nx[, fftbins])</code>	Return a window of a given length and type.
<code>barthann(M[, sym])</code>	Return a modified Bartlett-Hann window.
<code>bartlett(M[, sym])</code>	Return a Bartlett window.
<code>blackman(M[, sym])</code>	Return a Blackman window.
<code>blackmanharris(M[, sym])</code>	Return a minimum 4-term Blackman-Harris window.
<code>bohman(M[, sym])</code>	Return a Bohman window.
<code>boxcar(M[, sym])</code>	Return a boxcar or rectangular window.
<code>chebwin(M, at[, sym])</code>	Return a Dolph-Chebyshev window.
<code>cosine(M[, sym])</code>	Return a window with a simple cosine shape.
<code>exponential(M[, center, tau, sym])</code>	Return an exponential (or Poisson) window.
<code>flattop(M[, sym])</code>	Return a flat top window.
<code>gaussian(M, std[, sym])</code>	Return a Gaussian window.
<code>general_cosine(M, a[, sym])</code>	Generic weighted sum of cosine terms window
<code>general_gaussian(M, p, sig[, sym])</code>	Return a window with a generalized Gaussian shape.
<code>general_hamming(M, alpha[, sym])</code>	Return a generalized Hamming window.
<code>hamming(M[, sym])</code>	Return a Hamming window.
<code>hann(M[, sym])</code>	Return a Hann window.
<code>kaiser(M, beta[, sym])</code>	Return a Kaiser window.
<code>nuttall(M[, sym])</code>	Return a minimum 4-term Blackman-Harris window according to Nuttall.
<code>parzen(M[, sym])</code>	Return a Parzen window.
<code>taylor(M[, nbar, sll, norm, sym])</code>	Return a Taylor window.
<code>triang(M[, sym])</code>	Return a triangular window.
<code>tukey(M[, alpha, sym])</code>	Return a Tukey window, also known as a tapered cosine window.

## cupyx.scipy.signal.windows.get\_window

`cupyx.scipy.signal.windows.get_window(window, Nx, fftbins=True)`

Return a window of a given length and type.

### Parameters

- **window** (*string, float, or tuple*) – The type of window to create. See below for more details.
- **Nx** (*int*) – The number of samples in the window.
- **fftbins** (*bool, optional*) – If True (default), create a “periodic” window, ready to use with *ifftshift* and be multiplied by the result of an FFT (see also *fftpack.fftfreq*). If False, create a “symmetric” window, for use in filter design.

### Returns

**get\_window** – Returns a window of length *Nx* and type *window*

### Return type

*ndarray*

## Notes

Window types:

- `boxcar()`
- `triang()`
- `blackman()`
- `hamming()`
- `hann()`
- `bartlett()`
- `flattop()`
- `parzen()`
- `bohman()`
- `blackmanharris()`
- `nuttall()`
- `barthann()`
- `kaiser()` (needs beta)
- `gaussian()` (needs standard deviation)
- `general_gaussian()` (needs power, width)
- `chebwin()` (needs attenuation)
- `exponential()` (needs decay scale)
- `tukey()` (needs taper fraction)

If the window requires no parameters, then *window* can be a string.

If the window requires parameters, then *window* must be a tuple with the first argument the string name of the window, and the next arguments the needed parameters.

If *window* is a floating point number, it is interpreted as the beta parameter of the `kaiser()` window.

Each of the window types listed above is also the name of a function that can be called directly to create a window of that type.

## Examples

```
>>> import cupyx.scipy.signal.windows
>>> cupyx.scipy.signal.windows.get_window('triang', 7)
array([ 0.125,  0.375,  0.625,  0.875,  0.875,  0.625,  0.375])
>>> cupyx.scipy.signal.windows.get_window(('kaiser', 4.0), 9)
array([0.08848053, 0.32578323, 0.63343178, 0.89640418, 1.,
       0.89640418, 0.63343178, 0.32578323, 0.08848053])
>>> cupyx.scipy.signal.windows.get_window(4.0, 9)
array([0.08848053, 0.32578323, 0.63343178, 0.89640418, 1.,
       0.89640418, 0.63343178, 0.32578323, 0.08848053])
```

**cupyx.scipy.signal.windows.barthann**

`cupyx.scipy.signal.windows.barthann(M, sym=True)`

Return a modified Bartlett-Hann window.

**Parameters**

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

**Returns**

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

**Return type**

*ndarray*

**Examples**

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.barthann(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Bartlett-Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bartlett-Hann window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

**cupyx.scipy.signal.windows.bartlett**

`cupyx.scipy.signal.windows.bartlett(M, sym=True)`

Return a Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

**Parameters**

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

**Returns**

**w** – The triangular window, with the first and last samples equal to zero and the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

**Return type**

*ndarray*

See also:

*triang*

A triangular window that does not touch zero at the ends

**Notes**

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1} \left( \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right)$$

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The Fourier transform of the Bartlett is the product of two sinc functions. Note the excellent discussion in Kanasewich.<sup>2</sup>

For more information, see<sup>1</sup>, <sup>Page 600</sup>, <sup>2</sup>, <sup>3</sup>, <sup>4</sup> and<sup>5</sup>

**References****Examples**

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.bartlett(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Bartlett window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

---

<sup>2</sup> E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 109-110.

<sup>1</sup> M.S. Bartlett, “Periodogram Analysis and Continuous Spectra”, Biometrika 37, 1-16, 1950.

<sup>3</sup> A.V. Oppenheim and R.W. Schaffer, “Discrete-Time Signal Processing”, Prentice-Hall, 1999, pp. 468-471.

<sup>4</sup> Wikipedia, “Window function”, [https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

<sup>5</sup> W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 429.

```

>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bartlett window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")

```

## cupyx.scipy.signal.windows.blackman

cupyx.scipy.signal.windows.**blackman**(*M*, *sym*=True)

Return a Blackman window.

The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

The Blackman window is defined as

$$w(n) = 0.42 - 0.5 \cos(2\pi n/M) + 0.08 \cos(4\pi n/M)$$

The “exact Blackman” window was designed to null out the third and fourth sidelobes, but has discontinuities at the boundaries, resulting in a 6 dB/oct fall-off. This window is an approximation of the “exact” window, which does not null the sidelobes as well, but is smooth at the edges, improving the fall-off rate to 18 dB/oct.<sup>3</sup>

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a “near optimal” tapering function, almost as good (by some measures) as the Kaiser window.

For more information, see<sup>1, 2</sup>, and<sup>Page 601, 3</sup>

<sup>3</sup> Harris, Fredric J. (Jan 1978). “On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform”. Proceedings of the IEEE 66 (1): 51-83. [10.1109/PROC.1978.10837](https://doi.org/10.1109/PROC.1978.10837)

<sup>1</sup> Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

<sup>2</sup> Oppenheim, A.V., and R.W. Schaffer. Discrete-Time Signal Processing. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

## References

### Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal import blackman
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = blackman(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Blackman window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = cupy.abs(fftshift(A / cupy.abs(A).max()))
>>> response = 20 * cupy.log10(cupy.maximum(response, 1e-10))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Blackman window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.blackmanharris

cupyx.scipy.signal.windows.**blackmanharris**(*M*, *sym*=True)

Return a minimum 4-term Blackman-Harris window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import blackmanharris
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = blackmanharris(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Blackman-Harris window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Blackman-Harris window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.bohman

cupyx.scipy.signal.windows.**bohman**(*M*, *sym*=True)

Return a Bohman window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import bohman
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = bohman(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Bohman window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bohman window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

### cupyx.scipy.signal.windows.boxcar

`cupyx.scipy.signal.windows.boxcar(M, sym=True)`

Return a boxcar or rectangular window.

Also known as a rectangular window or Dirichlet window, this is equivalent to no window at all.

#### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – Whether the window is symmetric. (Has no effect for boxcar.)

#### Returns

**w** – The window, with the maximum value normalized to 1.

#### Return type

*ndarray*

### Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import boxcar
>>> import cupy
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = boxcar(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Boxcar window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
```

(continues on next page)



(continued from previous page)

```

>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the boxcar window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")

```

## cupyx.scipy.signal.windows.chebwin

cupyx.scipy.signal.windows.chebwin(*M*, *at*, *sym=True*)

Return a Dolph-Chebyshev window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **at** (*float*) – Attenuation (in dB).
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value always normalized to 1

### Return type

*ndarray*

## Notes

This window optimizes for the narrowest main lobe width for a given order *M* and sidelobe equiripple attenuation *at*, using Chebyshev polynomials. It was originally developed by Dolph to optimize the directionality of radio antenna arrays.

Unlike most windows, the Dolph-Chebyshev is defined in terms of its frequency response:

$$W(k) = \frac{\cos\{M \cos^{-1}[\beta \cos(\frac{\pi k}{M})]\}}{\cosh[M \cosh^{-1}(\beta)]}$$

where

$$\beta = \cosh \left[ \frac{1}{M} \cosh^{-1}(10^{\frac{A}{20}}) \right]$$

and  $0 \leq \text{abs}(k) \leq M-1$ . *A* is the attenuation in decibels (*at*).

The time domain window is then generated using the IFFT, so power-of-two *M* are the fastest to generate, and prime number *M* are the slowest.

The equiripple condition in the frequency domain creates impulses in the time domain, which appear at the ends of the window.

For more information, see<sup>1</sup>, <sup>2</sup> and <sup>3</sup>

<sup>1</sup> C. Dolph, “A current distribution for broadside arrays which optimizes the relationship between beam width and side-lobe level”, Proceedings of the IEEE, Vol. 34, Issue 6

<sup>2</sup> Peter Lynch, “The Dolph-Chebyshev Window: A Simple Optimal Filter”, American Meteorological Society (April 1997) <http://mathsci.ucd.ie/~plynch/Publications/Dolph.pdf>

<sup>3</sup> F. J. Harris, “On the use of windows for harmonic analysis with the discrete Fourier transforms”, Proceedings of the IEEE, Vol. 66, No. 1, January 1978

## References

### Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.chebwin(51, at=100)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Dolph-Chebyshev window (100 dB)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Dolph-Chebyshev window (100 dB)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.cosine

`cupyx.scipy.signal.windows.cosine(M, sym=True)`

Return a window with a simple cosine shape.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

New in version 0.13.0.

## Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.cosine(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Cosine window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the cosine window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.show()
```

## cupyx.scipy.signal.windows.exponential

`cupyx.scipy.signal.windows.exponential(M, center=None, tau=1.0, sym=True)`

Return an exponential (or Poisson) window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **center** (*float*, *optional*) – Parameter defining the center location of the window function. The default value if not given is `center = (M-1) / 2`. This parameter must take its default value for symmetric windows.
- **tau** (*float*, *optional*) – Parameter defining the decay. For `center = 0` use `tau = -(M-1) / ln(x)` if `x` is the fraction of the window remaining at the end.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

The Exponential window is defined as

$$w(n) = e^{-|n-center|/\tau}$$

## References

S. Gade and H. Herlufsen, “Windows to FFT analysis (Part I)”, Technical Review 3, Bruel & Kjaer, 1987.

## Examples

Plot the symmetric window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> M = 51
>>> tau = 3.0
>>> window = cupyx.scipy.signal.windows.exponential(M, tau=tau)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Exponential Window (tau=3.0)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -35, 0])
>>> plt.title("Frequency response of the Exponential window (tau=3.0)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

This function can also generate non-symmetric windows:

```
>>> tau2 = -(M-1) / np.log(0.01)
>>> window2 = cupyx.scipy.signal.windows.exponential(M, 0, tau2, False)
>>> plt.figure()
>>> plt.plot(cupy.asnumpy(window2))
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

**cupyx.scipy.signal.windows.flattop****cupyx.scipy.signal.windows.flattop**(*M*, *sym*=*True*)

Return a flat top window.

**Parameters**

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When *True* (default), generates a symmetric window, for use in filter design. When *False*, generates a periodic window, for use in spectral analysis.

**Returns**

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is *True*).

**Return type***ndarray***Notes**

Flat top windows are used for taking accurate measurements of signal amplitude in the frequency domain, with minimal scalloping error from the center of a frequency bin to its edges, compared to others. This is a 5th-order cosine window, with the 5 terms optimized to make the main lobe maximally flat.<sup>1</sup>

**References****Examples**

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import flattop
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = flattop(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Flat top window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the flat top window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

<sup>1</sup> D'Antona, Gabriele, and A. Ferrero, "Digital Signal Processing for Measurement Systems", Springer Media, 2006, p. 70 10.1007/0-387-28666-7

**cupyx.scipy.signal.windows.gaussian**`cupyx.scipy.signal.windows.gaussian(M, std, sym=True)`

Return a Gaussian window.

**Parameters**

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **std** (*float*) – The standard deviation, sigma.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

**Returns**

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

**Return type**

*ndarray*

**Notes**

The Gaussian window is defined as

$$w(n) = e^{-\frac{1}{2}\left(\frac{n}{\sigma}\right)^2}$$

**Examples**

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.gaussian(51, std=7)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title(r"Gaussian window ($\sigma$=7)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title(r"Frequency response of the Gaussian window ($\sigma$=7)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.general\_cosine

cupyx.scipy.signal.windows.**general\_cosine**(*M*, *a*, *sym*=*True*)

Generic weighted sum of cosine terms window

### Parameters

- **M** (*int*) – Number of points in the output window
- **a** (*array\_like*) – Sequence of weighting coefficients. This uses the convention of being centered on the origin, so these will typically all be positive numbers, not alternating sign.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Notes

For more information, see<sup>1</sup> and<sup>2</sup>

### References

### Examples

Heinzel describes a flat-top window named “HFT90D” with formula:<sup>Page 611, 2</sup>

$$w_j = 1 - 1.942604 \cos(z) + 1.340318 \cos(2z) - 0.440811 \cos(3z) + 0.043097 \cos(4z)$$

where

$$z = \frac{2\pi j}{N}, j = 0 \dots N - 1$$

Since this uses the convention of starting at the origin, to reproduce the window, we need to convert every other coefficient to a positive number:

```
>>> HFT90D = [1, 1.942604, 1.340318, 0.440811, 0.043097]
```

The paper states that the highest sidelobe is at -90.2 dB. Reproduce Figure 42 by plotting the window and its frequency response, and confirm the sidelobe level in red:

```
>>> from cupyx.scipy.signal.windows import general_cosine
>>> from cupy.fft import fft, fftshift
>>> import cupy
>>> import matplotlib.pyplot as plt
```

```
>>> window = general_cosine(1000, HFT90D, sym=False)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("HFT90D window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

<sup>1</sup> A. Nuttall, “Some windows with very good sidelobe behavior,” IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 29, no. 1, pp. 84-91, Feb 1981. [10.1109/TASSP.1981.1163506](https://doi.org/10.1109/TASSP.1981.1163506)

<sup>2</sup> Heinzel G. et al., “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows”, February 15, 2002 [https://holometer.fnal.gov/GH\\_FFT.pdf](https://holometer.fnal.gov/GH_FFT.pdf)

```
>>> plt.figure()
>>> A = fft(window, 10000) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = cupy.abs(fftshift(A / cupy.abs(A).max()))
>>> response = 20 * cupy.log10(cupy.maximum(response, 1e-10))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-50/1000, 50/1000, -140, 0])
>>> plt.title("Frequency response of the HFT90D window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axhline(-90.2, color='red')
>>> plt.show()
```

### cupyx.scipy.signal.windows.general\_gaussian

cupyx.scipy.signal.windows.**general\_gaussian**(*M*, *p*, *sig*, *sym=True*)

Return a window with a generalized Gaussian shape.

#### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **p** (*float*) – Shape parameter.  $p = 1$  is identical to *gaussian*,  $p = 0.5$  is the same shape as the Laplace distribution.
- **sig** (*float*) – The standard deviation, sigma.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

#### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

#### Return type

*ndarray*

### Notes

The generalized Gaussian window is defined as

$$w(n) = e^{-\frac{1}{2} \left| \frac{n}{\sigma} \right|^{2p}}$$

the half-power point is at

$$(2 \log(2))^{1/(2p)} \sigma$$



## Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.general_gaussian(51, p=1.5, sig=7)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title(r"Generalized Gaussian window (p=1.5, $\sigma$=7)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title(r"Freq. resp. of the gen. Gaussian "
...         r>window (p=1.5, $\sigma$=7)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.general\_hamming

cupyx.scipy.signal.windows.**general\_hamming**(*M*, *alpha*, *sym*=True)

Return a generalized Hamming window.

The generalized Hamming window is constructed by multiplying a rectangular window by one period of a cosine function<sup>1</sup>.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **alpha** (*float*) – The window coefficient,  $\alpha$
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

<sup>1</sup> DSPRelated, “Generalized Hamming Window Family”, [https://www.dsprelated.com/freebooks/sasp/Generalized\\_Hamming\\_Window\\_Family.html](https://www.dsprelated.com/freebooks/sasp/Generalized_Hamming_Window_Family.html)

## Notes

The generalized Hamming window is defined as

$$w(n) = \alpha - (1 - \alpha) \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

Both the common Hamming window and Hann window are special cases of the generalized Hamming window with  $\alpha = 0.54$  and  $\alpha = 0.5$ , respectively<sup>2</sup>.

See also:

*hamming*, *hann*

## Examples

The Sentinel-1A/B Instrument Processing Facility uses generalized Hamming windows in the processing of spaceborne Synthetic Aperture Radar (SAR) data<sup>3</sup>. The facility uses various values for the  $\alpha$  parameter based on operating mode of the SAR instrument. Some common  $\alpha$  values include 0.75, 0.7 and 0.52<sup>4</sup>. As an example, we plot these different windows.

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> fig1, spatial_plot = plt.subplots()
>>> spatial_plot.set_title("Generalized Hamming Windows")
>>> spatial_plot.set_ylabel("Amplitude")
>>> spatial_plot.set_xlabel("Sample")
```

```
>>> fig2, freq_plot = plt.subplots()
>>> freq_plot.set_title("Frequency Responses")
>>> freq_plot.set_ylabel("Normalized magnitude [dB]")
>>> freq_plot.set_xlabel("Normalized frequency [cycles per sample]")
```

```
>>> for alpha in [0.75, 0.7, 0.52]:
...     window = cupyx.scipy.signal.windows.general_hamming(41, alpha)
...     spatial_plot.plot(cupy.asnumpy(window), label="{:.2f}".format(alpha))
...     A = fft(window, 2048) / (len(window)/2.0)
...     freq = cupy.linspace(-0.5, 0.5, len(A))
...     response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
...     freq_plot.plot(
...         cupy.asnumpy(freq), cupy.asnumpy(response),
...         label="{:.2f}".format(alpha)
...     )
>>> freq_plot.legend(loc="upper right")
>>> spatial_plot.legend(loc="upper right")
```

---

<sup>2</sup> Wikipedia, "Window function", [https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

<sup>3</sup> Riccardo Piantanida ESA, "Sentinel-1 Level 1 Detailed Algorithm Definition", <https://sentinel.esa.int/documents/247904/1877131/Sentinel-1-Level-1-Detailed-Algorithm-Definition>

<sup>4</sup> Matthieu Bourbigot ESA, "Sentinel-1 Product Definition", <https://sentinel.esa.int/documents/247904/1877131/Sentinel-1-Product-Definition>

## References

### cupyx.scipy.signal.windows.hamming

`cupyx.scipy.signal.windows.hamming(M, sym=True)`

Return a Hamming window.

The Hamming window is a taper formed by using a raised cosine with non-zero endpoints, optimized to minimize the nearest side lobe.

#### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

#### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

#### Return type

*ndarray*

## Notes

The Hamming window is defined as

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

For more information, see<sup>1</sup>,<sup>2</sup>,<sup>3</sup> and<sup>4</sup>

## References

### Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

<sup>1</sup> Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

<sup>2</sup> E.R. Kanasevich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 109-110.

<sup>3</sup> Wikipedia, “Window function”, [https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

<sup>4</sup> W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 425.

```
>>> window = cupyx.scipy.signal.windows.hamming(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Hamming window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Hamming window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.hann

cupyx.scipy.signal.windows.**hann**(*M*, *sym*=*True*)

Return a Hann window.

The Hann window is a taper formed by using a raised cosine or sine-squared with ends that touch zero.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

The Hann window is defined as

$$w(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M-1$$

The window was named for Julius von Hann, an Austrian meteorologist. It is also known as the Cosine Bell. It is sometimes erroneously referred to as the “Hanning” window, from the use of “hann” as a verb in the original paper and confusion with the very similar Hamming window.

Most references to the Hann window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

For more information, see<sup>1, 2, 3</sup>, and<sup>4</sup>

<sup>1</sup> Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

<sup>2</sup> E.R. Kanasevich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 106-108.

<sup>3</sup> Wikipedia, “Window function”, [https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

<sup>4</sup> W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 425.

## References

## Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.hann(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = cupy.abs(fftshift(A / cupy.abs(A).max()))
>>> response = 20 * cupy.log10(np.maximum(response, 1e-10))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Hann window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.kaiser

cupyx.scipy.signal.windows.**kaiser**(*M*, *beta*, *sym*=True)

Return a Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **beta** (*float*) – Shape parameter, determines trade-off between main-lobe width and side lobe level. As beta gets large, the window narrows.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

The Kaiser window is defined as

$$w(n) = I_0 \left( \beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2},$$

where  $I_0$  is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple approximation to the DPSS window based on Bessel functions. The Kaiser window is a very good approximation to the Digital Prolate Spheroidal Sequence, or Slepian window, which is the transform which maximizes the energy in the main lobe of the window relative to total energy.

The Kaiser can approximate other windows by varying the beta parameter. (Some literature uses  $\alpha = \beta/\pi$ .)<sup>4</sup>

beta	Window shape
0	Rectangular
5	Similar to a Hamming
6	Similar to a Hann
8.6	Similar to a Blackman

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise NaNs will be returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

For more information, see<sup>1, 2, 3</sup>, and<sup>Page 618, 4</sup>

## References

### Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

---

<sup>4</sup> F. J. Harris, “On the use of windows for harmonic analysis with the discrete Fourier transform,” Proceedings of the IEEE, vol. 66, no. 1, pp. 51-83, Jan. 1978. 10.1109/PROC.1978.10837

<sup>1</sup> J. F. Kaiser, “Digital Filters” - Ch 7 in “Systems analysis by digital computer”, Editors: F.F. Kuo and J.F. Kaiser, p 218-285. John Wiley and Sons, New York, (1966).

<sup>2</sup> E.R. Kanasevich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 177-178.

<sup>3</sup> Wikipedia, “Window function”, [https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

```

>>> window = cupyx.scipy.signal.windows.kaiser(51, beta=14)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title(r"Kaiser window ($\beta=14$)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")

>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title(r"Frequency response of the Kaiser window ($\beta=14$)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")

```

### cupyx.scipy.signal.windows.nuttall

`cupyx.scipy.signal.windows.nuttall(M, sym=True)`

Return a minimum 4-term Blackman-Harris window according to Nuttall.

This variation is called “Nuttall4c” by Heinzel.<sup>2</sup>

#### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

#### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

#### Return type

*ndarray*

### Notes

For more information, see<sup>1</sup> and<sup>Page 619, 2</sup>

<sup>2</sup> Heinzel G. et al., “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows”, February 15, 2002 [https://holometer.fnal.gov/GH\\_FFT.pdf](https://holometer.fnal.gov/GH_FFT.pdf)

<sup>1</sup> A. Nuttall, “Some windows with very good sidelobe behavior,” IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 29, no. 1, pp. 84-91, Feb 1981. [10.1109/TASSP.1981.1163506](https://doi.org/10.1109/TASSP.1981.1163506)

## References

## Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import nuttall
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = nuttall(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Nuttall window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Nuttall window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.parzen

`cupyx.scipy.signal.windows.parzen(M, sym=True)`

Return a Parzen window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero, an empty array is returned. An exception is thrown when it is negative.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*



## Notes

For more information, see<sup>1</sup>.

## References

## Examples

Plot the window and its frequency response:

```
>>> import cupy as cp
>>> from cupyx.scipy import signal
>>> from cupyx.scipy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.windows.parzen(51)
>>> plt.plot(window)
>>> plt.title("Parzen window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cp.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cp.log10(cp.abs(fftshift(A / abs(A).max()))))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Parzen window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.taylor

`cupyx.scipy.signal.windows.taylor(M, nbar=4, sll=30, norm=True, sym=True)`

Return a Taylor window. The Taylor window taper function approximates the Dolph-Chebyshev window's constant sidelobe level for a parameterized number of near-in sidelobes, but then allows a taper beyond<sup>2</sup>. The SAR (synthetic aperture radar) community commonly uses Taylor weighting for image formation processing because it provides strong, selectable sidelobe suppression with minimum broadening of the mainlobe<sup>1</sup>.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **nbar** (*int*, *optional*) – Number of nearly constant level sidelobes adjacent to the mainlobe.
- **sll** (*float*, *optional*) – Desired suppression of sidelobe level in decibels (dB) relative to the DC gain of the mainlobe. This should be a positive number.

<sup>1</sup> E. Parzen, "Mathematical Considerations in the Estimation of Spectra", *Technometrics*, Vol. 3, No. 2 (May, 1961), pp. 167-190

<sup>2</sup> Armin Doerry, "Catalog of Window Taper Functions for Sidelobe Control", 2017. [https://www.researchgate.net/profile/Armin\\_Doerry/publication/316281181\\_Catalog\\_of\\_Window\\_Taper\\_Functions\\_for\\_Sidelobe\\_Control/links/58f92cb2a6fdccb121c9d54d/Catalog-of-Window-Taper-Functions-for-Sidelobe-Control.pdf](https://www.researchgate.net/profile/Armin_Doerry/publication/316281181_Catalog_of_Window_Taper_Functions_for_Sidelobe_Control/links/58f92cb2a6fdccb121c9d54d/Catalog-of-Window-Taper-Functions-for-Sidelobe-Control.pdf)

<sup>1</sup> W. Carrara, R. Goodman, and R. Majewski, "Spotlight Synthetic Aperture Radar: Signal Processing Algorithms" Pages 512-513, July 1995.

- **norm** (*bool*, *optional*) – When True (default), divides the window by the largest (middle) value for odd-length windows or the value that would occur between the two repeated middle values for even-length windows such that all values are less than or equal to 1. When False the DC gain will remain at 1 (0 dB) and the sidelobes will be *sll* dB down.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

**Returns**

**out** – The window. When *norm* is True (default), the maximum value is normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

**Return type**

array

See also:

*chebwin*, *kaiser*, *bartlett*, *blackman*, *hamming*, *hanning*

**References****Examples**

```
Plot the window and its frequency response: >>> from scipy import signal >>> from scipy.fft import
fft, fftshift >>> import matplotlib.pyplot as plt >>> window = signal.windows.taylor(51, nbar=20, sll=100,
norm=False) >>> plt.plot(window) >>> plt.title("Taylor window (100 dB)") >>> plt.ylabel("Amplitude") >>>
plt.xlabel("Sample") >>> plt.figure() >>> A = fft(window, 2048) / (len(window)/2.0) >>> freq = np.linspace(-
0.5, 0.5, len(A)) >>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())) >>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0]) >>> plt.title("Frequency response of the Taylor window (100 dB)") >>>
plt.ylabel("Normalized magnitude [dB]") >>> plt.xlabel("Normalized frequency [cycles per sample]")
```

**cupyx.scipy.signal.windows.triang**

cupyx.scipy.signal.windows.**triang**(*M*, *sym*=True)

Return a triangular window.

**Parameters**

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

**Returns**

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

**Return type**

*ndarray*

See also:

***bartlett***

A triangular window that touches zero

## Examples

Plot the window and its frequency response:

```
>>> from cupyx.scipy.signal.windows import triang
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = triang(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Triangular window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = cupy.abs(fftshift(A / cupy.abs(A).max()))
>>> response = 20 * cupy.log10(cupy.maximum(response, 1e-10))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the triangular window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## cupyx.scipy.signal.windows.tukey

`cupyx.scipy.signal.windows.tukey(M, alpha=0.5, sym=True)`

Return a Tukey window, also known as a tapered cosine window.

### Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **alpha** (*float*, *optional*) – Shape parameter of the Tukey window, representing the fraction of the window inside the cosine tapered region. If zero, the Tukey window is equivalent to a rectangular window. If one, the Tukey window is equivalent to a Hann window.
- **sym** (*bool*, *optional*) – When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

### Returns

**w** – The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

### Return type

*ndarray*

## Notes

For more information, see<sup>1</sup> and<sup>2</sup>.

## References

## Examples

Plot the window and its frequency response:

```
>>> import cupyx.scipy.signal.windows
>>> import cupy as cp
>>> from cupy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = cupyx.scipy.signal.windows.tukey(51)
>>> plt.plot(cupy.asnumpy(window))
>>> plt.title("Tukey window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.ylim([0, 1.1])
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = cupy.linspace(-0.5, 0.5, len(A))
>>> response = 20 * cupy.log10(cupy.abs(fftshift(A / cupy.abs(A).max()))))
>>> plt.plot(cupy.asnumpy(freq), cupy.asnumpy(response))
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Tukey window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

## 5.4.8 Sparse matrices (cupyx.scipy.sparse)

---

**Hint:** SciPy API Reference: Sparse matrices ([scipy.sparse](#))

---

CuPy supports sparse matrices using `cuSPARSE`. These matrices have the same interfaces of SciPy's `sparse matrices`.

---

<sup>1</sup> Harris, Fredric J. (Jan 1978). "On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform". *Proceedings of the IEEE* 66 (1): 51-83. 10.1109/PROC.1978.10837

<sup>2</sup> Wikipedia, "Window function", [https://en.wikipedia.org/wiki/Window\\_function#Tukey\\_window](https://en.wikipedia.org/wiki/Window_function#Tukey_window)

## Conversion to/from SciPy sparse matrices

`cupyx.scipy.sparse.*_matrix` and `scipy.sparse.*_matrix` are not implicitly convertible to each other. That means, SciPy functions cannot take `cupyx.scipy.sparse.*_matrix` objects as inputs, and vice versa.

- To convert SciPy sparse matrices to CuPy, pass it to the constructor of each CuPy sparse matrix class.
- To convert CuPy sparse matrices to SciPy, use `get` method of each CuPy sparse matrix class.

Note that converting between CuPy and SciPy incurs data transfer between the host (CPU) device and the GPU device, which is costly in terms of performance.

## Conversion to/from CuPy ndarrays

- To convert CuPy ndarray to CuPy sparse matrices, pass it to the constructor of each CuPy sparse matrix class.
- To convert CuPy sparse matrices to CuPy ndarray, use `toarray` of each CuPy sparse matrix instance (e.g., `cupyx.scipy.sparse.csr_matrix.toarray()`).

Converting between CuPy ndarray and CuPy sparse matrices does not incur data transfer; it is copied inside the GPU device.

## Contents

### Sparse matrix classes

<code>coo_matrix</code> (arg1[, shape, dtype, copy])	COOrdinate format sparse matrix.
<code>csc_matrix</code> (arg1[, shape, dtype, copy])	Compressed Sparse Column matrix.
<code>csr_matrix</code> (arg1[, shape, dtype, copy])	Compressed Sparse Row matrix.
<code>dia_matrix</code> (arg1[, shape, dtype, copy])	Sparse matrix with DIAGONAL storage.
<code>spmatrix</code> ([maxprint])	Base class of all sparse matrixes.

### cupyx.scipy.sparse.coo\_matrix

**class** `cupyx.scipy.sparse.coo_matrix`(arg1, shape=None, dtype=None, copy=False)

COOrdinate format sparse matrix.

This can be instantiated in several ways.

**coo\_matrix**(D)

D is a rank-2 `cupy.ndarray`.

**coo\_matrix**(S)

S is another sparse matrix. It is equivalent to `S.tocoo()`.

**coo\_matrix**((M, N), [dtype])

It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**coo\_matrix**((data, (row, col)))

All data, row and col are one-dimenaional `cupy.ndarray`.

#### Parameters

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.

- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If True, copies of given data are always used.

See also:

`scipy.sparse.coo_matrix`

## Methods

**`__len__()`**

**`__iter__()`**

**`arcsin()`**

Elementwise arcsin.

**`arcsinh()`**

Elementwise arcsinh.

**`arctan()`**

Elementwise arctan.

**`arctanh()`**

Elementwise arctanh.

**`asformat(format)`**

Return this matrix in a given sparse format.

### Parameters

**format** (*str* or *None*) – Format you need.

**`asfptype()`**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

### Returns

A matrix with float type.

### Return type

*cupyx.scipy.sparse.spmatrix*

**`astype(t)`**

Casts the array to given data type.

### Parameters

**dtype** – Type specifier.

### Returns

A copy of the array with a given type.

**`ceil()`**

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**copy**()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero**()

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns**

Number of non-zero entries.

**deg2rad**()

Elementwise deg2rad.

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*].) – 0 (the main diagonal).

**Returns**

The k-th diagonal.

**Return type***cupy.ndarray***dot(*other*)**

Ordinary dot product

**eliminate\_zeros()**

Removes zero entries in place.

**expm1()**

Elementwise expm1.

**floor()**

Elementwise floor.

**get(*stream=None*)**

Returns a copy of the array on host memory.

**Parameters****stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.**Returns**

Copy of the array on host memory.

**Return type***scipy.sparse.coo\_matrix***getH()****get\_shape()**

Returns the shape of the matrix.

**Returns**

Shape of the matrix.

**Return type***tuple***getformat()****getmaxprint()****getnnz(*axis=None*)**

Returns the number of stored values, including explicit zeros.

**log1p()**

Elementwise log1p.

**maximum(*other*)****mean(*axis=None, dtype=None, out=None*)**

Compute the arithmetic mean along the specified axis.

**Parameters****axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.**Returns**

Summed array.



**Return type***cupy.ndarray***See also:**`scipy.sparse.spmatrix.mean()`**minimum**(*other*)**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n*, *dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg**()

Elementwise rad2deg.

**reshape**(\**shape*, *order='C'*)

Gives a new shape to a sparse matrix without changing its data.

**Parameters**

- **shape** (*tuple*) – The new shape should be compatible with the original shape.
- **order** – { 'C', 'F' } (optional) Read the elements using this index order. 'C' means to read and write the elements using C-like index order. 'F' means to read and write the elements using Fortran-like index order. Default: C.

**Returns**

sparse matrix

**Return type***cupyx.scipy.sparse.coo\_matrix***rint**()

Elementwise rint.

**set\_shape**(*shape*)**setdiag**(*values*, *k=0*)

Set diagonal or off-diagonal elements of the array.

**Parameters**

- **values** (*ndarray*) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values are longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (*int*, *optional*) – Which off-diagonal to set, corresponding to elements  $a[i,i+k]$ . Default: 0 (the main diagonal).

**sign**()

Elementwise sign.

**sin()**

Elementwise sin.

**sinh()**

Elementwise sinh.

**sqrt()**

Elementwise sqrt.

**sum**(axis=None, dtype=None, out=None)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (cupy.ndarray) – Output matrix.

**Returns**

Summed array.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum\_duplicates()**

Eliminate duplicate matrix entries by adding them together.

**Warning:** When sorting the indices, CuPy follows the convention of cuSPARSE, which is different from that of SciPy. Therefore, the order of the output indices may differ:

```
>>> #      1 0 0
>>> # A = 1 1 0
>>> #      1 1 1
>>> data = cupy.array([1, 1, 1, 1, 1, 1], 'f')
>>> row = cupy.array([0, 1, 1, 2, 2, 2], 'i')
>>> col = cupy.array([0, 0, 1, 0, 1, 2], 'i')
>>> A = cupyx.scipy.sparse.coo_matrix((data, (row, col)),
...                                   shape=(3, 3))
>>> a = A.get()
>>> A.sum_duplicates()
>>> a.sum_duplicates() # a is scipy.sparse.coo_matrix
>>> A.row
array([0, 1, 1, 2, 2, 2], dtype=int32)
>>> a.row
array([0, 1, 2, 1, 2, 2], dtype=int32)
>>> A.col
array([0, 0, 1, 0, 1, 2], dtype=int32)
>>> a.col
array([0, 0, 0, 1, 1, 2], dtype=int32)
```

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.coo_matrix.sum_duplicates()`

**tan()**

Elementwise tan.

**tanh()**

Elementwise tanh.

**toarray**(*order=None, out=None*)

Returns a dense matrix representing the same value.

**Parameters**

- **order** (*str*) – Not supported.
- **out** – Not supported.

**Returns**

Dense array representing the same value.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.coo_matrix.toarray()`

**tobsr**(*blocksize=None, copy=False*)

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)

Converts the matrix to COOrdinate format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**tocsc**(*copy=False*)

Converts the matrix to Compressed Sparse Column format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csc conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**tocsr**(*copy=False*)

Converts the matrix to Compressed Sparse Row format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in coo to csr conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns**

Transpose matrix.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

`__nonzero__()``__bool__()`

### Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to `toarray()` method.

**H****T****device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format** = 'coo'**ndim****nnz****shape****size**

### `cupyx.scipy.sparse.csc_matrix`

**class** `cupyx.scipy.sparse.csc_matrix`(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Compressed Sparse Column matrix.

This can be instantiated in several ways.

**csc\_matrix(D)**

D is a rank-2 `cupy.ndarray`.

**csc\_matrix(S)**

S is another sparse matrix. It is equivalent to `S.tocsc()`.

**csc\_matrix((M, N), [dtype])**

It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**csc\_matrix((data, (row, col)))**

All data, row and col are one-dimensional `cupy.ndarray`.

**csc\_matrix((data, indices, indptr))**

All data, indices and indptr are one-dimensional `cupy.ndarray`.

#### Parameters

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.

- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If True, copies of given arrays are always used.

See also:

`scipy.sparse.csc_matrix`

## Methods

`__getitem__(key)`

`__setitem__(key, x)`

`__len__()`

`__iter__()`

`arcsin()`

Elementwise arcsin.

`arcsinh()`

Elementwise arcsinh.

`arctan()`

Elementwise arctan.

`arctanh()`

Elementwise arctanh.

`argmax(axis=None, out=None)`

Returns indices of maximum elements along an axis.

Implicit zero elements are taken into account. If there are several maximum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

### Parameters

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

### Returns

Indices of maximum elements. If array, its size along **axis** is 1.

### Return type

(`cupy.ndarray` or *int*)

`argmin(axis=None, out=None)`

Returns indices of minimum elements along an axis.

Implicit zero elements are taken into account. If there are several minimum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

### Parameters

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns**

Indices of minimum elements. If matrix, its size along *axis* is 1.

**Return type**

(cupy.ndarray or int)

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters**

**format** (*str* or *None*) – Format you need.

**asfptype**()

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns**

A matrix with float type.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**astype**(*t*)

Casts the array to given data type.

**Parameters**

**dtype** – Type specifier.

**Returns**

A copy of the array with a given type.

**ceil**()

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**copy()**

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero()**

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns**

Number of non-zero entries.

**deg2rad()**

Elementwise deg2rad.

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*].) – 0 (the main diagonal).

**Returns**

The k-th diagonal.

**Return type**

*cupy.ndarray*

**dot**(*other*)

Ordinary dot product

**eliminate\_zeros()**

Removes zero entories in place.

**expm1()**

Elementwise expm1.

**floor()**

Elementwise floor.

**get**(*stream=None*)

Returns a copy of the array on host memory.



**Warning:** You need to install SciPy to use this method.

**Parameters**

**stream** (`cupy.cuda.Stream`) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns**

Copy of the array on host memory.

**Return type**

*scipy.sparse.csc\_matrix*

**getH()**

**get\_shape()**

Returns the shape of the matrix.

**Returns**

Shape of the matrix.

**Return type**

*tuple*

**getcol(*i*)**

Returns a copy of column *i* of the matrix, as a (m x 1) CSC matrix (column vector).

**Parameters**

**i** (*integer*) – Column

**Returns**

Sparse matrix with single column

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**getformat()**

**getmaxprint()**

**getnnz(*axis=None*)**

Returns the number of stored values, including explicit zeros.

**Parameters**

**axis** – Not supported yet.

**Returns**

The number of stored values.

**Return type**

*int*

**getrow(*i*)**

Returns a copy of row *i* of the matrix, as a (1 x n) CSR matrix (row vector).

**Parameters**

**i** (*integer*) – Row

**Returns**

Sparse matrix with single row

**Return type***cupyx.scipy.sparse.csc\_matrix***log1p()**

Elementwise log1p.

**max**(*axis=None, out=None, \*, explicit=False*)

Returns the maximum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis = None*).
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the maximum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns**Maximum of *a*. If *axis* is *None*, the result is a scalar value. If *axis* is given, the result is an array of dimension *a.ndim - 1*. This differs from numpy for computational efficiency.**Return type***(cupy.ndarray or float)***See also:***min* : The minimum value of a sparse matrix along a given axis.**See also:***numpy.matrix.max* : NumPy's implementation of *max* for matrices**maximum**(*other*)**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters****axis** (*int* or *None*) – Axis along which the sum is computed. If it is *None*, it computes the average of all the elements. Select from {*None*, 0, 1, -2, -1}.**Returns**

Summed array.

**Return type***cupy.ndarray***See also:***scipy.sparse.spmatrix.mean()***min**(*axis=None, out=None, \*, explicit=False*)

Returns the minimum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis = None*).

- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the minimum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns**

Minimum of **a**. If **axis** is *None*, the result is a scalar value. If **axis** is given, the result is an array of dimension **a.ndim** - 1. This differs from `numpy` for computational efficiency.

**Return type**

(*cupy.ndarray* or *float*)

**See also:**

`max` : The maximum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.min` : NumPy's implementation of 'min' for matrices

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg**()

Elementwise `rad2deg`.

**reshape**(\**shape*, *order='C'*)

Gives a new shape to a sparse matrix without changing its data.

**Parameters**

- **shape** (*tuple*) – The new shape should be compatible with the original shape.
- **order** – { 'C', 'F' } (optional) Read the elements using this index order. 'C' means to read and write the elements using C-like index order. 'F' means to read and write the elements using Fortran-like index order. Default: C.

**Returns**

sparse matrix

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**rint**()

Elementwise `rint`.

**set\_shape**(*shape*)

**setdiag**(*values*, *k=0*)

Set diagonal or off-diagonal elements of the array.

**Parameters**

- **values** (`cupy.ndarray`) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (`int`, *optional*) – Which diagonal to set, corresponding to elements `a[i, i+k]`. Default: 0 (the main diagonal).

**sign**()

Elementwise sign.

**sin**()

Elementwise sin.

**sinh**()

Elementwise sinh.

**sort\_indices**()

Sorts the indices of this matrix *in place*.

**Warning:** Calling this function might synchronize the device.

**sorted\_indices**()

Return a copy of this matrix with sorted indices

**Warning:** Calling this function might synchronize the device.

**sqrt**()

Elementwise sqrt.

**sum**(*axis=None*, *dtype=None*, *out=None*)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns**

Summed array.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum\_duplicates()**

Eliminate duplicate matrix entries by adding them together.

---

**Note:** This is an *in place* operation.

---

**Warning:** Calling this function might synchronize the device.

**See also:**

`scipy.sparse.csr_matrix.sum_duplicates()`, `scipy.sparse.csc_matrix.sum_duplicates()`

**tan()**

Elementwise tan.

**tanh()**

Elementwise tanh.

**toarray(*order=None, out=None*)**

Returns a dense matrix representing the same value.

**Parameters**

- **order** (`{'C', 'F', None}`) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns**

Dense array representing the same matrix.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.csc_matrix.toarray()`

**tobsr(*blocksize=None, copy=False*)**

Convert this matrix to Block Sparse Row format.

**tocoo(*copy=False*)**

Converts the matrix to COOrdinate format.

**Parameters**

- **copy** (*bool*) – If `False`, it shares data arrays as much as possible.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**tocsc(*copy=None*)**

Converts the matrix to Compressed Sparse Column format.

**Parameters**

- **copy** (*bool*) – If `False`, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**tocsr**(*copy=False*)

Converts the matrix to Compressed Sparse Row format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns**

*self* with the dimensions reversed.

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return *self*==value.

**\_\_ne\_\_**(*other*)

Return *self*!=value.

**\_\_lt\_\_**(*other*)

Return *self*<value.

**\_\_le\_\_**(*other*)

Return *self*<=value.

`__gt__(other)`

Return self>value.

`__ge__(other)`

Return self>=value.

`__nonzero__()`

`__bool__()`

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to `toarray()` method.

**H**

**T**

**device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format = 'csc'**

**has\_canonical\_format**

Determine whether the matrix has sorted indices and no duplicates.

**Returns**

bool: True if the above applies, otherwise False.

---

**Note:** `has_canonical_format` implies `has_sorted_indices`, so if the latter flag is False, so will the former be; if the former is found True, the latter flag is also set.

---

**Warning:** Getting this property might synchronize the device.

**has\_sorted\_indices**

Determine whether the matrix has sorted indices.

**Returns**

**bool:**

True if the indices of the matrix are in sorted order, otherwise False.

**Warning:** Getting this property might synchronize the device.

**ndim**

**nnz**

**shape**

**size**

### `cupyx.scipy.sparse.csr_matrix`

**class** `cupyx.scipy.sparse.csr_matrix`(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Compressed Sparse Row matrix.

This can be instantiated in several ways.

**csr\_matrix**(D)

D is a rank-2 [cupy.ndarray](#).

**csr\_matrix**(S)

S is another sparse matrix. It is equivalent to `S.tocsr()`.

**csr\_matrix**((M, N), [dtype])

It constructs an empty matrix whose shape is (M, N). Default dtype is float64.

**csr\_matrix**((data, (row, col)))

All data, row and col are one-dimensional [cupy.ndarray](#).

**csr\_matrix**((data, indices, indptr))

All data, indices and indptr are one-dimensional [cupy.ndarray](#).

#### Parameters

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of [numpy.dtype](#).
- **copy** (*bool*) – If True, copies of given arrays are always used.

See also:

[scipy.sparse.csr\\_matrix](#)

#### Methods

**\_\_getitem\_\_**(*key*)

**\_\_setitem\_\_**(*key*, *x*)

**\_\_len\_\_**()

**\_\_iter\_\_**()

**arcsin**()

Elementwise arcsin.

**arcsinh**()

Elementwise arcsinh.



**arctan()**

Elementwise arctan.

**arctanh()**

Elementwise arctanh.

**argmax**(*axis=None, out=None*)

Returns indices of maximum elements along an axis.

Implicit zero elements are taken into account. If there are several maximum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns**

Indices of maximum elements. If array, its size along **axis** is 1.

**Return type**

(cupy.ndarray or *int*)

**argmin**(*axis=None, out=None*)

Returns indices of minimum elements along an axis.

Implicit zero elements are taken into account. If there are several minimum values, the index of the first occurrence is returned. If NaN values occur in the matrix, the output defaults to a zero entry for the row/column in which the NaN occurs.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

**Returns**

Indices of minimum elements. If matrix, its size along **axis** is 1.

**Return type**

(cupy.ndarray or *int*)

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters**

**format** (*str* or *None*) – Format you need.

**asfptype()**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns**

A matrix with float type.

**Return type***cupyx.scipy.sparse.spmatrix***astype(*t*)**

Casts the array to given data type.

**Parameters**

**dtype** – Type specifier.

**Returns**

A copy of the array with a given type.

**ceil()**

Elementwise ceil.

**conj(*copy=True*)**

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type***cupyx.scipy.sparse.spmatrix***conjugate(*copy=True*)**

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type***cupyx.scipy.sparse.spmatrix***copy()**

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero()**

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns**

Number of non-zero entries.

**deg2rad()**

Elementwise deg2rad.

**diagonal( $k=0$ )**

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*].) – 0 (the main diagonal).

**Returns**

The k-th diagonal.

**Return type**

*cupy.ndarray*

**dot(*other*)**

Ordinary dot product

**eliminate\_zeros()**

Removes zero entories in place.

**expm1()**

Elementwise expm1.

**floor()**

Elementwise floor.

**get(*stream=None*)**

Returns a copy of the array on host memory.

**Parameters**

**stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns**

Copy of the array on host memory.

**Return type**

*scipy.sparse.csr\_matrix*

**getH()****get\_shape()**

Returns the shape of the matrix.

**Returns**

Shape of the matrix.

**Return type**

*tuple*

**getcol(*i*)**

Returns a copy of column i of the matrix, as a (m x 1) CSR matrix (column vector).

**Parameters**

**i** (*integer*) – Column

**Returns**

Sparse matrix with single column

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**getformat()**

**getmaxprint()**

**getnnz**(*axis=None*)

Returns the number of stored values, including explicit zeros.

**Parameters**

**axis** – Not supported yet.

**Returns**

The number of stored values.

**Return type**

*int*

**getrow**(*i*)

Returns a copy of row *i* of the matrix, as a (1 x n) CSR matrix (row vector).

**Parameters**

**i** (*integer*) – Row

**Returns**

Sparse matrix with single row

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**log1p()**

Elementwise log1p.

**max**(*axis=None, out=None, \*, explicit=False*)

Returns the maximum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis = None*).
- **out** (*None*) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the maximum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns**

Maximum of *a*. If *axis* is *None*, the result is a scalar value. If *axis* is given, the result is an array of dimension *a.ndim - 1*. This differs from numpy for computational efficiency.

**Return type**

(*cupy.ndarray* or *float*)

**See also:**

*min* : The minimum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.max` : NumPy's implementation of `max` for matrices

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters**

**axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.

**Returns**

Summed array.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.mean()`

**min**(*axis=None, out=None, \*, explicit=False*)

Returns the minimum of the matrix or maximum along an axis.

**Parameters**

- **axis** (*int*) – {-2, -1, 0, 1, None} (optional) Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. `axis = None`).
- **out** (None) – (optional) This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.
- **explicit** (*bool*) – Return the minimum value explicitly specified and ignore all implicit zero entries. If the dimension has no explicit values, a zero is then returned to indicate that it is the only implicit value. This parameter is experimental and may change in the future.

**Returns**

Minimum of `a`. If `axis` is None, the result is a scalar value. If `axis` is given, the result is an array of dimension `a.ndim - 1`. This differs from `numpy` for computational efficiency.

**Return type**

(*cupy.ndarray* or *float*)

**See also:**

`max` : The maximum value of a sparse matrix along a given axis.

**See also:**

`numpy.matrix.min` : NumPy's implementation of 'min' for matrices

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix, vector or scalar

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.

- **dtype** – Type specifier.

**rad2deg()**

Elementwise rad2deg.

**reshape**(\*shape, order='C')

Gives a new shape to a sparse matrix without changing its data.

**Parameters**

- **shape** (*tuple*) – The new shape should be compatible with the original shape.
- **order** – { 'C', 'F' } (optional) Read the elements using this index order. 'C' means to read and write the elements using C-like index order. 'F' means to read and write the elements using Fortran-like index order. Default: C.

**Returns**

sparse matrix

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**rint()**

Elementwise rint.

**set\_shape**(shape)

**setdiag**(values, k=0)

Set diagonal or off-diagonal elements of the array.

**sign()**

Elementwise sign.

**sin()**

Elementwise sin.

**sinh()**

Elementwise sinh.

**sort\_indices()**

Sorts the indices of this matrix *in place*.

**Warning:** Calling this function might synchronize the device.

**sorted\_indices()**

Return a copy of this matrix with sorted indices

**Warning:** Calling this function might synchronize the device.

**sqrt()**

Elementwise sqrt.

**sum**(axis=None, dtype=None, out=None)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (`cupy.ndarray`) – Output matrix.

**Returns**

Summed array.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.sparse.spmatrix.sum()`

**sum\_duplicates()**

Eliminate duplicate matrix entries by adding them together.

---

**Note:** This is an *in place* operation.

---

<p><b>Warning:</b> Calling this function might synchronize the device.</p>
--

**See also:**

`scipy.sparse.csr_matrix.sum_duplicates()`,  
`sum_duplicates()`

`scipy.sparse.csc_matrix.`

**tan()**

Elementwise tan.

**tanh()**

Elementwise tanh.

**toarray(*order=None, out=None*)**

Returns a dense matrix representing the same value.

**Parameters**

- **order** ({'C', 'F', None}) – Whether to store data in C (row-major) order or F (column-major) order. Default is C-order.
- **out** – Not supported.

**Returns**

Dense array representing the same matrix.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.sparse.csr_matrix.toarray()`

**tobsr(*blocksize=None, copy=False*)**

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy=False*)

Converts the matrix to COOrdinate format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**tocsc**(*copy=False*)

Converts the matrix to Compressed Sparse Column format.

**Parameters**

**copy** (*bool*) – If *False*, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in csr to csc conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**tocsr**(*copy=False*)

Converts the matrix to Compressed Sparse Row format.

**Parameters**

**copy** (*bool*) – If *False*, the method returns itself. Otherwise it makes a copy of the matrix.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csr\_matrix*

**todense**(*order=None, out=None*)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Returns a transpose matrix.

**Parameters**

- **axes** – This option is not supported.
- **copy** (*bool*) – If *True*, a returned matrix shares no data. Otherwise, it shared data arrays as much as possible.

**Returns**

*self* with the dimensions reversed.



**Return type***cupyx.scipy.sparse.csc\_matrix***trunc()**

Elementwise trunc.

**\_\_eq\_\_(other)**

Return self==value.

**\_\_ne\_\_(other)**

Return self!=value.

**\_\_lt\_\_(other)**

Return self&lt;value.

**\_\_le\_\_(other)**

Return self&lt;=value.

**\_\_gt\_\_(other)**

Return self&gt;value.

**\_\_ge\_\_(other)**

Return self&gt;=value.

**\_\_nonzero\_\_()****\_\_bool\_\_()****Attributes****A**

Dense ndarray representation of this matrix.

This property is equivalent to *toarray()* method.**H****T****device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format = 'csr'****has\_canonical\_format**

Determine whether the matrix has sorted indices and no duplicates.

**Returns**

bool: True if the above applies, otherwise False.

---

**Note:** *has\_canonical\_format* implies *has\_sorted\_indices*, so if the latter flag is False, so will the former be; if the former is found True, the latter flag is also set.

---

**Warning:** Getting this property might synchronize the device.

**has\_sorted\_indices**

Determine whether the matrix has sorted indices.

**Returns****bool:**

True if the indices of the matrix are in sorted order, otherwise False.

**Warning:** Getting this property might synchronize the device.

**ndim****nnz****shape****size****cupyx.scipy.sparse.dia\_matrix****class** cupyx.scipy.sparse.dia\_matrix(*arg1, shape=None, dtype=None, copy=False*)

Sparse matrix with DIAgonal storage.

Now it has only one initializer format below:

dia\_matrix((data, offsets))

**Parameters**

- **arg1** – Arguments for the initializer.
- **shape** (*tuple*) – Shape of a matrix. Its length must be two.
- **dtype** – Data type. It must be an argument of `numpy.dtype`.
- **copy** (*bool*) – If True, copies of given arrays are always used.

See also:

`scipy.sparse.dia_matrix`

**Methods****\_\_len\_\_()****\_\_iter\_\_()****arcsin()**

Elementwise arcsin.

**arcsinh()**

Elementwise arcsinh.

**arctan()**

Elementwise arctan.

**arctanh()**

Elementwise arctanh.

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters**

**format** (*str* or *None*) – Format you need.

**asfptype()**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns**

A matrix with float type.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**astype**(*t*)

Casts the array to given data type.

**Parameters**

**dtype** – Type specifier.

**Returns**

A copy of the array with a given type.

**ceil()**

Elementwise ceil.

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type***cupyx.scipy.sparse.spmatrix***copy()**

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero()**

Returns number of non-zero entries.

---

**Note:** This method counts the actual number of non-zero entories, which does not include explicit zero entries. Instead `nnz` returns the number of entries including explicit zeros.

---

**Returns**

Number of non-zero entries.

**deg2rad()**

Elementwise `deg2rad`.

**diagonal(*k*=0)**

Returns the *k*-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*].) – 0 (the main diagonal).

**Returns**

The *k*-th diagonal.

**Return type***cupy.ndarray***dot(*other*)**

Ordinary dot product

**expm1()**

Elementwise `expm1`.

**floor()**

Elementwise `floor`.

**get(*stream*=None)**

Returns a copy of the array on host memory.

**Parameters**

**stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns**

Copy of the array on host memory.

**Return type***scipy.sparse.dia\_matrix*

**getH()**

**get\_shape()**

Returns the shape of the matrix.

**Returns**

Shape of the matrix.

**Return type**

`tuple`

**getformat()**

**getmaxprint()**

**getnnz**(*axis=None*)

Returns the number of stored values, including explicit zeros.

**Parameters**

**axis** – Not supported yet.

**Returns**

The number of stored values.

**Return type**

`int`

**log1p()**

Elementwise log1p.

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

**Parameters**

**axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the average of all the elements. Select from {None, 0, 1, -2, -1}.

**Returns**

Summed array.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.sparse.spmatrix.mean()`

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

Elementwise power function.

**Parameters**

- **n** – Exponent.
- **dtype** – Type specifier.

**rad2deg()**

Elementwise rad2deg.

**reshape(\*shape, order='C')**

Gives a new shape to a sparse matrix without changing its data.

**Parameters**

- **shape** (*tuple*) – The new shape should be compatible with the original shape.
- **order** – { 'C', 'F' } (optional) Read the elements using this index order. 'C' means to read and write the elements using C-like index order. 'F' means to read and write the elements using Fortran-like index order. Default: C.

**Returns**

sparse matrix

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**rint()**

Elementwise rint.

**set\_shape(shape)****setdiag(values, k=0)**

Set diagonal or off-diagonal elements of the array.

**Parameters**

- **values** (*cupy.ndarray*) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (*int*, *optional*) – Which diagonal to set, corresponding to elements  $a[i, i+k]$ . Default: 0 (the main diagonal).

**sign()**

Elementwise sign.

**sin()**

Elementwise sin.

**sinh()**

Elementwise sinh.

**sqrt()**

Elementwise sqrt.

**sum(axis=None, dtype=None, out=None)**

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (*cupy.ndarray*) – Output matrix.

**Returns**

Summed array.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**tan()**

Elementwise tan.

**tanh()**

Elementwise tanh.

**toarray(*order=None, out=None*)**

Returns a dense matrix representing the same value.

**tobsr(*blocksize=None, copy=False*)**

Convert this matrix to Block Sparse Row format.

**tocoo(*copy=False*)**

Convert this matrix to COOrdinate format.

**tocsc(*copy=False*)**

Converts the matrix to Compressed Sparse Column format.

**Parameters**

**copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csc conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**tocsr(*copy=False*)**

Converts the matrix to Compressed Sparse Row format.

**Parameters**

**copy** (*bool*) – If `False`, it shares data arrays as much as possible. Actually this option is ignored because all arrays in a matrix cannot be shared in dia to csr conversion.

**Returns**

Converted matrix.

**Return type**

*cupyx.scipy.sparse.csc\_matrix*

**todense(*order=None, out=None*)**

Return a dense matrix representation of this matrix.

**todia(*copy=False*)**

Convert this matrix to sparse DIAgonal format.

**todok(*copy=False*)**

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Reverses the dimensions of the sparse matrix.

**trunc**()

Elementwise trunc.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to [`toarray\(\)`](#) method.

**H**

**T**

**device**

CUDA device on which this array resides.

**dtype**

Data type of the matrix.

**format** = 'dia'

**ndim**

**nnz**

**shape**

**size**



**cupyx.scipy.sparse.spmatrix**

**class** cupyx.scipy.sparse.spmatrix(*maxprint=50*)

Base class of all sparse matrixes.

See `scipy.sparse.spmatrix`

**Methods**

**\_\_len\_\_()**

**\_\_iter\_\_()**

**asformat**(*format*)

Return this matrix in a given sparse format.

**Parameters**

**format** (*str* or *None*) – Format you need.

**asfptype()**

Upcasts matrix to a floating point format.

When the matrix has floating point type, the method returns itself. Otherwise it makes a copy with floating point type and the same format.

**Returns**

A matrix with float type.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**astype**(*t*)

Casts the array to given data type.

**Parameters**

**t** – Type specifier.

**Returns**

A copy of the array with the given type and the same format.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**conj**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**conjugate**(*copy=True*)

Element-wise complex conjugation.

If the matrix is of non-complex data type and *copy* is False, this method does nothing and the data is not copied.

**Parameters**

**copy** (*bool*) – If True, the result is guaranteed to not share data with self.

**Returns**

The element-wise complex conjugate.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**copy**()

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

**count\_nonzero**()

Number of non-zero entries, equivalent to

**diagonal**(*k=0*)

Returns the k-th diagonal of the matrix.

**Parameters**

- **k** (*int*, *optional*) – Which diagonal to get, corresponding to elements
- **a**[*i* – 0 (the main diagonal).
- **Default** (*i+k*].) – 0 (the main diagonal).

**Returns**

The k-th diagonal.

**Return type**

*cupy.ndarray*

**dot**(*other*)

Ordinary dot product

**get**(*stream=None*)

Return a copy of the array on host memory.

**Parameters**

**stream** (*cupy.cuda.Stream*) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

**Returns**

An array on host memory.

**Return type**

*scipy.sparse.spmatrix*

**getH**()**get\_shape**()**getformat**()

**getmaxprint()**

**getnnz**(*axis=None*)

Number of stored values, including explicit zeros.

**maximum**(*other*)

**mean**(*axis=None, dtype=None, out=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

#### Parameters

- **-2** (*axis*) – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **-1** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **0** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **1** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **None** – optional Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e., *axis = None*).
- **dtype** (*dtype*) – optional Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.
- **out** ([cupy.ndarray](#)) – optional Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

#### Returns

Output array of means

#### Return type

m ([cupy.ndarray](#))

#### See also:

`scipy.sparse.spmatrix.mean()`

**minimum**(*other*)

**multiply**(*other*)

Point-wise multiplication by another matrix

**power**(*n, dtype=None*)

**reshape**(*\*shape, order='C'*)

Gives a new shape to a sparse matrix without changing its data.

#### Parameters

- **shape** ([tuple](#)) – The new shape should be compatible with the original shape.
- **order** – { 'C', 'F' } (optional) Read the elements using this index order. 'C' means to read and write the elements using C-like index order. 'F' means to read and write the elements using Fortran-like index order. Default: C.

**Returns**

sparse matrix

**Return type**

*cupyx.scipy.sparse.coo\_matrix*

**set\_shape**(*shape*)

**setdiag**(*values*, *k*=0)

Set diagonal or off-diagonal elements of the array.

**Parameters**

- **values** (*cupy.ndarray*) – New values of the diagonal elements. Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values is longer than the diagonal, then the remaining values are ignored. If a scalar value is given, all of the diagonal is set to it.
- **k** (*int*, *optional*) – Which diagonal to set, corresponding to elements  $a[i, i+k]$ . Default: 0 (the main diagonal).

**sum**(*axis*=None, *dtype*=None, *out*=None)

Sums the matrix elements over a given axis.

**Parameters**

- **axis** (int or None) – Axis along which the sum is computed. If it is None, it computes the sum of all the elements. Select from {None, 0, 1, -2, -1}.
- **dtype** – The type of returned matrix. If it is not specified, type of the array is used.
- **out** (*cupy.ndarray*) – Output matrix.

**Returns**

Summed array.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.sparse.spmatrix.sum()`

**toarray**(*order*=None, *out*=None)

Return a dense ndarray representation of this matrix.

**tobsr**(*blocksize*=None, *copy*=False)

Convert this matrix to Block Sparse Row format.

**tocoo**(*copy*=False)

Convert this matrix to COOrdinate format.

**tocsc**(*copy*=False)

Convert this matrix to Compressed Sparse Column format.

**tocsr**(*copy*=False)

Convert this matrix to Compressed Sparse Row format.

**todense**(*order*=None, *out*=None)

Return a dense matrix representation of this matrix.

**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

**tolil**(*copy=False*)

Convert this matrix to LInked List format.

**transpose**(*axes=None, copy=False*)

Reverses the dimensions of the sparse matrix.

**\_\_eq\_\_**(*other*)

Return self==value.

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_lt\_\_**(*other*)

Return self<value.

**\_\_le\_\_**(*other*)

Return self<=value.

**\_\_gt\_\_**(*other*)

Return self>value.

**\_\_ge\_\_**(*other*)

Return self>=value.

**\_\_nonzero\_\_**()

**\_\_bool\_\_**()

## Attributes

**A**

Dense ndarray representation of this matrix.

This property is equivalent to [`toarray\(\)`](#) method.

**H**

**T**

**device**

CUDA device on which this array resides.

**ndim**

**nnz**

**shape**

**size**

## Functions

Building sparse matrices:

<code>eye(m[, n, k, dtype, format])</code>	Creates a sparse matrix with ones on diagonal.
<code>identity(n[, dtype, format])</code>	Creates an identity matrix in sparse format.
<code>kron(A, B[, format])</code>	Kronecker product of sparse matrices A and B.
<code>kronsum(A, B[, format])</code>	Kronecker sum of sparse matrices A and B.
<code>diags(diagonals[, offsets, shape, format, dtype])</code>	Construct a sparse matrix from diagonals.
<code>spdiags(data, diags, m, n[, format])</code>	Creates a sparse matrix from diagonals.
<code>tril(A[, k, format])</code>	Returns the lower triangular portion of a matrix in sparse format
<code>triu(A[, k, format])</code>	Returns the upper triangular portion of a matrix in sparse format
<code>bmat(blocks[, format, dtype])</code>	Builds a sparse matrix from sparse sub-blocks
<code>hstack(blocks[, format, dtype])</code>	Stacks sparse matrices horizontally (column wise)
<code>vstack(blocks[, format, dtype])</code>	Stacks sparse matrices vertically (row wise)
<code>rand(m, n[, density, format, dtype, ...])</code>	Generates a random sparse matrix.
<code>random(m, n[, density, format, dtype, ...])</code>	Generates a random sparse matrix.

### `cupyx.scipy.sparse.eye`

`cupyx.scipy.sparse.eye(m, n=None, k=0, dtype='d', format=None)`

Creates a sparse matrix with ones on diagonal.

#### Parameters

- **m** (*int*) – Number of rows.
- **n** (*int* or *None*) – Number of columns. If it is *None*, it makes a square matrix.
- **k** (*int*) – Diagonal to place ones on.
- **dtype** – Type of a matrix to create.
- **format** (*str* or *None*) – Format of the result, e.g. `format="csr"`.

#### Returns

Created sparse matrix.

#### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.eye()`

**cupyx.scipy.sparse.identity**

`cupyx.scipy.sparse.identity(n, dtype='d', format=None)`

Creates an identity matrix in sparse format.

---

**Note:** Currently it only supports csr, csc and coo formats.

---

**Parameters**

- **n** (*int*) – Number of rows and columns.
- **dtype** – Type of a matrix to create.
- **format** (*str* or *None*) – Format of the result, e.g. `format="csr"`.

**Returns**

Created identity matrix.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.identity()`

**cupyx.scipy.sparse.kron**

`cupyx.scipy.sparse.kron(A, B, format=None)`

Kronecker product of sparse matrices A and B.

**Parameters**

- **A** (*cupyx.scipy.sparse.spmatrix*) – a sparse matrix.
- **B** (*cupyx.scipy.sparse.spmatrix*) – a sparse matrix.
- **format** (*str*) – the format of the returned sparse matrix.

**Returns**

Generated sparse matrix with the specified format.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.kron()`

## cupyx.scipy.sparse.kronsum

`cupyx.scipy.sparse.kronsum(A, B, format=None)`

Kronecker sum of sparse matrices A and B.

Kronecker sum is the sum of two Kronecker products  $\text{kron}(I_n, A) + \text{kron}(B, I_m)$ , where  $I_n$  and  $I_m$  are identity matrices.

### Parameters

- **A** (`cupyx.scipy.sparse.spmatrix`) – a sparse matrix.
- **B** (`cupyx.scipy.sparse.spmatrix`) – a sparse matrix.
- **format** (`str`) – the format of the returned sparse matrix.

### Returns

Generated sparse matrix with the specified format.

### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.kronsum()`

## cupyx.scipy.sparse.diags

`cupyx.scipy.sparse.diags(diagonals, offsets=0, shape=None, format=None, dtype=None)`

Construct a sparse matrix from diagonals.

### Parameters

- **diagonals** (*sequence of array\_like*) – Sequence of arrays containing the matrix diagonals, corresponding to *offsets*.
- **offsets** (*sequence of int or an int*) –

#### Diagonals to set:

- $k = 0$  the main diagonal (default)
- $k > 0$  the  $k$ -th upper diagonal
- $k < 0$  the  $k$ -th lower diagonal

- **shape** (*tuple of int*) – Shape of the result. If omitted, a square matrix large enough to contain the diagonals is returned.
- **format** (`{"dia", "csr", "csc", "lil", ...}`) – Matrix format of the result. By default (format=None) an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype*) – Data type of the matrix.

### Returns

Generated matrix.

### Return type

`cupyx.scipy.sparse.spmatrix`



## Notes

This function differs from *spdiags* in the way it handles off-diagonals.

The result from *diags* is the sparse equivalent of:

```
cupy.diag(diagonals[0], offsets[0])
+ ...
+ cupy.diag(diagonals[k], offsets[k])
```

Repeated diagonal offsets are disallowed.

## cupyx.scipy.sparse.spdiags

`cupyx.scipy.sparse.spdiags(data, diags, m, n, format=None)`

Creates a sparse matrix from diagonals.

### Parameters

- **data** (`cupy.ndarray`) – Matrix diagonals stored row-wise.
- **diags** (`cupy.ndarray`) – Diagonals to set.
- **m** (`int`) – Number of rows.
- **n** (`int`) – Number of cols.
- **format** (`str` or `None`) – Sparse format, e.g. `format="csr"`.

### Returns

Created sparse matrix.

### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.spdiags()`

## cupyx.scipy.sparse.tril

`cupyx.scipy.sparse.tril(A, k=0, format=None)`

Returns the lower triangular portion of a matrix in sparse format

### Parameters

- **A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose lower triangular portion is desired.
- **k** (`integer`) – The top-most diagonal of the lower triangle.
- **format** (`string`) – Sparse format of the result, e.g. ‘csr’, ‘csc’, etc.

### Returns

Lower triangular portion of A in sparse format.

### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.tril()`

### `cupyx.scipy.sparse.triu`

`cupyx.scipy.sparse.triu(A, k=0, format=None)`

Returns the upper triangular portion of a matrix in sparse format

#### Parameters

- **A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose upper triangular portion is desired.
- **k** (*integer*) – The bottom-most diagonal of the upper triangle.
- **format** (*string*) – Sparse format of the result, e.g. ‘csr’, ‘csc’, etc.

#### Returns

Upper triangular portion of A in sparse format.

#### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.triu()`

### `cupyx.scipy.sparse.bmat`

`cupyx.scipy.sparse.bmat(blocks, format=None, dtype=None)`

Builds a sparse matrix from sparse sub-blocks

#### Parameters

- **blocks** (*array\_like*) – Grid of sparse matrices with compatible shapes. An entry of None implies an all-zero matrix.
- **format** (*{'bsr', 'coo', 'csc', 'csr', 'dia', 'dok', 'lil'}*, *optional*) – The sparse format of the result (e.g. “csr”). By default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype*, *optional*) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

#### Returns

bmat (sparse matrix)

See also:

`scipy.sparse.bmat()`

## Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, bmat
>>> A = csr_matrix(array([[1., 2.], [3., 4.])))
>>> B = csr_matrix(array([[5.], [6.])))
>>> C = csr_matrix(array([[7.])))
>>> bmat([A, B], [None, C]).toarray()
array([[1., 2., 5.],
       [3., 4., 6.],
       [0., 0., 7.]])
>>> bmat([A, None], [None, C]).toarray()
array([[1., 2., 0.],
       [3., 4., 0.],
       [0., 0., 7.]])
```

## cupyx.scipy.sparse.hstack

`cupyx.scipy.sparse.hstack(blocks, format=None, dtype=None)`

Stacks sparse matrices horizontally (column wise)

### Parameters

- **blocks** (sequence of `cupyx.scipy.sparse.spmatrix`) – sparse matrices to stack
- **format** (`str`) – sparse format of the result (e.g. “csr”) by default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (`dtype`, optional) – The data-type of the output matrix. If not given, the dtype is determined from that of blocks.

### Returns

the stacked sparse matrix

### Return type

`cupyx.scipy.sparse.spmatrix`

See also:

`scipy.sparse.hstack()`

## Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, hstack
>>> A = csr_matrix(array([[1., 2.], [3., 4.])))
>>> B = csr_matrix(array([[5.], [6.])))
>>> hstack([A, B]).toarray()
array([[1., 2., 5.],
       [3., 4., 6.]])
```

## cupyx.scipy.sparse.vstack

`cupyx.scipy.sparse.vstack(blocks, format=None, dtype=None)`

Stacks sparse matrices vertically (row wise)

### Parameters

- **blocks** (sequence of `cupyx.scipy.sparse.spmatrix`) – sparse matrices to stack
- **format** (*str*, optional) – sparse format of the result (e.g. “csr”) by default an appropriate sparse matrix format is returned. This choice is subject to change.
- **dtype** (*dtype*, optional) – The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

### Returns

the stacked sparse matrix

### Return type

*cupyx.scipy.sparse.spmatrix*

See also:

`scipy.sparse.vstack()`

## Examples

```
>>> from cupy import array
>>> from cupyx.scipy.sparse import csr_matrix, vstack
>>> A = csr_matrix(array([[1., 2.], [3., 4.]])
>>> B = csr_matrix(array([[5., 6.]])
>>> vstack([A, B]).toarray()
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
```

## cupyx.scipy.sparse.rand

`cupyx.scipy.sparse.rand(m, n, density=0.01, format='coo', dtype=None, random_state=None)`

Generates a random sparse matrix.

See `cupyx.scipy.sparse.random()` for detail.

### Parameters

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.
- **random\_state** (`cupy.random.RandomState` or *int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.

**Returns**

Generated matrix.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.rand()`

**See also:**

*cupyx.scipy.sparse.random()*

**cupyx.scipy.sparse.random**

`cupyx.scipy.sparse.random(m, n, density=0.01, format='coo', dtype=None, random_state=None, data_rvs=None)`

Generates a random sparse matrix.

This function generates a random sparse matrix. First it selects non-zero elements with given density from  $(m, n)$  elements. So the number of non-zero elements  $k$  is  $k = m * n * \text{density}$ . Value of each element is selected with `data_rvs` function.

**Parameters**

- **m** (*int*) – Number of rows.
- **n** (*int*) – Number of cols.
- **density** (*float*) – Ratio of non-zero entries.
- **format** (*str*) – Matrix format.
- **dtype** (*dtype*) – Type of the returned matrix values.
- **random\_state** (*cupy.random.RandomState* or *int*) – State of random number generator. If an integer is given, the method makes a new state for random number generator and uses it. If it is not given, the default state is used. This state is used to generate random indexes for nonzero entries.
- **data\_rvs** (*callable*) – A function to generate data for a random matrix. If it is not given, *random\_state.rand* is used.

**Returns**

Generated matrix.

**Return type**

*cupyx.scipy.sparse.spmatrix*

**See also:**

`scipy.sparse.random()`

Sparse matrix tools:

---

<i>find</i> (A)	Returns the indices and values of the nonzero elements of a matrix
-----------------	--

---

## cupyx.scipy.sparse.find

cupyx.scipy.sparse.find(*A*)

Returns the indices and values of the nonzero elements of a matrix

### Parameters

**A** (`cupy.ndarray` or `cupyx.scipy.sparse.spmatrix`) – Matrix whose nonzero elements are desired.

### Returns

It returns (I, J, V). I, J, and V contain respectively the row indices, column indices, and values of the nonzero matrix entries.

### Return type

tuple of `cupy.ndarray`

### See also:

`scipy.sparse.find()`

Identifying sparse matrices:

<code>issparse(x)</code>	Checks if a given matrix is a sparse matrix.
<code>isspmatrix(x)</code>	Checks if a given matrix is a sparse matrix.
<code>isspmatrix_csc(x)</code>	Checks if a given matrix is of CSC format.
<code>isspmatrix_csr(x)</code>	Checks if a given matrix is of CSR format.
<code>isspmatrix_coo(x)</code>	Checks if a given matrix is of COO format.
<code>isspmatrix_dia(x)</code>	Checks if a given matrix is of DIA format.

## cupyx.scipy.sparse.issparse

cupyx.scipy.sparse.issparse(*x*)

Checks if a given matrix is a sparse matrix.

### Returns

Returns if *x* is `cupyx.scipy.sparse.spmatrix` that is a base class of all sparse matrix classes.

### Return type

`bool`

## cupyx.scipy.sparse.isspmatrix

cupyx.scipy.sparse.isspmatrix(*x*)

Checks if a given matrix is a sparse matrix.

### Returns

Returns if *x* is `cupyx.scipy.sparse.spmatrix` that is a base class of all sparse matrix classes.

### Return type

`bool`

**cupyx.scipy.sparse.isspmatrix\_csc**

`cupyx.scipy.sparse.isspmatrix_csc(x)`

Checks if a given matrix is of CSC format.

**Returns**

Returns if `x` is `cupyx.scipy.sparse.csc_matrix`.

**Return type**

`bool`

**cupyx.scipy.sparse.isspmatrix\_csr**

`cupyx.scipy.sparse.isspmatrix_csr(x)`

Checks if a given matrix is of CSR format.

**Returns**

Returns if `x` is `cupyx.scipy.sparse.csr_matrix`.

**Return type**

`bool`

**cupyx.scipy.sparse.isspmatrix\_coo**

`cupyx.scipy.sparse.isspmatrix_coo(x)`

Checks if a given matrix is of COO format.

**Returns**

Returns if `x` is `cupyx.scipy.sparse.coo_matrix`.

**Return type**

`bool`

**cupyx.scipy.sparse.isspmatrix\_dia**

`cupyx.scipy.sparse.isspmatrix_dia(x)`

Checks if a given matrix is of DIA format.

**Returns**

Returns if `x` is `cupyx.scipy.sparse.dia_matrix`.

**Return type**

`bool`

## Submodules

---

*csgraph*

---

*linalg*

---

## Exceptions

- `scipy.sparse.SparseEfficiencyWarning`
- `scipy.sparse.SparseWarning`

## 5.4.9 Sparse linear algebra (`cupyx.scipy.sparse.linalg`)

---

**Hint:** SciPy API Reference: Sparse linear algebra (`scipy.sparse.linalg`)

---

### Abstract linear operators

<i>LinearOperator</i> (shape, matvec[, rmatvec, ...])	Common interface for performing matrix vector products
<i>aslinearoperator</i> (A)	Return A as a LinearOperator.

### `cupyx.scipy.sparse.linalg.LinearOperator`

**class** `cupyx.scipy.sparse.linalg.LinearOperator`(shape, matvec, rmatvec=None, matmat=None, dtype=None, rmatmat=None)

Common interface for performing matrix vector products

To construct a concrete LinearOperator, either pass appropriate callables to the constructor of this class, or subclass it.

#### Parameters

- **shape** (*tuple*) – Matrix dimensions (M, N).
- **matvec** (*callable*  $f(v)$ ) – Returns  $A * v$ .
- **rmatvec** (*callable*  $f(v)$ ) – Returns  $A^H * v$ , where  $A^H$  is the conjugate transpose of A.
- **matmat** (*callable*  $f(V)$ ) – Returns  $A * V$ , where V is a dense matrix with dimensions (N, K).
- **dtype** (*dtype*) – Data type of the matrix.
- **rmatmat** (*callable*  $f(V)$ ) – Returns  $A^H * V$ , where V is a dense matrix with dimensions (M, K).



See also:

`scipy.sparse.linalg.LinearOperator`

## Methods

**`__call__`**(*x*)

Call self as a function.

**`adjoint`**()

Hermitian adjoint.

**`dot`**(*x*)

Matrix-matrix or matrix-vector multiplication.

**`matmat`**(*X*)

Matrix-matrix multiplication.

**`matvec`**(*x*)

Matrix-vector multiplication.

**`rmatmat`**(*X*)

Adjoint matrix-matrix multiplication.

**`rmatvec`**(*x*)

Adjoint matrix-vector multiplication.

**`transpose`**()

Transpose this linear operator.

**`__eq__`**(*value*, /)

Return self==value.

**`__ne__`**(*value*, /)

Return self!=value.

**`__lt__`**(*value*, /)

Return self<value.

**`__le__`**(*value*, /)

Return self<=value.

**`__gt__`**(*value*, /)

Return self>value.

**`__ge__`**(*value*, /)

Return self>=value.

## Attributes

**H**

Hermitian adjoint.

**T**

Transpose this linear operator.

**ndim** = 2

## cupyx.scipy.sparse.linalg.aslinearoperator

cupyx.scipy.sparse.linalg.aslinearoperator(*A*)

Return *A* as a LinearOperator.

### Parameters

**A** (*array-like*) – The input array to be converted to a *LinearOperator* object. It may be any of the following types:

- `cupy.ndarray`
- sparse matrix (e.g. `csr_matrix`, `coo_matrix`, etc.)
- `cupyx.scipy.sparse.linalg.LinearOperator`
- object with `.shape` and `.matvec` attributes

### Returns

*LinearOperator* object

### Return type

`cupyx.scipy.sparse.linalg.LinearOperator`

See also:

`scipy.sparse.aslinearoperator`()`

## Matrix norms

---

<code>norm(x[, ord, axis])</code>	Norm of a cupy.scipy.spmatrix
-----------------------------------	-------------------------------

---

## cupyx.scipy.sparse.linalg.norm

cupyx.scipy.sparse.linalg.norm(*x*, *ord=None*, *axis=None*)

Norm of a cupy.scipy.spmatrix

This function is able to return one of seven different sparse matrix norms, depending on the value of the *ord* parameter.

### Parameters

- **x** (*sparse matrix*) – Input sparse matrix.
- **ord** (*non-zero int, inf, -inf, 'fro', optional*) – Order of the norm (see table under Notes). *inf* means numpy's *inf* object.

- **axis** – (int, 2-tuple of ints, None, optional): If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

**Returns**

0-D or 1-D array or norm(s).

**Return type**

*ndarray*

**See also:**

`scipy.sparse.linalg.norm()`

## Solving linear problems

Direct methods for linear equation systems:

<code>spsolve(A, b)</code>	Solves a sparse linear system $A \mathbf{x} = \mathbf{b}$
<code>spsolve_triangular(A, b[, lower, ...])</code>	Solves a sparse triangular system $A \mathbf{x} = \mathbf{b}$ .
<code>factorized(A)</code>	Return a function for solving a sparse linear system, with <i>A</i> pre-factorized.

### cupyx.scipy.sparse.linalg.spsolve

`cupyx.scipy.sparse.linalg.spsolve(A, b)`

Solves a sparse linear system  $A \mathbf{x} = \mathbf{b}$

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix with dimension (M, M).
- **b** (`cupy.ndarray`) – Dense vector or matrix with dimension (M) or (M, N).

**Returns**

Solution to the system  $A \mathbf{x} = \mathbf{b}$ .

**Return type**

*cupy.ndarray*

### cupyx.scipy.sparse.linalg.spsolve\_triangular

`cupyx.scipy.sparse.linalg.spsolve_triangular(A, b, lower=True, overwrite_A=False, overwrite_b=False, unit_diagonal=False)`

Solves a sparse triangular system  $A \mathbf{x} = \mathbf{b}$ .

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix with dimension (M, M).
- **b** (`cupy.ndarray`) – Dense vector or matrix with dimension (M) or (M, K).
- **lower** (*bool*) – Whether *A* is a lower or upper triangular matrix. If True, it is lower triangular, otherwise, upper triangular.
- **overwrite\_A** (*bool*) – (not supported)

- **overwrite\_b** (*bool*) – Allows overwriting data in *b*.
- **unit\_diagonal** (*bool*) – If True, diagonal elements of *A* are assumed to be 1 and will not be referenced.

**Returns**

Solution to the system  $A \mathbf{x} = \mathbf{b}$ . The shape is the same as *b*.

**Return type**

*cupy.ndarray*

## cupyx.scipy.sparse.linalg.factorized

`cupyx.scipy.sparse.linalg.factorized(A)`

Return a function for solving a sparse linear system, with *A* pre-factorized.

**Parameters**

**A** (*cupyx.scipy.sparse.spmatrix*) – Sparse matrix to factorize.

**Returns**

a function to solve the linear system of equations given in *A*.

**Return type**

callable

---

**Note:** This function computes LU decomposition of a sparse matrix on the CPU using *scipy.sparse.linalg.splu*. Therefore, LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the method returned by this function is performed on the GPU.

---

**See also:**

`scipy.sparse.linalg.factorized()`

Iterative methods for linear equation systems:

<code>cg(A, b[, x0, tol, maxiter, M, callback, atol])</code>	Uses Conjugate Gradient iteration to solve $A\mathbf{x} = \mathbf{b}$ .
<code>gmres(A, b[, x0, tol, restart, maxiter, M, ...])</code>	Uses Generalized Minimal RESidual iteration to solve $A\mathbf{x} = \mathbf{b}$ .
<code>cgs(A, b[, x0, tol, maxiter, M, callback, atol])</code>	Use Conjugate Gradient Squared iteration to solve $A\mathbf{x} = \mathbf{b}$ .
<code>minres(A, b[, x0, shift, tol, maxiter, M, ...])</code>	Uses MINimum RESidual iteration to solve $A\mathbf{x} = \mathbf{b}$ .

## cupyx.scipy.sparse.linalg.cg

`cupyx.scipy.sparse.linalg.cg(A, b, x0=None, tol=1e-05, maxiter=None, M=None, callback=None, atol=None)`

Uses Conjugate Gradient iteration to solve  $A\mathbf{x} = \mathbf{b}$ .

**Parameters**

- **A** (*ndarray*, *spmatrix* or *LinearOperator*) – The real or complex matrix of the linear system with shape  $(n, n)$ . *A* must be a hermitian, positive definitive matrix with type of *cupy.ndarray*, *cupyx.scipy.sparse.spmatrix* or *cupyx.scipy.sparse.linalg.LinearOperator*.

- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.
- **tol** (`float`) – Tolerance for convergence.
- **maxiter** (`int`) – Maximum number of iterations.
- **M** (`ndarray`, `spmatrix` or `LinearOperator`) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (`function`) – User-specified function to call after each iteration. It is called as `callback(xk)`, where `xk` is the current solution vector.
- **atol** (`float`) – Tolerance for convergence.

**Returns**

It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type**

`tuple`

See also:

`scipy.sparse.linalg.cg()`

**cupyx.scipy.sparse.linalg.gmres**

`cupyx.scipy.sparse.linalg.gmres(A, b, x0=None, tol=1e-05, restart=None, maxiter=None, M=None, callback=None, atol=None, callback_type=None)`

Uses Generalized Minimal RESidual iteration to solve  $Ax = b$ .

**Parameters**

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system with shape  $(n, n)$ . A must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.
- **tol** (`float`) – Tolerance for convergence.
- **restart** (`int`) – Number of iterations between restarts. Larger values increase iteration cost, but may be necessary for convergence.
- **maxiter** (`int`) – Maximum number of iterations.
- **M** (`ndarray`, `spmatrix` or `LinearOperator`) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (`function`) – User-specified function to call on every restart. It is called as `callback(arg)`, where `arg` is selected by `callback_type`.
- **callback\_type** (`str`) – ‘x’ or ‘pr\_norm’. If ‘x’, the current solution vector is used as an argument of callback function. if ‘pr\_norm’, relative (preconditioned) residual norm is used as an argument.
- **atol** (`float`) – Tolerance for convergence.

**Returns**

It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type**

`tuple`

**Reference:**

M. Wang, H. Klie, M. Parashar and H. Sudan, “Solving Sparse Linear Systems on NVIDIA Tesla GPUs”, ICCS 2009 (2009).

**See also:**

`scipy.sparse.linalg.gmres()`

**cupyx.scipy.sparse.linalg.cgs**

`cupyx.scipy.sparse.linalg.cgs(A, b, x0=None, tol=1e-05, maxiter=None, M=None, callback=None, atol=None)`

Use Conjugate Gradient Squared iteration to solve  $Ax = b$ .

**Parameters**

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system with shape  $(n, n)$ .
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.
- **tol** (`float`) – Tolerance for convergence.
- **maxiter** (`int`) – Maximum number of iterations.
- **M** (`ndarray`, `spmatrix` or `LinearOperator`) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (`function`) – User-specified function to call after each iteration. It is called as `callback(xk)`, where `xk` is the current solution vector.
- **atol** (`float`) – Tolerance for convergence.

**Returns**

It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type**

`tuple`

**See also:**

`scipy.sparse.linalg.cgs()`

**cupyx.scipy.sparse.linalg.minres**

`cupyx.scipy.sparse.linalg.minres(A, b, x0=None, shift=0.0, tol=1e-05, maxiter=None, M=None, callback=None, check=False)`

Uses MINimum RESidual iteration to solve  $Ax = b$ .

**Parameters**

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system with shape  $(n, n)$ .
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape  $(n,)$  or  $(n, 1)$ .
- **x0** (`cupy.ndarray`) – Starting guess for the solution.
- **shift** (`int` or `float`) – If  $\text{shift} \neq 0$  then the method solves  $(A - \text{shift} \cdot I)x = b$
- **tol** (`float`) – Tolerance for convergence.
- **maxiter** (`int`) – Maximum number of iterations.
- **M** (`ndarray`, `spmatrix` or `LinearOperator`) – Preconditioner for A. The preconditioner should approximate the inverse of A. M must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **callback** (`function`) – User-specified function to call after each iteration. It is called as `callback(xk)`, where `xk` is the current solution vector.

**Returns**

It returns `x` (`cupy.ndarray`) and `info` (`int`) where `x` is the converged solution and `info` provides convergence information.

**Return type**

`tuple`

See also:

`scipy.sparse.linalg.minres()`

Iterative methods for least-squares problems:

<code>lsqr(A, b)</code>	Solves linear system with QR decomposition.
<code>lsmr(A, b[, x0, damp, atol, btol, conlim, ...])</code>	Iterative solver for least-squares problems.

**cupyx.scipy.sparse.linalg.lsqr**

`cupyx.scipy.sparse.linalg.lsqr(A, b)`

Solves linear system with QR decomposition.

Find the solution to a large, sparse, linear system of equations. The function solves  $Ax = b$ . Given two-dimensional matrix A is decomposed into  $Q * R$ .

**Parameters**

- **A** (`cupy.ndarray` or `cupyx.scipy.sparse.csr_matrix`) – The input matrix with dimension  $(N, N)$
- **b** (`cupy.ndarray`) – Right-hand side vector.

**Returns**

Its length must be ten. It has same type elements as SciPy. Only the first element, the solution vector `x`, is available and other elements are expressed as `None` because the implementation of cuSOLVER is different from the one of SciPy. You can easily calculate the fourth element by `norm(b - Ax)` and the ninth element by `norm(x)`.

**Return type**

tuple

See also:

`scipy.sparse.linalg.lsqr()`

**cupyx.scipy.sparse.linalg.lsmr**

`cupyx.scipy.sparse.linalg.lsmr(A, b, x0=None, damp=0.0, atol=1e-06, btol=1e-06, conlim=100000000.0, maxiter=None)`

Iterative solver for least-squares problems.

`lsmr` solves the system of linear equations  $Ax = b$ . If the system is inconsistent, it solves the least-squares problem  $\min ||b - Ax||_2$ . `A` is a rectangular matrix of dimension `m`-by-`n`, where all cases are allowed: `m = n`, `m > n`, or `m < n`. `B` is a vector of length `m`. The matrix `A` may be dense or sparse (usually sparse).

**Parameters**

- **A** (`ndarray`, `spmatrix` or `LinearOperator`) – The real or complex matrix of the linear system. `A` must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **b** (`cupy.ndarray`) – Right hand side of the linear system with shape `(m,)` or `(m, 1)`.
- **x0** (`cupy.ndarray`) – Starting guess for the solution. If `None` zeros are used.
- **damp** (`float`) – Damping factor for regularized least-squares. `lsmr` solves the regularized least-squares problem

$$\min \begin{matrix} ||(b) - (A)x|| \\ ||(0) - (damp * I)x||_2 \end{matrix}$$

where `damp` is a scalar. If `damp` is `None` or `0`, the system is solved without regularization.

- **atol** (`float`) – Stopping tolerances. `lsmr` continues iterations until a certain backward error estimate is smaller than some quantity depending on `atol` and `btol`.
- **btol** (`float`) – Stopping tolerances. `lsmr` continues iterations until a certain backward error estimate is smaller than some quantity depending on `atol` and `btol`.
- **conlim** (`float`) – `lsmr` terminates if an estimate of `cond(A)` i.e. condition number of matrix exceeds `conlim`. If `conlim` is `None`, the default value is `1e+8`.
- **maxiter** (`int`) – Maximum number of iterations.

**Returns**

- `x` (`ndarray`): Least-square solution returned.
- `istop` (`int`): `istop` gives the reason for stopping:



```

0 means  $x=0$  is a solution.

1 means  $x$  is an approximate solution to  $A*x = B$ ,
according to atol and btol.

2 means  $x$  approximately solves the least-squares problem
according to atol.

3 means  $\text{COND}(A)$  seems to be greater than CONLIM.

4 is the same as 1 with atol = btol = eps (machine
precision)

5 is the same as 2 with atol = eps.

6 is the same as 3 with CONLIM = 1/eps.

7 means ITN reached maxiter before the other stopping
conditions were satisfied.

```

- *itn* (int): Number of iterations used.
- *normr* (float):  $\text{norm}(b - Ax)$
- *normar* (float):  $\text{norm}(A^T (b - Ax))$
- *norma* (float):  $\text{norm}(A)$
- *conda* (float): Condition number of  $A$ .
- *normx* (float):  $\text{norm}(x)$

**Return type**

tuple

**See also:**`scipy.sparse.linalg.lsqr()`**References**

D. C.-L. Fong and M. A. Saunders, “LSMR: An iterative algorithm for sparse least-squares problems”, SIAM J. Sci. Comput., vol. 33, pp. 2950-2971, 2011.

**Matrix factorizations**

Eigenvalue problems:

<code>eigsh(a[, k, which, v0, ncv, maxiter, tol, ...])</code>	Find $k$ eigenvalues and eigenvectors of the real symmetric square matrix or complex Hermitian matrix $A$ .
<code>lobpcg(A, X[, B, M, Y, tol, maxiter, ...])</code>	Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

### cupyx.scipy.sparse.linalg.eigsh

`cupyx.scipy.sparse.linalg.eigsh(a, k=6, *, which='LM', v0=None, ncv=None, maxiter=None, tol=0, return_eigenvectors=True)`

Find  $k$  eigenvalues and eigenvectors of the real symmetric square matrix or complex Hermitian matrix  $A$ .

Solves  $Ax = wx$ , the standard eigenvalue problem for  $w$  eigenvalues with corresponding eigenvectors  $x$ .

#### Parameters

- **a** (`ndarray`, `spmatrix` or `LinearOperator`) – A symmetric square matrix with dimension  $(n, n)$ . `a` must be `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **k** (`int`) – The number of eigenvalues and eigenvectors to compute. Must be  $1 \leq k < n$ .
- **which** (`str`) – 'LM' or 'LA'. 'LM': finds  $k$  largest (in magnitude) eigenvalues. 'LA': finds  $k$  largest (algebraic) eigenvalues. 'SA': finds  $k$  smallest (algebraic) eigenvalues.
- **v0** (`ndarray`) – Starting vector for iteration. If `None`, a random unit vector is used.
- **ncv** (`int`) – The number of Lanczos vectors generated. Must be  $k + 1 < ncv < n$ . If `None`, default value is used.
- **maxiter** (`int`) – Maximum number of Lanczos update iterations. If `None`, default value is used.
- **tol** (`float`) – Tolerance for residuals  $\|Ax - wx\|$ . If `0`, machine precision is used.
- **return\_eigenvectors** (`bool`) – If `True`, returns eigenvectors in addition to eigenvalues.

#### Returns

If `return_eigenvectors` is `True`, it returns  $w$  and  $x$  where  $w$  is eigenvalues and  $x$  is eigenvectors. Otherwise, it returns only  $w$ .

#### Return type

`tuple`

See also:

`scipy.sparse.linalg.eigsh()`

---

**Note:** This function uses the thick-restart Lanczos methods (<https://sdm.lbl.gov/~kewu/ps/trlan.html>).

---

### cupyx.scipy.sparse.linalg.lobpcg

`cupyx.scipy.sparse.linalg.lobpcg(A, X, B=None, M=None, Y=None, tol=None, maxiter=None, largest=True, verbosityLevel=0, retLambdaHistory=False, retResidualNormsHistory=False)`

Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

LOBPCG is a preconditioned eigensolver for large symmetric positive definite (SPD) generalized eigenproblems.

#### Parameters

- **A** (*array-like*) – The symmetric linear operator of the problem, usually a sparse matrix. Can be of the following types - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`

- **X** (`cupy.ndarray`) – Initial approximation to the  $k$  eigenvectors (non-sparse). If  $A$  has `shape=(n,n)` then  $X$  should have `shape=(n,k)`.
- **B** (*array-like*) – The right hand side operator in a generalized eigenproblem. By default,  $B = \text{Identity}$ . Can be of following types: - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`
- **M** (*array-like*) – Preconditioner to  $A$ ; by default  $M = \text{Identity}$ .  $M$  should approximate the inverse of  $A$ . Can be of the following types: - `cupy.ndarray` - `cupyx.scipy.sparse.csr_matrix` - `cupy.scipy.sparse.linalg.LinearOperator`
- **Y** (`cupy.ndarray`) –  $n$ -by- $sizeY$  matrix of constraints (non-sparse),  $sizeY < n$ . The iterations will be performed in the  $B$ -orthogonal complement of the column-space of  $Y$ .  $Y$  must be full rank.
- **tol** (`float`) – Solver tolerance (stopping criterion). The default is `tol=n*sqrt(eps)`.
- **maxiter** (`int`) – Maximum number of iterations. The default is `maxiter = 20`.
- **largest** (`bool`) – When `True`, solve for the largest eigenvalues, otherwise the smallest.
- **verbosityLevel** (`int`) – Controls solver output. The default is `verbosityLevel=0`.
- **retLambdaHistory** (`bool`) – Whether to return eigenvalue history. Default is `False`.
- **retResidualNormsHistory** (`bool`) – Whether to return history of residual norms. Default is `False`.

#### Returns

- $w$  (`cupy.ndarray`): Array of  $k$  eigenvalues
- $v$  (`cupy.ndarray`) An array of  $k$  eigenvectors.  $v$  has the same shape as  $X$ .
- $lambdas$  (list of `cupy.ndarray`): The eigenvalue history, if `retLambdaHistory` is `True`.
- $rnorms$  (list of `cupy.ndarray`): The history of residual norms, if `retResidualNormsHistory` is `True`.

#### Return type

`tuple`

#### See also:

`scipy.sparse.linalg.lobpcg()`

---

**Note:** If both `retLambdaHistory` and `retResidualNormsHistory` are `True` the return tuple has the following format (`lambda`, `V`, `lambda history`, `residual norms history`).

---

Singular values problems:

---

`svds(a[, k, ncv, tol, which, maxiter, ...])`

Finds the largest  $k$  singular values/vectors for a sparse matrix.

---

## cupyx.scipy.sparse.linalg.svds

`cupyx.scipy.sparse.linalg.svds(a, k=6, *, ncv=None, tol=0, which='LM', maxiter=None, return_singular_vectors=True)`

Finds the largest k singular values/vectors for a sparse matrix.

### Parameters

- **a** (`ndarray`, `spmatrix` or `LinearOperator`) – A real or complex array with dimension (m, n). a must `cupy.ndarray`, `cupyx.scipy.sparse.spmatrix` or `cupyx.scipy.sparse.linalg.LinearOperator`.
- **k** (`int`) – The number of singular values/vectors to compute. Must be  $1 \leq k < \min(m, n)$ .
- **ncv** (`int`) – The number of Lanczos vectors generated. Must be  $k + 1 < \text{ncv} < \min(m, n)$ . If `None`, default value is used.
- **tol** (`float`) – Tolerance for singular values. If 0, machine precision is used.
- **which** (`str`) – Only 'LM' is supported. 'LM': finds k largest singular values.
- **maxiter** (`int`) – Maximum number of Lanczos update iterations. If `None`, default value is used.
- **return\_singular\_vectors** (`bool`) – If `True`, returns singular vectors in addition to singular values.

### Returns

If `return_singular_vectors` is `True`, it returns u, s and vt where u is left singular vectors, s is singular values and vt is right singular vectors. Otherwise, it returns only s.

### Return type

`tuple`

### See also:

`scipy.sparse.linalg.svds()`

---

**Note:** This is a naive implementation using `cupyx.scipy.sparse.linalg.eigsh` as an eigensolver on  $a.H @ a$  or  $a @ a.H$ .

---

Complete or incomplete LU factorizations:

<code>splu(A[, permc_spec, diag_pivot_thresh, ...])</code>	Computes the LU decomposition of a sparse square matrix.
<code>spilu(A[, drop_tol, fill_factor, drop_rule, ...])</code>	Computes the incomplete LU decomposition of a sparse square matrix.
<code>SuperLU(obj)</code>	

---

**cupyx.scipy.sparse.linalg.splu**

`cupyx.scipy.sparse.linalg.splu(A, permc_spec=None, diag_pivot_thresh=None, relax=None, panel_size=None, options={})`

Computes the LU decomposition of a sparse square matrix.

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix to factorize.
- **permc\_spec** (`str`) – (For further augments, see `scipy.sparse.linalg.splu()`)
- **diag\_pivot\_thresh** (`float`) –
- **relax** (`int`) –
- **panel\_size** (`int`) –
- **options** (`dict`) –

**Returns**

Object which has a solve method.

**Return type**

`cupyx.scipy.sparse.linalg.SuperLU`

---

**Note:** This function LU-decomposes a sparse matrix on the CPU using `scipy.sparse.linalg.splu`. Therefore, LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the solve method, which this function returns, is performed on the GPU.

---

See also:

`scipy.sparse.linalg.splu()`

**cupyx.scipy.sparse.linalg.spilu**

`cupyx.scipy.sparse.linalg.spilu(A, drop_tol=None, fill_factor=None, drop_rule=None, permc_spec=None, diag_pivot_thresh=None, relax=None, panel_size=None, options={})`

Computes the incomplete LU decomposition of a sparse square matrix.

**Parameters**

- **A** (`cupyx.scipy.sparse.spmatrix`) – Sparse matrix to factorize.
- **drop\_tol** (`float`) – (For further augments, see `scipy.sparse.linalg.spilu()`)
- **fill\_factor** (`float`) –
- **drop\_rule** (`str`) –
- **permc\_spec** (`str`) –
- **diag\_pivot\_thresh** (`float`) –
- **relax** (`int`) –
- **panel\_size** (`int`) –
- **options** (`dict`) –

**Returns**

Object which has a `solve` method.

**Return type**

*cupyx.scipy.sparse.linalg.SuperLU*

---

**Note:** This function computes incomplete LU decomposition of a sparse matrix on the CPU using *scipy.sparse.linalg.spilu* (unless you set `fill_factor` to 1). Therefore, incomplete LU decomposition is not accelerated on the GPU. On the other hand, the computation of solving linear equations using the `solve` method, which this function returns, is performed on the GPU.

If you set `fill_factor` to 1, this function computes incomplete LU decomposition on the GPU, but without fill-in or pivoting.

---

**See also:**

`scipy.sparse.linalg.spilu()`

**cupyx.scipy.sparse.linalg.SuperLU**

```
class cupyx.scipy.sparse.linalg.SuperLU(obj)
```

**Methods**

```
solve(rhs, trans='N')
```

Solves linear system of equations with one or several right-hand sides.

**Parameters**

- **rhs** (`cupy.ndarray`) – Right-hand side(s) of equation with dimension (M) or (M, K).
- **trans** (`str`) – 'N', 'T' or 'H'. 'N': Solves  $A * x = rhs$ . 'T': Solves  $A.T * x = rhs$ . 'H': Solves  $A.conj().T * x = rhs$ .

**Returns**

Solution vector(s)

**Return type**

*cupy.ndarray*

```
__eq__(value, /)
```

Return self==value.

```
__ne__(value, /)
```

Return self!=value.

```
__lt__(value, /)
```

Return self<value.

```
__le__(value, /)
```

Return self<=value.

```
__gt__(value, /)
```

Return self>value.

```
__ge__(value, /)
```

Return self>=value.

### 5.4.10 Compressed sparse graph routines (`cupyx.scipy.sparse.csgraph`)

---

**Note:** The `csgraph` module uses `pylibcugraph` as a backend. You need to install *pylibcugraph* package <<https://anaconda.org/rapidsai/pylibcugraph>> from rapidsai Conda channel to use features listed on this page.

---



---

**Note:** Currently, the `csgraph` module is not supported on AMD ROCm platforms.

---



---

**Hint:** SciPy API Reference: Compressed sparse graph routines (`scipy.sparse.csgraph`)

---

#### Contents

---

<code>connected_components(csgraph[, directed, ...])</code>	Analyzes the connected components of a sparse graph
---	---

---

#### `cupyx.scipy.sparse.csgraph.connected_components`

`cupyx.scipy.sparse.csgraph.connected_components(csgraph, directed=True, connection='weak', return_labels=True)`

Analyzes the connected components of a sparse graph

##### Parameters

- **csgraph** (`cupy.ndarray` of `cupyx.scipy.sparse.csr_matrix`) – The adjacency matrix representing connectivity among nodes.
- **directed** (`bool`) – If True, it operates on a directed graph. If False, it operates on an undirected graph.
- **connection** (`str`) – 'weak' or 'strong'. For directed graphs, the type of connection to use. Nodes *i* and *j* are “strongly” connected only when a path exists both from *i* to *j* and from *j* to *i*. If `directed` is False, this argument is ignored.
- **return\_labels** (`bool`) – If True, it returns the labels for each of the connected components.

##### Returns

If `return_labels == True`, returns a tuple (`n`, `labels`), where `n` is the number of connected components and `labels` is labels of each connected components. Otherwise, returns `n`.

##### Return type

tuple of int and `cupy.ndarray`, or int

See also:

`scipy.sparse.csgraph.connected_components()`

### 5.4.11 Spatial algorithms and data structures (`cupyx.scipy.spatial`)

---

**Hint:** SciPy API Reference: `Spatial` (`scipy.spatial`)

---

---

**Note:** The `spatial` module uses `pylibraft` as a backend. You need to install *pylibraft* package <<https://anaconda.org/rapidsai/pylibraft>> from `rapidsai` Conda channel to use features listed on this page.

---

---

**Note:** Currently, the `spatial` module is not supported on AMD ROCm platforms.

---

#### Functions

---

<code>distance_matrix(x, y[, p])</code>	Compute the distance matrix.
---	------------------------------

---

#### `cupyx.scipy.spatial.distance_matrix`

`cupyx.scipy.spatial.distance_matrix(x, y, p=2.0)`

Compute the distance matrix.

Returns the matrix of all pair-wise distances.

##### Parameters

- **x** (*array\_like*) – Matrix of M vectors in K dimensions.
- **y** (*array\_like*) – Matrix of N vectors in K dimensions.
- **p** (*float*) – Which Minkowski p-norm to use ( $1 \leq p \leq \text{infinity}$ ). Default=2.0

##### Returns

**Matrix containing the distance from every**  
vector in *x* to every vector in *y*, (size M, N).

##### Return type

result (*cupy.ndarray*)

### 5.4.12 Distance computations (`cupyx.scipy.spatial.distance`)

---

**Note:** The `distance` module uses `pylibraft` as a backend. You need to install *pylibraft* package <<https://anaconda.org/rapidsai/pylibraft>> from `rapidsai` Conda channel to use features listed on this page.

---

---

**Note:** Currently, the `distance` module is not supported on AMD ROCm platforms.

---

---

**Hint:** SciPy API Reference: `Spatial distance routines` (`scipy.spatial.distance`)

---



## Distance matrix computations

Distance matrix computation from a collection of raw observation vectors stored in a rectangular array.

<code>pdist(X[, metric, out])</code>	Compute distance between observations in $n$ -dimensional space.
<code>cdist(XA, XB[, metric, out])</code>	Compute distance between each pair of the two collections of inputs.
<code>distance_matrix(x, y[, p])</code>	Compute the distance matrix.

### cupyx.scipy.spatial.distance.pdist

`cupyx.scipy.spatial.distance.pdist(X, metric='euclidean', *, out=None, **kwargs)`

Compute distance between observations in  $n$ -dimensional space.

#### Parameters

- **X** (*array\_like*) – An  $m$  by  $n$  array of  $m$  original observations in an  $n$ -dimensional space. Inputs are converted to float type.
- **metric** (*str, optional*) – The distance metric to use. The distance function can be ‘canberra’, ‘chebyshev’, ‘cityblock’, ‘correlation’, ‘cosine’, ‘euclidean’, ‘hamming’, ‘hellinger’, ‘jensenshannon’, ‘kl\_divergence’, ‘matching’, ‘minkowski’, ‘russellrao’, ‘sqeuclidean’.
- **out** (*cupy.ndarray, optional*) – The output array. If not None, the distance matrix  $Y$  is stored in this array.
- **\*\*kwargs** (*dict, optional*) – Extra arguments to *metric*: refer to each metric documentation for a list of all possible arguments. Some possible arguments:  $p$  (float): The  $p$ -norm to apply for Minkowski, weighted and unweighted. Default: 2.0

#### Returns

Returns a condensed distance matrix  $Y$ . For each  $i$  and  $j$  and (where  $i < j < m$ ), where  $m$  is the number of original observations. The metric  $\text{dist}(u=X[i], v=X[j])$  is computed and stored in entry  $m * i + j - ((i + 2) * (i + 1)) // 2$ .

#### Return type

$Y$  (*cupy.ndarray*)

### cupyx.scipy.spatial.distance.cdist

`cupyx.scipy.spatial.distance.cdist(XA, XB, metric='euclidean', out=None, **kwargs)`

Compute distance between each pair of the two collections of inputs.

#### Parameters

- **XA** (*array\_like*) – An  $m_A$  by  $n$  array of  $m_A$  original observations in an  $n$ -dimensional space. Inputs are converted to float type.
- **XB** (*array\_like*) – An  $m_B$  by  $n$  array of  $m_B$  original observations in an  $n$ -dimensional space. Inputs are converted to float type.
- **metric** (*str, optional*) – The distance metric to use. The distance function can be ‘canberra’, ‘chebyshev’, ‘cityblock’, ‘correlation’, ‘cosine’, ‘euclidean’, ‘hamming’, ‘hellinger’, ‘jensenshannon’, ‘kl\_divergence’, ‘matching’, ‘minkowski’, ‘russellrao’, ‘sqeuclidean’.

- **out** (`cupy.ndarray`, *optional*) – The output array. If not None, the distance matrix Y is stored in this array.
- **\*\*kwargs** (*dict*, *optional*) – Extra arguments to *metric*: refer to each metric documentation for a list of all possible arguments. Some possible arguments: *p* (float): The p-norm to apply for Minkowski, weighted and unweighted. Default: 2.0

**Returns**

A  $m_A$  by  $m_B$  distance matrix is

returned. For each  $i$  and  $j$ , the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the  $ij$  th entry.

**Return type**

Y (`cupy.ndarray`)

**cupyx.scipy.spatial.distance.distance\_matrix**

`cupyx.scipy.spatial.distance.distance_matrix(x, y, p=2.0)`

Compute the distance matrix.

Returns the matrix of all pair-wise distances.

**Parameters**

- **x** (*array\_like*) – Matrix of M vectors in K dimensions.
- **y** (*array\_like*) – Matrix of N vectors in K dimensions.
- **p** (*float*) – Which Minkowski p-norm to use ( $1 \leq p \leq \infty$ ). Default=2.0

**Returns**

Matrix containing the distance from every vector in *x* to every vector in *y*, (size M, N).

**Return type**

result (`cupy.ndarray`)

## Distance functions

Distance functions between two numeric vectors  $u$  and  $v$ . Computing distances over a large collection of vectors is inefficient for these functions. Use *cdist* for this purpose.

<i>minkowski</i> ( $u, v, p$ )	Compute the Minkowski distance between two 1-D arrays.
<i>canberra</i> ( $u, v$ )	Compute the Canberra distance between two 1-D arrays.
<i>chebyshev</i> ( $u, v$ )	Compute the Chebyshev distance between two 1-D arrays.
<i>cityblock</i> ( $u, v$ )	Compute the City Block (Manhattan) distance between two 1-D arrays.
<i>correlation</i> ( $u, v$ )	Compute the correlation distance between two 1-D arrays.
<i>cosine</i> ( $u, v$ )	Compute the Cosine distance between two 1-D arrays.
<i>hamming</i> ( $u, v$ )	Compute the Hamming distance between two 1-D arrays.
<i>euclidean</i> ( $u, v$ )	Compute the Euclidean distance between two 1-D arrays.
<i>jensenshannon</i> ( $u, v$ )	Compute the Jensen-Shannon distance between two 1-D arrays.
<i>russellrao</i> ( $u, v$ )	Compute the Russell-Rao distance between two 1-D arrays.
<i>squeuclidean</i> ( $u, v$ )	Compute the squared Euclidean distance between two 1-D arrays.
<i>hellinger</i> ( $u, v$ )	Compute the Hellinger distance between two 1-D arrays.
<i>kl_divergence</i> ( $u, v$ )	Compute the Kullback-Leibler divergence between two 1-D arrays.

### cupyx.scipy.spatial.distance.minkowski

`cupyx.scipy.spatial.distance.minkowski( $u, v, p$ )`

Compute the Minkowski distance between two 1-D arrays.

#### Parameters

- **$u$**  (*array\_like*) – Input array of size (N,)
- **$v$**  (*array\_like*) – Input array of size (N,)
- **$p$**  (*float*) – The order of the norm of the difference  $\|u - v\|_p$ . Note that for  $0 < p < 1$ , the triangle inequality only holds with an additional multiplicative factor, i.e. it is only a quasi-metric.

#### Returns

The Minkowski distance between vectors  $u$  and  $v$ .

#### Return type

minkowski (double)

**cupyx.scipy.spatial.distance.canberra**

`cupyx.scipy.spatial.distance.canberra(u, v)`

Compute the Canberra distance between two 1-D arrays.

The Canberra distance is defined as

$$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Canberra distance between vectors *u* and *v*.

**Return type**

canberra (double)

**cupyx.scipy.spatial.distance.chebyshev**

`cupyx.scipy.spatial.distance.chebyshev(u, v)`

Compute the Chebyshev distance between two 1-D arrays.

The Chebyshev distance is defined as

$$d(u, v) = \max_i |u_i - v_i|$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Chebyshev distance between vectors *u* and *v*.

**Return type**

chebyshev (double)

**cupyx.scipy.spatial.distance.cityblock**

`cupyx.scipy.spatial.distance.cityblock(u, v)`

Compute the City Block (Manhattan) distance between two 1-D arrays.

The City Block distance is defined as

$$d(u, v) = \sum_i |u_i - v_i|$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The City Block distance between vectors  $u$  and  $v$ .

**Return type**

cityblock (double)

**cupyx.scipy.spatial.distance.correlation**

`cupyx.scipy.spatial.distance.correlation( $u, v$ )`

Compute the correlation distance between two 1-D arrays.

The correlation distance is defined as

$$d(u, v) = 1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{\|(u - \bar{u})\|_2 \|(v - \bar{v})\|_2}$$

where  $\bar{u}$  is the mean of the elements of  $u$  and  $x \cdot y$  is the dot product.

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The correlation distance between vectors  $u$  and  $v$ .

**Return type**

correlation (double)

**cupyx.scipy.spatial.distance.cosine**

`cupyx.scipy.spatial.distance.cosine( $u, v$ )`

Compute the Cosine distance between two 1-D arrays.

The Cosine distance is defined as

$$d(u, v) = 1 - \frac{u \cdot v}{\|u\|_2 \|v\|_2}$$

where  $x \cdot y$  is the dot product.

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Cosine distance between vectors  $u$  and  $v$ .

**Return type**

cosine (double)

### cupyx.scipy.spatial.distance.hamming

cupyx.scipy.spatial.distance.hamming(*u*, *v*)

Compute the Hamming distance between two 1-D arrays.

The Hamming distance is defined as the proportion of elements in both *u* and *v* that are not in the exact same position:

$$d(u, v) = \frac{1}{n} \sum_{k=0}^n u_i \neq v_i$$

where  $x \neq y$  is one if *x* is different from *y* and zero otherwise.

#### Parameters

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

#### Returns

The Hamming distance between vectors *u* and *v*.

#### Return type

hamming (double)

### cupyx.scipy.spatial.distance.euclidean

cupyx.scipy.spatial.distance.euclidean(*u*, *v*)

Compute the Euclidean distance between two 1-D arrays.

The Euclidean distance is defined as

$$d(u, v) = \left( \sum_i (u_i - v_i)^2 \right)^{1/2}$$

#### Parameters

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

#### Returns

The Euclidean distance between vectors *u* and *v*.

#### Return type

euclidean (double)

### cupyx.scipy.spatial.distance.jensenshannon

cupyx.scipy.spatial.distance.jensenshannon(*u*, *v*)

Compute the Jensen-Shannon distance between two 1-D arrays.

The Jensen-Shannon distance is defined as

$$d(u, v) = \sqrt{\frac{KL(u||m) + KL(v||m)}{2}}$$

where *KL* is the Kullback-Leibler divergence and *m* is the pointwise mean of *u* and *v*.

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Jensen-Shannon distance between vectors  $u$  and  $v$ .

**Return type**

jensenshannon (double)

**cupyx.scipy.spatial.distance.russellrao**

`cupyx.scipy.spatial.distance.russellrao(u, v)`

Compute the Russell-Rao distance between two 1-D arrays.

The Russell-Rao distance is defined as the proportion of elements in both  $u$  and  $v$  that are in the exact same position:

$$d(u, v) = \frac{1}{n} \sum_{k=0}^n u_i = v_i$$

where  $x = y$  is one if  $x$  is different from  $y$  and zero otherwise.

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Hamming distance between vectors  $u$  and  $v$ .

**Return type**

hamming (double)

**cupyx.scipy.spatial.distance.sqeuclidean**

`cupyx.scipy.spatial.distance.sqeuclidean(u, v)`

Compute the squared Euclidean distance between two 1-D arrays.

The squared Euclidean distance is defined as

$$d(u, v) = \sum_i (u_i - v_i)^2$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The squared Euclidean distance between vectors  $u$  and  $v$ .

**Return type**

sqeuclidean (double)

**cupyx.scipy.spatial.distance.hellinger**

`cupyx.scipy.spatial.distance.hellinger(u, v)`

Compute the Hellinger distance between two 1-D arrays.

The Hellinger distance is defined as

$$d(u, v) = \frac{1}{\sqrt{2}} \sqrt{\sum_i (\sqrt{u_i} - \sqrt{v_i})^2}$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Hellinger distance between vectors *u* and *v*.

**Return type**

hellinger (double)

**cupyx.scipy.spatial.distance.kl\_divergence**

`cupyx.scipy.spatial.distance.kl_divergence(u, v)`

Compute the Kullback-Leibler divergence between two 1-D arrays.

The Kullback-Leibler divergence is defined as

$$KL(U\|V) = \sum_i U_i \log \left( \frac{U_i}{V_i} \right)$$

**Parameters**

- **u** (*array\_like*) – Input array of size (N,)
- **v** (*array\_like*) – Input array of size (N,)

**Returns**

The Kullback-Leibler divergence between vectors *u* and *v*.

**Return type**

kl\_divergence (double)

**5.4.13 Special functions (cupyx.scipy.special)**

---

**Hint:** SciPy API Reference: Special functions (`scipy.special`)

---



## Bessel functions

<code>j0</code>	Bessel function of the first kind of order 0.
<code>j1</code>	Bessel function of the first kind of order 1.
<code>k0</code>	Modified Bessel function of the second kind of order 0.
<code>k0e</code>	Exponentially scaled modified Bessel function K of order 0
<code>k1</code>	Modified Bessel function of the second kind of order 1.
<code>k1e</code>	Exponentially scaled modified Bessel function K of order 1
<code>y0</code>	Bessel function of the second kind of order 0.
<code>y1</code>	Bessel function of the second kind of order 1.
<code>yn</code>	Bessel function of the second kind of order n.
<code>i0</code>	Modified Bessel function of order 0.
<code>i0e</code>	Exponentially scaled modified Bessel function of order 0.
<code>i1</code>	Modified Bessel function of order 1.
<code>i1e</code>	Exponentially scaled modified Bessel function of order 1.
<code>spherical_yn(n, z[, derivative])</code>	Spherical Bessel function of the second kind or its derivative.

### `cupyx.scipy.special.j0`

`cupyx.scipy.special.j0()`

Bessel function of the first kind of order 0.

**See also:**

`scipy.special.j0()`

### `cupyx.scipy.special.j1`

`cupyx.scipy.special.j1()`

Bessel function of the first kind of order 1.

**See also:**

`scipy.special.j1()`

### `cupyx.scipy.special.k0`

`cupyx.scipy.special.k0()`

Modified Bessel function of the second kind of order 0.

#### Parameters

**x** (`cupy.ndarray`) – argument (float)

#### Returns

Value of the modified Bessel function K of order 0 at x.

**Return type***cupy.ndarray***See also:**`scipy.special.k0()`**cupyx.scipy.special.k0e**`cupyx.scipy.special.k0e()`

Exponentially scaled modified Bessel function K of order 0

**Parameters****x** (*cupy.ndarray*) – argument (float)**Returns**

Value at x.

**Return type***cupy.ndarray***See also:**`scipy.special.k0e()`**cupyx.scipy.special.k1**`cupyx.scipy.special.k1()`

Modified Bessel function of the second kind of order 1.

**Parameters****x** (*cupy.ndarray*) – argument (float)**Returns**

Value of the modified Bessel function K of order 1 at x.

**Return type***cupy.ndarray***See also:**`scipy.special.k1()`**cupyx.scipy.special.k1e**`cupyx.scipy.special.k1e()`

Exponentially scaled modified Bessel function K of order 1

**Parameters****x** (*cupy.ndarray*) – argument (float)**Returns**

Value at x.

**Return type***cupy.ndarray*

**See also:**

`scipy.special.k1e()`

### **cupyx.scipy.special.y0**

`cupyx.scipy.special.y0()`

Bessel function of the second kind of order 0.

**See also:**

`scipy.special.y0()`

### **cupyx.scipy.special.y1**

`cupyx.scipy.special.y1()`

Bessel function of the second kind of order 1.

**See also:**

`scipy.special.y1()`

### **cupyx.scipy.special.yn**

`cupyx.scipy.special.yn()`

Bessel function of the second kind of order  $n$ .

#### **Parameters**

- **n** (`cupy.ndarray`) – order (integer)
- **x** (`cupy.ndarray`) – argument (float)

#### **Returns**

The result.

#### **Return type**

*cupy.ndarray*

#### **Notes**

Unlike SciPy, no warning will be raised on unsafe casting of *order* to 32-bit integer.

**See also:**

`scipy.special.yn()`

### `cupyx.scipy.special.i0`

`cupyx.scipy.special.i0()`

Modified Bessel function of order 0.

**See also:**

`scipy.special.i0()`

### `cupyx.scipy.special.i0e`

`cupyx.scipy.special.i0e()`

Exponentially scaled modified Bessel function of order 0.

**See also:**

`scipy.special.i0e()`

### `cupyx.scipy.special.i1`

`cupyx.scipy.special.i1()`

Modified Bessel function of order 1.

**See also:**

`scipy.special.i1()`

### `cupyx.scipy.special.i1e`

`cupyx.scipy.special.i1e()`

Exponentially scaled modified Bessel function of order 1.

**See also:**

`scipy.special.i1e()`

### `cupyx.scipy.special.spherical_yn`

`cupyx.scipy.special.spherical_yn(n, z, derivative=False)`

Spherical Bessel function of the second kind or its derivative.

**Parameters**

- **n** (`cupy.ndarray`) – Order of the Bessel function.
- **z** (`cupy.ndarray`) – Argument of the Bessel function. Real-valued input.
- **derivative** (*bool*, *optional*) – If True, the value of the derivative (rather than the function itself) is returned.

**Returns**

**yn**

**Return type**

*cupy.ndarray*

See also:

`scipy.special.spherical_yn()`

## Raw statistical functions

See also:

`cupyx.scipy.stats`

<i>bdtr</i>	Binomial distribution cumulative distribution function.
<i>bdtrc</i>	Binomial distribution survival function.
<i>bdtri</i>	Inverse function to <i>bdtr</i> with respect to <i>p</i> .
<i>bdtr</i>	Cumulative distribution function of the beta distribution.
<i>bdtri</i>	The <i>p</i> -th quantile of the beta distribution.
<i>fdtr</i>	F cumulative distribution function.
<i>fdtrc</i>	F survival function.
<i>fdtri</i>	The <i>p</i> -th quantile of the F-distribution.
<i>gdtr</i>	Gamma distribution cumulative distribution function.
<i>gdtrc</i>	Gamma distribution survival function.
<i>nbdtr</i>	Negative binomial distribution cumulative distribution function.
<i>nbdtrc</i>	Negative binomial distribution survival function.
<i>nbdtri</i>	Inverse function to <i>nbdtr</i> with respect to <i>p</i> .
<i>pdtr</i>	Poisson cumulative distribution function.
<i>pdtrc</i>	Binomial distribution survival function.
<i>pdtri</i>	Inverse function to <i>pdtr</i> with respect to <i>m</i> .
<i>chdtr</i>	Chi-square cumulative distribution function.
<i>chdtrc</i>	Chi square survival function.
<i>chdtri</i>	Inverse to <i>chdtrc</i> with respect to <i>x</i> .
<i>ndtr</i>	Cumulative distribution function of normal distribution.
<i>log_ndtr</i>	Logarithm of Gaussian cumulative distribution function.
<i>ndtri</i>	Inverse of the cumulative distribution function of the standard
<i>logit</i>	Logit function.
<i>expit</i>	Logistic sigmoid function (expit).
<i>log_expit</i>	Logarithm of the logistic sigmoid function.
<i>boxcox</i>	Compute the Box-Cox transformation.
<i>boxcox1p</i>	Compute the Box-Cox transformation on $1 + x$ .
<i>inv_boxcox</i>	Compute the Box-Cox transformation.
<i>inv_boxcox1p</i>	Compute the Box-Cox transformation on $1 + x$ .

## cupyx.scipy.special.bdtr

`cupyx.scipy.special.bdtr()`

Binomial distribution cumulative distribution function.

### Parameters

- **k** (`cupy.ndarray`) – Number of successes (float), rounded down to the nearest integer.
- **n** (`cupy.ndarray`) – Number of events (int).
- **p** (`cupy.ndarray`) – Probability of success in a single event (float).

**Returns**

**y** – Probability of floor(**k**) or fewer successes in **n** independent events with success probabilities of **p**.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.bdtr()`

**cupyx.scipy.special.bdtrc**

`cupyx.scipy.special.bdtrc()`

Binomial distribution survival function.

Returns the complemented binomial distribution function (the integral of the density from **x** to infinity).

**Parameters**

- **k** (*cupy.ndarray*) – Number of successes (float), rounded down to the nearest integer.
- **n** (*cupy.ndarray*) – Number of events (int).
- **p** (*cupy.ndarray*) – Probability of success in a single event (float).

**Returns**

**y** – Probability of floor(**k**) + 1 or more successes in **n** independent events with success probabilities of **p**.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.bdtrc()`

**cupyx.scipy.special.bdtri**

`cupyx.scipy.special.bdtri()`

Inverse function to *bdtr* with respect to *p*.

**Parameters**

- **k** (*cupy.ndarray*) – Number of successes (float), rounded down to the nearest integer.
- **n** (*cupy.ndarray*) – Number of events (int).
- **y** (*cupy.ndarray*) – Cumulative probability (probability of **k** or fewer successes in **n** events).

**Returns**

**p** – The event probability such that `bdtr(floor(k), n, p) = y`.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.bdtri()`

**cupyx.scipy.special.btdtr**`cupyx.scipy.special.btdtr()`

Cumulative distribution function of the beta distribution.

**Parameters**

- **a** (`cupy.ndarray`) – Shape parameter ( $a > 0$ ).
- **b** (`cupy.ndarray`) – Shape parameter ( $b > 0$ ).
- **x** (`cupy.ndarray`) – Upper limit of integration, in  $[0, 1]$ .

**Returns**

**I** – Cumulative distribution function of the beta distribution with parameters *a* and *b* at *x*.

**Return type**

*cupy.ndarray*

See also:

`scipy.special.btdtr()`

**cupyx.scipy.special.btdtri**`cupyx.scipy.special.btdtri()`

The *p*-th quantile of the beta distribution.

This function is the inverse of the beta cumulative distribution function, *btdtr*, returning the value of *x* for which  $\text{btdtr}(a, b, x) = p$ .

**Parameters**

- **a** (`cupy.ndarray`) – Shape parameter ( $a > 0$ ).
- **b** (`cupy.ndarray`) – Shape parameter ( $b > 0$ ).
- **p** (`cupy.ndarray`) – Cumulative probability, in  $[0, 1]$ .

**Returns**

**x** – The quantile corresponding to *p*.

**Return type**

*cupy.ndarray*

See also:

`scipy.special.btdtri()`

**cupyx.scipy.special.fdtr**`cupyx.scipy.special.fdtr()`

F cumulative distribution function.

**Parameters**

- **dfn** (`cupy.ndarray`) – First parameter (positive float).
- **dfd** (`cupy.ndarray`) – Second parameter (positive float).
- **x** (`cupy.ndarray`) – Argument (nonnegative float).

**Returns**

- **y** (*cupy.ndarray*) – The CDF of the F-distribution with parameters *dfn* and *dfd* at *x*.
- **See also:**  
`scipy.special.fctr()`

**cupyx.scipy.special.fctr**`cupyx.scipy.special.fctr()`

F survival function.

Returns the complemented F-distribution function (the integral of the density from *x* to infinity).

**Parameters**

- **dfn** (*cupy.ndarray*) – First parameter (positive float).
- **dfd** (*cupy.ndarray*) – Second parameter (positive float).
- **x** (*cupy.ndarray*) – Argument (nonnegative float).

**Returns**

- **y** (*cupy.ndarray*) – The complemented F-distribution function with parameters *dfn* and *dfd* at *x*.
- **See also:**  
`scipy.special.fctr()`

**cupyx.scipy.special.fctri**`cupyx.scipy.special.fctri()`

The *p*-th quantile of the F-distribution.

This function is the inverse of the F-distribution CDF, *fctr*, returning the *x* such that  $fctr(dfn, dfd, x) = p$ .

**Parameters**

- **dfn** (*cupy.ndarray*) – First parameter (positive float).
- **dfd** (*cupy.ndarray*) – Second parameter (positive float).
- **p** (*cupy.ndarray*) – Cumulative probability, in  $[0, 1]$ .

**Returns**

- **y** (*cupy.ndarray*) – The quantile corresponding to *p*.
- **See also:**  
`scipy.special.fctri()`



**cupyx.scipy.special.gdtr**`cupyx.scipy.special.gdtr()`

Gamma distribution cumulative distribution function.

**Parameters**

- **a** (`cupy.ndarray`) – The rate parameter of the gamma distribution, sometimes denoted beta (float). It is also the reciprocal of the scale parameter theta.
- **b** (`cupy.ndarray`) – The shape parameter of the gamma distribution, sometimes denoted alpha (float).
- **x** (`cupy.ndarray`) – The quantile (upper limit of integration; float).

**Returns****F** – The CDF of the gamma distribution with parameters *a* and *b* evaluated at *x*.**Return type***cupy.ndarray***See also:**`scipy.special.gdtr()`**cupyx.scipy.special.gdtrc**`cupyx.scipy.special.gdtrc()`

Gamma distribution survival function.

**Parameters**

- **a** (`cupy.ndarray`) – The rate parameter of the gamma distribution, sometimes denoted beta (float). It is also the reciprocal of the scale parameter theta.
- **b** (`cupy.ndarray`) – The shape parameter of the gamma distribution, sometimes denoted alpha (float).
- **x** (`cupy.ndarray`) – The quantile (lower limit of integration; float).

**Returns****I** – The survival function of the gamma distribution with parameters *a* and *b* at *x*.**Return type***cupy.ndarray***See also:**`scipy.special.gdtrc()`

**cupyx.scipy.special.nbdtr**`cupyx.scipy.special.nbdtr()`

Negative binomial distribution cumulative distribution function.

**Parameters**

- **k** (`cupy.ndarray`) – The maximum number of allowed failures (nonnegative int).
- **n** (`cupy.ndarray`) – The target number of successes (positive int).
- **p** (`cupy.ndarray`) – Probability of success in a single event (float).

**Returns**

**F** – The probability of  $k$  or fewer failures before  $n$  successes in a sequence of events with individual success probability  $p$ .

**Return type**

*cupy.ndarray*

See also:

`scipy.special.nbdtr()`

**cupyx.scipy.special.nbdtrc**`cupyx.scipy.special.nbdtrc()`

Negative binomial distribution survival function.

**Parameters**

- **k** (`cupy.ndarray`) – The maximum number of allowed failures (nonnegative int).
- **n** (`cupy.ndarray`) – The target number of successes (positive int).
- **p** (`cupy.ndarray`) – Probability of success in a single event (float).

**Returns**

**F** – The probability of  $k + 1$  or more failures before  $n$  successes in a sequence of events with individual success probability  $p$ .

**Return type**

*cupy.ndarray*

See also:

`scipy.special.nbdtrc()`

**cupyx.scipy.special.nbdtri**`cupyx.scipy.special.nbdtri()`

Inverse function to *nbdtr* with respect to  $p$ .

**Parameters**

- **k** (`cupy.ndarray`) – The maximum number of allowed failures (nonnegative int).
- **n** (`cupy.ndarray`) – The target number of successes (positive int).
- **y** (`cupy.ndarray`) – The probability of  $k$  or fewer failures before  $n$  successes (float).

**Returns**

**p** – Probability of success in a single event (float) such that `nbdtr(k, n, p) = y`.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.nbdtri()`

**cupyx.scipy.special.pdtr**

`cupyx.scipy.special.pdtr()`

Poisson cumulative distribution function.

**Parameters**

- **k** (*cupy.ndarray*) – Nonnegative real argument.
- **m** (*cupy.ndarray*) – Nonnegative real shape parameter.

**Returns**

**y** – Values of the Poisson cumulative distribution function.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.pdtr()`

**cupyx.scipy.special.pdtrc**

`cupyx.scipy.special.pdtrc()`

Binomial distribution survival function.

Returns the complemented binomial distribution function (the integral of the density from x to infinity).

**Parameters**

- **k** (*cupy.ndarray*) – Nonnegative real argument.
- **m** (*cupy.ndarray*) – Nonnegative real shape parameter.

**Returns**

**y** – The sum of the terms from k+1 to infinity of the Poisson distribution.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.pdtrc()`

### cupyx.scipy.special.pdtri

cupyx.scipy.special.pdtri()

Inverse function to *pdtr* with respect to *m*.

#### Parameters

- **k** (`cupy.ndarray`) – Nonnegative real argument.
- **y** (`cupy.ndarray`) – Cumulative probability.

#### Returns

**m** – The Poisson variable *m* such that the sum from 0 to *k* of the Poisson density is equal to the given probability *y*.

#### Return type

*cupy.ndarray*

See also:

`scipy.special.pdtri()`

### cupyx.scipy.special.chdtr

cupyx.scipy.special.chdtr()

Chi-square cumulative distribution function.

#### Parameters

- **v** (`cupy.ndarray`) – Degrees of freedom.
- **x** (`cupy.ndarray`) – Upper bound of the integral (nonnegative float).

#### Returns

**y** – The CDF of the chi-squared distribution with parameter *df* at *x*.

#### Return type

*cupy.ndarray*

See also:

`scipy.special.chdtr()`

### cupyx.scipy.special.chdtrc

cupyx.scipy.special.chdtrc()

Chi square survival function.

Returns the complemented chi-squared distribution function (the integral of the density from *x* to infinity).

#### Parameters

- **v** (`cupy.ndarray`) – Degrees of freedom.
- **x** (`cupy.ndarray`) – Upper bound of the integral (nonnegative float).

#### Returns

**y** – The complemented chi-squared distribution function with parameter *df* at *x*.

#### Return type

*cupy.ndarray*

See also:

`scipy.special.chdtrc()`

### `cupyx.scipy.special.chdtri`

`cupyx.scipy.special.chdtri()`

Inverse to *chdtrc* with respect to *x*.

#### Parameters

- **v** (`cupy.ndarray`) – Degrees of freedom.
- **p** (`cupy.ndarray`, *optional*) – Probability.
- **p** – Optional output array for the function results.

#### Returns

*x* – Value so that the probability a Chi square random variable with *v* degrees of freedom is greater than *x* equals *p*.

#### Return type

*cupy.ndarray*

See also:

`scipy.special.chdtri()`

### `cupyx.scipy.special.ndtr`

`cupyx.scipy.special.ndtr()`

Cumulative distribution function of normal distribution.

See also:

`scipy.special.ndtr`

### `cupyx.scipy.special.log_ndtr`

`cupyx.scipy.special.log_ndtr()`

Logarithm of Gaussian cumulative distribution function.

Returns the log of the area under the standard Gaussian propability density function.

#### Parameters

**x** (*array-like*) – The input array

#### Returns

*y* – The value of the log of the normal cumulative distribution function evaluated at *x*

#### Return type

*cupy.ndarray*

See also:

`scipy.special.log_ndtr()`

**cupyx.scipy.special.ndtri**`cupyx.scipy.special.ndtri()`

**Inverse of the cumulative distribution function of the standard normal distribution.**

**See also:**

`scipy.special.ndtri`

**cupyx.scipy.special.logit**`cupyx.scipy.special.logit()`

Logit function.

**Parameters**

**x** (`cupy.ndarray`) – input data

**Returns**

values of `logit(x)`

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.logit`

**cupyx.scipy.special.expit**`cupyx.scipy.special.expit()`

Logistic sigmoid function (`expit`).

**Parameters**

**x** (`cupy.ndarray`) – input data (must be real)

**Returns**

values of `expit(x)`

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.expit`

---

**Note:** `expit` is the inverse of `logit`.

---

**cupyx.scipy.special.log\_expit****cupyx.scipy.special.log\_expit()**

Logarithm of the logistic sigmoid function.

**Parameters****x** ([cupy.ndarray](#)) – input data (must be real)**Returns**values of  $\log(\text{expit}(x))$ **Return type***cupy.ndarray***See also:**[scipy.special.log\\_expit](#)


---

**Note:** The function is mathematically equivalent to  $\log(\text{expit}(x))$ , but is formulated to avoid loss of precision for inputs with large (positive or negative) magnitude.

---

**cupyx.scipy.special.boxcox****cupyx.scipy.special.boxcox()**

Compute the Box-Cox transformation.

**Parameters****x** ([cupy.ndarray](#)) – input data (must be real)**Returns**values of  $\text{boxcox}(x)$ **Return type***cupy.ndarray***See also:**[scipy.special.boxcox](#)**cupyx.scipy.special.boxcox1p****cupyx.scipy.special.boxcox1p()**Compute the Box-Cox transformation on  $1 + x$ .**Parameters****x** ([cupy.ndarray](#)) – input data (must be real)**Returns**values of  $\text{boxcox1p}(x)$ **Return type***cupy.ndarray***See also:**[scipy.special.boxcox1p](#)

**cupyx.scipy.special.inv\_boxcox****cupyx.scipy.special.inv\_boxcox()**

Compute the Box-Cox transformation.

**Parameters****x** ([cupy.ndarray](#)) – input data (must be real)**Returns**values of `inv_boxcox(x)`**Return type***cupy.ndarray***See also:**[scipy.special.inv\\_boxcox](#)**cupyx.scipy.special.inv\_boxcox1p****cupyx.scipy.special.inv\_boxcox1p()**Compute the Box-Cox transformation on  $1 + x$ .**Parameters****x** ([cupy.ndarray](#)) – input data (must be real)**Returns**values of `inv_boxcox1p(x)`**Return type***cupy.ndarray***See also:**[scipy.special.inv\\_boxcox1p](#)**Information Theory functions**

<a href="#">entr</a>	Elementwise function for computing entropy.
<a href="#">rel_entr</a>	Elementwise function for computing relative entropy.
<a href="#">kl_div</a>	Elementwise function for computing Kullback-Leibler divergence.
<a href="#">huber</a>	Elementwise function for computing the Huber loss.
<a href="#">pseudo_huber</a>	Elementwise function for computing the Pseudo-Huber loss.



**cupyx.scipy.special.entr**

`cupyx.scipy.special.entr()`

Elementwise function for computing entropy.

**See also:**

`scipy.special.entr()`

**cupyx.scipy.special.rel\_entr**

`cupyx.scipy.special.rel_entr()`

Elementwise function for computing relative entropy.

**See also:**

`scipy.special.rel_entr()`

**cupyx.scipy.special.kl\_div**

`cupyx.scipy.special.kl_div()`

Elementwise function for computing Kullback-Leibler divergence.

**See also:**

`scipy.special.kl_div()`

**cupyx.scipy.special.huber**

`cupyx.scipy.special.huber()`

Elementwise function for computing the Huber loss.

**See also:**

`scipy.special.huber()`

**cupyx.scipy.special.pseudo\_huber**

`cupyx.scipy.special.pseudo_huber()`

Elementwise function for computing the Pseudo-Huber loss.

**See also:**

`scipy.special.pseudo_huber()`

## Gamma and related functions

<a href="#"><i>gamma</i></a>	Gamma function.
<a href="#"><i>gammaln</i></a>	Logarithm of the absolute value of the Gamma function.
<a href="#"><i>loggamma</i></a>	Principal branch of the logarithm of the gamma function.
<a href="#"><i>gammainc</i></a>	Elementwise function for <code>scipy.special.gammainc</code>
<a href="#"><i>gammaincinv</i></a>	Elementwise function for <code>scipy.special.gammaincinv</code>
<a href="#"><i>gammaincc</i></a>	Elementwise function for <code>scipy.special.gammaincc</code>
<a href="#"><i>gammainccinv</i></a>	Elementwise function for <code>scipy.special.gammainccinv</code>
<a href="#"><i>beta</i></a>	Beta function.
<a href="#"><i>betaln</i></a>	Natural logarithm of absolute value of beta function.
<a href="#"><i>betainc</i></a>	Incomplete beta function.
<a href="#"><i>betaincinv</i></a>	Inverse of the incomplete beta function.
<a href="#"><i>psi</i></a>	The digamma function.
<a href="#"><i>rgamma</i></a>	Reciprocal gamma function.
<a href="#"><i>polygamma</i>(n, x)</a>	Polygamma function n.
<a href="#"><i>multigammaln</i>(a, d)</a>	Returns the log of multivariate gamma, also sometimes called the generalized gamma.
<a href="#"><i>digamma</i></a>	The digamma function.
<a href="#"><i>poch</i></a>	Elementwise function for <code>scipy.special.poch</code> (Pochhammer symbol)

### cupyx.scipy.special.gamma

`cupyx.scipy.special.gamma()`

Gamma function.

**Parameters**

**z** (`cupy.ndarray`) – The input of gamma function.

**Returns**

Computed value of gamma function.

**Return type**

`cupy.ndarray`

**See also:**

`scipy.special.gamma()`

### cupyx.scipy.special.gammaln

`cupyx.scipy.special.gammaln()`

Logarithm of the absolute value of the Gamma function.

**Parameters**

- **x** (`cupy.ndarray`) – Values on the real line at which to compute
- **gammaln.** –

**Returns**

Values of `gammaln` at x.

**Return type***cupy.ndarray***See also:**`scipy.special.gammaln`**cupyx.scipy.special.loggamma**`cupyx.scipy.special.loggamma()`

Principal branch of the logarithm of the gamma function.

**Parameters**

- **z** (*cupy.ndarray*) – Values in the complex plane at which to compute loggamma
- **out** (*cupy.ndarray*, *optional*) – Output array for computed values of loggamma

**Returns**

Values of loggamma at z.

**Return type***cupy.ndarray***See also:**`scipy.special.loggamma()`**cupyx.scipy.special.gammainc**`cupyx.scipy.special.gammainc()`Elementwise function for `scipy.special.gammainc`

Regularized lower incomplete gamma function.

**See also:**`scipy.special.gammainc()`**cupyx.scipy.special.gammaincinv**`cupyx.scipy.special.gammaincinv()`Elementwise function for `scipy.special.gammaincinv`Inverse to `gammainc`.**See also:**`scipy.special.gammaincinv()`

### cupyx.scipy.special.gammaincc

cupyx.scipy.special.gammaincc()

Elementwise function for scipy.special.gammaincc

Regularized upper incomplete gamma function.

**See also:**

scipy.special.gammaincc()

### cupyx.scipy.special.gammainccinv

cupyx.scipy.special.gammainccinv()

Elementwise function for scipy.special.gammainccinv

Inverse to gammaincc.

**See also:**

scipy.special.gammainccinv()

### cupyx.scipy.special.beta

cupyx.scipy.special.beta()

Beta function.

#### Parameters

- **a** ([cupy.ndarray](#)) – Real-valued arguments
- **b** ([cupy.ndarray](#)) – Real-valued arguments
- **out** ([cupy.ndarray](#), *optional*) – Optional output array for the function result

#### Returns

Value of the beta function

#### Return type

scalar or [ndarray](#)

**See also:**

scipy.special.beta()

### cupyx.scipy.special.betaln

cupyx.scipy.special.betaln()

Natural logarithm of absolute value of beta function.

Computes  $\ln(\text{abs}(\text{beta}(a, b)))$ .

#### Parameters

- **a** ([cupy.ndarray](#)) – Real-valued arguments
- **b** ([cupy.ndarray](#)) – Real-valued arguments
- **out** ([cupy.ndarray](#), *optional*) – Optional output array for the function result

**Returns**

Value of the natural log of the magnitude of beta.

**Return type**

scalar or *ndarray*

**See also:**

`scipy.special.betaln()`

**cupyx.scipy.special.betainc**

`cupyx.scipy.special.betainc()`

Incomplete beta function.

**Parameters**

- **a** (*cupy.ndarray*) – Positive, real-valued parameters
- **b** (*cupy.ndarray*) – Positive, real-valued parameters
- **x** (*cupy.ndarray*) – Real-valued such that  $0 \leq x \leq 1$ , the upper limit of integration.
- **out** (*ndarray*, *optional*) – Optional output array for the function result

**Returns**

Value of the incomplete beta function

**Return type**

scalar or *ndarray*

**See also:**

`scipy.special.betainc()`

**cupyx.scipy.special.betaincinv**

`cupyx.scipy.special.betaincinv()`

Inverse of the incomplete beta function.

**Parameters**

- **a** (*cupy.ndarray*) – Positive, real-valued parameters
- **b** (*cupy.ndarray*) – Positive, real-valued parameters
- **y** (*cupy.ndarray*) – Real-valued input.
- **out** (*ndarray*, *optional*) – Optional output array for the function result

**Returns**

Value of the inverse of the incomplete beta function

**Return type**

scalar or *ndarray*

**See also:**

`scipy.special.betaincinv()`

**cupyx.scipy.special.psi****cupyx.scipy.special.psi()**

The digamma function.

**Parameters****x** ([cupy.ndarray](#)) – The input of digamma function.**Returns**

Computed value of digamma function.

**Return type***cupy.ndarray***See also:**[scipy.special.digamma](#)**cupyx.scipy.special.rgamma****cupyx.scipy.special.rgamma()**

Reciprocal gamma function.

**Parameters****z** ([cupy.ndarray](#)) – The input to the rgamma function.**Returns**

Computed value of the rgamma function.

**Return type***cupy.ndarray***See also:**[scipy.special.rgamma\(\)](#)**cupyx.scipy.special.polygamma****cupyx.scipy.special.polygamma(*n*, *x*)**Polygamma function *n*.**Parameters**

- **n** ([cupy.ndarray](#)) – The order of the derivative of *psi*.
- **x** ([cupy.ndarray](#)) – Where to evaluate the polygamma function.

**Returns**

The result.

**Return type***cupy.ndarray***See also:**[scipy.special.polygamma](#)

**cupyx.scipy.special.multigammaln**`cupyx.scipy.special.multigammaln(a, d)`

Returns the log of multivariate gamma, also sometimes called the generalized gamma.

**Parameters**

- **a** (`cupy.ndarray`) – The multivariate gamma is computed for each item of *a*.
- **d** (`int`) – The dimension of the space of integration.

**Returns**

**res** – The values of the log multivariate gamma at the given points *a*.

**Return type**

*ndarray*

**See also:**

`scipy.special.multigammaln()`

**cupyx.scipy.special.digamma**`cupyx.scipy.special.digamma()`

The digamma function.

**Parameters**

- **x** (`cupy.ndarray`) – The input of digamma function.

**Returns**

Computed value of digamma function.

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.digamma`

**cupyx.scipy.special.poch**`cupyx.scipy.special.poch()`

Elementwise function for `scipy.special.poch` (Pochhammer symbol)

**See also:**

`scipy.special.poch()`

## Elliptic integrals

`ellipk ellipkm1 ellipj`

## Error function and Fresnel integrals

<i>erf</i>	Error function.
<i>erfc</i>	Complementary error function.
<i>erfcx</i>	Scaled complementary error function.
<i>erfinv</i>	Inverse function of error function.
<i>erfcinv</i>	Inverse function of complementary error function.

### `cupyx.scipy.special.erf`

`cupyx.scipy.special.erf()`

Error function.

**See also:**

`scipy.special.erf()`

### `cupyx.scipy.special.erfc`

`cupyx.scipy.special.erfc()`

Complementary error function.

**See also:**

`scipy.special.erfc()`

### `cupyx.scipy.special.erfcx`

`cupyx.scipy.special.erfcx()`

Scaled complementary error function.

**See also:**

`scipy.special.erfcx()`

### `cupyx.scipy.special.erfinv`

`cupyx.scipy.special.erfinv()`

Inverse function of error function.

**See also:**

`scipy.special.erfinv()`

---

**Note:** The behavior close to (and outside) the domain follows that of SciPy v1.4.0+.

---



**cupyx.scipy.special.erfcinv**`cupyx.scipy.special.erfcinv()`

Inverse function of complementary error function.

**See also:**`scipy.special.erfcinv()`

---

**Note:** The behavior close to (and outside) the domain follows that of SciPy v1.4.0+.

---

**Legendre functions**

<i>lpmv</i>	Associated Legendre function of integer order and real degree.
<i>sph_harm</i>	Spherical Harmonic.

**cupyx.scipy.special.lpmv**`cupyx.scipy.special.lpmv()`

Associated Legendre function of integer order and real degree.

**See also:**`scipy.special.lpmv()`**cupyx.scipy.special.sph\_harm**`cupyx.scipy.special.sph_harm()`

Spherical Harmonic.

**See also:**`scipy.special.sph_harm()`**Other special functions**

<i>exp1</i>	Exponential integral E1.
<i>expi</i>	Exponential integral Ei.
<i>expn</i>	Generalized exponential integral En.
<i>exprel</i>	Computes $(\exp(x) - 1) / x$ .
<i>softmax</i> (x[, axis])	Softmax function.
<i>log_softmax</i> (x[, axis])	Compute logarithm of softmax function
<i>zeta</i>	Hurwitz zeta function.
<i>zetac</i>	Riemann zeta function minus 1.

**cupyx.scipy.special.exp1**`cupyx.scipy.special.exp1()`

Exponential integral E1.

**Parameters****x** (`cupy.ndarray`) – Real argument**Returns****y** – Values of the exponential integral E1**Return type**scalar or `cupy.ndarray`**See also:**`scipy.special.exp1()`**cupyx.scipy.special.expi**`cupyx.scipy.special.expi()`

Exponential integral Ei.

**Parameters****x** (`cupy.ndarray`) – Real argument**Returns****y** – Values of exponential integral**Return type**scalar or `cupy.ndarray`**See also:**`scipy.special.expi()`**cupyx.scipy.special.expn**`cupyx.scipy.special.expn()`

Generalized exponential integral En.

**Parameters**

- **n** (`cupy.ndarray`) – Non-negative integers
- **x** (`cupy.ndarray`) – Real argument

**Returns****y** – Values of the generalized exponential integral**Return type**scalar or `cupy.ndarray`**See also:**`scipy.special.expn()`

**cupyx.scipy.special.exprel**`cupyx.scipy.special.exprel()`Computes  $(\exp(x) - 1) / x$ .**See also:**`scipy.special.exprel()`**cupyx.scipy.special.softmax**`cupyx.scipy.special.softmax(x, axis=None)`

Softmax function.

The softmax function transforms each element of a collection by computing the exponential of each element divided by the sum of the exponentials of all the elements.

**Parameters**

- **x** (*array-like*) – The input array
- **axis** (*int or tuple of ints, optional*) – Axis to compute values along. Default is None

**Returns**

**s** – Returns an array with same shape as input. The result will sum to 1 along the provided axis

**Return type***cupy.ndarray***cupyx.scipy.special.log\_softmax**`cupyx.scipy.special.log_softmax(x, axis=None)`

Compute logarithm of softmax function

**Parameters**

- **x** (*array-like*) – Input array
- **axis** (*int or tuple of ints, optional*) – Axis to compute values along. Default is None and softmax will be computed over the entire array *x*

**Returns**

**s** – An array with the same shape as *x*. Exponential of the result will sum to 1 along the specified axis. If *x* is a scalar, a scalar is returned

**Return type***cupy.ndarry*

**cupyx.scipy.special.zeta**`cupyx.scipy.special.zeta()`

Hurwitz zeta function.

**Parameters**

- **x** (`cupy.ndarray`) – Input data, must be real.
- **q** (`cupy.ndarray`) – Input data, must be real.

**Returns**

Values of  $\zeta(x, q)$ .

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.zeta`

**cupyx.scipy.special.zetac**`cupyx.scipy.special.zetac()`

Riemann zeta function minus 1.

**Parameters**

- **x** (`cupy.ndarray`) – Input data, must be real.

**Returns**

Values of  $\zeta(x)-1$ .

**Return type**

*cupy.ndarray*

**See also:**

`scipy.special.zetac`

## Convenience functions

<code>cbrt</code>	Cube root.
<code>exp10</code>	Computes $10^{**x}$ .
<code>exp2</code>	Computes $2^{**x}$ .
<code>radian</code>	Degrees, minutes, seconds to radians:
<code>cosdg</code>	Cosine of $x$ with $x$ in degrees.
<code>sindg</code>	Sine of $x$ with $x$ in degrees.
<code>tandg</code>	Tangent of $x$ with $x$ in degrees.
<code>cotdg</code>	Cotangent of $x$ with $x$ in degrees.
<code>log1p</code>	Elementwise function for <code>scipy.special.log1p</code>
<code>expm1</code>	Computes $\exp(x) - 1$ .
<code>cosm1</code>	Computes $\cos(x) - 1$ .
<code>round(a[, decimals, out])</code>	
<code>xlogy</code>	Compute $x \cdot \log(y)$ so that the result is 0 if $x = 0$ .
<code>xlog1py</code>	Compute $x \cdot \log1p(y)$ so that the result is 0 if $x = 0$ .
<code>logsumexp(a[, axis, b, keepdims, return_sign])</code>	Compute the log of the sum of exponentials of input elements.
<code>sinc(x, /[, out, casting, dtype])</code>	Elementwise sinc function.

### cupyx.scipy.special.cbrt

`cupyx.scipy.special.cbrt()`

Cube root.

**See also:**

`scipy.special.cbrt()`

### cupyx.scipy.special.exp10

`cupyx.scipy.special.exp10()`

Computes  $10^{**x}$ .

**See also:**

`scipy.special.exp10()`

### cupyx.scipy.special.exp2

`cupyx.scipy.special.exp2()`

Computes  $2^{**x}$ .

**See also:**

`scipy.special.exp2()`

### **cupyx.scipy.special.radian**

`cupyx.scipy.special.radian()`

Degrees, minutes, seconds to radians:

**See also:**

`scipy.special.radian()`

### **cupyx.scipy.special.cosdg**

`cupyx.scipy.special.cosdg()`

Cosine of  $x$  with  $x$  in degrees.

**See also:**

`scipy.special.cosdg()`

### **cupyx.scipy.special.sindg**

`cupyx.scipy.special.sindg()`

Sine of  $x$  with  $x$  in degrees.

**See also:**

`scipy.special.sindg()`

### **cupyx.scipy.special.tandg**

`cupyx.scipy.special.tandg()`

Tangent of  $x$  with  $x$  in degrees.

**See also:**

`scipy.special.tandg()`

### **cupyx.scipy.special.cotdg**

`cupyx.scipy.special.cotdg()`

Cotangent of  $x$  with  $x$  in degrees.

**See also:**

`scipy.special.cotdg()`

**cupyx.scipy.special.log1p****cupyx.scipy.special.log1p()**Elementwise function for `scipy.special.log1p`Calculates  $\log(1 + x)$  for use when  $x$  is near zero.**Notes**This implementation currently does not support complex-valued  $x$ .**See also:**`scipy.special.log1p()`**cupyx.scipy.special.expm1****cupyx.scipy.special.expm1()**Computes  $\exp(x) - 1$ .**See also:**`scipy.special.expm1()`**cupyx.scipy.special.cosm1****cupyx.scipy.special.cosm1()**Computes  $\cos(x) - 1$ .**See also:**`scipy.special.cosm1()`**cupyx.scipy.special.round****cupyx.scipy.special.round**(*a*, *decimals*=0, *out*=None)**cupyx.scipy.special.xlogy****cupyx.scipy.special.xlogy()**Compute  $x \cdot \log(y)$  so that the result is 0 if  $x = 0$ .**Parameters****x** (`cupy.ndarray`) – input data**Returns**values of  $x \cdot \log(y)$ **Return type**`cupy.ndarray`**See also:**`scipy.special.xlogy`

### cupyx.scipy.special.xlog1py

cupyx.scipy.special.xlog1py()

Compute  $x \cdot \log_{1p}(y)$  so that the result is 0 if  $x = 0$ .

**Parameters**

**x** (`cupy.ndarray`) – input data

**Returns**

values of  $x \cdot \log_{1p}(y)$

**Return type**

`cupy.ndarray`

See also:

`scipy.special.xlog1py`

### cupyx.scipy.special.logsumexp

cupyx.scipy.special.logsumexp(*a*, *axis=None*, *b=None*, *keepdims=False*, *return\_sign=False*)

Compute the log of the sum of exponentials of input elements.

**Parameters**

- **a** (`cupy.ndarray`) – Input array
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes over which the sum is taken. By default *axis* is None, and all elements are summed
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array
- **b** (`cupy.ndarray`, *optional*) – Scaling factor for  $\exp(a)$  must be of the same shape as *a* or broadcastable to *a*. These values may be negative in order to implement subtraction
- **return\_sign** (*bool, optional*) – If this is set to True, the result will be a pair containing sign information; if False, results that are negative will be returned as NaN. Default is False

**Returns**

- **res** (`cupy.ndarray`) – The result,  $\text{cp.log}(\text{cp.sum}(\text{cp.exp}(a)))$  calculated in a numerically more stable way. If *b* is given then  $\text{cp.log}(\text{cp.sum}(b * \text{cp.exp}(a)))$  is returned
- **sgn** (`cupy.ndarray`) – If *return\_sign* is True, this will be an array of floating-point numbers matching *res* and +1, 0, or -1 depending on the sign of the result. If False, only onw result is returned

See also:

`scipy.special.logsumexp`



**cupyx.scipy.special.sinc**

`cupyx.scipy.special.sinc(x, /, out=None, *, casting='same_kind', dtype=None)`

Elementwise sinc function.

**See also:**

`numpy.sinc()`

**5.4.14 Statistical functions (cupyx.scipy.stats)**


---

**Hint:** SciPy API Reference: Statistical functions (`scipy.stats`)

---

**Summary statistics**

<code>trim_mean(a, proportiontocut[, axis])</code>	Return mean of array after trimming distribution from both tails.
<code>entropy(pk[, qk, base, axis])</code>	Calculate the entropy of a distribution for given probability values.

**cupyx.scipy.stats.trim\_mean**

`cupyx.scipy.stats.trim_mean(a, proportiontocut, axis=0)`

Return mean of array after trimming distribution from both tails.

If *proportiontocut* = 0.1, slices off ‘leftmost’ and ‘rightmost’ 10% of scores. The input is sorted before slicing. Slices off less if proportion results in a non-integer slice index (i.e., conservatively slices off *proportiontocut* ).

**Parameters**

- **a** (`cupy.ndarray`) – Input array.
- **proportiontocut** (`float`) – Fraction to cut off of both tails of the distribution.
- **axis** (`int` or `None`, *optional*) – Axis along which the trimmed means are computed. Default is 0. If `None`, compute over the whole array *a*.

**Returns**

**trim\_mean** – Mean of trimmed array.

**Return type**

`ndarray`

**See also:**

**trimboth**

**tmean**

Compute the trimmed mean ignoring values outside given *limits*.

## Examples

```
>>> import cupy as cp
>>> from cupyx.scipy import stats
>>> x = cp.arange(20)
>>> stats.trim_mean(x, 0.1)
array(9.5)
>>> x2 = x.reshape(5, 4)
>>> x2
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> stats.trim_mean(x2, 0.25)
array([ 8.,  9., 10., 11.])
>>> stats.trim_mean(x2, 0.25, axis=1)
array([ 1.5,  5.5,  9.5, 13.5, 17.5])
```

## cupyx.scipy.stats.entropy

cupyx.scipy.stats.**entropy**(pk, qk=None, base=None, axis=0)

Calculate the entropy of a distribution for given probability values.

If only probabilities pk are given, the entropy is calculated as  $S = -\sum(pk * \log(pk))$ , axis=axis).

If qk is not None, then compute the Kullback-Leibler divergence  $S = \sum(pk * \log(pk / qk))$ , axis=axis).

This routine will normalize pk and qk if they don't sum to 1.

### Parameters

- **pk** ([ndarray](#)) – Defines the (discrete) distribution. pk[i] is the (possibly unnormalized) probability of event i.
- **qk** ([ndarray](#), *optional*) – Sequence against which the relative entropy is computed. Should be in the same format as pk.
- **base** ([float](#), *optional*) – The logarithmic base to use, defaults to e (natural logarithm).
- **axis** ([int](#), *optional*) – The axis along which the entropy is calculated. Default is 0.

### Returns

The calculated entropy.

### Return type

S ([cupy.ndarray](#))

## Other statistical functionality

<code>boxcox_llf(lmb, data)</code>	The boxcox log-likelihood function.
<code>zmap(scores, compare[, axis, ddof, nan_policy])</code>	Calculate the relative z-scores.
<code>zscore(a[, axis, ddof, nan_policy])</code>	Compute the z-score.

### cupyx.scipy.stats.boxcox\_llf

`cupyx.scipy.stats.boxcox_llf(lmb, data)`

The boxcox log-likelihood function.

#### Parameters

- **lmb** (*scalar*) – Parameter for Box-Cox transformation
- **data** (*array-like*) – Data to calculate Box-Cox log-likelihood for. If *data* is multi-dimensional, the log-likelihood is calculated along the first axis

#### Returns

**llf** – Box-Cox log-likelihood of *data* given *lmb*. A float for 1-D *data*, an array otherwise

#### Return type

float or *cupy.ndarray*

#### See also:

`scipy.stats.boxcox_llf`

### cupyx.scipy.stats.zmap

`cupyx.scipy.stats.zmap(scores, compare, axis=0, ddof=0, nan_policy='propagate')`

Calculate the relative z-scores.

Return an array of z-scores, i.e., scores that are standardized to zero mean and unit variance, where mean and variance are calculated from the comparison array.

#### Parameters

- **scores** (*array-like*) – The input for which z-scores are calculated
- **compare** (*array-like*) – The input from which the mean and standard deviation of the normalization are taken; assumed to have the same dimension as *scores*
- **axis** (*int* or *None*, *optional*) – Axis over which mean and variance of *compare* are calculated. Default is 0. If *None*, compute over the whole array *scores*
- **ddof** (*int*, *optional*) – Degrees of freedom correction in the calculation of the standard deviation. Default is 0
- **nan\_policy** (*{'propagate', 'raise', 'omit'}*, *optional*) – Defines how to handle the occurrence of nans in *compare*. ‘propagate’ returns nan, ‘raise’ raises an exception, ‘omit’ performs the calculations ignoring nan values. Default is ‘propagate’. Note that when the value is ‘omit’, nans in *scores* also propagate to the output, but they do not affect the z-scores computed for the non-nan values

#### Returns

**zscore** – Z-scores, in the same shape as *scores*

**Return type**  
array-like

### cupyx.scipy.stats.zscore

`cupyx.scipy.stats.zscore(a, axis=0, ddof=0, nan_policy='propagate')`

Compute the z-score.

Compute the z-score of each value in the sample, relative to the sample mean and standard deviation.

#### Parameters

- **a** (*array-like*) – An array like object containing the sample data
- **axis** (*int* or *None*, *optional*) – Axis along which to operate. Default is 0. If None, compute over the whole array *a*
- **ddof** (*int*, *optional*) – Degrees of freedom correction in the calculation of the standard deviation. Default is 0
- **nan\_policy** (*{'propagate', 'raise', 'omit'}*, *optional*) – Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'. Note that when the value is 'omit', nans in the input also propagate to the output, but they do not affect the z-scores computed for the non-nan values

#### Returns

**zscore** – The z-scores, standardized by mean and standard deviation of input array *a*

**Return type**  
array-like

## 5.5 CuPy-specific functions

CuPy-specific functions are placed under `cupyx` namespace.

<code>cupyx.rsqrt</code>	Returns the reciprocal square root.
<code>cupyx.scatter_add(a, slices, value)</code>	Adds given values to specified elements of an array.
<code>cupyx.scatter_max(a, slices, value)</code>	Stores a maximum value of elements specified by indices to an array.
<code>cupyx.scatter_min(a, slices, value)</code>	Stores a minimum value of elements specified by indices to an array.
<code>cupyx.empty_pinned(shape[, dtype, order])</code>	Returns a new, uninitialized NumPy array with the given shape and dtype.
<code>cupyx.empty_like_pinned(a[, dtype, order, ...])</code>	Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.
<code>cupyx.zeros_pinned(shape[, dtype, order])</code>	Returns a new, zero-initialized NumPy array with the given shape and dtype.
<code>cupyx.zeros_like_pinned(a[, dtype, order, ...])</code>	Returns a new, zero-initialized NumPy array with the same shape and dtype as those of the given array.

### 5.5.1 cupyx.rsqrt

`cupyx.rsqrt()`

Returns the reciprocal square root.

### 5.5.2 cupyx.scatter\_add

`cupyx.scatter_add(a, slices, value)`

Adds given values to specified elements of an array.

It adds value to the specified elements of a. If all of the indices target different locations, the operation of `scatter_add()` is equivalent to `a[slices] = a[slices] + value`. If there are multiple elements targeting the same location, `scatter_add()` uses all of these values for addition. On the other hand, `a[slices] = a[slices] + value` only adds the contribution from one of the indices targeting the same location.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_add()` behaves identically to `numpy.add.at()`.

#### Example

```
>>> import cupy
>>> import cupyx
>>> a = cupy.zeros((6,), dtype=cupy.float32)
>>> i = cupy.array([1, 0, 1])
>>> v = cupy.array([1., 1., 1.])
>>> cupyx.scatter_add(a, i, v);
>>> a
array([1., 2., 0., 0., 0., 0.], dtype=float32)
```

#### Parameters

- **a** (*ndarray*) – An array that gets added.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – Values to increment a at referenced locations.

---

**Note:** It only supports types that are supported by CUDA's `atomicAdd` when an integer array is included in slices. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

---



---

**Note:** `scatter_add()` does not raise an error when indices exceed size of axes. Instead, it wraps indices.

---

See also:

`numpy.ufunc.at()`.

### 5.5.3 cupyx.scatter\_max

`cupyx.scatter_max(a, slices, value)`

Stores a maximum value of elements specified by indices to an array.

It stores the maximum value of elements in `value` array indexed by `slices` to `a`. If all of the indices target different locations, the operation of `scatter_max()` is equivalent to `a[slices] = cupy.maximum(a[slices], value)`. If there are multiple elements targeting the same location, `scatter_max()` stores the maximum of all of these values to the given index of `a`, the initial element of `a` is also taken in account.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_max()` behaves identically to `numpy.maximum.at()`.

#### Example

```
>>> import numpy
>>> import cupy
>>> a = cupy.zeros((6,), dtype=numpy.float32)
>>> i = cupy.array([1, 0, 1, 2])
>>> v = cupy.array([1., 2., 3., -1.])
>>> cupyx.scatter_max(a, i, v);
>>> a
array([2., 3., 0., 0., 0., 0.], dtype=float32)
```

#### Parameters

- **a** (*ndarray*) – An array to store the results.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – An array used for reference.

### 5.5.4 cupyx.scatter\_min

`cupyx.scatter_min(a, slices, value)`

Stores a minimum value of elements specified by indices to an array.

It stores the minimum value of elements in `value` array indexed by `slices` to `a`. If all of the indices target different locations, the operation of `scatter_min()` is equivalent to `a[slices] = cupy.minimum(a[slices], value)`. If there are multiple elements targeting the same location, `scatter_min()` stores the minimum of all of these values to the given index of `a`, the initial element of `a` is also taken in account.

Note that just like an array indexing, negative indices are interpreted as counting from the end of an array.

Also note that `scatter_min()` behaves identically to `numpy.minimum.at()`.

### Example

```

>>> import numpy
>>> import cupy
>>> a = cupy.zeros((6,), dtype=numpy.float32)
>>> i = cupy.array([1, 0, 1, 2])
>>> v = cupy.array([1., 2., 3., -1.])
>>> cupyx.scatter_min(a, i, v);
>>> a
array([ 0.,  0., -1.,  0.,  0.,  0.], dtype=float32)

```

#### Parameters

- **a** (`ndarray`) – An array to store the results.
- **slices** – It is integer, slices, ellipsis, `numpy.newaxis`, integer array-like, boolean array-like or tuple of them. It works for slices used for `cupy.ndarray.__getitem__()` and `cupy.ndarray.__setitem__()`.
- **v** (*array-like*) – An array used for reference.

### 5.5.5 cupyx.empty\_pinned

`cupyx.empty_pinned(shape, dtype=<class 'float'>, order='C')`

Returns a new, uninitialized NumPy array with the given shape and dtype.

This is a convenience function which is just `numpy.empty()`, except that the underlying memory is pinned/pagelocked.

#### Parameters

- **shape** (*int* or *tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (`{'C', 'F'}`) – Row-major (C-style) or column-major (Fortran-style) order.

#### Returns

A new array with elements not initialized.

#### Return type

`numpy.ndarray`

#### See also:

`numpy.empty()`

### 5.5.6 cupyx.empty\_like\_pinned

`cupyx.empty_like_pinned(a, dtype=None, order='K', subok=None, shape=None)`

Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.

This is a convenience function which is just `numpy.empty_like()`, except that the underlying memory is pinned/pagelocked.

This function currently does not support `subok` option.

#### Parameters

- **a** (*numpy.ndarray or cupy.ndarray*) – Base array.
- **dtype** – Data type specifier. The data type of **a** is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if **a** is Fortran contiguous, 'C' otherwise. 'K' means match the layout of **a** as closely as possible.
- **subok** – Not supported yet, must be None.
- **shape** (*int or tuple of ints*) – Overrides the shape of the result. If **order**='K' and the number of dimensions is unchanged, will try to keep order, otherwise, **order**='C' is implied.

**Returns**

A new array with same shape and dtype of **a** with elements not initialized.

**Return type**

*numpy.ndarray*

**See also:**

*numpy.empty\_like()*

## 5.5.7 cupyx.zeros\_pinned

`cupyx.zeros_pinned(shape, dtype=<class 'float'>, order='C')`

Returns a new, zero-initialized NumPy array with the given shape and dtype.

This is a convenience function which is just `numpy.zeros()`, except that the underlying memory is pinned/pagelocked.

**Parameters**

- **shape** (*int or tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.
- **order** (*{'C', 'F'}*) – Row-major (C-style) or column-major (Fortran-style) order.

**Returns**

An array filled with zeros.

**Return type**

*numpy.ndarray*

**See also:**

*numpy.zeros()*

## 5.5.8 cupyx.zeros\_like\_pinned

`cupyx.zeros_like_pinned(a, dtype=None, order='K', subok=None, shape=None)`

Returns a new, zero-initialized NumPy array with the same shape and dtype as those of the given array.

This is a convenience function which is just `numpy.zeros_like()`, except that the underlying memory is pinned/pagelocked.

This function currently does not support **subok** option.

**Parameters**



- **a** (*numpy.ndarray* or *cupy.ndarray*) – Base array.
- **dtype** – Data type specifier. The dtype of a is used by default.
- **order** (*{'C', 'F', 'A', or 'K'}*) – Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if a is Fortran contiguous, 'C' otherwise. 'K' means match the layout of a as closely as possible.
- **subok** – Not supported yet, must be None.
- **shape** (*int* or *tuple of ints*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

**Returns**

An array filled with zeros.

**Return type**

*numpy.ndarray*

**See also:**

*numpy.zeros\_like()*

### 5.5.9 non-SciPy compat Signal API

The functions under *cupyx.signal* are non-SciPy compat signal API ported from cuSignal through the courtesy of Nvidia cuSignal team.

<i>cupyx.signal.channelize_poly</i> (x, h, n_chans)	Polyphase channelize signal into n channels
<i>cupyx.signal.convolve1d3o</i> (in1, in2[, mode, ...])	Convolve a 1-dimensional array with a 3rd order filter.
<i>cupyx.signal.pulse_compression</i> (x, template)	Pulse Compression is used to increase the range resolution and SNR by performing matched filtering of the transmitted pulse (template) with the received signal (x)
<i>cupyx.signal.pulse_doppler</i> (x[, window, nfft])	Pulse doppler processing yields a range/doppler data matrix that represents moving target data that's separated from clutter.
<i>cupyx.signal.cfar_alpha</i> (pfa, N)	Computes the value of alpha corresponding to a given probability of false alarm and number of reference cells N.
<i>cupyx.signal.ca_cfar</i> (array, guard_cells, ...)	Computes the cell-averaged constant false alarm rate (CA CFAR) detector threshold and returns for a given array.
<i>cupyx.signal.freq_shift</i> (x, freq, fs)	Frequency shift signal by freq at fs sample rate

#### **cupyx.signal.channelize\_poly**

*cupyx.signal.channelize\_poly*(x, h, n\_chans)

Polyphase channelize signal into n channels

**Parameters**

- **x** (*array\_like*) – The input data to be channelized
- **h** (*array\_like*) – The 1-D input filter; will be split into n channels of int number of taps
- **n\_chans** (*int*) – Number of channels for channelizer

**Returns**

yy

**Return type**

channelized output matrix

**Notes**

Currently only supports simple channelizer where channel spacing is equivalent to the number of channels used (zero overlap). Number of filter taps (len of filter / n\_chans) must be <=32.

**cupyx.signal.convolve1d3o**

`cupyx.signal.convolve1d3o(in1, in2, mode='valid', method='direct')`

Convolve a 1-dimensional array with a 3rd order filter. This results in a third order convolution.

Convolve *in1* and *in2*, with the output size determined by the *mode* argument.

**Parameters**

- **in1** (*array\_like*) – First input. Should have one dimension.
- **in2** (*array\_like*) – Second input. Should have three dimensions.
- **mode** (*str* {'full', 'valid', 'same'}, *optional*) – A string indicating the size of the output:

**full**

The output is the full discrete linear convolution of the inputs. (Default)

**valid**

The output consists only of those elements that do not rely on the zero-padding. In 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

**same**

The output is the same size as *in1*, centered with respect to the 'full' output.

- **method** (*str* {'auto', 'direct', 'fft'}, *optional*) – A string indicating which method to use to calculate the convolution.

**direct**

The convolution is determined directly from sums, the definition of convolution.

**fft**

The Fourier Transform is used to perform the convolution by calling *fftconvolve*.

**auto**

Automatically chooses direct or Fourier method based on an estimate of which is faster (default).

**Returns**

**out** – A 1-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

**Return type**

*ndarray*

**See also:**

`convolve`, `convolve1d2o`, [`convolve1d3o`](#)

### cupyx.signal.pulse\_compression

`cupyx.signal.pulse_compression(x, template, normalize=False, window=None, nfft=None)`

Pulse Compression is used to increase the range resolution and SNR by performing matched filtering of the transmitted pulse (template) with the received signal (x)

#### Parameters

- **x** (`ndarray`) – Received signal, assume 2D array with [num\_pulses, sample\_per\_pulse]
- **template** (`ndarray`) – Transmitted signal, assume 1D array
- **normalize** (`bool`) – Normalize transmitted signal
- **window** (`array_like`, `callable`, `string`, `float`, or `tuple`, `optional`) – Specifies the window applied to the signal in the Fourier domain.
- **nfft** (`int`, *size of FFT for pulse compression. Default is number of*) – samples per pulse

#### Returns

**compressedIQ** – Pulse compressed output

#### Return type

`ndarray`

### cupyx.signal.pulse\_doppler

`cupyx.signal.pulse_doppler(x, window=None, nfft=None)`

Pulse doppler processing yields a range/doppler data matrix that represents moving target data that's separated from clutter. An estimation of the doppler shift can also be obtained from pulse doppler processing. FFT taken across slow-time (pulse) dimension.

#### Parameters

- **x** (`ndarray`) – Received signal, assume 2D array with [num\_pulses, sample\_per\_pulse]
- **window** (`array_like`, `callable`, `string`, `float`, or `tuple`, `optional`) – Specifies the window applied to the signal in the Fourier domain.
- **nfft** (`int`, *size of FFT for pulse compression. Default is number of*) – samples per pulse

#### Returns

**pd\_dataMatrix** – Pulse-doppler output (range/doppler matrix)

#### Return type

`ndarray`

### cupyx.signal.cfar\_alpha

`cupyx.signal.cfar_alpha(pfa, N)`

Computes the value of alpha corresponding to a given probability of false alarm and number of reference cells N.

#### Parameters

- **pfa** (`float`) – Probability of false alarm.
- **N** (`int`) – Number of reference cells.

**Returns**

**alpha** – Alpha value.

**Return type**

float

**cupyx.signal.ca\_cfar**

`cupyx.signal.ca_cfar(array, guard_cells, reference_cells, pfa=0.001)`

Computes the cell-averaged constant false alarm rate (CA CFAR) detector threshold and returns for a given array. :param array: Array containing data to be processed. :type array: ndarray :param guard\_cells\_x: One-sided guard cell count in the first dimension. :type guard\_cells\_x: int :param guard\_cells\_y: One-sided guard cell count in the second dimension. :type guard\_cells\_y: int :param reference\_cells\_x: one-sided reference cell count in the first dimension. :type reference\_cells\_x: int :param reference\_cells\_y: one-sided reference cell count in the second dimension. :type reference\_cells\_y: int :param pfa: Probability of false alarm. :type pfa: float

**Returns**

- **threshold** (ndarray) – CFAR threshold
- **return** (ndarray) – CFAR detections

**cupyx.signal.freq\_shift**

`cupyx.signal.freq_shift(x, freq, fs)`

Frequency shift signal by freq at fs sample rate

**Parameters**

- **x** (array\_like, complex valued) – The data to be shifted.
- **freq** (float) – Shift by this many (Hz)
- **fs** (float) – Sampling rate of the signal
- **domain** (string) – freq or time

## 5.5.10 Profiling utilities

<code>cupyx.profiler.benchmark(func[, args, ...])</code>	Timing utility for measuring time spent by both CPU and GPU.
<code>cupyx.profiler.time_range([message, ...])</code>	Mark function calls with ranges using NVTX/roctx.
<code>cupyx.profiler.profile()</code>	Enable CUDA profiling during with statement.

**cupyx.profiler.benchmark**

`cupyx.profiler.benchmark(func, args=(), kwargs={}, n_repeat=10000, *, name=None, n_warmup=10, max_duration=inf, devices=None)`

Timing utility for measuring time spent by both CPU and GPU.

This function is a very convenient helper for setting up a timing test. The GPU time is properly recorded by synchronizing internal streams. As a result, to time a multi-GPU function all participating devices must be passed as the `devices` argument so that this helper knows which devices to record. A simple example is given as follows:

```
import cupy as cp
from cupyx.profiler import benchmark

def f(a, b):
    return 3 * cp.sin(-a) * b

a = 0.5 - cp.random.random((100,))
b = cp.random.random((100,))
print(benchmark(f, (a, b), n_repeat=1000))
```

### Parameters

- **func** (*callable*) – a callable object to be timed.
- **args** (*tuple*) – positional arguments to be passed to the callable.
- **kwargs** (*dict*) – keyword arguments to be passed to the callable.
- **n\_repeat** (*int*) – number of times the callable is called. Increasing this value would improve the collected statistics at the cost of longer test time.
- **name** (*str*) – the function name to be reported. If not given, the callable’s `__name__` attribute is used.
- **n\_warmup** (*int*) – number of times the callable is called. The warm-up runs are not timed.
- **max\_duration** (*float*) – the maximum time (in seconds) that the entire test can use. If the taken time is longer than this limit, the test is stopped and the statistics collected up to the breakpoint is reported.
- **devices** (*tuple*) – a tuple of device IDs (int) that will be timed during the timing test. If not given, the current device is used.

### Returns

an object collecting all test results.

### Return type

`_PerfCaseResult`

## cupyx.profiler.time\_range

**class** `cupyx.profiler.time_range`(*message=None, color\_id=None, argb\_color=None, sync=False*)

Mark function calls with ranges using NVTX/roctx. This object can be used either as a decorator or a context manager.

When used as a decorator, the decorated function calls are marked as ranges:

```
>>> from cupyx.profiler import time_range
>>> @time_range()
... def function_to_profile():
...     pass
```

When used as a context manager, it describes the enclosed block as a nested range:

```
>>> from cupyx.profiler import time_range
>>> with time_range('some range in green', color_id=0):
```

(continues on next page)

(continued from previous page)

```
...     # do something you want to measure
...     pass
```

The marked ranges are visible in the profiler (such as nvvp, nsys-ui, etc) timeline.

#### Parameters

- **message** (*str*) – Name of a range. When used as a decorator, the default is `func.__name__`.
- **color\_id** – range color ID
- **argb\_color** – range color in ARGB (e.g. 0xFF00FF00 for green)
- **sync** (*bool*) – If True, waits for completion of all outstanding processing on GPU before calling `cupy.cuda.nvtx.RangePush()` or `cupy.cuda.nvtx.RangePop()`

See also:

`cupy.cuda.nvtx.RangePush()`, `cupy.cuda.nvtx.RangePop()`

#### Methods

**\_\_call\_\_** (*func*)

Call self as a function.

**\_\_enter\_\_** ()

**\_\_exit\_\_** (*exc\_type, exc\_value, traceback*)

**\_\_eq\_\_** (*value, /*)

Return self==value.

**\_\_ne\_\_** (*value, /*)

Return self!=value.

**\_\_lt\_\_** (*value, /*)

Return self<value.

**\_\_le\_\_** (*value, /*)

Return self<=value.

**\_\_gt\_\_** (*value, /*)

Return self>value.

**\_\_ge\_\_** (*value, /*)

Return self>=value.

### cupyx.profiler.profile

**cupyx.profiler.profile()**

Enable CUDA profiling during with statement.

This function enables profiling on entering a with statement, and disables profiling on leaving the statement.

```
>>> with cupyx.profiler.profile():
...     # do something you want to measure
...     pass
```

---

**Note:** When starting `nvprof` from the command line, manually setting `--profile-from-start off` may be required for the desired behavior. Likewise, when using `nsys profile` setting `-c cudaProfilerApi` may be required.

---

**See also:**

`cupy.cuda.runtime.profilerStart()`, `cupy.cuda.runtime.profilerStop()`

### 5.5.11 DLPack utilities

Below are helper functions for creating a `cupy.ndarray` from either a DLPack tensor or any object supporting the DLPack data exchange protocol. For further detail see [DLPack](#).

---

<code>cupy.from_dlpack(array)</code>	Zero-copy conversion between array objects compliant with the DLPack data exchange protocol.
--------------------------------------	--

---

#### `cupy.from_dlpack`

`cupy.from_dlpack(array)`

Zero-copy conversion between array objects compliant with the DLPack data exchange protocol.

**Parameters**

**array** (*object*) – an array object that implements two methods: `__dlpack__()` and `__dlpack_device__()`.

**Returns**

a CuPy array that can be safely accessed on CuPy’s current stream.

**Return type**

`cupy.ndarray`

---

**Note:** This function is different from CuPy’s legacy `fromDlpack()` function. This function takes any object implementing the DLPack data exchange protocol, as well as a raw `PyCapsule` object that contains the DLPack tensor as input (for backward compatibility), whereas `fromDlpack()` only accepts `PyCapsule` objects. If the input object is not compliant with the protocol, users are responsible to ensure data safety.

---

**See also:**

`numpy.from_dlpack()`, [Python Specification for DLPack](#), [Data interchange mechanisms](#)

### 5.5.12 Automatic Kernel Parameters Optimizations (`cupyx.optimizing`)

---

<code>cupyx.optimizing.optimize(*[, key, path, ...])</code>	Context manager that optimizes kernel launch parameters.
---	--

---

## cupyx.optimizing.optimize

`cupyx.optimizing.optimize(*, key=None, path=None, readonly=False, **config_dict)`

Context manager that optimizes kernel launch parameters.

In this context, CuPy's routines find the best kernel launch parameter values (e.g., the number of threads and blocks). The found values are cached and reused with keys as the shapes, strides and dtypes of the given inputs arrays.

### Parameters

- **key** (*string or None*) – The cache key of optimizations.
- **path** (*string or None*) – The path to save optimization cache records. When path is specified and exists, records will be loaded from the path. When readonly option is set to False, optimization cache records will be saved to the path after the optimization.
- **readonly** (*bool*) – See the description of path option.
- **max\_trials** (*int*) – The number of trials that defaults to 100.
- **timeout** (*float*) – Stops study after the given number of seconds. Default is 1.
- **max\_total\_time\_per\_trial** (*float*) – Repeats measuring the execution time of the routine for the given number of seconds. Default is 0.1.

### Examples

```
>>> import cupy
>>> from cupyx import optimizing
>>>
>>> x = cupy.arange(100)
>>> with optimizing.optimize():
...     cupy.sum(x)
...
array(4950)
```

---

**Note:** Optuna (<https://optuna.org>) installation is required. Currently it works for reduction operations only.

---

## 5.6 Low-level CUDA support

### 5.6.1 Device management

---

`cupy.cuda.Device([device])`

Object that represents a CUDA device.

---



## cupy.cuda.Device

**class** `cupy.cuda.Device(device=None)`

Object that represents a CUDA device.

This class provides some basic manipulations on CUDA devices.

It supports the context protocol. For example, the following code is an example of temporarily switching the current device:

```
with Device(0):
    do_something_on_device_0()
```

After the *with* statement gets done, the current device is reset to the original one.

### Parameters

**device** (*int* or `cupy.cuda.Device`) – Index of the device to manipulate. Be careful that the device ID (a.k.a. GPU ID) is zero origin. If it is a Device object, then its ID is used. The current device is selected by default.

### Variables

**id** (*int*) – ID of this device.

## Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \*args)

**from\_pci\_bus\_id**(*type cls*, *pci\_bus\_id*)

Returns a new device instance based on a PCI Bus ID

### Parameters

**pci\_bus\_id** (*str*) – The string for a device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

### Returns

An instance of the Device class that has the PCI Bus ID as given by the argument `pci_bus_id`.

### Return type

device (*Device*)

**synchronize**(*self*)

Synchronizes the current thread to the device.

**use**(*self*)

Makes this device current.

If you want to switch a device temporarily, use the *with* statement.

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(value, /)

Return self<=value.

**\_\_gt\_\_**(value, /)

Return self>value.

**\_\_ge\_\_**(value, /)

Return self>=value.

## Attributes

### **attributes**

A dictionary of device attributes.

#### **Returns**

Dictionary of attribute values with the names as keys. The string *cudaDevAttr* has been trimmed from the names. For example, the attribute corresponding to the enumerated value *cudaDevAttrMaxThreadsPerBlock* will have key *MaxThreadsPerBlock*.

#### **Return type**

attributes (dict)

### **compute\_capability**

Compute capability of this device.

The capability is represented by a string containing the major index and the minor index. For example, compute capability 3.5 is represented by the string '35'.

### **cublas\_handle**

The cuBLAS handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusolver\_handle**

The cuSOLVER handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusolver\_sp\_handle**

The cuSOLVER Sphandle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **cusparse\_handle**

The cuSPARSE handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

### **id**

'int'

#### **Type**

id

### **mem\_info**

The device memory info.

#### **Returns**

The amount of free memory, in bytes. total: The total amount of memory, in bytes.

**Return type**

free

**pci\_bus\_id**

A string of the PCI Bus ID

**Returns**

Returned identifier string for the device in the following format [domain]:[bus]:[device].[function] where domain, bus, device, and function are all hexadecimal values.

**Return type**

pci\_bus\_id (str)

## 5.6.2 Memory management

<code>cupy.get_default_memory_pool()</code>	Returns CuPy default memory pool for GPU memory.
<code>cupy.get_default_pinned_memory_pool()</code>	Returns CuPy default memory pool for pinned memory.
<code>cupy.cuda.Memory(size_t size)</code>	Memory allocation on a CUDA device.
<code>cupy.cuda.MemoryAsync(size_t size, stream)</code>	Asynchronous memory allocation on a CUDA device.
<code>cupy.cuda.ManagedMemory(size_t size)</code>	Managed memory (Unified memory) allocation on a CUDA device.
<code>cupy.cuda.UnownedMemory(intptr_t ptr, ...)</code>	CUDA memory that is not owned by CuPy.
<code>cupy.cuda.PinnedMemory(size[, flags])</code>	Pinned memory allocation on host.
<code>cupy.cuda.MemoryPointer(BaseMemory mem, ...)</code>	Pointer to a point on a device memory.
<code>cupy.cuda.PinnedMemoryPointer(mem, ...)</code>	Pointer of a pinned memory.
<code>cupy.cuda.malloc_managed(size_t size)</code>	Allocate managed memory (unified memory).
<code>cupy.cuda.malloc_async(size_t size)</code>	(Experimental) Allocate memory from Stream Ordered Memory Allocator.
<code>cupy.cuda.alloc(size)</code>	Calls the current allocator.
<code>cupy.cuda.alloc_pinned_memory(size_t size)</code>	Calls the current allocator.
<code>cupy.cuda.get_allocator()</code>	Returns the current allocator for GPU memory.
<code>cupy.cuda.set_allocator([allocator])</code>	Sets the current allocator for GPU memory.
<code>cupy.cuda.using_allocator([allocator])</code>	Sets a thread-local allocator for GPU memory inside
<code>cupy.cuda.set_pinned_memory_allocator([...])</code>	Sets the current allocator for the pinned memory.
<code>cupy.cuda.MemoryPool([allocator])</code>	Memory pool for all GPU devices on the host.
<code>cupy.cuda.MemoryAsyncPool([pool_handles])</code>	(Experimental) CUDA memory pool for all GPU devices on the host.
<code>cupy.cuda.PinnedMemoryPool([allocator])</code>	Memory pool for pinned memory on the host.
<code>cupy.cuda.PythonFunctionAllocator(...)</code>	Allocator with python functions to perform memory allocation.
<code>cupy.cuda.CFunctionAllocator(intptr_t param, ...)</code>	Allocator with C function pointers to allocation routines.

### cupy.get\_default\_memory\_pool

`cupy.get_default_memory_pool()`

Returns CuPy default memory pool for GPU memory.

**Returns**

The memory pool object.

**Return type**

*cupy.cuda.MemoryPool*

---

**Note:** If you want to disable memory pool, please use the following code.

```
>>> cupy.cuda.set_allocator(None)
```

---

### cupy.get\_default\_pinned\_memory\_pool

`cupy.get_default_pinned_memory_pool()`

Returns CuPy default memory pool for pinned memory.

**Returns**

The memory pool object.

**Return type**

*cupy.cuda.PinnedMemoryPool*

---

**Note:** If you want to disable memory pool, please use the following code.

```
>>> cupy.cuda.set_pinned_memory_allocator(None)
```

---

### cupy.cuda.Memory

**class** `cupy.cuda.Memory`(*size\_t size*)

Memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

**Parameters**

**size** (*int*) – Size of the memory allocation in bytes.

**Methods**

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

```

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

```

device
device_id
    'int'
    Type
        device_id

ptr
    'intptr_t'
    Type
        ptr

size
    'size_t'
    Type
        size

```

## cupy.cuda.MemoryAsync

**class** cupy.cuda.**MemoryAsync**(*size\_t size, stream*)

Asynchronous memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

### Parameters

- **size** (*int*) – Size of the memory allocation in bytes.
- **stream** (*Stream*) – The stream on which the memory is allocated and freed.

### Methods

```

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

```

```
__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## Attributes

**device**

**device\_id**

‘int’

**Type**

device\_id

**ptr**

‘intptr\_t’

**Type**

ptr

**size**

‘size\_t’

**Type**

size

**stream\_ref**

## cupy.cuda.ManagedMemory

**class** cupy.cuda.**ManagedMemory**(size\_t size)

Managed memory (Unified memory) allocation on a CUDA device.

This class provides an RAII interface of the CUDA managed memory allocation.

### Parameters

**size** (*int*) – Size of the memory allocation in bytes.

## Methods

**advise**(self, int advise, Device dev)

(experimental) Advise about the usage of this memory.

### Parameters

- **advics** (*int*) – Advise to be applied for this memory.
- **dev** (cupy.cuda.Device) – Device to apply the advice for.

**prefetch**(*self*, *stream*)

(experimental) Prefetch memory.

**Parameters**

**stream** (`cupy.cuda.Stream`) – CUDA stream.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**device**

**device\_id**

‘int’

**Type**

device\_id

**ptr**

‘intptr\_t’

**Type**

ptr

**size**

‘size\_t’

**Type**

size

## cupy.cuda.UnownedMemory

**class** `cupy.cuda.UnownedMemory`(*intptr\_t ptr*, *size\_t size*, *owner*, *int device\_id=-1*)

CUDA memory that is not owned by CuPy.

**Parameters**

- **ptr** (*int*) – Pointer to the buffer.
- **size** (*int*) – Size of the buffer.
- **owner** (*object*) – Reference to the owner object to keep the memory alive.

- **device\_id** (*int*) – CUDA device ID of the buffer. If omitted, the device associated to the pointer is retrieved.

## Methods

**\_\_eq\_\_**(*value*, /)  
Return self==value.

**\_\_ne\_\_**(*value*, /)  
Return self!=value.

**\_\_lt\_\_**(*value*, /)  
Return self<value.

**\_\_le\_\_**(*value*, /)  
Return self<=value.

**\_\_gt\_\_**(*value*, /)  
Return self>value.

**\_\_ge\_\_**(*value*, /)  
Return self>=value.

## Attributes

**device**

**device\_id**

‘int’

**Type**

device\_id

**ptr**

‘intptr\_t’

**Type**

ptr

**size**

‘size\_t’

**Type**

size

## cupy.cuda.PinnedMemory

**class** cupy.cuda.**PinnedMemory**(*size*, *flags*=0)

Pinned memory allocation on host.

This class provides a RAII interface of the pinned memory allocation.

### Parameters

**size** (*int*) – Size of the memory allocation in bytes.



## Methods

```
__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## cupy.cuda.MemoryPointer

**class** `cupy.cuda.MemoryPointer`(*BaseMemory mem, ptrdiff\_t offset*)

Pointer to a point on a device memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

### Parameters

- **mem** (*BaseMemory*) – The device memory buffer.
- **offset** (*int*) – An offset from the head of the buffer to the place this pointer refers.

### Variables

- `~MemoryPointer.device` (*Device*) – Device whose memory the pointer refers to.
- `~MemoryPointer.mem` (*BaseMemory*) – The device memory buffer.
- `~MemoryPointer.ptr` (*int*) – Pointer to the place within the buffer.

## Methods

**copy\_from**(*self, mem, size\_t size*)

Copies a memory sequence from a (possibly different) device or host.

This function is a useful interface that selects appropriate one from `copy_from_device()` and `copy_from_host()`.

### Parameters

- **mem** (*int* or *ctypes.c\_void\_p* or `cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_async` instead if you are using streams in your code, or have PTDS enabled.

`copy_from_async(self, mem, size_t size, stream=None)`

Copies a memory sequence from an arbitrary place asynchronously.

This function is a useful interface that selects appropriate one from `copy_from_device_async()` and `copy_from_host_async()`.

**Parameters**

- **mem** (`int` or `ctypes.c_void_p` or `cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (`int`) – Size of the sequence in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

`copy_from_device(self, MemoryPointer src, size_t size)`

Copies a memory sequence from a (possibly different) device.

**Parameters**

- **src** (`cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (`int`) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_device_async` instead if you are using streams in your code, or have PTDS enabled.

`copy_from_device_async(self, MemoryPointer src, size_t size, stream=None)`

Copies a memory from a (possibly different) device asynchronously.

**Parameters**

- **src** (`cupy.cuda.MemoryPointer`) – Source memory pointer.
- **size** (`int`) – Size of the sequence in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

`copy_from_host(self, mem, size_t size)`

Copies a memory sequence from the host memory.

**Parameters**

- **mem** (`int` or `ctypes.c_void_p`) – Source memory pointer.
- **size** (`int`) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use `copy_from_host_async` instead if you are using streams in your code, or have PTDS enabled.

**copy\_from\_host\_async**(*self, mem, size\_t size, stream=None*)

Copies a memory sequence from the host memory asynchronously.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Source memory pointer. It must point to pinned memory.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (*cupy.cuda.Stream*) – CUDA stream. The default uses CUDA stream of the current context.

**copy\_to\_host**(*self, mem, size\_t size*)

Copies a memory sequence to the host memory.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Target memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use *copy\_to\_host\_async* instead if you are using streams in your code, or have PTDS enabled.

**copy\_to\_host\_async**(*self, mem, size\_t size, stream=None*)

Copies a memory sequence to the host memory asynchronously.

**Parameters**

- **mem** (*int* or *ctypes.c\_void\_p*) – Target memory pointer. It must point to pinned memory.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (*cupy.cuda.Stream*) – CUDA stream. The default uses CUDA stream of the current context.

**memset**(*self, int value, size\_t size*)

Fills a memory sequence by constant byte value.

**Parameters**

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

**Warning:** This function always uses the legacy default stream and does not honor the current stream. Use *memset\_async* instead if you are using streams in your code, or have PTDS enabled.

**memset\_async**(*self, int value, size\_t size, stream=None*)

Fills a memory sequence by constant byte value asynchronously.

**Parameters**

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

- **stream** (`cupy.cuda.Stream`) – CUDA stream. The default uses CUDA stream of the current context.

`__eq__(value, /)`

Return self==value.

`__ne__(value, /)`

Return self!=value.

`__lt__(value, /)`

Return self<value.

`__le__(value, /)`

Return self<=value.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

### Attributes

**device**

**device\_id**

**mem**

**ptr**

## `cupy.cuda.PinnedMemoryPointer`

**class** `cupy.cuda.PinnedMemoryPointer`(*mem, ptrdiff\_t offset*)

Pointer of a pinned memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

### Parameters

- **mem** (`PinnedMemory`) – The device memory buffer.
- **offset** (`int`) – An offset from the head of the buffer to the place this pointer refers.

### Variables

- `~PinnedMemoryPointer.mem` (`PinnedMemory`) – The device memory buffer.
- `~PinnedMemoryPointer.ptr` (`int`) – Pointer to the place within the buffer.

## Methods

**size**(*self*) → *size\_t*

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**mem**

**ptr**

## cupy.cuda.malloc\_managed

**cupy.cuda.malloc\_managed**(*size\_t* *size*) → *MemoryPointer*

Allocate managed memory (unified memory).

This method can be used as a CuPy memory allocator. The simplest way to use a managed memory as the default allocator is the following code:

```
set_allocator(malloc_managed)
```

The advantage using managed memory in CuPy is that device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. CUDA >= 8.0 with GPUs later than or equal to Pascal is preferable.

Read more at: [https://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDART\\_\\_MEMORY.html#axzz4qygc1Ry1](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#axzz4qygc1Ry1) # NOQA

### Parameters

**size** (*int*) – Size of the memory allocation in bytes.

### Returns

Pointer to the allocated buffer.

### Return type

*MemoryPointer*

### cupy.cuda.malloc\_async

`cupy.cuda.malloc_async(size_t size) → MemoryPointer`

(Experimental) Allocate memory from Stream Ordered Memory Allocator.

This method can be used as a CuPy memory allocator. The simplest way to use CUDA's Stream Ordered Memory Allocator as the default allocator is the following code:

```
set_allocator(malloc_async)
```

Using this feature requires CUDA  $\geq 11.2$  with a supported GPU and platform. If it is not supported, an error will be raised.

The current CuPy stream is used to allocate/free the memory.

**Parameters**

**size** (*int*) – Size of the memory allocation in bytes.

**Returns**

Pointer to the allocated buffer.

**Return type**

*MemoryPointer*

**Warning:** This feature is currently experimental and subject to change.

See also:

[Stream Ordered Memory Allocator](#)

### cupy.cuda.alloc

`cupy.cuda.alloc(size) → MemoryPointer`

Calls the current allocator.

Use `set_allocator()` to change the current allocator.

**Parameters**

**size** (*int*) – Size of the memory allocation.

**Returns**

Pointer to the allocated buffer.

**Return type**

*MemoryPointer*

### cupy.cuda.alloc\_pinned\_memory

`cupy.cuda.alloc_pinned_memory(size_t size) → PinnedMemoryPointer`

Calls the current allocator.

Use `set_pinned_memory_allocator()` to change the current allocator.

**Parameters**

**size** (*int*) – Size of the memory allocation.

**Returns**

Pointer to the allocated buffer.

**Return type**

[PinnedMemoryPointer](#)

**cupy.cuda.get\_allocator**

`cupy.cuda.get_allocator()`

Returns the current allocator for GPU memory.

**Returns**

CuPy memory allocator.

**Return type**

function

**cupy.cuda.set\_allocator**

`cupy.cuda.set_allocator(allocator=None)`

Sets the current allocator for GPU memory.

**Parameters**

**allocator** (*function*) – CuPy memory allocator. It must have the same interface as the [cupy.cuda.alloc\(\)](#) function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator will be used (i.e., memory pool is disabled).

**cupy.cuda.using\_allocator**

`cupy.cuda.using_allocator(allocator=None)`

Sets a thread-local allocator for GPU memory inside context manager

**Parameters**

**allocator** (*function*) – CuPy memory allocator. It must have the same interface as the [cupy.cuda.alloc\(\)](#) function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator will be used (i.e., memory pool is disabled).

**cupy.cuda.set\_pinned\_memory\_allocator**

`cupy.cuda.set_pinned_memory_allocator(allocator=None)`

Sets the current allocator for the pinned memory.

**Parameters**

**allocator** (*function*) – CuPy pinned memory allocator. It must have the same interface as the [cupy.cuda.alloc\\_pinned\\_memory\(\)](#) function, which takes the buffer size as an argument and returns the device buffer of that size. When `None` is specified, raw memory allocator is used (i.e., memory pool is disabled).

## cupy.cuda.MemoryPool

**class** `cupy.cuda.MemoryPool`(*allocator=None*)

Memory pool for all GPU devices on the host.

A memory pool preserves any allocations even if they are freed by the user. Freed memory buffers are held by the memory pool as *free blocks*, and they are reused for further memory allocations of the same sizes. The allocated blocks are managed for each device, so one instance of this class can be used for multiple devices.

---

**Note:** When the allocation is skipped by reusing the pre-allocated block, it does not call `cudaMalloc` and therefore CPU-GPU synchronization does not occur. It makes interleaves of memory allocations and kernel invocations very fast.

---

---

**Note:** The memory pool holds allocated blocks without freeing as much as possible. It makes the program hold most of the device memory, which may make other CUDA programs running in parallel out-of-memory situation.

---

### Parameters

**allocator** (*function*) – The base CuPy memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

### Methods

**free\_all\_blocks**(*self*, *stream=None*)

Releases free blocks.

### Parameters

**stream** (`cupy.cuda.Stream`) – Release free blocks in the arena of the given stream. The default releases blocks in all arenas.

---

**Note:** A memory pool may split a free block for space efficiency. A split block is not released until all its parts are merged back into one even if `free_all_blocks()` is called.

---

**free\_all\_free**(*self*)

(Deprecated) Use `free_all_blocks()` instead.

**free\_bytes**(*self*) → `size_t`

Gets the total number of bytes acquired but not used by the pool.

### Returns

The total number of bytes acquired but not used by the pool.

### Return type

`int`

**get\_limit**(*self*) → `size_t`

Gets the upper limit of memory allocation of the current device.

### Returns

The number of bytes



**Return type**`int`**malloc**(*self*, *size\_t* *size*) → *MemoryPointer*

Allocates the memory, from the pool if possible.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryPool().malloc)
```

Also, the way to use a memory pool of Managed memory (Unified memory) as the default allocator is the following code:

```
set_allocator(MemoryPool(malloc_managed).malloc)
```

**Parameters**

**size** (*int*) – Size of the memory buffer to allocate in bytes.

**Returns**

Pointer to the allocated buffer.

**Return type***MemoryPointer***n\_free\_blocks**(*self*) → *size\_t*

Counts the total number of free blocks.

**Returns**

The total number of free blocks.

**Return type**`int`**set\_limit**(*self*, *size=None*, *fraction=None*)

Sets the upper limit of memory allocation of the current device.

When *fraction* is specified, its value will become a fraction of the amount of GPU memory that is available for allocation. For example, if you have a GPU with 2 GiB memory, you can either use `set_limit(fraction=0.5)` or `set_limit(size=1024**3)` to limit the memory size to 1 GiB.

*size* and *fraction* cannot be specified at the same time. If both of them are **not** specified or `0` is specified, the limit will be disabled.

---

**Note:** You can also set the limit by using CUPY\_GPU\_MEMORY\_LIMIT environment variable, see [Environment variables](#) for the details. The limit set by this method supersedes the value specified in the environment variable.

Also note that this method only changes the limit for the current device, whereas the environment variable sets the default limit for all devices.

---

**Parameters**

- **size** (*int*) – Limit size in bytes.
- **fraction** (*float*) – Fraction in the range of `[0, 1]`.

**total\_bytes**(*self*) → size\_t

Gets the total number of bytes acquired by the pool.

**Returns**

The total number of bytes acquired by the pool.

**Return type**

int

**used\_bytes**(*self*) → size\_t

Gets the total number of bytes used by the pool.

**Returns**

The total number of bytes used by the pool.

**Return type**

int

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## cupy.cuda.MemoryAsyncPool

**class** cupy.cuda.**MemoryAsyncPool**(*pool\_handles*='current')

(Experimental) CUDA memory pool for all GPU devices on the host.

A memory pool preserves any allocations even if they are freed by the user. One instance of this class can be used for multiple devices. This class uses CUDA's Stream Ordered Memory Allocator (supported on CUDA 11.2+). The simplest way to use this pool as CuPy's default allocator is the following code:

```
set_allocator(MemoryAsyncPool().malloc)
```

Using this feature requires CUDA >= 11.2 with a supported GPU and platform. If it is not supported, an error will be raised.

The current CuPy stream is used to allocate/free the memory.

**Parameters**

**pool\_handles** (*str* or *int*) – A flag to indicate which mempool to use. 'default' is for the device's default mempool, 'current' is for the current mempool (which could be the default one), and an *int* that represents cudaMemPool\_t created from elsewhere for an external mempool. A list consisting of these flags can also be accepted, in which case the list length must equal to the total number of visible devices so that the mempools for each device can be set independently.

**Warning:** This feature is currently experimental and subject to change.

---

**Note:** [MemoryAsyncPool](#) currently cannot work with memory hooks.

---

**See also:**

[Stream Ordered Memory Allocator](#)

## Methods

**free\_all\_blocks**(*self*, *stream=None*)

Releases free memory.

**Parameters**

**stream** ([cupy.cuda.Stream](#)) – Release memory freed on the given stream. If stream is None, the current stream is used.

**See also:**

[Physical Page Caching Behavior](#)

**free\_bytes**(*self*) → size\_t

Gets the total number of bytes acquired but not used by the pool.

**Returns**

The total number of bytes acquired but not used by the pool.

**Return type**

int

**get\_limit**(*self*) → size\_t

Gets the upper limit of memory allocation of the current device.

**Returns**

The number of bytes

**Return type**

int

---

**Note:** Unlike with [MemoryPool](#), [MemoryAsyncPool](#)’s [set\\_limit\(\)](#) method can only impose a *soft* limit. If other (non-CuPy) applications are also allocating memory from the same mempool, this limit may not be respected.

---

**malloc**(*self*, *size\_t size*) → [MemoryPointer](#)

Allocate memory from the current device’s pool on the current stream.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryAsyncPool().malloc)
```

**Parameters**

**size** ([int](#)) – Size of the memory buffer to allocate in bytes.

**Returns**

Pointer to the allocated buffer.

**Return type**

[MemoryPointer](#)

**n\_free\_blocks**(*self*) → size\_t

**set\_limit**(*self*, *size=None*, *fraction=None*)

Sets the upper limit of memory allocation of the current device.

When *fraction* is specified, its value will become a fraction of the amount of GPU memory that is available for allocation. For example, if you have a GPU with 2 GiB memory, you can either use `set_limit(fraction=0.5)` or `set_limit(size=1024**3)` to limit the memory size to 1 GiB.

*size* and *fraction* cannot be specified at the same time. If both of them are **not** specified or `0` is specified, the limit will be disabled.

---

**Note:** Unlike with [MemoryPool](#), [MemoryAsyncPool](#)'s `set_limit()` method can only impose a *soft* limit. If other (non-CuPy) applications are also allocating memory from the same mempool, this limit may not be respected. Internally, this limit is set via the `cudaMemPoolAttrReleaseThreshold` attribute.

---

---

**Note:** You can also set the limit by using `CUPY_GPU_MEMORY_LIMIT` environment variable, see [Environment variables](#) for the details. The limit set by this method supersedes the value specified in the environment variable.

Also note that this method only changes the limit for the current device, whereas the environment variable sets the default limit for all devices.

---

**Parameters**

- **size** (*int*) – Limit size in bytes.
- **fraction** (*float*) – Fraction in the range of `[0, 1]`.

**total\_bytes**(*self*) → size\_t

Gets the total number of bytes acquired by the pool.

**Returns**

The total number of bytes acquired by the pool.

**Return type**

[int](#)

**used\_bytes**(*self*) → size\_t

Gets the total number of bytes used by the pool.

**Returns**

The total number of bytes used by the pool.

**Return type**

[int](#)

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

```

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

## Attributes

**memoryAsyncHasStat**

## cupy.cuda.PinnedMemoryPool

**class** `cupy.cuda.PinnedMemoryPool(allocator=_malloc)`

Memory pool for pinned memory on the host.

Note that it preserves all allocated memory buffers even if the user explicitly release the one. Those released memory buffers are held by the memory pool as *free blocks*, and reused for further memory allocations of the same size.

### Parameters

**allocator** (*function*) – The base CuPy pinned memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

## Methods

**free**(*self*, *intptr\_t ptr*, *size\_t size*)

**free\_all\_blocks**(*self*)

Release free all blocks.

**malloc**(*self*, *size\_t size*) → *PinnedMemoryPointer*

**n\_free\_blocks**(*self*)

Count the total number of free blocks.

### Returns

The total number of free blocks.

### Return type

`int`

```
__eq__(value, /)
```

Return self==value.

```
__ne__(value, /)
```

Return self!=value.

```
__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## cupy.cuda.PythonFunctionAllocator

**class** `cupy.cuda.PythonFunctionAllocator(malloc_func, free_func)`

Allocator with python functions to perform memory allocation.

This allocator keeps functions corresponding to *malloc* and *free*, delegating the actual allocation to external sources while only handling the timing of the resource allocation and deallocation.

*malloc* should follow the signature `malloc(int, int) -> int` returning the pointer to the allocated memory given the *param* object, the number of bytes to allocate and the device id on which the allocation should take place.

Similarly, *free* should follow the signature `free(int, int)` with no return, taking the pointer to the allocated memory and the device id on which the memory was allocated.

If the external memory management supports asynchronous operations, the current CuPy stream can be retrieved inside *malloc\_func* and *free\_func* by calling `cupy.cuda.get_current_stream()`. To use external streams, wrap them with `cupy.cuda.ExternalStream()`.

### Parameters

- **malloc\_func** (*function*) – *malloc* function to be called.
- **free\_func** (*function*) – *free* function to be called.

### Methods

**malloc**(*self*, *size\_t* *size*) → *MemoryPointer*

```
__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.
```

## cupy.cuda.CFunctionAllocator

**class** `cupy.cuda.CFunctionAllocator(intptr_t param, intptr_t malloc_func, intptr_t free_func, owner)`

Allocator with C function pointers to allocation routines.

This allocator keeps raw pointers to a *param* object along with functions pointers to *malloc* and *free*, delegating the actual allocation to external sources while only handling the timing of the resource allocation and deallocation.

*malloc* should follow the signature `void*(*malloc)(void*, size_t, int)` returning the pointer to the allocated memory given the pointer to *param*, the number of bytes to allocate and the device id on which the allocation should take place.

Similarly, *free* should follow the signature `void(*free)(void*, void*, int)` with no return, taking the pointer to *param*, the pointer to the allocated memory and the device id on which the memory was allocated.

### Parameters

- **param** (*int*) – Address of *param*.
- **malloc\_func** (*int*) – Address of *malloc*.
- **free\_func** (*int*) – Address of *free*.
- **owner** (*object*) – Reference to the owner object to keep the param and the functions alive.

### Methods

**malloc**(*self*, *size\_t* size) → *MemoryPointer*

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## 5.6.3 Memory hook

<code>cupy.cuda.MemoryHook()</code>	Base class of hooks for Memory allocations.
<code>cupy.cuda.memory_hooks.DebugPrintHook([...])</code>	Memory hook that prints debug information.
<code>cupy.cuda.memory_hooks.LineProfileHook([...])</code>	Code line CuPy memory profiler.

## cupy.cuda.MemoryHook

### class cupy.cuda.MemoryHook

Base class of hooks for Memory allocations.

*MemoryHook* is an callback object. Registered memory hooks are invoked before and after memory is allocated from GPU device, and memory is retrieved from memory pool, and memory is released to memory pool.

Memory hooks that derive *MemoryHook* are required to implement six methods: *alloc\_preprocess()*, *alloc\_postprocess()*, *malloc\_preprocess()*, *malloc\_postprocess()*, *free\_preprocess()*, and *free\_postprocess()*. By default, these methods do nothing.

Specifically, *alloc\_preprocess()* (resp. *alloc\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is allocated from GPU device.

Likewise, *malloc\_preprocess()* (resp. *malloc\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is retrieved from memory pool.

Below is a pseudo code to describe how malloc and hooks work. Please note that *alloc\_preprocess()* and *alloc\_postprocess()* are not invoked if a cached free chunk is found:

```
def malloc(size):
    Call malloc_preprocess of all memory hooks
    Try to find a cached free chunk from memory pool
    if chunk is not found:
        Call alloc_preprocess for all memory hooks
        Invoke actual memory allocation to get a new chunk
        Call alloc_postprocess for all memory hooks
    Call malloc_postprocess for all memory hooks
```

Moreover, *free\_preprocess()* (resp. *free\_postprocess()*) of all memory hooks registered are called before (resp. after) memory is released to memory pool.

Below is a pseudo code to describe how free and hooks work:

```
def free(ptr):
    Call free_preprocess of all memory hooks
    Push a memory chunk of a given pointer back to memory pool
    Call free_postprocess for all memory hooks
```

To register a memory hook, use with statement. Memory hooks are registered to all method calls within with statement and are unregistered at the end of with statement.

---

**Note:** CuPy stores the dictionary of registered function hooks as a thread local object. So, memory hooks registered can be different depending on threads.

---



## Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \*\_)

**alloc\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

**alloc\_preprocess**(*self*, \*\**kwargs*)

Callback function invoked before allocating memory from GPU device.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**free\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after releasing memory to memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**free\_preprocess**(*self*, \*\**kwargs*)

Callback function invoked before releasing memory to memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**malloc\_postprocess**(*self*, \*\**kwargs*)

Callback function invoked after retrieving memory from memory pool.

### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in malloc.
- **pmem\_id** (*int*) – Pooled memory object ID. 0 if an error occurred in malloc.

**malloc\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before retrieving memory from memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

**Attributes**

**name** = 'MemoryHook'

**cupy.cuda.memory\_hooks.DebugPrintHook**

**class** cupy.cuda.memory\_hooks.DebugPrintHook(*file=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, flush=True*)

Memory hook that prints debug information.

This memory hook outputs the debug information of input arguments of **malloc** and **free** methods involved in the hooked functions at postprocessing time (that is, just after each method is called).

**Example**

The basic usage is to use it with **with** statement.

Code example:

```
>>> import cupy
>>> from cupy.cuda import memory_hooks
>>>
>>> cupy.cuda.set_allocator(cupy.cuda.MemoryPool().malloc)
>>> with memory_hooks.DebugPrintHook():
...     x = cupy.array([1, 2, 3])
...     del x
```

Output example:

```
{ "hook": "alloc", "device_id": 0, "mem_size": 512, "mem_ptr": 150496608256 }
{ "hook": "malloc", "device_id": 0, "size": 24, "mem_size": 512, "mem_ptr": 150496608256,
  ↪ "pmem_id": "0x7f39200c5278" }
{ "hook": "free", "device_id": 0, "mem_size": 512, "mem_ptr": 150496608256, "pmem_id":
  ↪ "0x7f39200c5278" }
```

where the output format is JSONL (JSON Lines) and `hook` is the name of hook point, and `device_id` is the CUDA Device ID, and `size` is the requested memory size to allocate, and `mem_size` is the rounded memory size to be allocated, and `mem_ptr` is the memory pointer, and `pmem_id` is the pooled memory object ID.

#### Variables

- **file** – Output file\_like object that redirect to.
- **flush** – If True, this hook forcibly flushes the text stream at the end of print. The default is True.

#### Methods

`__enter__(self)`

`__exit__(self, *_)`

`alloc_postprocess(self, **kwargs)`

Callback function invoked after allocating memory from GPU device.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

`alloc_preprocess(self, **kwargs)`

Callback function invoked before allocating memory from GPU device.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

`free_postprocess(self, **kwargs)`

Callback function invoked after releasing memory to memory pool.

##### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

`free_preprocess(self, **kwargs)`

Callback function invoked before releasing memory to memory pool.

##### Keyword Arguments

- `device_id` (*int*) – CUDA device ID
- `mem_size` (*int*) – Memory bytesize
- `mem_ptr` (*int*) – Memory pointer to free
- `pmem_id` (*int*) – Pooled memory object ID.

`malloc_postprocess`(*self*, *\*\*kwargs*)

Callback function invoked after retrieving memory from memory pool.

#### Keyword Arguments

- `device_id` (*int*) – CUDA device ID
- `size` (*int*) – Requested memory bytesize to allocate
- `mem_size` (*int*) – Rounded memory bytesize allocated
- `mem_ptr` (*int*) – Obtained memory pointer. 0 if an error occurred in `malloc`.
- `pmem_id` (*int*) – Pooled memory object ID. 0 if an error occurred in `malloc`.

`malloc_preprocess`(*self*, *\*\*kwargs*)

Callback function invoked before retrieving memory from memory pool.

#### Keyword Arguments

- `device_id` (*int*) – CUDA device ID
- `size` (*int*) – Requested memory bytesize to allocate
- `mem_size` (*int*) – Rounded memory bytesize to be allocated

`__eq__`(*value*, /)

Return `self==value`.

`__ne__`(*value*, /)

Return `self!=value`.

`__lt__`(*value*, /)

Return `self<value`.

`__le__`(*value*, /)

Return `self<=value`.

`__gt__`(*value*, /)

Return `self>value`.

`__ge__`(*value*, /)

Return `self>=value`.

#### Attributes

`name` = 'DebugPrintHook'

## cupy.cuda.memory\_hooks.LineProfileHook

**class** cupy.cuda.memory\_hooks.LineProfileHook(*max\_depth=0*)

Code line CuPy memory profiler.

This profiler shows line-by-line GPU memory consumption using traceback module. But, note that it can trace only CPython level, no Cython level. ref. <https://github.com/cython/cython/issues/1755>

### Example

Code example:

```
from cupy.cuda import memory_hooks
hook = memory_hooks.LineProfileHook()
with hook:
    # some CuPy codes
hook.print_report()
```

Output example:

```
_root (4.00KB, 4.00KB)
  lib/python3.6/unittest/__main__.py:18:<module> (4.00KB, 4.00KB)
    lib/python3.6/unittest/main.py:255:runTests (4.00KB, 4.00KB)
      tests/cupy_tests/test.py:37:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:38:test (1.00KB, 1.00KB)
      tests/cupy_tests/test.py:39:test (2.00KB, 2.00KB)
```

Each line shows:

```
{filename}:{lineno}:{func_name} ({used_bytes}, {acquired_bytes})
```

where *used\_bytes* is the memory bytes used from CuPy memory pool, and *acquired\_bytes* is the actual memory bytes the CuPy memory pool acquired from GPU device. *\_root* is a root node of the stack trace to show total memory usage.

#### Parameters

**max\_depth** (*int*) – maximum depth to follow stack traces. Default is 0 (no limit).

### Methods

**\_\_enter\_\_**(*self*)

**\_\_exit\_\_**(*self*, \*\_)

**alloc\_postprocess**(*self*, \*\*kwargs)

Callback function invoked after allocating memory from GPU device.

#### Keyword Arguments

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in allocation.

**alloc\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before allocating memory from GPU device.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**free\_postprocess**(*self*, *\*\*kwargs*)

Callback function invoked after releasing memory to memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**free\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before releasing memory to memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **mem\_size** (*int*) – Memory bytesize
- **mem\_ptr** (*int*) – Memory pointer to free
- **pmem\_id** (*int*) – Pooled memory object ID.

**malloc\_postprocess**(*self*, *\*\*kwargs*)

Callback function invoked after retrieving memory from memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize allocated
- **mem\_ptr** (*int*) – Obtained memory pointer. 0 if an error occurred in malloc.
- **pmem\_id** (*int*) – Pooled memory object ID. 0 if an error occurred in malloc.

**malloc\_preprocess**(*self*, *\*\*kwargs*)

Callback function invoked before retrieving memory from memory pool.

**Keyword Arguments**

- **device\_id** (*int*) – CUDA device ID
- **size** (*int*) – Requested memory bytesize to allocate
- **mem\_size** (*int*) – Rounded memory bytesize to be allocated

**print\_report**(*file*=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)

Prints a report of line memory profiling.

**\_\_eq\_\_**(*value*, /)

Return self==value.

```

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

```
name = 'LineProfileHook'
```

## 5.6.4 Streams and events

<code>cupy.cuda.Stream([null, non_blocking, ptds])</code>	CUDA stream.
<code>cupy.cuda.ExternalStream(ptr[, device_id])</code>	CUDA stream not managed by CuPy.
<code>cupy.cuda.get_current_stream(int device_id=-1)</code>	Gets the current CUDA stream for the specified CUDA device.
<code>cupy.cuda.Event([block, disable_timing, ...])</code>	CUDA event, a synchronization point of CUDA streams.
<code>cupy.cuda.get_elapsed_time(start_event, ...)</code>	Gets the elapsed time between two events.

### cupy.cuda.Stream

```
class cupy.cuda.Stream(null=False, non_blocking=False, ptds=False)
```

CUDA stream.

This class handles the CUDA stream handle in RAII way, i.e., when an Stream instance is destroyed by the GC, its handle is also destroyed.

Note that if both `null` and `ptds` are `False`, a plain new stream is created.

#### Parameters

- **null** (*bool*) – If `True`, the stream is a null stream (i.e. the default stream that synchronizes with all streams). Note that you can also use the `Stream.null` singleton object instead of creating a new null stream object.
- **ptds** (*bool*) – If `True` and `null` is `False`, the per-thread default stream is used. Note that you can also use the `Stream.ptds` singleton object instead of creating a new per-thread default stream object.
- **non\_blocking** (*bool*) – If `True` and both `null` and `ptds` are `False`, the stream does not synchronize with the `NULL` stream.

#### Variables

- **~Stream.ptr** (*intptr\_t*) – Raw stream handle.

- `~Stream.device_id` (*int*) – The ID of the device that the stream was created on. The value -1 is used for the singleton stream objects.

## Methods

`__enter__(self)`

`__exit__(self, *args)`

`add_callback(self, callback, arg)`

Adds a callback that is called when all queued work is done.

### Parameters

- **callback** (*function*) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, use the `launch_host_func()` method instead of this one, as it may be deprecated and removed from CUDA at some point.

---

`begin_capture(self, mode=None)`

Begin stream capture to construct a CUDA graph.

A call to this function must be paired with a call to `end_capture()` to complete the capture.

```
# create a non-blocking stream for the purpose of capturing
s1 = cp.cuda.Stream(non_blocking=True)
with s1:
    s1.begin_capture()
    # ... perform operations to construct a graph ...
    g = s1.end_capture()

# the returned graph can be launched on any stream (including s1)
g.launch(stream=s1)
s1.synchronize()

s2 = cp.cuda.Stream()
with s2:
    g.launch()
s2.synchronize()
```

### Parameters

- **mode** (*int*) – The stream capture mode. Default is `streamCaptureModeRelaxed`.

---

**Note:** During the stream capture, synchronous device-host transfers are not allowed. This has a particular implication for CuPy APIs, as some functions that internally require synchronous transfer would not work as expected and an exception would be raised. For further constraints of CUDA stream capture, please refer to the CUDA Programming Guide.

---



---

**Note:** Currently this capability is not supported on HIP.

---

**See also:**

[`cudaStreamBeginCapture\(\)`](#)

**`end_capture(self)`**

End stream capture and retrieve the constructed CUDA graph.

**Returns**

A CUDA graph object that encapsulates the captured work.

**Return type**

[`cupy.cuda.Graph`](#)

---

**Note:** Currently this capability is not supported on HIP.

---

**See also:**

[`cudaStreamEndCapture\(\)`](#)

**`is_capturing(self)`**

Check if the stream is capturing.

**Returns**

If the capturing status is successfully queried, the returned value indicates the capturing status. An exception could be raised if such a query is illegal, please refer to the CUDA Programming Guide for detail.

**Return type**

[`bool`](#)

**`launch_host_func(self, callback, arg)`**

Launch a callback on host when all queued work is done.

**Parameters**

- **`callback`** (*function*) – Callback function. It must take only one argument (user data object), and returns nothing.
- **`arg`** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, this method is recommended over [`add\_callback\(\)`](#), which may be deprecated and removed from CUDA at some point.

---

**See also:**

[`cudaLaunchHostFunc\(\)`](#)

**`record(self, event=None)`**

Records an event on the stream.

**Parameters**

**`event`** (*None* or [`cupy.cuda.Event`](#)) – CUDA event. If *None*, then a new plain event is created and used.

**Returns**

The recorded event.

**Return type***cupy.cuda.Event***See also:***cupy.cuda.Event.record()***synchronize(*self*)**

Waits for the stream completing all queued work.

**use(*self*)**

Makes this stream current.

If you want to switch a stream temporarily, use the *with* statement.

**wait\_event(*self*, *event*)**

Makes the stream wait for an event.

The future work on this stream will be done after the event.

**Parameters**

**event** (*cupy.cuda.Event*) – CUDA event.

**\_\_eq\_\_(*self*, *other*)****\_\_ne\_\_(*value*, /)**

Return self!=value.

**\_\_lt\_\_(*value*, /)**

Return self<value.

**\_\_le\_\_(*value*, /)**

Return self<=value.

**\_\_gt\_\_(*value*, /)**

Return self>value.

**\_\_ge\_\_(*value*, /)**

Return self>=value.

**Attributes****done**

True if all work on this stream has been done.

**null** = <Stream 0 (device -1)>

**ptds** = <Stream 2 (device -1)>

## cupy.cuda.ExternalStream

**class** cupy.cuda.ExternalStream(ptr, device\_id=-1)

CUDA stream not managed by CuPy.

This class allows to use external streams in CuPy by providing the stream pointer obtained from the CUDA runtime call. The user is in charge of managing the life-cycle of the stream.

### Parameters

- **ptr** (*intptr\_t*) – Address of the *cudaStream\_t* object.
- **device\_id** (*int*) – The ID of the device that the stream was created on. Default is -1, indicating it is unknown.

### Variables

- **~Stream.ptr** (*intptr\_t*) – Raw stream handle.
- **~Stream.device\_id** (*int*) – The ID of the device that the stream was created on. The value -1 is used to indicate it is unknown.

**Warning:** If `device_id` is not specified, the user is required to ensure legal operations of the stream. Specifically, the stream must be used on the device that it was created on.

## Methods

**\_\_enter\_\_**(self)

**\_\_exit\_\_**(self, \*args)

**add\_callback**(self, callback, arg)

Adds a callback that is called when all queued work is done.

### Parameters

- **callback** (*function*) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, use the `launch_host_func()` method instead of this one, as it may be deprecated and removed from CUDA at some point.

---

**begin\_capture**(self, mode=None)

Begin stream capture to construct a CUDA graph.

A call to this function must be paired with a call to `end_capture()` to complete the capture.

```
# create a non-blocking stream for the purpose of capturing
s1 = cp.cuda.Stream(non_blocking=True)
with s1:
    s1.begin_capture()
    # ... perform operations to construct a graph ...
    g = s1.end_capture()
```

(continues on next page)

(continued from previous page)

```
# the returned graph can be launched on any stream (including s1)
g.launch(stream=s1)
s1.synchronize()

s2 = cp.cuda.Stream()
with s2:
    g.launch()
s2.synchronize()
```

---

**Parameters**

**mode** (*int*) – The stream capture mode. Default is `streamCaptureModeRelaxed`.

---

**Note:** During the stream capture, synchronous device-host transfers are not allowed. This has a particular implication for CuPy APIs, as some functions that internally require synchronous transfer would not work as expected and an exception would be raised. For further constraints of CUDA stream capture, please refer to the CUDA Programming Guide.

---

---

**Note:** Currently this capability is not supported on HIP.

---

**See also:**

`cudaStreamBeginCapture()`

**end\_capture**(*self*)

End stream capture and retrieve the constructed CUDA graph.

**Returns**

A CUDA graph object that encapsulates the captured work.

**Return type**

*cupy.cuda.Graph*

---

**Note:** Currently this capability is not supported on HIP.

---

**See also:**

`cudaStreamEndCapture()`

**is\_capturing**(*self*)

Check if the stream is capturing.

**Returns**

If the capturing status is successfully queried, the returned value indicates the capturing status. An exception could be raised if such a query is illegal, please refer to the CUDA Programming Guide for detail.

**Return type**

*bool*

**launch\_host\_func**(*self*, *callback*, *arg*)

Launch a callback on host when all queued work is done.

**Parameters**

- **callback** (*function*) – Callback function. It must take only one argument (user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

---

**Note:** Whenever possible, this method is recommended over `add_callback()`, which may be deprecated and removed from CUDA at some point.

---

**See also:**

`cudaLaunchHostFunc()`

**record**(*self*, *event=None*)

Records an event on the stream.

**Parameters**

**event** (*None* or `cupy.cuda.Event`) – CUDA event. If *None*, then a new plain event is created and used.

**Returns**

The recorded event.

**Return type**

`cupy.cuda.Event`

**See also:**

`cupy.cuda.Event.record()`

**synchronize**(*self*)

Waits for the stream completing all queued work.

**use**(*self*)

Makes this stream current.

If you want to switch a stream temporarily, use the *with* statement.

**wait\_event**(*self*, *event*)

Makes the stream wait for an event.

The future work on this stream will be done after the event.

**Parameters**

**event** (`cupy.cuda.Event`) – CUDA event.

**\_\_eq\_\_**(*self*, *other*)

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

`__ge__(value, /)`  
Return self>=value.

## Attributes

**done**  
True if all work on this stream has been done.

## `cupy.cuda.get_current_stream`

`cupy.cuda.get_current_stream(int device_id=-1)`  
Gets the current CUDA stream for the specified CUDA device.

### Parameters

**device\_id** (*int*, *optional*) – Index of the device to check for the current stream. The currently active device is selected by default.

### Returns

The current CUDA stream.

### Return type

*cupy.cuda.Stream*

## `cupy.cuda.Event`

**class** `cupy.cuda.Event`(*block=False, disable\_timing=False, interprocess=False*)

CUDA event, a synchronization point of CUDA streams.

This class handles the CUDA event handle in RAII way, i.e., when an Event instance is destroyed by the GC, its handle is also destroyed.

### Parameters

- **block** (*bool*) – If True, the event blocks on the `synchronize()` method.
- **disable\_timing** (*bool*) – If True, the event does not prepare the timing data.
- **interprocess** (*bool*) – If True, the event can be passed to other processes.

### Variables

**~Event.ptr** (*intptr\_t*) – Raw event handle.

## Methods

**record**(*self, stream=None*)  
Records the event to a stream.

### Parameters

**stream** (*cupy.cuda.Stream*) – CUDA stream to record event. The null stream is used by default.

### See also:

*cupy.cuda.Stream.record()*

**synchronize(*self*)**

Synchronizes all device work to the event.

If the event is created as a blocking event, it also blocks the CPU thread until the event is done.

**\_\_eq\_\_(*value*, /)**

Return self==value.

**\_\_ne\_\_(*value*, /)**

Return self!=value.

**\_\_lt\_\_(*value*, /)**

Return self<value.

**\_\_le\_\_(*value*, /)**

Return self<=value.

**\_\_gt\_\_(*value*, /)**

Return self>value.

**\_\_ge\_\_(*value*, /)**

Return self>=value.

**Attributes****done**

True if the event is done.

**cupy.cuda.get\_elapsed\_time**

`cupy.cuda.get_elapsed_time(start_event, end_event)`

Gets the elapsed time between two events.

**Parameters**

- **start\_event** ([Event](#)) – Earlier event.
- **end\_event** ([Event](#)) – Later event.

**Returns**

Elapsed time in milliseconds.

**Return type**

[float](#)

**5.6.5 Graphs**


---

`cupy.cuda.Graph(*args, **kwargs)`

The CUDA graph object.

---

## cupy.cuda.Graph

**class** cupy.cuda.**Graph**(\*args, \*\*kwargs)

The CUDA graph object.

Currently this class cannot be initiated by the user and must be created via stream capture. See [begin\\_capture\(\)](#) for detail.

### Methods

**launch**(self, stream=None)

Launch the CUDA graph on the given stream.

#### Parameters

**stream** ([Stream](#)) – A CuPy stream object. If not specified (using the default value *None*), the graph is launched on the current stream.

#### See also:

[cudaGraphLaunch\(\)](#)

**upload**(self, stream=None)

Upload the CUDA graph to the given stream.

#### Parameters

**stream** ([Stream](#)) – A CuPy stream object. If not specified (using the default value *None*), the graph is uploaded the current stream.

#### See also:

[cudaGraphUpload\(\)](#)

**\_\_eq\_\_**(value, /)

Return self==value.

**\_\_ne\_\_**(value, /)

Return self!=value.

**\_\_lt\_\_**(value, /)

Return self<value.

**\_\_le\_\_**(value, /)

Return self<=value.

**\_\_gt\_\_**(value, /)

Return self>value.

**\_\_ge\_\_**(value, /)

Return self>=value.



**Attributes****graph****graphExec****5.6.6 Texture and surface memory**

<code>cupy.cuda.texture.ChannelFormatDescriptor(...)</code>	A class that holds the channel format description.
<code>cupy.cuda.texture.CUDAarray(...)</code>	Allocate a CUDA array ( <i>cudaArray_t</i> ) that can be used as texture memory.
<code>cupy.cuda.texture.ResourceDescriptor(...)</code>	A class that holds the resource description.
<code>cupy.cuda.texture.TextureDescriptor([...])</code>	A class that holds the texture description.
<code>cupy.cuda.texture.TextureObject(...)</code>	A class that holds a texture object.
<code>cupy.cuda.texture.SurfaceObject(...)</code>	A class that holds a surface object.

**cupy.cuda.texture.ChannelFormatDescriptor**

**class** `cupy.cuda.texture.ChannelFormatDescriptor`(*int x, int y, int z, int w, int f*)

A class that holds the channel format description. Equivalent to `cudaChannelFormatDesc`.

**Parameters**

- **x** (*int*) – the number of bits for the x channel.
- **y** (*int*) – the number of bits for the y channel.
- **z** (*int*) – the number of bits for the z channel.
- **w** (*int*) – the number of bits for the w channel.
- **f** (*int*) – the channel format. Use one of the values in `cudaChannelFormat*`, such as `cupy.cuda.runtime.cudaChannelFormatKindFloat`.

See also:

`cudaCreateChannelDesc()`

**Methods**

**get\_channel\_format**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value, /*)

Return `self==value`.

**\_\_ne\_\_**(*value, /*)

Return `self!=value`.

**\_\_lt\_\_**(*value, /*)

Return `self<value`.

**\_\_le\_\_**(*value, /*)

Return `self<=value`.

`__gt__(value, /)`

Return self>value.

`__ge__(value, /)`

Return self>=value.

## Attributes

`ptr`

## `cupy.cuda.texture.CUDAArray`

**class** `cupy.cuda.texture.CUDAArray`(*ChannelFormatDescriptor desc, size\_t width, size\_t height=0, size\_t depth=0, unsigned int flags=0*)

Allocate a CUDA array (*cudaArray\_t*) that can be used as texture memory. Depending on the input, either 1D, 2D, or 3D CUDA array is returned.

### Parameters

- **desc** (`ChannelFormatDescriptor`) – an instance of `ChannelFormatDescriptor`.
- **width** (`int`) – the width (in elements) of the array.
- **height** (`int`, *optional*) – the height (in elements) of the array.
- **depth** (`int`, *optional*) – the depth (in elements) of the array.
- **flags** (`int`, *optional*) – the flag for extensions. Use one of the values in `cudaArray*`, such as `cupy.cuda.runtime.cudaArrayDefault`.

**Warning:** The memory allocation of `CUDAArray` is done outside of CuPy’s memory management (enabled by default) due to CUDA’s limitation. Users of `CUDAArray` should be cautious about any out-of-memory possibilities.

See also:

`cudaMalloc3DArray()`

## Methods

**copy\_from**(*self, in\_arr, stream=None*)

Copy data from device or host array to CUDA array.

### Parameters

- **in\_arr** (`cupy.ndarray` or `numpy.ndarray`) –
- **stream** (`cupy.cuda.Stream`) – if not `None`, an asynchronous copy is performed.

---

**Note:** For CUDA arrays with different dimensions, the requirements for the shape of the input array are given as follows:

- 1D: (nch \* width,)
- 2D: (height, nch \* width)

- 3D: (depth, height, nch \* width)

where nch is the number of channels specified in *desc*.

---

**copy\_to**(*self*, *out\_arr*, *stream=None*)

Copy data from CUDA array to device or host array.

#### Parameters

- **out\_arr** (`cupy.ndarray` or `numpy.ndarray`) – must be C-contiguous
- **stream** (`cupy.cuda.Stream`) – if not `None`, an asynchronous copy is performed.

---

**Note:** For CUDA arrays with different dimensions, the requirements for the shape of the output array are given as follows:

- 1D: (nch \* width,)
- 2D: (height, nch \* width)
- 3D: (depth, height, nch \* width)

where nch is the number of channels specified in *desc*.

---

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

#### Attributes

**depth**

**desc**

**flags**

**height**

**ndim**

**ptr**

**width**

## cupy.cuda.texture.ResourceDescriptor

```
class cupy.cuda.texture.ResourceDescriptor(int restype, CUDAarray cuArr=None, ndarray arr=None,
                                          ChannelFormatDescriptor chDesc=None, size_t
                                          sizeInBytes=0, size_t width=0, size_t height=0, size_t
                                          pitchInBytes=0)
```

A class that holds the resource description. Equivalent to `cudaResourceDesc`.

### Parameters

- **restype** (*int*) – the resource type. Use one of the values in `cudaResourceType*`, such as `cupy.cuda.runtime.cudaResourceTypeArray`.
- **cuArr** (*CUDAarray*, *optional*) – An instance of *CUDAarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeArray`.
- **arr** (*cupy.ndarray*, *optional*) – An instance of *ndarray*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **chDesc** (*ChannelFormatDescriptor*, *optional*) – an instance of *ChannelFormatDescriptor*, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear` or `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **sizeInBytes** (*int*, *optional*) – total bytes in the linear memory, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypeLinear`.
- **width** (*int*, *optional*) – the width (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **height** (*int*, *optional*) – the height (in elements) of the 2D array, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.
- **pitchInBytes** (*int*, *optional*) – the number of bytes per pitch-aligned row, required if `restype` is set to `cupy.cuda.runtime.cudaResourceTypePitch2D`.

---

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

---

See also:

`cudaCreateTextureObject()`

### Methods

**get\_resource\_desc**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

```

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

```

arr
chDesc
cuArr
ptr

```

### cupy.cuda.texture.TextureDescriptor

```

class cupy.cuda.texture.TextureDescriptor(addressModes=None, int filterMode=0, int readMode=0,
                                          sRGB=None, borderColors=None, normalizedCoords=None,
                                          maxAnisotropy=None)

```

A class that holds the texture description. Equivalent to `cudaTextureDesc`.

#### Parameters

- **addressModes** (*tuple* or *list*) – an iterable with length up to 3, each element is one of the values in `cudaAddressMode*`, such as `cupy.cuda.runtime.cudaAddressModeWrap`.
- **filterMode** (*int*) – the filter mode. Use one of the values in `cudaFilterMode*`, such as `cupy.cuda.runtime.cudaFilterModePoint`.
- **readMode** (*int*) – the read mode. Use one of the values in `cudaReadMode*`, such as `cupy.cuda.runtime.cudaReadModeElementType`.
- **normalizedCoords** (*int*) – whether coordinates are normalized or not.
- **sRGB** (*int*, *optional*) –
- **borderColors** (*tuple* or *list*, *optional*) – an iterable with length up to 4.
- **maxAnisotropy** (*int*, *optional*) –

---

**Note:** A texture backed by *mipmap* arrays is currently not supported in CuPy.

---

See also:

`cudaCreateTextureObject()`

## Methods

**get\_texture\_desc**(*self*)

Returns a dict containing the input.

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

**ptr**

## cupy.cuda.texture.TextureObject

**class** `cupy.cuda.texture.TextureObject`(*ResourceDescriptor ResDesc*, *TextureDescriptor TexDesc*)

A class that holds a texture object. Equivalent to `cudaTextureObject_t`. The returned *TextureObject* instance can be passed as a argument when launching *RawKernel* or *ElementwiseKernel*.

### Parameters

- **ResDesc** (*ResourceDescriptor*) – an instance of the resource descriptor.
- **TexDesc** (*TextureDescriptor*) – an instance of the texture descriptor.

See also:

`cudaCreateTextureObject()`

## Methods

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

```

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

### Attributes

**ResDesc**

**TexDesc**

**ptr**

## cupy.cuda.texture.SurfaceObject

**class** cupy.cuda.texture.**SurfaceObject**(*ResourceDescriptor ResDesc*)

A class that holds a surface object. Equivalent to `cudaSurfaceObject_t`. The returned *SurfaceObject* instance can be passed as a argument when launching *RawKernel*.

### Parameters

**ResDesc** (*ResourceDescriptor*) – an intance of the resource descriptor.

**See also:**

`cudaCreateSurfaceObject()`

### Methods

```

__eq__(value, /)
    Return self==value.

__ne__(value, /)
    Return self!=value.

__lt__(value, /)
    Return self<value.

__le__(value, /)
    Return self<=value.

__gt__(value, /)
    Return self>value.

__ge__(value, /)
    Return self>=value.

```

## Attributes

**ResDesc**

**ptr**

## 5.6.7 NVTX

<code>cupy.cuda.nvtx.Mark</code> (message, int id_color=-1)	Marks an instantaneous event (marker) in the application.
<code>cupy.cuda.nvtx.MarkC</code> (message, uint32_t color=0)	Marks an instantaneous event (marker) in the application.
<code>cupy.cuda.nvtx.RangePush</code> (message, ...)	Starts a nested range.
<code>cupy.cuda.nvtx.RangePushC</code> (message, ...)	Starts a nested range.
<code>cupy.cuda.nvtx.RangePop</code> ()	Ends a nested range started by a <code>RangePush*()</code> call.

### `cupy.cuda.nvtx.Mark`

`cupy.cuda.nvtx.Mark`(message, int id\_color=-1)

Marks an instantaneous event (marker) in the application.

Markers are used to describe events at a specific time during execution of the application.

#### Parameters

- **message** (*str*) – Name of a marker.
- **id\_color** (*int*) – ID of color for a marker.

### `cupy.cuda.nvtx.MarkC`

`cupy.cuda.nvtx.MarkC`(message, uint32\_t color=0)

Marks an instantaneous event (marker) in the application.

Markers are used to describe events at a specific time during execution of the application.

#### Parameters

- **message** (*str*) – Name of a marker.
- **color** (*uint32*) – Color code for a marker.

### `cupy.cuda.nvtx.RangePush`

`cupy.cuda.nvtx.RangePush`(message, int id\_color=-1)

Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. This is particularly useful when profiling with Nsight Systems to help connect user-specified ranges with CuPy's internal CUDA-kernels. The duration of a range is defined by the corresponding pair of `RangePush()` to `RangePop()` calls, which can be nested.



```

from cupy.cuda.nvtx import RangePush, RangePop

RangePush("Nested Powers of A")
for i in range(N):
    RangePush("Iter {}: Double A".format(i))
    A = 2*A
    RangePop()
RangePop()

```

#### Parameters

- **message** (*str*) – Name of a range.
- **id\_color** (*int*) – ID of color for a range.

### cupy.cuda.nvtx.RangePushC

`cupy.cuda.nvtx.RangePushC(message, uint32_t color=0)`

Starts a nested range.

Ranges are used to describe events over a time span during execution of the application. This is particularly useful when profiling with Nsight Systems to help connect user-specified ranges with CuPy's internal CUDA-kernels. The duration of a range is defined by the corresponding pair of `RangePushC()` to `RangePop()` calls, which can be nested.

```

from cupy.cuda.nvtx import RangePushC, RangePop

RangePush("Nested Powers of A")
for i in range(N):
    RangePushC("Iter {}: Double A".format(i))
    A = 2*A
    RangePop()
RangePop()

```

#### Parameters

- **message** (*str*) – Name of a range.
- **color** (*uint32*) – ARGB color for a range.

### cupy.cuda.nvtx.RangePop

`cupy.cuda.nvtx.RangePop()`

Ends a nested range started by a `RangePush*()` call.

## 5.6.8 NCCL

<code>cupy.cuda.nccl.NcclCommunicator(int ndev, ...)</code>	Initialize an NCCL communicator for one device controlled by one process.
<code>cupy.cuda.nccl.get_build_version()</code>	
<code>cupy.cuda.nccl.get_version()</code>	Returns the runtime version of NCCL.
<code>cupy.cuda.nccl.get_unique_id()</code>	
<code>cupy.cuda.nccl.groupStart()</code>	Start a group of NCCL calls.
<code>cupy.cuda.nccl.groupEnd()</code>	End a group of NCCL calls.

### `cupy.cuda.nccl.NcclCommunicator`

**class** `cupy.cuda.nccl.NcclCommunicator(int ndev, tuple commId, int rank)`

Initialize an NCCL communicator for one device controlled by one process.

#### Parameters

- **ndev** (*int*) – Total number of GPUs to be used.
- **commId** (*tuple*) – The unique ID returned by `get_unique_id()`.
- **rank** (*int*) – The rank of the GPU managed by the current process.

#### Returns

An `NcclCommunicator` instance.

#### Return type

*NcclCommunicator*

---

**Note:** This method is for creating an NCCL communicator in a multi-process environment, typically managed by MPI or multiprocessing. For controlling multiple devices by one process, use `initAll()` instead.

---

#### See also:

`ncclCommInitRank`

#### Methods

**abort**(*self*)

**allGather**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *intptr\_t* stream)

**allReduce**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* op, *intptr\_t* stream)

**bcast**(*self*, *intptr\_t* buff, *int* count, *int* datatype, *int* root, *intptr\_t* stream)

**broadcast**(*self*, *intptr\_t* sendbuff, *intptr\_t* recvbuff, *int* count, *int* datatype, *int* root, *intptr\_t* stream)

**check\_async\_error**(*self*)

**destroy**(*self*)

**device\_id**(*self*)

**static initAll**(*devices*)

Initialize NCCL communicators for multiple devices in a single process.

**Parameters**

**devices** (*int* or *list of int*) – The number of GPUs or a list of GPUs to be used. For the former case, the first *devices* GPUs will be used.

**Returns**

A list of `NcclCommunicator` instances.

**Return type**

`list`

---

**Note:** This method is for creating a group of NCCL communicators, each controlling one device, in a single process like this:

```
from cupy.cuda import nccl
# Use 3 GPUs: #0, #2, and #3
comms = nccl.NcclCommunicator.initAll([0, 2, 3])
assert len(comms) == 3
```

In a multi-process setup, use the default initializer instead.

---

**See also:**

`ncclCommInitAll`

**rank\_id**(*self*)

**recv**(*self*, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* peer, *intptr\_t* stream)

**reduce**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* count, *int* datatype, *int* op, *int* root, *intptr\_t* stream)

**reduceScatter**(*self*, *intptr\_t* sendbuf, *intptr\_t* recvbuf, *size\_t* recvcnt, *int* datatype, *int* op, *intptr\_t* stream)

**send**(*self*, *intptr\_t* sendbuf, *size\_t* count, *int* datatype, *int* peer, *intptr\_t* stream)

**size**(*self*)

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## Attributes

`comm`

### `cupy.cuda.nccl.get_build_version`

`cupy.cuda.nccl.get_build_version()`

### `cupy.cuda.nccl.get_version`

`cupy.cuda.nccl.get_version()`

Returns the runtime version of NCCL.

This function will return 0 when built with NCCL version earlier than 2.3.4, which does not support `ncclGetVersion` API.

### `cupy.cuda.nccl.get_unique_id`

`cupy.cuda.nccl.get_unique_id()`

### `cupy.cuda.nccl.groupStart`

`cupy.cuda.nccl.groupStart()`

Start a group of NCCL calls. Must be paired with [`groupEnd\(\)`](#).

---

**Note:** This method is useful when the `NcclCommunicator` instances are created via [`initAll\(\)`](#). A typical usage pattern is like this:

```
comms = cupy.cuda.nccl.NcclCommunicator.initAll(n, dev_list)
# ... do some preparation work
cupy.cuda.nccl.groupStart()
for rank, comm in enumerate(comms):
    # ... make some collective calls ...
cupy.cuda.nccl.groupEnd()
```

Other use cases include fusing several NCCL calls into one, and point-to-point communications using [`send\(\)`](#) and [`recv\(\)`](#) (with NCCL 2.7+).

---

#### See also:

[ncclGroupStart](#), [Group Calls](#)

## cupy.cuda.nccl.groupEnd

`cupy.cuda.nccl.groupEnd()`

End a group of NCCL calls. Must be paired with [`groupStart\(\)`](#).

**Note:** This method is useful when the `NcclCommunicator` instances are created via [`initAll\(\)`](#). A typical usage pattern is like this:

```

comms = cupy.cuda.nccl.NcclCommunicator.initAll(n, dev_list)
# ... do some preparation work
cupy.cuda.nccl.groupStart()
for rank, comm in enumerate(comms):
    # ... make some collective calls ...
cupy.cuda.nccl.groupEnd()

```

Other use cases include fusing several NCCL calls into one, and point-to-point communications using [`send\(\)`](#) and [`recv\(\)`](#) (with NCCL 2.7+).

**See also:**

[ncclGroupEnd](#), [Group Calls](#)

## 5.6.9 Version

---

`cupy.cuda.get_local_runtime_version()`

Returns the version of the CUDA Runtime installed in the environment.

---

## cupy.cuda.get\_local\_runtime\_version

`cupy.cuda.get_local_runtime_version()`

Returns the version of the CUDA Runtime installed in the environment.

Unlike [`cupy.cuda.runtime.runtimeGetVersion\(\)`](#), which returns the CUDA Runtime version statically linked to CuPy, this function returns the version retrieved from the shared library installed on the host. Use this method to probe the CUDA Runtime version installed in the environment.

**Return type**

`int`

## 5.6.10 Runtime API

CuPy wraps CUDA Runtime APIs to provide the native CUDA operations. Please check the [CUDA Runtime API documentation](#) to use these functions.

---

`cupy.cuda.runtime.driverGetVersion()`

---

`cupy.cuda.runtime.runtimeGetVersion()`

Returns the version of the CUDA Runtime statically linked to CuPy.

continues on next page

Table 3 – continued from previous page

<code>cupy.cuda.runtime.getDevice()</code>	
<code>cupy.cuda.runtime.getDeviceProperties(int device)</code>	
<code>cupy.cuda.runtime.deviceGetAttribute(...)</code>	
<code>cupy.cuda.runtime.deviceGetByPCIBusId(...)</code>	
<code>cupy.cuda.runtime.deviceGetPCIBusId(int device)</code>	
<code>cupy.cuda.runtime.deviceGetDefaultMemPool(...)</code>	Get the default mempool on the current device.
<code>cupy.cuda.runtime.deviceGetMemPool(int device)</code>	Get the current mempool on the current device.
<code>cupy.cuda.runtime.deviceSetMemPool(...)</code>	Set the current mempool on the current device to pool.
<code>cupy.cuda.runtime.memPoolCreate(...)</code>	
<code>cupy.cuda.runtime.memPoolDestroy(intptr_t pool)</code>	
<code>cupy.cuda.runtime.memPoolTrimTo(...)</code>	
<code>cupy.cuda.runtime.getDeviceCount()</code>	
<code>cupy.cuda.runtime.setDevice(int device)</code>	
<code>cupy.cuda.runtime.deviceSynchronize()</code>	
<code>cupy.cuda.runtime.deviceCanAccessPeer(...)</code>	
<code>cupy.cuda.runtime.deviceEnablePeerAccess(...)</code>	
<code>cupy.cuda.runtime.deviceGetLimit(int limit)</code>	
<code>cupy.cuda.runtime.deviceSetLimit(int limit, ...)</code>	
<code>cupy.cuda.runtime.malloc(size_t size)</code>	
<code>cupy.cuda.runtime.mallocManaged(size_t size, ...)</code>	
<code>cupy.cuda.runtime.malloc3DArray(...)</code>	
<code>cupy.cuda.runtime.mallocArray(...)</code>	
<code>cupy.cuda.runtime.mallocAsync(size_t size, ...)</code>	
<code>cupy.cuda.runtime.mallocFromPoolAsync(...)</code>	
<code>cupy.cuda.runtime.hostAlloc(size_t size, ...)</code>	
<code>cupy.cuda.runtime.hostRegister(intptr_t ptr, ...)</code>	

continues on next page

Table 3 – continued from previous page

<code>cupy.cuda.runtime.hostUnregister(intptr_t ptr)</code>
<code>cupy.cuda.runtime.free(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeHost(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeArray(intptr_t ptr)</code>
<code>cupy.cuda.runtime.freeAsync(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memGetInfo()</code>
<code>cupy.cuda.runtime.memcpy(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyAsync(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyPeer(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpyPeerAsync(...)</code>
<code>cupy.cuda.runtime.memcpy2D(intptr_t dst, ...)</code>
<code>cupy.cuda.runtime.memcpy2DAsync(...)</code>
<code>cupy.cuda.runtime.memcpy2DFromArray(...)</code>
<code>cupy.cuda.runtime.memcpy2DFromArrayAsync(...)</code>
<code>cupy.cuda.runtime.memcpy2DToArray(...)</code>
<code>cupy.cuda.runtime.memcpy2DToArrayAsync(...)</code>
<code>cupy.cuda.runtime.memcpy3D(...)</code>
<code>cupy.cuda.runtime.memcpy3DAsync(...)</code>
<code>cupy.cuda.runtime.memset(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memsetAsync(intptr_t ptr, ...)</code>
<code>cupy.cuda.runtime.memPrefetchAsync(...)</code>
<code>cupy.cuda.runtime.memAdvise(intptr_t devPtr, ...)</code>
<code>cupy.cuda.runtime.pointerGetAttributes(...)</code>
<code>cupy.cuda.runtime.streamCreate()</code>
<code>cupy.cuda.runtime.streamCreateWithFlags(...)</code>

continues on next page

Table 3 – continued from previous page

<code>cupy.cuda.runtime.streamDestroy(intptr_t stream)</code>	
<code>cupy.cuda.runtime.streamSynchronize(...)</code>	
<code>cupy.cuda.runtime.streamAddCallback(...)</code>	
<code>cupy.cuda.runtime.streamQuery(intptr_t stream)</code>	
<code>cupy.cuda.runtime.streamWaitEvent(...)</code>	
<code>cupy.cuda.runtime.launchHostFunc(...)</code>	
<code>cupy.cuda.runtime.eventCreate()</code>	
<code>cupy.cuda.runtime.eventCreateWithFlags(...)</code>	
<code>cupy.cuda.runtime.eventDestroy(intptr_t event)</code>	
<code>cupy.cuda.runtime.eventElapsedTime(...)</code>	
<code>cupy.cuda.runtime.eventQuery(intptr_t event)</code>	
<code>cupy.cuda.runtime.eventRecord(...)</code>	
<code>cupy.cuda.runtime.eventSynchronize(...)</code>	
<code>cupy.cuda.runtime.ipcGetMemHandle(...)</code>	
<code>cupy.cuda.runtime.ipcOpenMemHandle(...)</code>	
<code>cupy.cuda.runtime.ipcCloseMemHandle(...)</code>	
<code>cupy.cuda.runtime.ipcGetEventHandle(...)</code>	
<code>cupy.cuda.runtime.ipcOpenEventHandle(...)</code>	
<code>cupy.cuda.runtime.profilerStart()</code>	Enable profiling.
<code>cupy.cuda.runtime.profilerStop()</code>	Disable profiling.

### `cupy.cuda.runtime.driverGetVersion`

`cupy.cuda.runtime.driverGetVersion()` → int



**cupy.cuda.runtime.runtimeGetVersion**

`cupy.cuda.runtime.runtimeGetVersion()` → int

Returns the version of the CUDA Runtime statically linked to CuPy.

**See also:**

`cupy.cuda.get_local_runtime_version()`

**cupy.cuda.runtime.getDevice**

`cupy.cuda.runtime.getDevice()` → int

**cupy.cuda.runtime.getDeviceProperties**

`cupy.cuda.runtime.getDeviceProperties(int device)`

**cupy.cuda.runtime.deviceGetAttribute**

`cupy.cuda.runtime.deviceGetAttribute(int attrib, int device)` → int

**cupy.cuda.runtime.deviceGetByPCIBusId**

`cupy.cuda.runtime.deviceGetByPCIBusId(unicode pci_bus_id)` → int

**cupy.cuda.runtime.deviceGetPCIBusId**

`cupy.cuda.runtime.deviceGetPCIBusId(int device)` → unicode

**cupy.cuda.runtime.deviceGetDefaultMemPool**

`cupy.cuda.runtime.deviceGetDefaultMemPool(int device)` → intptr\_t

Get the default mempool on the current device.

**cupy.cuda.runtime.deviceGetMemPool**

`cupy.cuda.runtime.deviceGetMemPool(int device)` → intptr\_t

Get the current mempool on the current device.

**cupy.cuda.runtime.deviceSetMemPool**

`cupy.cuda.runtime.deviceSetMemPool(int device, intptr_t pool)`  
Set the current mempool on the current device to pool.

**cupy.cuda.runtime.memPoolCreate**

`cupy.cuda.runtime.memPoolCreate(MemPoolProps props) → intptr_t`

**cupy.cuda.runtime.memPoolDestroy**

`cupy.cuda.runtime.memPoolDestroy(intptr_t pool)`

**cupy.cuda.runtime.memPoolTrimTo**

`cupy.cuda.runtime.memPoolTrimTo(intptr_t pool, size_t size)`

**cupy.cuda.runtime.getDeviceCount**

`cupy.cuda.runtime.getDeviceCount() → int`

**cupy.cuda.runtime.setDevice**

`cupy.cuda.runtime.setDevice(int device)`

**cupy.cuda.runtime.deviceSynchronize**

`cupy.cuda.runtime.deviceSynchronize()`

**cupy.cuda.runtime.deviceCanAccessPeer**

`cupy.cuda.runtime.deviceCanAccessPeer(int device, int peerDevice) → int`

**cupy.cuda.runtime.deviceEnablePeerAccess**

`cupy.cuda.runtime.deviceEnablePeerAccess(int peerDevice)`

**cupy.cuda.runtime.deviceGetLimit**

`cupy.cuda.runtime.deviceGetLimit(int limit) → size_t`

**cupy.cuda.runtime.deviceSetLimit**

`cupy.cuda.runtime.deviceSetLimit(int limit, size_t value)`

**cupy.cuda.runtime.malloc**

`cupy.cuda.runtime.malloc(size_t size) → intptr_t`

**cupy.cuda.runtime.mallocManaged**

`cupy.cuda.runtime.mallocManaged(size_t size, unsigned int flags=cudaMemAttachGlobal) → intptr_t`

**cupy.cuda.runtime.malloc3DArray**

`cupy.cuda.runtime.malloc3DArray(intptr_t descPtr, size_t width, size_t height, size_t depth, unsigned int flags=0) → intptr_t`

**cupy.cuda.runtime.mallocArray**

`cupy.cuda.runtime.mallocArray(intptr_t descPtr, size_t width, size_t height, unsigned int flags=0) → intptr_t`

**cupy.cuda.runtime.mallocAsync**

`cupy.cuda.runtime.mallocAsync(size_t size, intptr_t stream) → intptr_t`

**cupy.cuda.runtime.mallocFromPoolAsync**

`cupy.cuda.runtime.mallocFromPoolAsync(size_t size, intptr_t pool, intptr_t stream) → intptr_t`

**cupy.cuda.runtime.hostAlloc**

`cupy.cuda.runtime.hostAlloc(size_t size, unsigned int flags) → intptr_t`

### **cupy.cuda.runtime.hostRegister**

`cupy.cuda.runtime.hostRegister(intptr_t ptr, size_t size, unsigned int flags)`

### **cupy.cuda.runtime.hostUnregister**

`cupy.cuda.runtime.hostUnregister(intptr_t ptr)`

### **cupy.cuda.runtime.free**

`cupy.cuda.runtime.free(intptr_t ptr)`

### **cupy.cuda.runtime.freeHost**

`cupy.cuda.runtime.freeHost(intptr_t ptr)`

### **cupy.cuda.runtime.freeArray**

`cupy.cuda.runtime.freeArray(intptr_t ptr)`

### **cupy.cuda.runtime.freeAsync**

`cupy.cuda.runtime.freeAsync(intptr_t ptr, intptr_t stream)`

### **cupy.cuda.runtime.memGetInfo**

`cupy.cuda.runtime.memGetInfo()`

### **cupy.cuda.runtime.memcpy**

`cupy.cuda.runtime.memcpy(intptr_t dst, intptr_t src, size_t size, int kind)`

### **cupy.cuda.runtime.memcpyAsync**

`cupy.cuda.runtime.memcpyAsync(intptr_t dst, intptr_t src, size_t size, int kind, intptr_t stream)`

### **cupy.cuda.runtime.memcpyPeer**

`cupy.cuda.runtime.memcpyPeer(intptr_t dst, int dstDevice, intptr_t src, int srcDevice, size_t size)`

**cupy.cuda.runtime.memcpyPeerAsync**

`cupy.cuda.runtime.memcpyPeerAsync(intptr_t dst, int dstDevice, intptr_t src, int srcDevice, size_t size, intptr_t stream)`

**cupy.cuda.runtime.memcpy2D**

`cupy.cuda.runtime.memcpy2D(intptr_t dst, size_t dpitch, intptr_t src, size_t spitch, size_t width, size_t height, MemoryKind kind)`

**cupy.cuda.runtime.memcpy2DAsync**

`cupy.cuda.runtime.memcpy2DAsync(intptr_t dst, size_t dpitch, intptr_t src, size_t spitch, size_t width, size_t height, MemoryKind kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy2DFromArray**

`cupy.cuda.runtime.memcpy2DFromArray(intptr_t dst, size_t dpitch, intptr_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, int kind)`

**cupy.cuda.runtime.memcpy2DFromArrayAsync**

`cupy.cuda.runtime.memcpy2DFromArrayAsync(intptr_t dst, size_t dpitch, intptr_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, int kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy2DToArray**

`cupy.cuda.runtime.memcpy2DToArray(intptr_t dst, size_t wOffset, size_t hOffset, intptr_t src, size_t spitch, size_t width, size_t height, int kind)`

**cupy.cuda.runtime.memcpy2DToArrayAsync**

`cupy.cuda.runtime.memcpy2DToArrayAsync(intptr_t dst, size_t wOffset, size_t hOffset, intptr_t src, size_t spitch, size_t width, size_t height, int kind, intptr_t stream)`

**cupy.cuda.runtime.memcpy3D**

`cupy.cuda.runtime.memcpy3D(intptr_t Memcpy3DParmsPtr)`

**cupy.cuda.runtime.memcpy3DAsync**

cupy.cuda.runtime.**memcpy3DAsync**(*intptr\_t Memcpy3DParmsPtr, intptr\_t stream*)

**cupy.cuda.runtime.memset**

cupy.cuda.runtime.**memset**(*intptr\_t ptr, int value, size\_t size*)

**cupy.cuda.runtime.memsetAsync**

cupy.cuda.runtime.**memsetAsync**(*intptr\_t ptr, int value, size\_t size, intptr\_t stream*)

**cupy.cuda.runtime.memPrefetchAsync**

cupy.cuda.runtime.**memPrefetchAsync**(*intptr\_t devPtr, size\_t count, int dstDevice, intptr\_t stream*)

**cupy.cuda.runtime.memAdvise**

cupy.cuda.runtime.**memAdvise**(*intptr\_t devPtr, size\_t count, int advice, int device*)

**cupy.cuda.runtime.pointerGetAttributes**

cupy.cuda.runtime.**pointerGetAttributes**(*intptr\_t ptr*) → PointerAttributes

**cupy.cuda.runtime.streamCreate**

cupy.cuda.runtime.**streamCreate**() → intptr\_t

**cupy.cuda.runtime.streamCreateWithFlags**

cupy.cuda.runtime.**streamCreateWithFlags**(*unsigned int flags*) → intptr\_t

**cupy.cuda.runtime.streamDestroy**

cupy.cuda.runtime.**streamDestroy**(*intptr\_t stream*)

**cupy.cuda.runtime.streamSynchronize**

cupy.cuda.runtime.**streamSynchronize**(*intptr\_t stream*)

**cupy.cuda.runtime.streamAddCallback**

`cupy.cuda.runtime.streamAddCallback(intptr_t stream, callback, intptr_t arg, unsigned int flags=0)`

**cupy.cuda.runtime.streamQuery**

`cupy.cuda.runtime.streamQuery(intptr_t stream)`

**cupy.cuda.runtime.streamWaitEvent**

`cupy.cuda.runtime.streamWaitEvent(intptr_t stream, intptr_t event, unsigned int flags=0)`

**cupy.cuda.runtime.launchHostFunc**

`cupy.cuda.runtime.launchHostFunc(intptr_t stream, callback, intptr_t arg)`

**cupy.cuda.runtime.eventCreate**

`cupy.cuda.runtime.eventCreate() → intptr_t`

**cupy.cuda.runtime.eventCreateWithFlags**

`cupy.cuda.runtime.eventCreateWithFlags(unsigned int flags) → intptr_t`

**cupy.cuda.runtime.eventDestroy**

`cupy.cuda.runtime.eventDestroy(intptr_t event)`

**cupy.cuda.runtime.eventElapsedTime**

`cupy.cuda.runtime.eventElapsedTime(intptr_t start, intptr_t end) → float`

**cupy.cuda.runtime.eventQuery**

`cupy.cuda.runtime.eventQuery(intptr_t event)`

**cupy.cuda.runtime.eventRecord**

`cupy.cuda.runtime.eventRecord(intptr_t event, intptr_t stream)`

### **cupy.cuda.runtime.eventSynchronize**

`cupy.cuda.runtime.eventSynchronize(intptr_t event)`

### **cupy.cuda.runtime.ipcGetMemHandle**

`cupy.cuda.runtime.ipcGetMemHandle(intptr_t devPtr)`

### **cupy.cuda.runtime.ipcOpenMemHandle**

`cupy.cuda.runtime.ipcOpenMemHandle(bytes handle, unsigned int flags=cudaIpcMemLazyEnablePeerAccess)`

### **cupy.cuda.runtime.ipcCloseMemHandle**

`cupy.cuda.runtime.ipcCloseMemHandle(intptr_t devPtr)`

### **cupy.cuda.runtime.ipcGetEventHandle**

`cupy.cuda.runtime.ipcGetEventHandle(intptr_t event)`

### **cupy.cuda.runtime.ipcOpenEventHandle**

`cupy.cuda.runtime.ipcOpenEventHandle(bytes handle)`

### **cupy.cuda.runtime.profilerStart**

`cupy.cuda.runtime.profilerStart()`

Enable profiling.

A user can enable CUDA profiling. When an error occurs, it raises an exception.

See the CUDA document for detail.

### **cupy.cuda.runtime.profilerStop**

`cupy.cuda.runtime.profilerStop()`

Disable profiling.

A user can disable CUDA profiling. When an error occurs, it raises an exception.

See the CUDA document for detail.



## 5.7 Custom kernels

<code>cupy.ElementwiseKernel</code> ( <i>in_params</i> , ...[, ...])	User-defined elementwise kernel.
<code>cupy.ReductionKernel</code> ( <i>unicode in_params</i> , ...)	User-defined reduction kernel.
<code>cupy.RawKernel</code> ( <i>unicode code</i> , <i>unicode name</i> , ...)	User-defined custom kernel.
<code>cupy.RawModule</code> ( <i>unicode code</i> =None, *, ...[, ...])	User-defined custom module.
<code>cupy.fuse</code> (*args, **kwargs)	Decorator that fuses a function.

### 5.7.1 cupy.ElementwiseKernel

```
class cupy.ElementwiseKernel(in_params, out_params, operation, name='kernel', reduce_dims=True,
                             preamble="", no_return=False, return_tuple=False, **kwargs)
```

User-defined elementwise kernel.

This class can be used to define an elementwise kernel with or without broadcasting.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

#### Parameters

- **in\_params** (*str*) – Input argument list.
- **out\_params** (*str*) – Output argument list.
- **operation** (*str*) – The body in the loop written in CUDA-C/C++.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce\_dims** (*bool*) – If `False`, the shapes of array arguments are kept within the kernel invocation. The shapes are reduced (i.e., the arrays are reshaped without copy to the minimum dimension) by default. It may make the kernel fast by reducing the index calculations.
- **options** (*tuple*) – Compile options passed to NVRTC. For details, see [https://docs.nvidia.com/cuda/nVRTC/index.html#group\\_\\_options](https://docs.nvidia.com/cuda/nVRTC/index.html#group__options).
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **no\_return** (*bool*) – If `True`, `__call__` returns `None`.
- **return\_tuple** (*bool*) – If `True`, `__call__` always returns tuple of array even if single value is returned.
- **loop\_prep** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the kernel function definition and above the `for` loop.
- **after\_loop** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the bottom of the kernel function definition.

## Methods

### `__call__()`

Compiles and invokes the elementwise kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes or dimensions are not compatible. It means that single `ElementwiseKernel` object may be compiled into multiple kernel binaries.

#### Parameters

- **args** – Arguments of the kernel.
- **size** (*int*) – Range size of the indices. By default, the range size is automatically determined from the result of broadcasting. This parameter must be specified if and only if all ndarrays are *raw* and the range size cannot be determined automatically.
- **block\_size** (*int*) – Number of threads per block. By default, the value is set to 128.

#### Returns

If `no_return` has not set, arrays are returned according to the `out_params` argument of the `__init__` method. If `no_return` has set, `None` is returned.

### `__eq__(value, /)`

Return `self==value`.

### `__ne__(value, /)`

Return `self!=value`.

### `__lt__(value, /)`

Return `self<value`.

### `__le__(value, /)`

Return `self<=value`.

### `__gt__(value, /)`

Return `self>value`.

### `__ge__(value, /)`

Return `self>=value`.

## Attributes

### `cached_code`

Returns `next(iter(self.cached_codes.values()))`.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

### `cached_codes`

Returns a dict that has input types as keys and codes values.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

### `in_params`

### `kwargs`

**name**  
**nargs**  
**nin**  
**no\_return**  
**nout**  
**operation**  
**out\_params**  
**params**  
**preamble**  
**reduce\_dims**  
**return\_tuple**

### 5.7.2 cupy.ReductionKernel

```
class cupy.ReductionKernel(unicode in_params, unicode out_params, map_expr, reduce_expr,
                           post_map_expr, identity, name=u'reduce_kernel', reduce_type=None,
                           reduce_dims=True, preamble=u'', options=())
```

User-defined reduction kernel.

This class can be used to define a reduction kernel with or without broadcasting.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

#### Parameters

- **in\_params** (*str*) – Input argument list.
- **out\_params** (*str*) – Output argument list.
- **map\_expr** (*str*) – Mapping expression for input values.
- **reduce\_expr** (*str*) – Reduction expression.
- **post\_map\_expr** (*str*) – Mapping expression for reduced values.
- **identity** (*str*) – Identity value for starting the reduction.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce\_type** (*str*) – Type of values to be used for reduction. This type is used to store the special variables `a`.
- **reduce\_dims** (*bool*) – If True, input arrays are reshaped without copy to smaller dimensions for efficiency.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **options** (*tuple of str*) – Additional compilation options.

## Methods

### `__call__()`

Compiles and invokes the reduction kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes, ndims, or axis are not compatible. It means that single `ReductionKernel` object may be compiled into multiple kernel binaries.

#### Parameters

- **args** – Arguments of the kernel.
- **out** (`cupy.ndarray`) – The output array. This can only be specified if `args` does not contain the output array.
- **axis** (`int` or `tuple of ints`) – Axis or axes along which the reduction is performed.
- **keepdims** (`bool`) – If `True`, the specified axes are remained as axes of length one.
- **stream** (`cupy.cuda.Stream`, *optional*) – The CUDA stream to launch the kernel on. If not given, the current stream will be used.

#### Returns

Arrays are returned according to the `out_params` argument of the `__init__` method.

### `__eq__(value, /)`

Return `self==value`.

### `__ne__(value, /)`

Return `self!=value`.

### `__lt__(value, /)`

Return `self<value`.

### `__le__(value, /)`

Return `self<=value`.

### `__gt__(value, /)`

Return `self>value`.

### `__ge__(value, /)`

Return `self>=value`.

## Attributes

### `cached_code`

Returns `next(iter(self.cached_codes.values()))`.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

### `cached_codes`

Returns a dict that has input types as keys and codes values.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

```

identity
    unicode
        Type
            identity
in_params
map_expr
name
nargs
nin
nout
options
out_params
params
post_map_expr
preamble
reduce_dims
reduce_expr
reduce_type

```

### 5.7.3 `cupy.RawKernel`

```

class cupy.RawKernel(unicode code, unicode name, tuple options=(), unicode backend=u'nvrtc', bool
                      translate_cucomplex=False, *, bool enable_cooperative_groups=False, bool
                      jitify=False)

```

User-defined custom kernel.

This class can be used to define a custom kernel using raw CUDA source.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

#### Parameters

- **code** (*str*) – CUDA source code.
- **name** (*str*) – Name of the kernel function.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see [https://docs.nvidia.com/cuda/nvrtc/index.html#group\\_options](https://docs.nvidia.com/cuda/nvrtc/index.html#group_options) or <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#command-option-description>
- **backend** (*str*) – Either `nvrtc` or `nvcc`. Defaults to `nvrtc`

- **translate\_cucomplex** (*bool*) – Whether the CUDA source includes the header *cuComplex.h* or not. If set to `True`, any code that uses the functions from *cuComplex.h* will be translated to its Thrust counterpart. Defaults to `False`.
- **enable\_cooperative\_groups** (*bool*) – Whether to enable cooperative groups in the CUDA source. If set to `True`, compile options are configured properly and the kernel is launched with `cuLaunchCooperativeKernel` so that cooperative groups can be used from the CUDA source. This feature is only supported in CUDA 9 or later.
- **jitify** (*bool*) – Whether or not to use `Jitify` to assist NVRTC to compile C++ kernels. Defaults to `False`.

---

**Note:** Starting CuPy v13.0.0, *RawKernel* by default compiles with the C++11 standard (`-std=c++11`) if it's not specified in options.

---

## Methods

**\_\_call\_\_**(*self*, *grid*, *block*, *args*, \*, *shared\_mem*=0)

Compiles and invokes the kernel.

The compilation runs only if the kernel is not cached.

### Parameters

- **grid** (*tuple*) – Size of grid in blocks.
- **block** (*tuple*) – Dimensions of each thread block.
- **args** (*tuple*) – Arguments of the kernel.
- **shared\_mem** (*int*) – Dynamic shared-memory size per thread block in bytes.

**compile**(*self*, *log\_stream*=None)

Compile the current kernel.

In general, you don't have to call this method; kernels are compiled implicitly on the first call.

### Parameters

**log\_stream** (*object*) – Pass either `sys.stdout` or a file object to which the compiler output will be written. Defaults to `None`.

**\_\_eq\_\_**(*value*, /)

Return `self==value`.

**\_\_ne\_\_**(*value*, /)

Return `self!=value`.

**\_\_lt\_\_**(*value*, /)

Return `self<value`.

**\_\_le\_\_**(*value*, /)

Return `self<=value`.

**\_\_gt\_\_**(*value*, /)

Return `self>value`.

**\_\_ge\_\_**(*value*, /)

Return `self>=value`.

## Attributes

### attributes

Returns a dictionary containing runtime kernel attributes. This is a read-only property; to overwrite the attributes, use

```
kernel = RawKernel(...) # arguments omitted
kernel.max_dynamic_shared_size_bytes = ...
kernel.preferred_shared_memory_carveout = ...
```

Note that the two attributes shown in the above example are the only two currently settable in CUDA.

Any attribute not existing in the present CUDA toolkit version will have the value -1.

#### Returns

A dictionary containing the kernel's attributes.

#### Return type

dict

### backend

### binary\_version

The binary architecture version that was used during compilation, in the format: 10\*major + minor.

### cache\_mode\_ca

Indicates whether option “-Xptxas -dlcm=ca” was set during compilation.

### code

### const\_size\_bytes

The size in bytes of constant memory used by the function.

### enable\_cooperative\_groups

### file\_path

### kernel

### local\_size\_bytes

The size in bytes of local memory used by the function.

### max\_dynamic\_shared\_size\_bytes

The maximum dynamically-allocated shared memory size in bytes that can be used by the function. Can be set.

### max\_threads\_per\_block

The maximum number of threads per block that can successfully launch the function on the device.

### name

### num\_regs

The number of registers used by the function.

### options

**preferred\_shared\_memory\_carveout**

On devices that have a unified L1 cache and shared memory, indicates the fraction to be used for shared memory as a *percentage* of the total. If the fraction does not exactly equal a supported shared memory capacity, then the next larger supported capacity is used. Can be set.

**ptx\_version**

The PTX virtual architecture version that was used during compilation, in the format: 10\*major + minor.

**shared\_size\_bytes**

The size in bytes of the statically-allocated shared memory used by the function. This is separate from any dynamically-allocated shared memory, which must be specified when the function is called.

## 5.7.4 cupy.RawModule

```
class cupy.RawModule(unicode code=None, *, unicode path=None, tuple options=(), unicode backend=u'nvrtc',
                    bool translate_cucomplex=False, bool enable_cooperative_groups=False,
                    name_expressions=None, bool jitify=False)
```

User-defined custom module.

This class can be used to either compile raw CUDA sources or load CUDA modules (\*.cubin, \*.ptx). This class is useful when a number of CUDA kernels in the same source need to be retrieved.

For the former case, the CUDA source code is compiled when any method is called. For the latter case, an existing CUDA binary (\*.cubin) or a PTX file can be loaded by providing its path.

CUDA kernels in a *RawModule* can be retrieved by calling *get\_function()*, which will return an instance of *RawKernel*. (Same as in *RawKernel*, the generated binary is also cached.)

**Parameters**

- **code** (*str*) – CUDA source code. Mutually exclusive with *path*.
- **path** (*str*) – Path to cubin/ptx. Mutually exclusive with *code*.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see [https://docs.nvidia.com/cuda/nvrtc/index.html#group\\_options](https://docs.nvidia.com/cuda/nvrtc/index.html#group_options) or <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#command-option-description>.
- **backend** (*str*) – Either *nvrtc* or *nvcc*. Defaults to *nvrtc*.
- **translate\_cucomplex** (*bool*) – Whether the CUDA source includes the header *cuComplex.h* or not. If set to *True*, any code that uses the functions from *cuComplex.h* will be translated to its Thrust counterpart. Defaults to *False*.
- **enable\_cooperative\_groups** (*bool*) – Whether to enable cooperative groups in the CUDA source. If set to *True*, compile options are configured properly and the kernel is launched with *cuLaunchCooperativeKernel* so that cooperative groups can be used from the CUDA source. This feature is only supported in CUDA 9 or later.
- **name\_expressions** (*sequence of str*) – A sequence (e.g. list) of strings referring to the names of C++ global/template kernels. For example, *name\_expressions=['func1<int>', 'func1<double>', 'func2']* for the template kernel *func1<T>* and non-template kernel *func2*. Strings in this tuple must then be passed, one at a time, to *get\_function()* to retrieve the corresponding kernel.
- **jitify** (*bool*) – Whether or not to use *Jitify* to assist NVRTC to compile C++ kernels. Defaults to *False*.



---

**Note:** Starting CuPy v13.0.0, *RawModule* by default compiles with the C++11 standard (`-std=c++11`) if it's not specified in options.

---



---

**Note:** Each kernel in *RawModule* possesses independent function attributes.

---



---

**Note:** Before CuPy v8.0.0, the compilation happens at initialization. Now, it happens at the first time retrieving any object (kernels or pointers) from the module.

---

## Methods

**compile**(*self*, *log\_stream=None*)

Compile the current module.

In general, you don't have to call this method; kernels are compiled implicitly on the first call.

### Parameters

**log\_stream** (*object*) – Pass either `sys.stdout` or a file object to which the compiler output will be written. Defaults to `None`.

---

**Note:** Calling *compile()* will reset the internal state of a *RawKernel*.

---

**get\_function**(*self*, *unicode name*)

Retrieve a CUDA kernel by its name from the module.

### Parameters

**name** (*str*) – Name of the kernel function. For C++ global/template kernels, *name* refers to one of the name expressions specified when initializing the present *RawModule* instance.

### Returns

An *RawKernel* instance.

### Return type

*RawKernel*

---

**Note:** The following example shows how to retrieve one of the specialized C++ template kernels:

```
code = r'''
template<typename T>
__global__ void func(T* in_arr) { /* do something */ }
'''

kers = ('func<int>', 'func<float>', 'func<double>')
mod = cupy.RawModule(code=code, options=('--std=c++11',),
                     name_expressions=kers)

// retrieve func<int>
ker_int = mod.get_function(kers[0])
```

**See also:**

`nVRTCAddNameExpression` and `nVRTCGetLoweredName` from [Accessing Lowered Names](#) of the NVRTC documentation.

**get\_global**(*self*, *name*)

Retrieve a pointer to a global symbol by its name from the module.

**Parameters**

**name** (*str*) – Name of the global symbol.

**Returns**

A handle to the global symbol.

**Return type**

`MemoryPointer`

---

**Note:** This method can be used to access, for example, constant memory:

```
# to get a pointer to "arr" declared in the source like this:
# __constant__ float arr[10];
memptr = mod.get_global("arr")
# ...wrap it using cupy.ndarray with a known shape
arr_ndarray = cp.ndarray((10,), cp.float32, memptr)
# ...perform data transfer to initialize it
arr_ndarray[...] = cp.random.random((10,), dtype=cp.float32)
# ...and arr is ready to be accessed by RawKernels
```

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

### Attributes

`backend`

`code`

`enable_cooperative_groups`

`file_path`

`module`

`name_expressions`

`options`

## 5.7.5 `cupy.fuse`

`cupy.fuse(*args, **kwargs)`

Decorator that fuses a function.

This decorator can be used to define an elementwise or reduction kernel more easily than [ElementwiseKernel](#) or [ReductionKernel](#).

Since the fused kernels are cached and reused, it is recommended to reuse the same decorated functions instead of e.g. decorating local functions that are defined multiple times.

### Parameters

**kernel\_name** (*str*) – Name of the fused kernel function. If omitted, the name of the decorated function is used.

### Example

```
>>> @cupy.fuse(kernel_name='squared_diff')
... def squared_diff(x, y):
...     return (x - y) * (x - y)
...
>>> x = cupy.arange(10)
>>> y = cupy.arange(10)[::-1]
>>> squared_diff(x, y)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
```

## 5.7.6 JIT kernel definition

Supported Python built-in functions include: `range`, `len()`, `max()`, `min()`.

**Note:** If loop unrolling is needed, use `cupyx.jit.range()` instead of the built-in `range`.

`cupyx.jit.rawkernel`(*\*, mode, device*)

A decorator compiles a Python function into CUDA kernel.

continues on next page

Table 4 – continued from previous page

<code>cupyx.jit.threadIdx</code>	dim3 threadIdx
<code>cupyx.jit.blockDim</code>	dim3 blockDim
<code>cupyx.jit.blockIdx</code>	dim3 blockIdx
<code>cupyx.jit.gridDim</code>	dim3 gridDim
<code>cupyx.jit.grid(ndim)</code>	Compute the thread index in the grid.
<code>cupyx.jit.gridsize(ndim)</code>	Compute the grid size.
<code>cupyx.jit.laneid()</code>	Returns the lane ID of the calling thread, ranging in <code>[0, jit.warpsize)</code> .
<code>cupyx.jit.warpsize</code>	Returns the number of threads in a warp.
<code>cupyx.jit.range(*args[, unroll])</code>	Range with loop unrolling support.
<code>cupyx.jit.syncthreads()</code>	Calls <code>__syncthreads()</code> .
<code>cupyx.jit.syncwarp(*[, mask])</code>	Calls <code>__syncwarp()</code> .
<code>cupyx.jit.shfl_sync(mask, var, val_id, *[, ...])</code>	Calls the <code>__shfl_sync</code> function.
<code>cupyx.jit.shfl_up_sync(mask, var, val_id, *)</code>	Calls the <code>__shfl_up_sync</code> function.
<code>cupyx.jit.shfl_down_sync(mask, var, val_id, *)</code>	Calls the <code>__shfl_down_sync</code> function.
<code>cupyx.jit.shfl_xor_sync(mask, var, val_id, *)</code>	Calls the <code>__shfl_xor_sync</code> function.
<code>cupyx.jit.shared_memory(dtype, size[, alignment])</code>	Allocates shared memory and returns it as a 1-D array.
<code>cupyx.jit.atomic_add(array, index, value[, ...])</code>	Calls the <code>atomicAdd</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_sub(array, index, value[, ...])</code>	Calls the <code>atomicSub</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_exch(array, index, value[, ...])</code>	Calls the <code>atomicExch</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_min(array, index, value[, ...])</code>	Calls the <code>atomicMin</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_max(array, index, value[, ...])</code>	Calls the <code>atomicMax</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_inc(array, index, value[, ...])</code>	Calls the <code>atomicInc</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_dec(array, index, value[, ...])</code>	Calls the <code>atomicDec</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_cas(array, index, value[, ...])</code>	Calls the <code>atomicCAS</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_and(array, index, value[, ...])</code>	Calls the <code>atomicAnd</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_or(array, index, value[, ...])</code>	Calls the <code>atomicOr</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.atomic_xor(array, index, value[, ...])</code>	Calls the <code>atomicXor</code> function to operate atomically on <code>array[index]</code> .
<code>cupyx.jit.cg.this_grid()</code>	Returns the current grid group ( <code>_GridGroup</code> ).
<code>cupyx.jit.cg.this_thread_block()</code>	Returns the current thread block group ( <code>_ThreadBlockGroup</code> ).
<code>cupyx.jit.cg.sync(group)</code>	Calls <code>cg::sync()</code> .
<code>cupyx.jit.cg.memcpy_async(group, dst, ...[, ...])</code>	Calls <code>cg::memcpy_sync()</code> .
<code>cupyx.jit.cg.wait(group)</code>	Calls <code>cg::wait()</code> .
<code>cupyx.jit.cg.wait_prior(group)</code>	Calls <code>cg::wait_prior&lt;N&gt;()</code> .
<code>cupyx.jit._interface._JitRawKernel(func, ...)</code>	JIT CUDA kernel object.

### **cupyx.jit.rawkernel**

`cupyx.jit.rawkernel(*, mode='cuda', device=False)`

A decorator compiles a Python function into CUDA kernel.

### **cupyx.jit.threadIdx**

`cupyx.jit.threadIdx = <Data code = "threadIdx", type = dim3>`

dim3 threadIdx

An integer vector type based on uint3 that is used to specify dimensions.

#### **Variables**

- `x (uint32)` –
- `y (uint32)` –
- `z (uint32)` –

### **cupyx.jit.blockDim**

`cupyx.jit.blockDim = <Data code = "blockDim", type = dim3>`

dim3 blockDim

An integer vector type based on uint3 that is used to specify dimensions.

#### **Variables**

- `x (uint32)` –
- `y (uint32)` –
- `z (uint32)` –

### **cupyx.jit.blockIdx**

`cupyx.jit.blockIdx = <Data code = "blockIdx", type = dim3>`

dim3 blockIdx

An integer vector type based on uint3 that is used to specify dimensions.

#### **Variables**

- `x (uint32)` –
- `y (uint32)` –
- `z (uint32)` –

### cupyx.jit.gridDim

`cupyx.jit.gridDim` = <Data code = "gridDim", type = dim3>  
dim3 gridDim

An integer vector type based on uint3 that is used to specify dimensions.

#### Variables

- `x` (uint32) –
- `y` (uint32) –
- `z` (uint32) –

### cupyx.jit.grid

`cupyx.jit.grid(ndim)` = <cupyx.jit function>

Compute the thread index in the grid.

Computation of the first integer is as follows:

```
jit.threadIdx.x + jit.blockIdx.x * jit.blockDim.x
```

and for the other two integers the y and z attributes are used.

#### Parameters

**ndim** (*int*) – The dimension of the grid. Only 1, 2, or 3 is allowed.

#### Returns

If ndim is 1, an integer is returned, otherwise a tuple.

#### Return type

*int* or *tuple*

---

**Note:** This function follows the convention of Numba's `numba.cuda.grid()`.

---

### cupyx.jit.gridsize

`cupyx.jit.gridsize(ndim)` = <cupyx.jit function>

Compute the grid size.

Computation of the first integer is as follows:

```
jit.blockDim.x * jit.gridDim.x
```

and for the other two integers the y and z attributes are used.

#### Parameters

**ndim** (*int*) – The dimension of the grid. Only 1, 2, or 3 is allowed.

#### Returns

If ndim is 1, an integer is returned, otherwise a tuple.

#### Return type

*int* or *tuple*

---

**Note:** This function follows the convention of Numba's `numba.cuda.gridsize()`.

---

### `cupyx.jit.laneid`

`cupyx.jit.laneid` = <cupyx.jit function>

Returns the lane ID of the calling thread, ranging in `[0, jit.warpsize)`.

---

**Note:** Unlike `numba.cuda.laneid`, this is a callable function instead of a property.

---

### `cupyx.jit.warpsize`

`cupyx.jit.warpsize` = <Data code = "warpSize", type = int>

Returns the number of threads in a warp.

**See also:**

`numba.cuda.warpsize`

### `cupyx.jit.range`

`cupyx.jit.range(*args, unroll=None)` = <cupyx.jit function>

Range with loop unrolling support.

#### Parameters

- **start** (*int*) – Same as that of built-in `range`.
- **stop** (*int*) – Same as that of built-in `range`.
- **step** (*int*) – Same as that of built-in `range`.
- **unroll** (*int or bool or None*) –
  - If `True`, add `#pragma unroll` directive before the loop.
  - If `False`, add `#pragma unroll(1)` directive before the loop to disable unrolling.
  - If an *int*, add `#pragma unroll(n)` directive before the loop, where the integer *n* means the number of iterations to unroll.
  - If `None` (default), leave the control of loop unrolling to the compiler (no `#pragma`).

**See also:**

`#pragma unroll`

### cupyx.jit.syncthreads

`cupyx.jit.syncthreads` = <cupyx.jit function>

Calls `__syncthreads()`.

See also:

[Synchronization functions](#)

### cupyx.jit.syncwarp

`cupyx.jit.syncwarp(*, mask=4294967295)` = <cupyx.jit function>

Calls `__syncwarp()`.

**Parameters**

**mask** (*int*) – Active threads in a warp. Default is 0xffffffff.

See also:

[Synchronization functions](#)

### cupyx.jit.shfl\_sync

`cupyx.jit.shfl_sync(mask, var, val_id, *, width=32)` = <cupyx.jit function>

Calls the `__shfl_sync` function. Please refer to [Warp Shuffle Functions](#) for detailed explanation.

### cupyx.jit.shfl\_up\_sync

`cupyx.jit.shfl_up_sync(mask, var, val_id, *, width=32)` = <cupyx.jit function>

Calls the `__shfl_up_sync` function. Please refer to [Warp Shuffle Functions](#) for detailed explanation.

### cupyx.jit.shfl\_down\_sync

`cupyx.jit.shfl_down_sync(mask, var, val_id, *, width=32)` = <cupyx.jit function>

Calls the `__shfl_down_sync` function. Please refer to [Warp Shuffle Functions](#) for detailed explanation.

### cupyx.jit.shfl\_xor\_sync

`cupyx.jit.shfl_xor_sync(mask, var, val_id, *, width=32)` = <cupyx.jit function>

Calls the `__shfl_xor_sync` function. Please refer to [Warp Shuffle Functions](#) for detailed explanation.

### cupyx.jit.shared\_memory

`cupyx.jit.shared_memory(dtype, size, alignment=None)` = <cupyx.jit function>

Allocates shared memory and returns it as a 1-D array.

**Parameters**

- **dtype** (*dtype*) – The dtype of the returned array.
- **size** (*int* or *None*) – If *int* type, the size of static shared memory. If *None*, declares the shared memory with extern specifier.



- **alignment** (*int* or *None*) – Enforce the alignment via `__align__(N)`.

### cupyx.jit.atomic\_add

`cupyx.jit.atomic_add(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicAdd` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A [cupy.ndarray](#) to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of [atomic\\_cas](#), this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in [atomic\\_cas](#) to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### cupyx.jit.atomic\_sub

`cupyx.jit.atomic_sub(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicSub` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A [cupy.ndarray](#) to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of [atomic\\_cas](#), this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in [atomic\\_cas](#) to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### cupyx.jit.atomic\_exch

`cupyx.jit.atomic_exch(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicExch` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A [cupy.ndarray](#) to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.

- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### `cupyx.jit.atomic_min`

`cupyx.jit.atomic_min(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicMin` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### `cupyx.jit.atomic_max`

`cupyx.jit.atomic_max(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicMax` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### cupyx.jit.atomic\_inc

`cupyx.jit.atomic_inc(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicInc` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### cupyx.jit.atomic\_dec

`cupyx.jit.atomic_dec(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicDec` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### cupyx.jit.atomic\_cas

`cupyx.jit.atomic_cas(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicCAS` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### `cupyx.jit.atomic_and`

`cupyx.jit.atomic_and(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicAnd` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### `cupyx.jit.atomic_or`

`cupyx.jit.atomic_or(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicOr` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.
- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

[Numba's corresponding atomic functions](#)

### `cupyx.jit.atomic_xor`

`cupyx.jit.atomic_xor(array, index, value, alt_value=None) = <cupyx.jit function>`

Calls the `atomicXor` function to operate atomically on `array[index]`. Please refer to [Atomic Functions](#) for detailed explanation.

#### Parameters

- **array** – A `cupy.ndarray` to index over.
- **index** – A valid index such that the address to the corresponding array element `array[index]` can be computed.

- **value** – Represent the value to use for the specified operation. For the case of `atomic_cas`, this is the value for `array[index]` to compare with.
- **alt\_value** – Only used in `atomic_cas` to represent the value to swap to.

See also:

Numba’s corresponding atomic functions

### `cupyx.jit.cg.this_grid`

`cupyx.jit.cg.this_grid` = <cupyx.jit function>

Returns the current grid group (`_GridGroup`).

See also:

`cupyx.jit.cg._GridGroup`, `numba.cuda.cg.this_grid()`

### `cupyx.jit.cg.this_thread_block`

`cupyx.jit.cg.this_thread_block` = <cupyx.jit function>

Returns the current thread block group (`_ThreadBlockGroup`).

See also:

`cupyx.jit.cg._ThreadBlockGroup`

### `cupyx.jit.cg.sync`

`cupyx.jit.cg.sync(group)` = <cupyx.jit function>

Calls `cg::sync()`.

#### Parameters

**group** – a valid cooperative group

See also:

`cg::sync`

### `cupyx.jit.cg.memcpy_async`

`cupyx.jit.cg.memcpy_async(group, dst, dst_idx, src, src_idx, size, *, aligned_size=None)` = <cupyx.jit function>

Calls `cg::memcpy_sync()`.

#### Parameters

- **group** – a valid cooperative group
- **dst** – the destination array that can be viewed as a 1D C-contiguous array
- **dst\_idx** – the start index of the destination array element
- **src** – the source array that can be viewed as a 1D C-contiguous array
- **src\_idx** – the start index of the source array element
- **size** (*int*) – the number of bytes to be copied from `src[src_index]` to `dst[dst_idx]`

- **aligned\_size** (*int*) – Use `cuda::aligned_size_t<N>` to guarantee the compiler that `src/dst` are at least `N`-bytes aligned. The behavior is undefined if the guarantee is not held.

See also:

`cg::memcpy_sync`

### `cupyx.jit.cg.wait`

`cupyx.jit.cg.wait(group)` = <cupyx.jit function>

Calls `cg::wait()`.

#### Parameters

**group** – a valid cooperative group

### `cupyx.jit.cg.wait_prior`

`cupyx.jit.cg.wait_prior(group)` = <cupyx.jit function>

Calls `cg::wait_prior<N>()`.

#### Parameters

- **group** – a valid cooperative group
- **step** (*int*) – wait for the first `N` steps to finish

### `cupyx.jit._interface._JitRawKernel`

**class** `cupyx.jit._interface._JitRawKernel(func, mode, device)`

JIT CUDA kernel object.

The decorator `:func:cupyx.jit.rawkernel` converts the target function to an object of this class. This class is not intended to be instantiated by users.

#### Methods

**\_\_call\_\_**(*grid, block, args, shared\_mem=0, stream=None*)

Calls the CUDA kernel.

The compilation will be deferred until the first function call. CuPy's JIT compiler infers the types of arguments at the call time, and will cache the compiled kernels for speeding up any subsequent calls.

#### Parameters

- **grid** (*tuple of int*) – Size of grid in blocks.
- **block** (*tuple of int*) – Dimensions of each thread block.
- **args** (*tuple*) – Arguments of the kernel. The type of all elements must be `bool`, `int`, `float`, `complex`, NumPy scalar or `cupy.ndarray`.
- **shared\_mem** (*int*) – Dynamic shared-memory size per thread block in bytes.
- **stream** (`cupy.cuda.Stream`) – CUDA stream.

See also:

*JIT kernel definition*

**`__getitem__`**(*grid\_and\_block*)

Numba-style kernel call.

**See also:**

*JIT kernel definition*

**`__eq__`**(*value*, /)

Return self==value.

**`__ne__`**(*value*, /)

Return self!=value.

**`__lt__`**(*value*, /)

Return self<value.

**`__le__`**(*value*, /)

Return self<=value.

**`__gt__`**(*value*, /)

Return self>value.

**`__ge__`**(*value*, /)

Return self>=value.

## Attributes

### **cached\_code**

Returns *next(iter(self.cached\_codes.values()))*.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

### **cached\_codes**

Returns a dict that has input types as keys and codes values.

This property method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

## 5.7.7 Kernel binary memoization

<code>cupy.memoize</code> (bool for_each_device=False)	Makes a function memoizing the result for each argument and device.
<code>cupy.clear_memo</code> ()	Clears the memoized results for all functions decorated by memoize.

## cupy.memoize

`cupy.memoize(bool for_each_device=False)`

Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

### Parameters

**for\_each\_device** (*bool*) – If `True`, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

## cupy.clear\_memo

`cupy.clear_memo()`

Clears the memoized results for all functions decorated by `memoize`.

# 5.8 Distributed

## 5.8.1 Communication between processes

<code>init_process_group(n_devices, rank, *, ...)</code>	Start <i>cupyx.distributed</i> and obtain a communicator.
<code>NCCLBackend(n_devices, rank[, host, port, ...])</code>	Interface that uses NVIDIA's NCCL to perform communications.

## cupyx.distributed.init\_process\_group

`cupyx.distributed.init_process_group(n_devices, rank, *, backend='nccl', host=None, port=None, use_mpi=False)`

Start *cupyx.distributed* and obtain a communicator.

This call initializes the distributed environment, it needs to be called for every process that is involved in the communications.

A single device per returned communication is only allowed. It is the user responsibility of setting the appropriated gpu to be used before creating and using the communicator.

Currently the user needs to specify each process rank and the total number of processes, and start all the processes in different hosts manually.

The process with rank 0 will spawn a TCP server using a subprocess that listens in the port indicated by the env var `CUPYX_DISTRIBUTED_PORT`, the rank 0 must be executed in the host determined by the env var `CUPYX_DISTRIBUTED_HOST`. In case their values are not specified, `'127.0.0.1'` and `13333` will be used by default.

Note that this feature is expected to be used within a trusted cluster environment.



### Example

```

>>> import cupy
>>> def process_0():
...     import cupyx.distributed
...     cupy.cuda.Device(0).use()
...     comm = cupyx.distributed.init_process_group(2, 0)
...     array = cupy.ones(1)
...     comm.broadcast(array, 0)
...
>>> def process_1():
...     import cupyx.distributed
...     cupy.cuda.Device(1).use()
...     comm = cupyx.distributed.init_process_group(2, 1)
...     array = cupy.zeros(1)
...     comm.broadcast(array, 0)
...     cupy.equal(array, cupy.ones(1))

```

#### Parameters

- **n\_devices** (*int*) – Total number of devices that will be used in the distributed execution.
- **rank** (*int*) – Unique id of the GPU that the communicator is associated to its value needs to be  $0 \leq \text{rank} < n\_devices$ .
- **backend** (*str*) – Backend to use for the communications. Optional, defaults to “nccl”.
- **host** (*str*) – host address for the process rendezvous on initialization defaults to *None*.
- **port** (*int*) – port for the process rendezvous on initialization defaults to *None*.
- **use\_mpi** (*bool*) – if *False*, it avoids using MPI for synchronization and uses the provided TCP server for exchanging CPU only information. defaults to *False*.

#### Returns

**object used to perform communications, adheres to the**  
Backend specification:

**Return type**  
Backend

### cupyx.distributed.NCCLBackend

**class** cupyx.distributed.NCCLBackend(*n\_devices*, *rank*, *host*='127.0.0.1', *port*=13333, *use\_mpi*=*False*)

Interface that uses NVIDIA’s NCCL to perform communications.

#### Parameters

- **n\_devices** (*int*) – Total number of devices that will be used in the distributed execution.
- **rank** (*int*) – Unique id of the GPU that the communicator is associated to its value needs to be  $0 \leq \text{rank} < n\_devices$ .
- **host** (*str*, *optional*) – host address for the process rendezvous on initialization. Defaults to “127.0.0.1”.
- **port** (*int*, *optional*) – port used for the process rendezvous on initialization. Defaults to 13333.

- **use\_mpi** (*bool*, *optional*) – switch between MPI and use the included TCP server for initialization & synchronization. Defaults to *False*.

## Methods

**all\_gather**(*in\_array*, *out\_array*, *count*, *stream=None*)

Performs an all gather operation.

### Parameters

- **in\_array** (*cupy.ndarray*) – array to be sent.
- **out\_array** (*cupy.ndarray*) – array where the result with be stored.
- **count** (*int*) – Number of elements to send to each rank.
- **stream** (*cupy.cuda.Stream*, *optional*) – if supported, stream to perform the communication.

**all\_reduce**(*in\_array*, *out\_array*, *op='sum'*, *stream=None*)

Performs an all reduce operation.

### Parameters

- **in\_array** (*cupy.ndarray*) – array to be sent.
- **out\_array** (*cupy.ndarray*) – array where the result with be stored.
- **op** (*str*) – reduction operation, can be one of ('sum', 'prod', 'min' 'max'), arrays of complex type only support 'sum'. Defaults to 'sum'.
- **stream** (*cupy.cuda.Stream*, *optional*) – if supported, stream to perform the communication.

**all\_to\_all**(*in\_array*, *out\_array*, *stream=None*)

Performs an all to all operation.

### Parameters

- **in\_array** (*cupy.ndarray*) – array to be sent. Its shape must be (*total\_ranks*, ...).
- **out\_array** (*cupy.ndarray*) – array where the result with be stored. Its shape must be (*total\_ranks*, ...).
- **stream** (*cupy.cuda.Stream*, *optional*) – if supported, stream to perform the communication.

**barrier**()

Performs a barrier operation.

The barrier is done in the cpu and is a explicit synchronization mechanism that halts the thread progression.

**broadcast**(*in\_out\_array*, *root=0*, *stream=None*)

Performs a broadcast operation.

### Parameters

- **in\_out\_array** (*cupy.ndarray*) – array to be sent for *root* rank. Other ranks will receive the broadcast data here.
- **root** (*int*, *optional*) – rank of the process that will send the broadcast. Defaults to 0.
- **stream** (*cupy.cuda.Stream*, *optional*) – if supported, stream to perform the communication.

**gather**(*in\_array*, *out\_array*, *root*=0, *stream*=None)

Performs a gather operation.

#### Parameters

- **in\_array** (`cupy.ndarray`) – array to be sent.
- **out\_array** (`cupy.ndarray`) – array where the result will be stored. Its shape must be (*total\_ranks*, ...).
- **root** (`int`) – rank that will receive *in\_array* from other ranks.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**recv**(*out\_array*, *peer*, *stream*=None)

Performs a receive operation.

#### Parameters

- **array** (`cupy.ndarray`) – array used to receive data.
- **peer** (`int`) – rank of the process *array* will be received from.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**reduce**(*in\_array*, *out\_array*, *root*=0, *op*='sum', *stream*=None)

Performs a reduce operation.

#### Parameters

- **in\_array** (`cupy.ndarray`) – array to be sent.
- **out\_array** (`cupy.ndarray`) – array where the result will be stored. will only be modified by the *root* process.
- **root** (`int`, *optional*) – rank of the process that will perform the reduction. Defaults to 0.
- **op** (`str`) – reduction operation, can be one of ('sum', 'prod', 'min', 'max'), arrays of complex type only support 'sum'. Defaults to 'sum'.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**reduce\_scatter**(*in\_array*, *out\_array*, *count*, *op*='sum', *stream*=None)

Performs a reduce scatter operation.

#### Parameters

- **in\_array** (`cupy.ndarray`) – array to be sent.
- **out\_array** (`cupy.ndarray`) – array where the result will be stored.
- **count** (`int`) – Number of elements to send to each rank.
- **op** (`str`) – reduction operation, can be one of ('sum', 'prod', 'min', 'max'), arrays of complex type only support 'sum'. Defaults to 'sum'.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**scatter**(*in\_array*, *out\_array*, *root*=0, *stream*=None)

Performs a scatter operation.

**Parameters**

- **in\_array** (`cupy.ndarray`) – array to be sent. Its shape must be *(total\_ranks, ...)*.
- **out\_array** (`cupy.ndarray`) – array where the result will be stored.
- **root** (`int`) – rank that will send the *in\_array* to other ranks.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**send**(*array*, *peer*, *stream*=None)

Performs a send operation.

**Parameters**

- **array** (`cupy.ndarray`) – array to be sent.
- **peer** (`int`) – rank of the process *array* will be sent to.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**send\_recv**(*in\_array*, *out\_array*, *peer*, *stream*=None)

Performs a send and receive operation.

**Parameters**

- **in\_array** (`cupy.ndarray`) – array to be sent.
- **out\_array** (`cupy.ndarray`) – array used to receive data.
- **peer** (`int`) – rank of the process to send *in\_array* and receive *out\_array*.
- **stream** (`cupy.cuda.Stream`, *optional*) – if supported, stream to perform the communication.

**stop**()

**\_\_eq\_\_**(*value*, /)

Return self==value.

**\_\_ne\_\_**(*value*, /)

Return self!=value.

**\_\_lt\_\_**(*value*, /)

Return self<value.

**\_\_le\_\_**(*value*, /)

Return self<=value.

**\_\_gt\_\_**(*value*, /)

Return self>value.

**\_\_ge\_\_**(*value*, /)

Return self>=value.

## 5.8.2 ndarray distributed across devices

<code>distributed_array(array, index_map[, mode])</code>	Creates a distributed array from the given data.
<code>DistributedArray(self, shape, dtype, chunks_map)</code>	Multi-dimensional array distributed across multiple CUDA devices.
<code>make_2d_index_map(i_partitions, ...)</code>	Create an <code>index_map</code> for a 2D matrix with a specified blocking.
<code>matmul(a, b[, out])</code>	Matrix multiplication between distributed arrays.

### cupyx.distributed.array.distributed\_array

`cupyx.distributed.array.distributed_array(array, index_map, mode=None)`

Creates a distributed array from the given data.

This function does not check if all elements of the given array are stored in some of the chunks.

#### Parameters

- **array** (*array\_like*) – `DistributedArray` object, `cupy.ndarray` object or any other object that can be passed to `numpy.array()`.
- **index\_map** (*dict from int to array indices*) – Indices for the chunks that devices with designated IDs own. One device can have multiple chunks, which can be specified as a list of array indices.
- **mode** (*mode object, optional*) – Mode that determines how overlaps of the chunks are interpreted. Defaults to `cupyx.distributed.array.REPLICA`.

#### Return type

`DistributedArray`

See also:

`DistributedArray.mode` for details about modes.

#### Example

```
>>> array = cupy.arange(9).reshape(3, 3)
>>> A = distributed_array(
...     array,
...     {0: [(slice(2), slice(2)), # array[:2, :2]
...          slice(None, None, 2)], # array[:, :2]
...     1: (slice(1, None), 2)}) # array[1:, 2]
```

### cupyx.distributed.array.DistributedArray

`class cupyx.distributed.array.DistributedArray(self, shape, dtype, chunks_map, mode=REPLICA, comms=None)`

Multi-dimensional array distributed across multiple CUDA devices.

This class implements some elementary operations that `cupy.ndarray` provides. The array content is split into chunks, contiguous arrays corresponding to slices of the original array. Note that one device can hold multiple chunks.

This direct constructor is designed for internal calls. Users should create distributed arrays using `distributed_array()`.

#### Parameters

- **shape** (*tuple of ints*) – Shape of created array.
- **dtype** (*dtype\_like*) – Any object that can be interpreted as a numpy data type.
- **chunks\_map** (*dict from int to list of chunks*) – Lists of chunk objects associated with each device.
- **mode** (*mode object, optional*) – Mode that determines how overlaps of the chunks are interpreted. Defaults to `cupyx.distributed.array.REPLICA`.
- **comms** (*optional*) – Communicator objects which a distributed array hold internally. Sharing them with other distributed arrays can save time because their initialization is a costly operation.

#### Return type

*DistributedArray*

See also:

*DistributedArray.mode* for details about modes.

#### Methods

`__getitem__(*args, **kwargs)`

Not supported.

`__setitem__(*args, **kwargs)`

Not supported.

`__len__(*args, **kwargs)`

Not supported.

`__iter__(*args, **kwargs)`

Not supported.

`__copy__(*args, **kwargs)`

Not supported.

`all(*args, **kwargs)`

Not supported.

`all_chunks()`

Return the chunks with all buffered data flushed.

Buffered data are created in situations such as resharding and mode changing.

#### Return type

*dict[int, list[`cupy.ndarray`]]*

`any(*args, **kwargs)`

Not supported.

`argmax(*args, **kwargs)`

Not supported.

**argmin**(\*args, \*\*kwargs)

Not supported.

**argpartition**(\*args, \*\*kwargs)

Not supported.

**argsort**(\*args, \*\*kwargs)

Not supported.

**astype**(\*args, \*\*kwargs)

Not supported.

**change\_mode**(mode)

Return a view or a copy in the given mode.

**Parameters**

**mode** (mode *Object*) – How overlaps of the chunks are interpreted.

**Return type**

[DistributedArray](#)

**See also:**

[DistributedArray.mode](#) for details about modes.

**choose**(\*args, \*\*kwargs)

Not supported.

**clip**(\*args, \*\*kwargs)

Not supported.

**compress**(\*args, \*\*kwargs)

Not supported.

**conj**(self) → [ndarray](#)

**conjugate**(self) → [ndarray](#)

**copy**(\*args, \*\*kwargs)

Not supported.

**cumprod**(\*args, \*\*kwargs)

Not supported.

**cumsum**(\*args, \*\*kwargs)

Not supported.

**diagonal**(\*args, \*\*kwargs)

Not supported.

**dot**(\*args, \*\*kwargs)

Not supported.

**dump**(\*args, \*\*kwargs)

Not supported.

**dumps**(\*args, \*\*kwargs)

Not supported.

**fill**(\*args, \*\*kwargs)

Not supported.

**flatten**(\*args, \*\*kwargs)

Not supported.

**get**(stream=None, order='C', out=None, blocking=True)

Return a copy of the array on the host memory.

**Return type**

*ndarray*

**item**(\*args, \*\*kwargs)

Not supported.

**max**(axis=None, out=None, keepdims=False)

Return the maximum along a given axis.

---

**Note:** Currently, it only supports non-None values for `axis` and the default values for `out` and `keepdims`.

---

**See also:**

*cupy.ndarray.max()*, *numpy.ndarray.max()*

**mean**(\*args, \*\*kwargs)

Not supported.

**min**(axis=None, out=None, keepdims=False)

Return the minimum along a given axis.

---

**Note:** Currently, it only supports non-None values for `axis` and the default values for `out` and `keepdims`.

---

**See also:**

*cupy.ndarray.min()*, *numpy.ndarray.min()*

**nonzero**(\*args, \*\*kwargs)

Not supported.

**partition**(\*args, \*\*kwargs)

Not supported.

**prod**(axis=None, dtype=None, out=None, keepdims=None)

Return the minimum along a given axis.

---

**Note:** Currently, it only supports non-None values for `axis` and the default values for `out` and `keepdims`.

---

**See also:**

*cupy.ndarray.prod()*, *numpy.ndarray.prod()*

**ptp**(\*args, \*\*kwargs)

Not supported.



**put**(\*args, \*\*kwargs)

Not supported.

**ravel**(\*args, \*\*kwargs)

Not supported.

**reduced\_view**(\*args, \*\*kwargs)

Not supported.

**repeat**(\*args, \*\*kwargs)

Not supported.

**reshape**(\*args, \*\*kwargs)

Not supported.

**reshard**(index\_map)

Return a view or a copy having the given index\_map.

Data transfers across devices are done on separate streams created internally. To make them asynchronous, transferred data is buffered and reflected to the chunks when necessary.

#### Parameters

**index\_map** (*dict from int to array indices*) – Indices for the chunks that devices with designated IDs own. The current index\_map of a distributed array can be obtained from [`DistributedArray.index\_map`](#).

#### Return type

[`DistributedArray`](#)

**round**(\*args, \*\*kwargs)

Not supported.

**scatter\_add**(\*args, \*\*kwargs)

Not supported.

**scatter\_max**(\*args, \*\*kwargs)

Not supported.

**scatter\_min**(\*args, \*\*kwargs)

Not supported.

**searchsorted**(\*args, \*\*kwargs)

Not supported.

**set**(\*args, \*\*kwargs)

Not supported.

**sort**(\*args, \*\*kwargs)

Not supported.

**squeeze**(\*args, \*\*kwargs)

Not supported.

**std**(\*args, \*\*kwargs)

Not supported.

**sum**(axis=None, dtype=None, out=None, keepdims=False)

Return the minimum along a given axis.

---

**Note:** Currently, it only supports non-None values for `axis` and the default values for `out` and `keepdims`.

---

**See also:**

[`cupy.ndarray.sum\(\)`](#), [`numpy.ndarray.sum\(\)`](#)

**swapaxes**(\*args, \*\*kwargs)

Not supported.

**take**(\*args, \*\*kwargs)

Not supported.

**toDlpack**(\*args, \*\*kwargs)

Not supported.

**tobytes**(\*args, \*\*kwargs)

Not supported.

**tofile**(\*args, \*\*kwargs)

Not supported.

**tolist**(\*args, \*\*kwargs)

Not supported.

**trace**(\*args, \*\*kwargs)

Not supported.

**transpose**(\*args, \*\*kwargs)

Not supported.

**var**(\*args, \*\*kwargs)

Not supported.

**view**(\*args, \*\*kwargs)

Not supported.

**\_\_eq\_\_**(value, /)

Return self==value.

**\_\_ne\_\_**(value, /)

Return self!=value.

**\_\_lt\_\_**(value, /)

Return self<value.

**\_\_le\_\_**(value, /)

Return self<=value.

**\_\_gt\_\_**(value, /)

Return self>value.

**\_\_ge\_\_**(value, /)

Return self>=value.

**\_\_bool\_\_**()

True if self else False

## Attributes

**T**

Not supported.

**base**

Not supported.

**cstruct**

Not supported.

**data**

Not supported.

**device**

Not supported.

**devices**

A collection of device IDs holding part of the data.

**dtype**

**flags**

Not supported.

**flat**

Not supported.

**imag**

Not supported.

**index\_map**

Indices for the chunks that devices with designated IDs own.

**itemsize**

Size of each element in bytes.

**See also:**

`numpy.ndarray.itemsize`

**mode**

Describe how overlaps of the chunks are interpreted.

In the replica mode, chunks are guaranteed to have identical values on their overlapping segments. In other modes, they are not necessarily identical and represent the original data as their max, sum, etc.

*DistributedArray* currently supports `cupyx.distributed.array.REPLICA`, `cupyx.distributed.array.MIN`, `cupyx.distributed.array.MAX`, `cupyx.distributed.array.SUM`, `cupyx.distributed.array.PROD` modes.

Many operations on distributed arrays including `cupy.ufunc` and `matmul()` involve changing their mode beforehand. These mode conversions are done automatically, so in most cases users do not have to manage modes manually.

### Example

```
>>> A = distributed_array(
...     cupy.arange(6).reshape(2, 3),
...     make_2d_index_map([0, 2], [0, 1, 3],
...                        [{0}, {1, 2}]])
>>> B = distributed_array(
...     cupy.arange(12).reshape(3, 4),
...     make_2d_index_map([0, 1, 3], [0, 2, 4],
...                        [{0}, {0}],
...                        [{1}, {2}]])
>>> C = A @ B
>>> C
array([[20, 23, 26, 29],
       [56, 68, 80, 92]])
>>> C.mode
'sum'
>>> C.all_chunks()
{0: [array([[0, 0],
           [0, 3]]),      # left half
     array([[0, 0],
           [6, 9]])],    # right half
 1: [array([[20, 23],
           [56, 65]])],  # left half
 2: [array([[26, 29],
           [74, 83]])]} # right half
>>> C_replica = C.change_mode('replica')
>>> C_replica.mode
'replica'
>>> C_replica.all_chunks()
{0: [array([[20, 23],
           [56, 68]])],  # left half
     array([[26, 29],
           [80, 92]])],  # right half
 1: [array([[20, 23],
           [56, 68]])],  # left half
 2: [array([[26, 29],
           [80, 92]])]} # right half
```

### nbytes

Total size of all elements in bytes.

It does not count skips between elements.

#### See also:

[numpy.ndarray.nbytes](#)

### ndim

Number of dimensions.

`a.ndim` is equivalent to `len(a.shape)`.

#### See also:

[numpy.ndarray.ndim](#)

**real**

Not supported.

**shape**

Tuple of array dimensions.

Assignment to this property is currently not supported.

**size****strides**

Not supported.

**cupyx.distributed.array.make\_2d\_index\_map**

`cupyx.distributed.array.make_2d_index_map(i_partitions, j_partitions, devices)`

Create an `index_map` for a 2D matrix with a specified blocking.

**Parameters**

- **`i_partitions`** (*list of ints*) – boundaries of blocks on the *i* axis
- **`j_partitions`** (*list of ints*) – boundaries of blocks on the *j* axis
- **`devices`** (*2D list of sets of ints*) – devices owning each block

**Returns****index\_map**

Indices for the chunks that devices with designated IDs are going to own.

**Return type**

dict from int to array indices

**Example**

```
>>> index_map = make_2d_index_map(
...     [0, 2, 4], [0, 3, 5],
...     [[{0}, {1}],
...      [{2}, {0, 1}]]
>>> pprint(index_map)
{0: [(slice(0, 2, None), slice(0, 3, None)),
      (slice(2, 4, None), slice(3, 5, None))],
 1: [(slice(0, 2, None), slice(3, 5, None)),
      (slice(2, 4, None), slice(3, 5, None))],
 2: [(slice(2, 4, None), slice(0, 3, None))]}
```

## cupyx.distributed.array.matmul

`cupyx.distributed.array.matmul(a, b, out=None, **kwargs)`

Matrix multiplication between distributed arrays.

The arguments must have compatible [shape](#) and [index\\_map](#).

This operation converts its operands into the replica mode, and compute their product in the sum mode.

### Parameters

- **a** ([DistributedArray](#)) – Input distributed arrays.
- **b** ([DistributedArray](#)) – Input distributed arrays.
- **out** (*optional*) – A location into which the result is stored. This option is currently not supported.

### Returns

The matrix product of the inputs.

### Return type

[DistributedArray](#)

## Example

```
>>> A = distributed_array(  
...     cupy.arange(6).reshape(2, 3),  
...     make_2d_index_map([0, 2], [0, 1, 3],  
...                        [{0}, {1, 2}]])  
>>> B = distributed_array(  
...     cupy.arange(12).reshape(3, 4),  
...     make_2d_index_map([0, 1, 3], [0, 2, 4],  
...                        [{0}, {0}],  
...                        [{1}, {2}]])  
>>> C = A @ B  
>>> C.mode  
'sum'  
>>> C.all_chunks()  
{0: [array([[0, 0],  
           [0, 3]]),  
     array([[0, 0],  
           [6, 9]])],  
 1: [array([[20, 23],  
           [56, 65]])],  
 2: [array([[26, 29],  
           [74, 83]])]}  
>>> C  
array([[20, 23, 26, 29],  
       [56, 68, 80, 92]])
```

See also:

[numpy.matmul](#)

## 5.9 Environment variables

### 5.9.1 For runtime

Here are the environment variables that CuPy uses at runtime.

#### **CUDA\_PATH**

Path to the directory containing CUDA. The parent of the directory containing `nvcc` is used as default. When `nvcc` is not found, `/usr/local/cuda` is used. See [Working with Custom CUDA Installation](#) for details.

#### **CUPY\_CACHE\_DIR**

Default: `${HOME}/.cupy/kernel_cache`

Path to the directory to store kernel cache. See [Performance Best Practices](#) for details.

#### **CUPY\_CACHE\_SAVE\_CUDA\_SOURCE**

Default: `0`

If set to 1, CUDA source file will be saved along with compiled binary in the cache directory for debug purpose. Note: the source file will not be saved if the compiled binary is already stored in the cache.

#### **CUPY\_CACHE\_IN\_MEMORY**

Default: `0`

If set to 1, [CUPY\\_CACHE\\_DIR](#) and [CUPY\\_CACHE\\_SAVE\\_CUDA\\_SOURCE](#) will be ignored, and the cache is in memory. This environment variable allows reducing disk I/O, but is ignored when `nvcc` is set to be the compiler backend.

#### **CUPY\_DISABLE\_JITIFY\_CACHE**

Default: `0`

If set to 1, headers loaded by Jitify would not be cached on disk (to [CUPY\\_CACHE\\_DIR](#)). The default is to always cache.

#### **CUPY\_DUMP\_CUDA\_SOURCE\_ON\_ERROR**

Default: `0`

If set to 1, when CUDA kernel compilation fails, CuPy dumps CUDA kernel code to standard error.

#### **CUPY\_CUDA\_COMPILE\_WITH\_DEBUG**

Default: `0`

If set to 1, CUDA kernel will be compiled with debug information (`--device-debug` and `--generate-line-info`).

#### **CUPY\_GPU\_MEMORY\_LIMIT**

Default: `0` (unlimited)

The amount of memory that can be allocated for each device. The value can be specified in absolute bytes or fraction (e.g., `"90%"`) of the total memory of each GPU. See [Memory Management](#) for details.

#### **CUPY\_SEED**

Set the seed for random number generators.

#### **CUPY\_EXPERIMENTAL\_SLICE\_COPY**

Default: `0`

If set to 1, the following syntax is enabled:

```
cupy_ndarray[:] = numpy_ndarray
```

## CUPY\_ACCELERATORS

Default: "cub" (In ROCm HIP environment, the default value is "". i.e., no accelerators are used.)

A comma-separated string of backend names (`cub`, `cutensor`, or `cutensornet`) which indicates the acceleration backends used in CuPy operations and its priority (in descending order). By default, all accelerators are disabled on HIP and only CUB is enabled on CUDA.

## CUPY\_TF32

Default: 0

If set to 1, it allows CUDA libraries to use Tensor Cores TF32 compute for 32-bit floating point compute.

## CUPY\_CUDA\_ARRAY\_INTERFACE\_SYNC

Default: 1

This controls CuPy's behavior as a Consumer. If set to 0, a stream synchronization will *not* be performed when a device array provided by an external library that implements the CUDA Array Interface is being consumed by CuPy. For more detail, see the [Synchronization](#) requirement in the CUDA Array Interface v3 documentation.

## CUPY\_CUDA\_ARRAY\_INTERFACE\_EXPORT\_VERSION

Default: 3

This controls CuPy's behavior as a Producer. If set to 2, the CuPy stream on which the data is being operated will not be exported and thus the Consumer (another library) will not perform any stream synchronization. For more detail, see the [Synchronization](#) requirement in the CUDA Array Interface v3 documentation.

## CUPY\_DLPACK\_EXPORT\_VERSION

Default: 0.6

This controls CuPy's DLPack support. Currently, setting a value smaller than 0.6 would disguise managed memory as normal device memory, which enables data exchanges with libraries that have not updated their DLPack support, whereas starting 0.6 CUDA managed memory can be correctly recognized as a valid device type.

## NVCC

Default: nvcc

Define the compiler to use when compiling CUDA source. Note that most CuPy kernels are built with NVRTC; this environment variable is only effective for [RawKernel/RawModule](#) with the nvcc backend or when using cub as the accelerator.

## CUPY\_CUDA\_PER\_THREAD\_DEFAULT\_STREAM

Default: 0

If set to 1, CuPy will use the CUDA per-thread default stream, effectively causing each host thread to automatically execute in its own stream, unless the CUDA default (`null`) stream or a user-created stream is specified. If set to 0 (default), the CUDA default (`null`) stream is used, unless the per-thread default stream (`ptds`) or a user-created stream is specified.

## CUPY\_COMPILE\_WITH\_PTX

Default: 0

By default, CuPy directly compiles kernels into SASS (CUBIN) to support [CUDA Enhanced Compatibility](#). If set to 1, CuPy instead compiles kernels into PTX and lets CUDA Driver assemble SASS from PTX. This option is only effective for CUDA 11.1 or later; CuPy always compiles into PTX on earlier CUDA versions. Also, this option only applies when NVRTC is selected as the compilation backend. NVCC backend always compiles into SASS (CUBIN).



### CUDA Toolkit Environment Variables

In addition to the environment variables listed above, as in any CUDA programs, all of the CUDA environment variables listed in the [CUDA Toolkit Documentation](#) will also be honored.

---

**Note:** When `CUPY_ACCELERATORS` or `NVCC` environment variables are set, g++-6 or later is required as the runtime host compiler. Please refer to [Installing CuPy from Source](#) for the details on how to install g++.

---

## 5.9.2 For installation

These environment variables are used during installation (building CuPy from source).

### CUTENSOR\_PATH

Path to the cuTENSOR root directory that contains `lib` and `include` directories. (experimental)

### CUPY\_INSTALL\_USE\_HIP

Default: 0

If set to 1, CuPy is built for AMD ROCm Platform (experimental). For building the ROCm support, see [Installing Binary Packages](#) for further detail.

### CUPY\_USE\_CUDA\_PYTHON

Default: 0

If set to 1, CuPy is built using [CUDA Python](#).

### CUPY\_NVCC\_GENERATE\_CODE

Build CuPy for a particular CUDA architecture. For example:

```
CUPY_NVCC_GENERATE_CODE="arch=compute_60,code=sm_60"
```

For specifying multiple archs, concatenate the `arch=...` strings with semicolons (;). If `current` is specified, then it will automatically detect the currently installed GPU architectures in build time. When this is not set, the default is to support all architectures.

### CUPY\_NUM\_BUILD\_JOBS

Default: 4

To enable or disable parallel build, sets the number of processes used to build the extensions in parallel.

### CUPY\_NUM\_NVCC\_THREADS

Default: 2

To enable or disable nvcc parallel compilation, sets the number of threads used to compile files using nvcc.

Additionally, the environment variables [CUDA\\_PATH](#) and [NVCC](#) are also respected at build time.

## 5.10 Comparison Table

Here is a list of NumPy / SciPy APIs and its corresponding CuPy implementations.

- in CuPy column denotes that CuPy implementation is not provided yet. We welcome contributions for these functions.

### 5.10.1 NumPy / CuPy APIs

#### Module-Level

NumPy	CuPy
<code>numpy.DataSource</code>	<code>cupy.DataSource</code> ( <i>alias of</i> <code>numpy.DataSource</code> )
<code>numpy.ScalarType</code>	-
<code>numpy.abs</code>	<code>cupy.abs</code>
<code>numpy.absolute</code>	<code>cupy.absolute</code>
<code>numpy.add</code>	<code>cupy.add</code>
<code>numpy.all</code>	<code>cupy.all</code>
<code>numpy.allclose</code>	<code>cupy.allclose</code>
<code>numpy.alltrue</code>	<code>cupy.alltrue</code>
<code>numpy.amax</code>	<code>cupy.amax</code>
<code>numpy.amin</code>	<code>cupy.amin</code>
<code>numpy.angle</code>	<code>cupy.angle</code>
<code>numpy.any</code>	<code>cupy.any</code>
<code>numpy.append</code>	<code>cupy.append</code>
<code>numpy.apply_along_axis</code>	<code>cupy.apply_along_axis</code>
<code>numpy.apply_over_axes</code>	-
<code>numpy.arange</code>	<code>cupy.arange</code>
<code>numpy.arccos</code>	<code>cupy.arccos</code>
<code>numpy.arccosh</code>	<code>cupy.arccosh</code>
<code>numpy.arcsin</code>	<code>cupy.arcsin</code>
<code>numpy.arcsinh</code>	<code>cupy.arcsinh</code>
<code>numpy.arctan</code>	<code>cupy.arctan</code>
<code>numpy.arctan2</code>	<code>cupy.arctan2</code>
<code>numpy.arctanh</code>	<code>cupy.arctanh</code>
<code>numpy.argmax</code>	<code>cupy.argmax</code>
<code>numpy.argmin</code>	<code>cupy.argmin</code>
<code>numpy.argpartition</code>	<code>cupy.argpartition</code>
<code>numpy.argsort</code>	<code>cupy.argsort</code>
<code>numpy.argwhere</code>	<code>cupy.argwhere</code>
<code>numpy.around</code>	<code>cupy.around</code>
<code>numpy.array</code>	<code>cupy.array</code>
<code>numpy.array2string</code>	<code>cupy.array2string</code>
<code>numpy.array_equal</code>	<code>cupy.array_equal</code>
<code>numpy.array_equiv</code>	<code>cupy.array_equiv</code>
<code>numpy.array_repr</code>	<code>cupy.array_repr</code>
<code>numpy.array_split</code>	<code>cupy.array_split</code>
<code>numpy.array_str</code>	<code>cupy.array_str</code>
<code>numpy.asanyarray</code>	<code>cupy.asanyarray</code>
<code>numpy.asarray</code>	<code>cupy.asarray</code>
<code>numpy.asarray_chkfinite</code>	<code>cupy.asarray_chkfinite</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.ascontiguousarray</code>	<code>cupy.ascontiguousarray</code>
<code>numpy.asfarray</code>	<code>cupy.asfarray</code>
<code>numpy.asfortranarray</code>	<code>cupy.asfortranarray</code>
<code>numpy.asmatrix</code>	<code>_1</code>
<code>numpy.atleast_1d</code>	<code>cupy.atleast_1d</code>
<code>numpy.atleast_2d</code>	<code>cupy.atleast_2d</code>
<code>numpy.atleast_3d</code>	<code>cupy.atleast_3d</code>
<code>numpy.average</code>	<code>cupy.average</code>
<code>numpy.bartlett</code>	<code>cupy.bartlett</code>
<code>numpy.base_repr</code>	<code>cupy.base_repr</code>
<code>numpy.binary_repr</code>	<code>cupy.binary_repr</code>
<code>numpy.bincount</code>	<code>cupy.bincount</code>
<code>numpy.bitwise_and</code>	<code>cupy.bitwise_and</code>
<code>numpy.bitwise_not</code>	<code>cupy.bitwise_not</code>
<code>numpy.bitwise_or</code>	<code>cupy.bitwise_or</code>
<code>numpy.bitwise_xor</code>	<code>cupy.bitwise_xor</code>
<code>numpy.blackman</code>	<code>cupy.blackman</code>
<code>numpy.block</code>	<code>-</code>
<code>numpy.bmat</code>	<code>_Page 864, 1</code>
<code>numpy.bool_</code>	<code>cupy.bool_ (alias of numpy.bool_)</code>
<code>numpy.broadcast</code>	<code>cupy.broadcast</code>
<code>numpy.broadcast_arrays</code>	<code>cupy.broadcast_arrays</code>
<code>numpy.broadcast_shapes</code>	<code>cupy.broadcast_shapes (alias of numpy.broadcast_shapes)</code>
<code>numpy.broadcast_to</code>	<code>cupy.broadcast_to</code>
<code>numpy.busday_count</code>	<code>_2</code>
<code>numpy.busday_offset</code>	<code>_Page 864, 2</code>
<code>numpy.busdaycalendar</code>	<code>_Page 864, 2</code>
<code>numpy.byte</code>	<code>cupy.byte (alias of numpy.byte)</code>
<code>numpy.byte_bounds</code>	<code>cupy.byte_bounds</code>
<code>numpy.bytes_</code>	<code>_3</code>
<code>numpy.c_</code>	<code>cupy.c_</code>
<code>numpy.can_cast</code>	<code>cupy.can_cast</code>
<code>numpy.cast</code>	<code>-</code>
<code>numpy.cbrt</code>	<code>cupy.cbrt</code>
<code>numpy.cdouble</code>	<code>cupy.cdouble (alias of numpy.cdouble)</code>
<code>numpy.ceil</code>	<code>cupy.ceil</code>
<code>numpy.cfloat</code>	<code>cupy.cfloat (alias of numpy.cfloat)</code>
<code>numpy.character</code>	<code>_Page 864, 3</code>
<code>numpy.chararray</code>	<code>_Page 864, 3</code>
<code>numpy.choose</code>	<code>cupy.choose</code>
<code>numpy.clip</code>	<code>cupy.clip</code>
<code>numpy.clongdouble</code>	<code>-</code>
<code>numpy.clongfloat</code>	<code>-</code>
<code>numpy.column_stack</code>	<code>cupy.column_stack</code>
<code>numpy.common_type</code>	<code>cupy.common_type</code>
<code>numpy.compare_chararrays</code>	<code>_Page 864, 3</code>
<code>numpy.complex128</code>	<code>cupy.complex128 (alias of numpy.complex128)</code>
<code>numpy.complex256</code>	<code>-</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.complex64</code>	<code>cupy.complex64</code> ( <i>alias of</i> <code>numpy.complex64</code> )
<code>numpy.complex_</code>	<code>cupy.complex_</code> ( <i>alias of</i> <code>numpy.complex_</code> )
<code>numpy.complexfloating</code>	<code>cupy.complexfloating</code> ( <i>alias of</i> <code>numpy.complexfloating</code> )
<code>numpy.compress</code>	<code>cupy.compress</code>
<code>numpy.concatenate</code>	<code>cupy.concatenate</code>
<code>numpy.conj</code>	<code>cupy.conj</code>
<code>numpy.conjugate</code>	<code>cupy.conjugate</code>
<code>numpy.convolve</code>	<code>cupy.convolve</code>
<code>numpy.copy</code>	<code>cupy.copy</code>
<code>numpy.copysign</code>	<code>cupy.copysign</code>
<code>numpy.copyto</code>	<code>cupy.copyto</code>
<code>numpy.corrccoef</code>	<code>cupy.corrccoef</code>
<code>numpy.correlate</code>	<code>cupy.correlate</code>
<code>numpy.cos</code>	<code>cupy.cos</code>
<code>numpy.cosh</code>	<code>cupy.cosh</code>
<code>numpy.count_nonzero</code>	<code>cupy.count_nonzero</code>
<code>numpy.cov</code>	<code>cupy.cov</code>
<code>numpy.cross</code>	<code>cupy.cross</code>
<code>numpy.csingle</code>	<code>cupy.csingle</code> ( <i>alias of</i> <code>numpy.csingle</code> )
<code>numpy.cumprod</code>	<code>cupy.cumprod</code>
<code>numpy.cumproduct</code>	<code>cupy.cumproduct</code>
<code>numpy.cumsum</code>	<code>cupy.cumsum</code>
<code>numpy.datetime64</code>	<code>_Page 864, 2</code>
<code>numpy.datetime_as_string</code>	<code>_Page 864, 2</code>
<code>numpy.datetime_data</code>	<code>_Page 864, 2</code>
<code>numpy.deg2rad</code>	<code>cupy.deg2rad</code>
<code>numpy.degrees</code>	<code>cupy.degrees</code>
<code>numpy.delete</code>	<code>cupy.delete</code>
<code>numpy.deprecate</code>	-
<code>numpy.deprecate_with_doc</code>	-
<code>numpy.diag</code>	<code>cupy.diag</code>
<code>numpy.diag_indices</code>	<code>cupy.diag_indices</code>
<code>numpy.diag_indices_from</code>	<code>cupy.diag_indices_from</code>
<code>numpy.diagflat</code>	<code>cupy.diagflat</code>
<code>numpy.diagonal</code>	<code>cupy.diagonal</code>
<code>numpy.diff</code>	<code>cupy.diff</code>
<code>numpy.digitize</code>	<code>cupy.digitize</code>
<code>numpy.disp</code>	<code>cupy.disp</code> ( <i>alias of</i> <code>numpy.disp</code> )
<code>numpy.divide</code>	<code>cupy.divide</code>
<code>numpy.divmod</code>	<code>cupy.divmod</code>
<code>numpy.dot</code>	<code>cupy.dot</code>
<code>numpy.double</code>	<code>cupy.double</code> ( <i>alias of</i> <code>numpy.double</code> )
<code>numpy.dsplitt</code>	<code>cupy.dsplitt</code>
<code>numpy.dstack</code>	<code>cupy.dstack</code>
<code>numpy.dtype</code>	<code>cupy.dtype</code> ( <i>alias of</i> <code>numpy.dtype</code> )
<code>numpy.ediff1d</code>	<code>cupy.ediff1d</code>
<code>numpy.einsum</code>	<code>cupy.einsum</code>
<code>numpy.einsum_path</code>	-

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.empty</code>	<code>cupy.empty</code>
<code>numpy.empty_like</code>	<code>cupy.empty_like</code>
<code>numpy.equal</code>	<code>cupy.equal</code>
<code>numpy.errstate</code>	<code>_4</code>
<code>numpy.exp</code>	<code>cupy.exp</code>
<code>numpy.exp2</code>	<code>cupy.exp2</code>
<code>numpy.expand_dims</code>	<code>cupy.expand_dims</code>
<code>numpy.expm1</code>	<code>cupy.expm1</code>
<code>numpy.extract</code>	<code>cupy.extract</code>
<code>numpy.eye</code>	<code>cupy.eye</code>
<code>numpy.fabs</code>	<code>cupy.fabs</code>
<code>numpy.fill_diagonal</code>	<code>cupy.fill_diagonal</code>
<code>numpy.find_common_type</code>	<code>cupy.find_common_type</code> ( <i>alias of</i> <code>numpy.find_common_type</code> )
<code>numpy.finfo</code>	<code>cupy.finfo</code> ( <i>alias of</i> <code>numpy.finfo</code> )
<code>numpy.fix</code>	<code>cupy.fix</code>
<code>numpy.flatiter</code>	<code>cupy.flatiter</code>
<code>numpy.flatnonzero</code>	<code>cupy.flatnonzero</code>
<code>numpy.flexible</code>	<code>_Page 864, 3</code>
<code>numpy.flip</code>	<code>cupy.flip</code>
<code>numpy.fliplr</code>	<code>cupy.fliplr</code>
<code>numpy.flipud</code>	<code>cupy.flipud</code>
<code>numpy.float128</code>	-
<code>numpy.float16</code>	<code>cupy.float16</code> ( <i>alias of</i> <code>numpy.float16</code> )
<code>numpy.float32</code>	<code>cupy.float32</code> ( <i>alias of</i> <code>numpy.float32</code> )
<code>numpy.float64</code>	<code>cupy.float64</code> ( <i>alias of</i> <code>numpy.float64</code> )
<code>numpy.float_</code>	<code>cupy.float_</code> ( <i>alias of</i> <code>numpy.float_</code> )
<code>numpy.float_power</code>	<code>cupy.float_power</code>
<code>numpy.floating</code>	<code>cupy.floating</code> ( <i>alias of</i> <code>numpy.floating</code> )
<code>numpy.floor</code>	<code>cupy.floor</code>
<code>numpy.floor_divide</code>	<code>cupy.floor_divide</code>
<code>numpy.fmax</code>	<code>cupy.fmax</code>
<code>numpy.fmin</code>	<code>cupy.fmin</code>
<code>numpy.fmod</code>	<code>cupy.fmod</code>
<code>numpy.format_float_positional</code>	<code>cupy.format_float_positional</code>
<code>numpy.format_float_scientific</code>	<code>cupy.format_float_scientific</code>
<code>numpy.format_parser</code>	<code>cupy.format_parser</code> ( <i>alias of</i> <code>numpy.format_parser</code> )
<code>numpy.frexp</code>	<code>cupy.frexp</code>
<code>numpy.from_dlpack</code>	<code>cupy.from_dlpack</code>
<code>numpy.frombuffer</code>	<code>cupy.frombuffer</code>
<code>numpy.fromfile</code>	<code>cupy.fromfile</code>
<code>numpy.fromfunction</code>	<code>cupy.fromfunction</code>
<code>numpy.fromiter</code>	<code>cupy.fromiter</code>
<code>numpy.frompyfunc</code>	-
<code>numpy.fromregex</code>	<code>_5</code>
<code>numpy.fromstring</code>	<code>cupy.fromstring</code>
<code>numpy.full</code>	<code>cupy.full</code>
<code>numpy.full_like</code>	<code>cupy.full_like</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.gcd</code>	<code>cupy.gcd</code>
<code>numpy.generic</code>	<code>cupy.generic</code> ( <i>alias of</i> <code>numpy.generic</code> )
<code>numpy.genfromtxt</code>	<code>cupy.genfromtxt</code>
<code>numpy.geomspace</code>	-
<code>numpy.get_array_wrap</code>	<code>cupy.get_array_wrap</code> ( <i>alias of</i> <code>numpy.get_array_wrap</code> )
<code>numpy.get_include</code>	-
<code>numpy.get_printoptions</code>	<code>cupy.get_printoptions</code> ( <i>alias of</i> <code>numpy.get_printoptions</code> )
<code>numpy.getbufsize</code>	-
<code>numpy.geterr</code>	<code>_Page 864, 4</code>
<code>numpy.geterrcall</code>	<code>_Page 864, 4</code>
<code>numpy.geterrobj</code>	<code>_Page 864, 4</code>
<code>numpy.gradient</code>	<code>cupy.gradient</code>
<code>numpy.greater</code>	<code>cupy.greater</code>
<code>numpy.greater_equal</code>	<code>cupy.greater_equal</code>
<code>numpy.half</code>	<code>cupy.half</code> ( <i>alias of</i> <code>numpy.half</code> )
<code>numpy.hamming</code>	<code>cupy.hamming</code>
<code>numpy.hanning</code>	<code>cupy.hanning</code>
<code>numpy.heaviside</code>	<code>cupy.heaviside</code>
<code>numpy.histogram</code>	<code>cupy.histogram</code>
<code>numpy.histogram2d</code>	<code>cupy.histogram2d</code>
<code>numpy.histogram_bin_edges</code>	-
<code>numpy.histogramdd</code>	<code>cupy.histogramdd</code>
<code>numpy.hsplit</code>	<code>cupy.hsplit</code>
<code>numpy.hstack</code>	<code>cupy.hstack</code>
<code>numpy.hypot</code>	<code>cupy.hypot</code>
<code>numpy.i0</code>	<code>cupy.i0</code>
<code>numpy.identity</code>	<code>cupy.identity</code>
<code>numpy.iinfo</code>	<code>cupy.iinfo</code> ( <i>alias of</i> <code>numpy.iinfo</code> )
<code>numpy.imag</code>	<code>cupy.imag</code>
<code>numpy.in1d</code>	<code>cupy.in1d</code>
<code>numpy.index_exp</code>	<code>cupy.index_exp</code> ( <i>alias of</i> <code>numpy.index_exp</code> )
<code>numpy.indices</code>	<code>cupy.indices</code>
<code>numpy.inexact</code>	<code>cupy.inexact</code> ( <i>alias of</i> <code>numpy.inexact</code> )
<code>numpy.info</code>	-
<code>numpy.inner</code>	<code>cupy.inner</code>
<code>numpy.insert</code>	-
<code>numpy.int16</code>	<code>cupy.int16</code> ( <i>alias of</i> <code>numpy.int16</code> )
<code>numpy.int32</code>	<code>cupy.int32</code> ( <i>alias of</i> <code>numpy.int32</code> )
<code>numpy.int64</code>	<code>cupy.int64</code> ( <i>alias of</i> <code>numpy.int64</code> )
<code>numpy.int8</code>	<code>cupy.int8</code> ( <i>alias of</i> <code>numpy.int8</code> )
<code>numpy.int_</code>	<code>cupy.int_</code> ( <i>alias of</i> <code>numpy.int_</code> )
<code>numpy.intc</code>	<code>cupy.intc</code> ( <i>alias of</i> <code>numpy.intc</code> )
<code>numpy.integer</code>	<code>cupy.integer</code> ( <i>alias of</i> <code>numpy.integer</code> )
<code>numpy.interp</code>	<code>cupy.interp</code>
<code>numpy.intersect1d</code>	<code>cupy.intersect1d</code>
<code>numpy.intp</code>	<code>cupy.intp</code> ( <i>alias of</i> <code>numpy.intp</code> )
<code>numpy.invert</code>	<code>cupy.invert</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.is_busday</code>	<code>_Page 864, 2</code>
<code>numpy.isclose</code>	<code>cupy.isclose</code>
<code>numpy.iscomplex</code>	<code>cupy.iscomplex</code>
<code>numpy.iscomplexobj</code>	<code>cupy.iscomplexobj</code>
<code>numpy.isfinite</code>	<code>cupy.isfinite</code>
<code>numpy.isfortran</code>	<code>cupy.isfortran</code>
<code>numpy.isin</code>	<code>cupy.isin</code>
<code>numpy.isinf</code>	<code>cupy.isinf</code>
<code>numpy.isnan</code>	<code>cupy.isnan</code>
<code>numpy.isnat</code>	<code>_Page 864, 2</code>
<code>numpy.isneginf</code>	<code>cupy.isneginf</code>
<code>numpy.isposinf</code>	<code>cupy.isposinf</code>
<code>numpy.isreal</code>	<code>cupy.isreal</code>
<code>numpy.isrealobj</code>	<code>cupy.isrealobj</code>
<code>numpy.isscalar</code>	<code>cupy.isscalar</code>
<code>numpy.issctype</code>	<code>cupy.issctype</code> ( <i>alias of</i> <code>numpy.issctype</code> )
<code>numpy.issubclass_</code>	<code>cupy.issubclass_</code> ( <i>alias of</i> <code>numpy.issubclass_</code> )
<code>numpy.issubdtype</code>	<code>cupy.issubdtype</code> ( <i>alias of</i> <code>numpy.issubdtype</code> )
<code>numpy.issubscdtype</code>	<code>cupy.issubscdtype</code> ( <i>alias of</i> <code>numpy.issubscdtype</code> )
<code>numpy.iterable</code>	<code>cupy.iterable</code> ( <i>alias of</i> <code>numpy.iterable</code> )
<code>numpy.ix_</code>	<code>cupy.ix_</code>
<code>numpy.kaiser</code>	<code>cupy.kaiser</code>
<code>numpy.kron</code>	<code>cupy.kron</code>
<code>numpy.lcm</code>	<code>cupy.lcm</code>
<code>numpy.ldexp</code>	<code>cupy.ldexp</code>
<code>numpy.left_shift</code>	<code>cupy.left_shift</code>
<code>numpy.less</code>	<code>cupy.less</code>
<code>numpy.less_equal</code>	<code>cupy.less_equal</code>
<code>numpy.lexsort</code>	<code>cupy.lexsort</code>
<code>numpy.linspace</code>	<code>cupy.linspace</code>
<code>numpy.load</code>	<code>cupy.load</code>
<code>numpy.loadtxt</code>	<code>cupy.loadtxt</code>
<code>numpy.log</code>	<code>cupy.log</code>
<code>numpy.log10</code>	<code>cupy.log10</code>
<code>numpy.log1p</code>	<code>cupy.log1p</code>
<code>numpy.log2</code>	<code>cupy.log2</code>
<code>numpy.logaddexp</code>	<code>cupy.logaddexp</code>
<code>numpy.logaddexp2</code>	<code>cupy.logaddexp2</code>
<code>numpy.logical_and</code>	<code>cupy.logical_and</code>
<code>numpy.logical_not</code>	<code>cupy.logical_not</code>
<code>numpy.logical_or</code>	<code>cupy.logical_or</code>
<code>numpy.logical_xor</code>	<code>cupy.logical_xor</code>
<code>numpy.logspace</code>	<code>cupy.logspace</code>
<code>numpy.longcomplex</code>	-
<code>numpy.longdouble</code>	-
<code>numpy.longfloat</code>	<code>cupy.longfloat</code> ( <i>alias of</i> <code>numpy.longfloat</code> )
<code>numpy.longlong</code>	<code>cupy.longlong</code> ( <i>alias of</i> <code>numpy.longlong</code> )
<code>numpy.lookfor</code>	-
<code>numpy.mask_indices</code>	<code>cupy.mask_indices</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.mat</code>	<code>_Page 864, 1</code>
<code>numpy.matmul</code>	<code>cupy.matmul</code>
<code>numpy.matrix</code>	<code>_Page 864, 1</code>
<code>numpy.max</code>	<code>cupy.max</code>
<code>numpy.maximum</code>	<code>cupy.maximum</code>
<code>numpy.maximum_sctype</code>	-
<code>numpy.may_share_memory</code>	<code>cupy.may_share_memory</code>
<code>numpy.mean</code>	<code>cupy.mean</code>
<code>numpy.median</code>	<code>cupy.median</code>
<code>numpy.memmap</code>	-
<code>numpy.meshgrid</code>	<code>cupy.meshgrid</code>
<code>numpy.mgrid</code>	<code>cupy.mgrid</code>
<code>numpy.min</code>	<code>cupy.min</code>
<code>numpy.min_scalar_type</code>	<code>cupy.min_scalar_type</code>
<code>numpy.minimum</code>	<code>cupy.minimum</code>
<code>numpy.mintypecode</code>	<code>cupy.mintypecode</code> ( <i>alias of</i> <code>numpy.mintypecode</code> )
<code>numpy.mod</code>	<code>cupy.mod</code>
<code>numpy.modf</code>	<code>cupy.modf</code>
<code>numpy.moveaxis</code>	<code>cupy.moveaxis</code>
<code>numpy.msort</code>	<code>cupy.msort</code>
<code>numpy.multiply</code>	<code>cupy.multiply</code>
<code>numpy.nan_to_num</code>	<code>cupy.nan_to_num</code>
<code>numpy.nanargmax</code>	<code>cupy.nanargmax</code>
<code>numpy.nanargmin</code>	<code>cupy.nanargmin</code>
<code>numpy.nancumprod</code>	<code>cupy.nancumprod</code>
<code>numpy.nancumsum</code>	<code>cupy.nancumsum</code>
<code>numpy.nanmax</code>	<code>cupy.nanmax</code>
<code>numpy.nanmean</code>	<code>cupy.nanmean</code>
<code>numpy.nanmedian</code>	<code>cupy.nanmedian</code>
<code>numpy.nanmin</code>	<code>cupy.nanmin</code>
<code>numpy.nanpercentile</code>	-
<code>numpy.nanprod</code>	<code>cupy.nanprod</code>
<code>numpy.nanquantile</code>	-
<code>numpy.nanstd</code>	<code>cupy.nanstd</code>
<code>numpy.nansum</code>	<code>cupy.nansum</code>
<code>numpy.nanvar</code>	<code>cupy.nanvar</code>
<code>numpy.nbytes</code>	-
<code>numpy.ndarray</code>	<code>cupy.ndarray</code>
<code>numpy.ndenumerate</code>	-
<code>numpy.ndim</code>	<code>cupy.ndim</code>
<code>numpy.ndindex</code>	<code>cupy.ndindex</code> ( <i>alias of</i> <code>numpy.ndindex</code> )
<code>numpy.nditer</code>	-
<code>numpy.negative</code>	<code>cupy.negative</code>
<code>numpy.nested_iters</code>	-
<code>numpy.newaxis</code>	<code>cupy.newaxis</code> ( <i>alias of</i> <code>numpy.newaxis</code> )
<code>numpy.nextafter</code>	<code>cupy.nextafter</code>
<code>numpy.nonzero</code>	<code>cupy.nonzero</code>
<code>numpy.not_equal</code>	<code>cupy.not_equal</code>
<code>numpy.number</code>	<code>cupy.number</code> ( <i>alias of</i> <code>numpy.number</code> )

continues on next page



Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.obj2sctype</code>	<code>cupy.obj2sctype</code> ( <i>alias of</i> <code>numpy.obj2sctype</code> )
<code>numpy.object_</code>	<code>_Page 864, 3</code>
<code>numpy.ogrid</code>	<code>cupy.ogrid</code>
<code>numpy.ones</code>	<code>cupy.ones</code>
<code>numpy.ones_like</code>	<code>cupy.ones_like</code>
<code>numpy.outer</code>	<code>cupy.outer</code>
<code>numpy.packbits</code>	<code>cupy.packbits</code>
<code>numpy.pad</code>	<code>cupy.pad</code>
<code>numpy.partition</code>	<code>cupy.partition</code>
<code>numpy.percentile</code>	<code>cupy.percentile</code>
<code>numpy.piecewise</code>	<code>cupy.piecewise</code>
<code>numpy.place</code>	<code>cupy.place</code>
<code>numpy.poly</code>	<code>cupy.poly</code> <sup>6</sup>
<code>numpy.poly1d</code>	<code>cupy.poly1d</code>
<code>numpy.polyadd</code>	<code>cupy.polyadd</code>
<code>numpy.polyder</code>	<code>_Page 864, 6</code>
<code>numpy.polydiv</code>	<code>_Page 864, 6</code>
<code>numpy.polyfit</code>	<code>cupy.polyfit</code>
<code>numpy.polyint</code>	<code>_Page 864, 6</code>
<code>numpy.polymul</code>	<code>cupy.polymul</code>
<code>numpy.polysub</code>	<code>cupy.polysub</code>
<code>numpy.polyval</code>	<code>cupy.polyval</code>
<code>numpy.positive</code>	<code>cupy.positive</code>
<code>numpy.power</code>	<code>cupy.power</code>
<code>numpy.printoptions</code>	<code>cupy.printoptions</code> ( <i>alias of</i> <code>numpy.printoptions</code> )
<code>numpy.prod</code>	<code>cupy.prod</code>
<code>numpy.product</code>	<code>cupy.product</code>
<code>numpy.promote_types</code>	<code>cupy.promote_types</code> ( <i>alias of</i> <code>numpy.promote_types</code> )
<code>numpy.ptp</code>	<code>cupy.ptp</code>
<code>numpy.put</code>	<code>cupy.put</code>
<code>numpy.put_along_axis</code>	-
<code>numpy.putmask</code>	<code>cupy.putmask</code>
<code>numpy.quantile</code>	<code>cupy.quantile</code>
<code>numpy.r_</code>	<code>cupy.r_</code>
<code>numpy.rad2deg</code>	<code>cupy.rad2deg</code>
<code>numpy.radians</code>	<code>cupy.radians</code>
<code>numpy.ravel</code>	<code>cupy.ravel</code>
<code>numpy.ravel_multi_index</code>	<code>cupy.ravel_multi_index</code>
<code>numpy.real</code>	<code>cupy.real</code>
<code>numpy.real_if_close</code>	<code>cupy.real_if_close</code>
<code>numpy.recarray</code>	<code>_Page 864, 5</code>
<code>numpy.recfromcsv</code>	<code>_Page 864, 5</code>
<code>numpy.recfromtxt</code>	<code>_Page 864, 5</code>
<code>numpy.reciprocal</code>	<code>cupy.reciprocal</code>
<code>numpy.record</code>	<code>_Page 864, 5</code>
<code>numpy.remainder</code>	<code>cupy.remainder</code>
<code>numpy.repeat</code>	<code>cupy.repeat</code>

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.require</code>	<code>cupy.require</code>
<code>numpy.reshape</code>	<code>cupy.reshape</code>
<code>numpy.resize</code>	<code>cupy.resize</code>
<code>numpy.result_type</code>	<code>cupy.result_type</code>
<code>numpy.right_shift</code>	<code>cupy.right_shift</code>
<code>numpy rint</code>	<code>cupy.rint</code>
<code>numpy.roll</code>	<code>cupy.roll</code>
<code>numpy.rollaxis</code>	<code>cupy.rollaxis</code>
<code>numpy.roots</code>	<code>cupy.roots</code>
<code>numpy.rot90</code>	<code>cupy.rot90</code>
<code>numpy.round</code>	<code>cupy.round</code>
<code>numpy.round_</code>	<code>cupy.round_</code>
<code>numpy.row_stack</code>	<code>cupy.row_stack</code>
<code>numpy.s_</code>	<code>cupy.s_</code> ( <i>alias of</i> <code>numpy.s_</code> )
<code>numpy.safe_eval</code>	<code>cupy.safe_eval</code> ( <i>alias of</i> <code>numpy.safe_eval</code> )
<code>numpy.save</code>	<code>cupy.save</code>
<code>numpy.savetxt</code>	<code>cupy.savetxt</code>
<code>numpy.savez</code>	<code>cupy.savez</code>
<code>numpy.savez_compressed</code>	<code>cupy.savez_compressed</code>
<code>numpy.sctype2char</code>	<code>cupy.sctype2char</code> ( <i>alias of</i> <code>numpy.sctype2char</code> )
<code>numpy.sctypeDict</code>	-
<code>numpy.sctypes</code>	-
<code>numpy.searchsorted</code>	<code>cupy.searchsorted</code>
<code>numpy.select</code>	<code>cupy.select</code>
<code>numpy.set_numeric_ops</code>	<code>_1</code>
<code>numpy.set_printoptions</code>	<code>cupy.set_printoptions</code> ( <i>alias of</i> <code>numpy.set_printoptions</code> )
<code>numpy.set_string_function</code>	<code>cupy.set_string_function</code> ( <i>alias of</i> <code>numpy.set_string_function</code> )
<code>numpy.setbufsize</code>	-
<code>numpy.setdiffld</code>	<code>cupy.setdiffld</code>
<code>numpy.seterr</code>	<code>_Page 864, 4</code>
<code>numpy.seterrcall</code>	<code>_Page 864, 4</code>
<code>numpy.seterrobj</code>	<code>_Page 864, 4</code>
<code>numpy.setxor1d</code>	<code>cupy.setxor1d</code>
<code>numpy.shape</code>	<code>cupy.shape</code>
<code>numpy.shares_memory</code>	<code>cupy.shares_memory</code>
<code>numpy.short</code>	<code>cupy.short</code> ( <i>alias of</i> <code>numpy.short</code> )
<code>numpy.show_config</code>	<code>cupy.show_config</code>
<code>numpy.show_runtime</code>	-
<code>numpy.sign</code>	<code>cupy.sign</code>
<code>numpy.signbit</code>	<code>cupy.signbit</code>
<code>numpy.signedinteger</code>	<code>cupy.signedinteger</code> ( <i>alias of</i> <code>numpy.signedinteger</code> )
<code>numpy.sin</code>	<code>cupy.sin</code>
<code>numpy.sinc</code>	<code>cupy.sinc</code>
<code>numpy.single</code>	<code>cupy.single</code> ( <i>alias of</i> <code>numpy.single</code> )
<code>numpy.singlecomplex</code>	<code>cupy.singlecomplex</code> ( <i>alias of</i> <code>numpy.singlecomplex</code> )

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.sinh</code>	<code>cupy.sinh</code>
<code>numpy.size</code>	<code>cupy.size</code>
<code>numpy.sometrue</code>	<code>cupy.sometrue</code>
<code>numpy.sort</code>	<code>cupy.sort</code>
<code>numpy.sort_complex</code>	<code>cupy.sort_complex</code>
<code>numpy.source</code>	-
<code>numpy.spacing</code>	-
<code>numpy.split</code>	<code>cupy.split</code>
<code>numpy.sqrt</code>	<code>cupy.sqrt</code>
<code>numpy.square</code>	<code>cupy.square</code>
<code>numpy.squeeze</code>	<code>cupy.squeeze</code>
<code>numpy.stack</code>	<code>cupy.stack</code>
<code>numpy.std</code>	<code>cupy.std</code>
<code>numpy.str_</code>	<small>_Page 864, 3</small>
<code>numpy.string_</code>	<small>_Page 864, 3</small>
<code>numpy.subtract</code>	<code>cupy.subtract</code>
<code>numpy.sum</code>	<code>cupy.sum</code>
<code>numpy.swapaxes</code>	<code>cupy.swapaxes</code>
<code>numpy.take</code>	<code>cupy.take</code>
<code>numpy.take_along_axis</code>	<code>cupy.take_along_axis</code>
<code>numpy.tan</code>	<code>cupy.tan</code>
<code>numpy.tanh</code>	<code>cupy.tanh</code>
<code>numpy.tensordot</code>	<code>cupy.tensordot</code>
<code>numpy.tile</code>	<code>cupy.tile</code>
<code>numpy.timedelta64</code>	<small>_Page 864, 2</small>
<code>numpy.trace</code>	<code>cupy.trace</code>
<code>numpy.transpose</code>	<code>cupy.transpose</code>
<code>numpy.trapz</code>	<code>cupy.trapz</code>
<code>numpy.tri</code>	<code>cupy.tri</code>
<code>numpy.tril</code>	<code>cupy.tril</code>
<code>numpy.tril_indices</code>	<code>cupy.tril_indices</code>
<code>numpy.tril_indices_from</code>	<code>cupy.tril_indices_from</code>
<code>numpy.trim_zeros</code>	<code>cupy.trim_zeros</code>
<code>numpy.triu</code>	<code>cupy.triu</code>
<code>numpy.triu_indices</code>	<code>cupy.triu_indices</code>
<code>numpy.triu_indices_from</code>	<code>cupy.triu_indices_from</code>
<code>numpy.true_divide</code>	<code>cupy.true_divide</code>
<code>numpy.trunc</code>	<code>cupy.trunc</code>
<code>numpy.typecodes</code>	-
<code>numpy.typename</code>	<code>cupy.typename</code> ( <i>alias of</i> <code>numpy.typename</code> )
<code>numpy.ubyte</code>	<code>cupy.ubyte</code> ( <i>alias of</i> <code>numpy.ubyte</code> )
<code>numpy.ufunc</code>	<code>cupy.ufunc</code>
<code>numpy.uint</code>	<code>cupy.uint</code> ( <i>alias of</i> <code>numpy.uint</code> )
<code>numpy.uint16</code>	<code>cupy.uint16</code> ( <i>alias of</i> <code>numpy.uint16</code> )
<code>numpy.uint32</code>	<code>cupy.uint32</code> ( <i>alias of</i> <code>numpy.uint32</code> )
<code>numpy.uint64</code>	<code>cupy.uint64</code> ( <i>alias of</i> <code>numpy.uint64</code> )
<code>numpy.uint8</code>	<code>cupy.uint8</code> ( <i>alias of</i> <code>numpy.uint8</code> )
<code>numpy.uintc</code>	<code>cupy.uintc</code> ( <i>alias of</i> <code>numpy.uintc</code> )
<code>numpy.uintp</code>	<code>cupy.uintp</code> ( <i>alias of</i> <code>numpy.uintp</code> )

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
<code>numpy.ulonglong</code>	<code>cupy.ulonglong</code> ( <i>alias of</i> <code>numpy.ulonglong</code> )
<code>numpy.unicode_</code>	<code>_Page 864, 3</code>
<code>numpy.union1d</code>	<code>cupy.union1d</code>
<code>numpy.unique</code>	<code>cupy.unique</code>
<code>numpy.unpackbits</code>	<code>cupy.unpackbits</code>
<code>numpy.unravel_index</code>	<code>cupy.unravel_index</code>
<code>numpy.unsignedinteger</code>	<code>cupy.unsignedinteger</code> ( <i>alias of</i> <code>numpy.unsignedinteger</code> )
<code>numpy.unwrap</code>	<code>cupy.unwrap</code>
<code>numpy.ushort</code>	<code>cupy.ushort</code> ( <i>alias of</i> <code>numpy.ushort</code> )
<code>numpy.vander</code>	<code>cupy.vander</code>
<code>numpy.var</code>	<code>cupy.var</code>
<code>numpy.vdot</code>	<code>cupy.vdot</code>
<code>numpy.vectorize</code>	<code>cupy.vectorize</code>
<code>numpy.void</code>	<code>_3</code>
<code>numpy.vsplit</code>	<code>cupy.vsplit</code>
<code>numpy.vstack</code>	<code>cupy.vstack</code>
<code>numpy.where</code>	<code>cupy.where</code>
<code>numpy.who</code>	<code>cupy.who</code>
<code>numpy.zeros</code>	<code>cupy.zeros</code>
<code>numpy.zeros_like</code>	<code>cupy.zeros_like</code>

## Multi-Dimensional Array

NumPy	CuPy
<code>numpy.ndarray.T</code>	<code>cupy.ndarray.T</code>
<code>numpy.ndarray.all</code>	<code>cupy.ndarray.all</code>
<code>numpy.ndarray.any</code>	<code>cupy.ndarray.any</code>
<code>numpy.ndarray.argmax</code>	<code>cupy.ndarray.argmax</code>
<code>numpy.ndarray.argmin</code>	<code>cupy.ndarray.argmin</code>
<code>numpy.ndarray.argpartition</code>	<code>cupy.ndarray.argpartition</code>
<code>numpy.ndarray.argsort</code>	<code>cupy.ndarray.argsort</code>
<code>numpy.ndarray.astype</code>	<code>cupy.ndarray.astype</code>
<code>numpy.ndarray.base</code>	<code>cupy.ndarray.base</code>
<code>numpy.ndarray.byteswap</code>	<code>_8</code>
<code>numpy.ndarray.choose</code>	<code>cupy.ndarray.choose</code>
<code>numpy.ndarray.clip</code>	<code>cupy.ndarray.clip</code>
<code>numpy.ndarray.compress</code>	<code>cupy.ndarray.compress</code>
<code>numpy.ndarray.conj</code>	<code>cupy.ndarray.conj</code>
<code>numpy.ndarray.conjugate</code>	<code>cupy.ndarray.conjugate</code>
<code>numpy.ndarray.copy</code>	<code>cupy.ndarray.copy</code>

continues on next page

<sup>1</sup> Use of `numpy.matrix` is discouraged in NumPy and thus we have no plan to add it to CuPy.

<sup>2</sup> `datetime64` and `timedelta64` dtypes are currently unsupported.

<sup>3</sup> `object` and string dtypes are not supported in GPU and thus left unimplemented in CuPy.

<sup>4</sup> Floating point error handling depends on CPU-specific features which is not available in GPU.

<sup>5</sup> Structured arrays and record arrays are currently unsupported.

<sup>6</sup> Use of `numpy.poly1d` is discouraged in NumPy and thus we have stopped adding functions with the interface.

<sup>7</sup> Not supported as it has been deprecated in NumPy.

Table 6 – continued from previous page

NumPy	CuPy
<code>numpy.ndarray.ctypes</code>	-
<code>numpy.ndarray.cumprod</code>	<code>cupy.ndarray.cumprod</code>
<code>numpy.ndarray.cumsum</code>	<code>cupy.ndarray.cumsum</code>
<code>numpy.ndarray.data</code>	<code>cupy.ndarray.data</code>
<code>numpy.ndarray.diagonal</code>	<code>cupy.ndarray.diagonal</code>
<code>numpy.ndarray.dot</code>	<code>cupy.ndarray.dot</code>
<code>numpy.ndarray.dtype</code>	<code>cupy.ndarray.dtype</code>
<code>numpy.ndarray.dump</code>	<code>cupy.ndarray.dump</code>
<code>numpy.ndarray.dumps</code>	<code>cupy.ndarray.dumps</code>
<code>numpy.ndarray.fill</code>	<code>cupy.ndarray.fill</code>
<code>numpy.ndarray.flags</code>	<code>cupy.ndarray.flags</code>
<code>numpy.ndarray.flat</code>	<code>cupy.ndarray.flat</code>
<code>numpy.ndarray.flatten</code>	<code>cupy.ndarray.flatten</code>
<code>numpy.ndarray.getfield</code>	-
<code>numpy.ndarray.imag</code>	<code>cupy.ndarray.imag</code>
<code>numpy.ndarray.item</code>	<code>cupy.ndarray.item</code>
<code>numpy.ndarray.itemset</code>	-
<code>numpy.ndarray.itemsize</code>	<code>cupy.ndarray.itemsize</code>
<code>numpy.ndarray.max</code>	<code>cupy.ndarray.max</code>
<code>numpy.ndarray.mean</code>	<code>cupy.ndarray.mean</code>
<code>numpy.ndarray.min</code>	<code>cupy.ndarray.min</code>
<code>numpy.ndarray.nbytes</code>	<code>cupy.ndarray.nbytes</code>
<code>numpy.ndarray.ndim</code>	<code>cupy.ndarray.ndim</code>
<code>numpy.ndarray.newbyteorder</code>	<small>_Page 866, 8</small>
<code>numpy.ndarray.nonzero</code>	<code>cupy.ndarray.nonzero</code>
<code>numpy.ndarray.partition</code>	<code>cupy.ndarray.partition</code>
<code>numpy.ndarray.prod</code>	<code>cupy.ndarray.prod</code>
<code>numpy.ndarray.ptp</code>	<code>cupy.ndarray.ptp</code>
<code>numpy.ndarray.put</code>	<code>cupy.ndarray.put</code>
<code>numpy.ndarray.ravel</code>	<code>cupy.ndarray.ravel</code>
<code>numpy.ndarray.real</code>	<code>cupy.ndarray.real</code>
<code>numpy.ndarray.repeat</code>	<code>cupy.ndarray.repeat</code>
<code>numpy.ndarray.reshape</code>	<code>cupy.ndarray.reshape</code>
<code>numpy.ndarray.resize</code>	-
<code>numpy.ndarray.round</code>	<code>cupy.ndarray.round</code>
<code>numpy.ndarray.searchsorted</code>	<code>cupy.ndarray.searchsorted</code>
<code>numpy.ndarray.setfield</code>	-
<code>numpy.ndarray.setflags</code>	-
<code>numpy.ndarray.shape</code>	<code>cupy.ndarray.shape</code>
<code>numpy.ndarray.size</code>	<code>cupy.ndarray.size</code>
<code>numpy.ndarray.sort</code>	<code>cupy.ndarray.sort</code>
<code>numpy.ndarray.squeeze</code>	<code>cupy.ndarray.squeeze</code>
<code>numpy.ndarray.std</code>	<code>cupy.ndarray.std</code>
<code>numpy.ndarray.strides</code>	<code>cupy.ndarray.strides</code>
<code>numpy.ndarray.sum</code>	<code>cupy.ndarray.sum</code>
<code>numpy.ndarray.swapaxes</code>	<code>cupy.ndarray.swapaxes</code>
<code>numpy.ndarray.take</code>	<code>cupy.ndarray.take</code>
<code>numpy.ndarray.tobytes</code>	<code>cupy.ndarray.tobytes</code>
<code>numpy.ndarray.tofile</code>	<code>cupy.ndarray.tofile</code>

continues on next page

Table 6 – continued from previous page

NumPy	CuPy
<code>numpy.ndarray.tolist</code>	<code>cupy.ndarray.tolist</code>
<code>numpy.ndarray.tostring</code>	<code>_Page 864, 7</code>
<code>numpy.ndarray.trace</code>	<code>cupy.ndarray.trace</code>
<code>numpy.ndarray.transpose</code>	<code>cupy.ndarray.transpose</code>
<code>numpy.ndarray.var</code>	<code>cupy.ndarray.var</code>
<code>numpy.ndarray.view</code>	<code>cupy.ndarray.view</code>

## Linear Algebra

NumPy	CuPy
<code>numpy.linalg.cholesky</code>	<code>cupy.linalg.cholesky</code>
<code>numpy.linalg.cond</code>	-
<code>numpy.linalg.det</code>	<code>cupy.linalg.det</code>
<code>numpy.linalg.eig</code>	-
<code>numpy.linalg.eigh</code>	<code>cupy.linalg.eigh</code>
<code>numpy.linalg.eigvals</code>	-
<code>numpy.linalg.eigvalsh</code>	<code>cupy.linalg.eigvalsh</code>
<code>numpy.linalg.inv</code>	<code>cupy.linalg.inv</code>
<code>numpy.linalg.lstsq</code>	<code>cupy.linalg.lstsq</code>
<code>numpy.linalg.matrix_power</code>	<code>cupy.linalg.matrix_power</code>
<code>numpy.linalg.matrix_rank</code>	<code>cupy.linalg.matrix_rank</code>
<code>numpy.linalg.multi_dot</code>	-
<code>numpy.linalg.norm</code>	<code>cupy.linalg.norm</code>
<code>numpy.linalg.pinv</code>	<code>cupy.linalg.pinv</code>
<code>numpy.linalg.qr</code>	<code>cupy.linalg.qr</code>
<code>numpy.linalg.slogdet</code>	<code>cupy.linalg.slogdet</code>
<code>numpy.linalg.solve</code>	<code>cupy.linalg.solve</code>
<code>numpy.linalg.svd</code>	<code>cupy.linalg.svd</code>
<code>numpy.linalg.tensorinv</code>	<code>cupy.linalg.tensorinv</code>
<code>numpy.linalg.tensorsolve</code>	<code>cupy.linalg.tensorsolve</code>

---

<sup>8</sup> Not supported as GPUs only support little-endian byte-encoding.

## Discrete Fourier Transform

NumPy	CuPy
<code>numpy.fft.fft</code>	<code>cupy.fft.fft</code>
<code>numpy.fft.fft2</code>	<code>cupy.fft.fft2</code>
<code>numpy.fft.fftfreq</code>	<code>cupy.fft.fftfreq</code>
<code>numpy.fft.fftn</code>	<code>cupy.fft.fftn</code>
<code>numpy.fft.fftshift</code>	<code>cupy.fft.fftshift</code>
<code>numpy.fft.hfft</code>	<code>cupy.fft.hfft</code>
<code>numpy.fft.ifft</code>	<code>cupy.fft.ifft</code>
<code>numpy.fft.ifft2</code>	<code>cupy.fft.ifft2</code>
<code>numpy.fft.ifftn</code>	<code>cupy.fft.ifftn</code>
<code>numpy.fft.ifftshift</code>	<code>cupy.fft.ifftshift</code>
<code>numpy.fft.ihfft</code>	<code>cupy.fft.ihfft</code>
<code>numpy.fft.irfft</code>	<code>cupy.fft.irfft</code>
<code>numpy.fft.irfft2</code>	<code>cupy.fft.irfft2</code>
<code>numpy.fft.irfftn</code>	<code>cupy.fft.irfftn</code>
<code>numpy.fft.rfft</code>	<code>cupy.fft.rfft</code>
<code>numpy.fft.rfft2</code>	<code>cupy.fft.rfft2</code>
<code>numpy.fft.rfftfreq</code>	<code>cupy.fft.rfftfreq</code>
<code>numpy.fft.rfftn</code>	<code>cupy.fft.rfftn</code>

## Random Sampling

NumPy	CuPy
<code>numpy.random.BitGenerator</code>	<code>cupy.random.BitGenerator</code>
<code>numpy.random.Generator</code>	-
<code>numpy.random.MT19937</code>	-
<code>numpy.random.PCG64</code>	-
<code>numpy.random.PCG64DXSM</code>	-
<code>numpy.random.Philox</code>	-
<code>numpy.random.RandomState</code>	<code>cupy.random.RandomState</code>
<code>numpy.random.SFC64</code>	-
<code>numpy.random.SeedSequence</code>	-
<code>numpy.random.beta</code>	<code>cupy.random.beta</code>
<code>numpy.random.binomial</code>	<code>cupy.random.binomial</code>
<code>numpy.random.bytes</code>	<code>cupy.random.bytes</code>
<code>numpy.random.chisquare</code>	<code>cupy.random.chisquare</code>
<code>numpy.random.choice</code>	<code>cupy.random.choice</code>
<code>numpy.random.default_rng</code>	<code>cupy.random.default_rng</code>
<code>numpy.random.dirichlet</code>	<code>cupy.random.dirichlet</code>
<code>numpy.random.exponential</code>	<code>cupy.random.exponential</code>
<code>numpy.random.f</code>	<code>cupy.random.f</code>
<code>numpy.random.gamma</code>	<code>cupy.random.gamma</code>
<code>numpy.random.geometric</code>	<code>cupy.random.geometric</code>
<code>numpy.random.get_bit_generator</code>	-
<code>numpy.random.get_state</code>	-
<code>numpy.random.gumbel</code>	<code>cupy.random.gumbel</code>
<code>numpy.random.hypergeometric</code>	<code>cupy.random.hypergeometric</code>

continues on next page

Table 7 – continued from previous page

NumPy	CuPy
<code>numpy.random.laplace</code>	<code>cupy.random.laplace</code>
<code>numpy.random.logistic</code>	<code>cupy.random.logistic</code>
<code>numpy.random.lognormal</code>	<code>cupy.random.lognormal</code>
<code>numpy.random.logseries</code>	<code>cupy.random.logseries</code>
<code>numpy.random.multinomial</code>	<code>cupy.random.multinomial</code>
<code>numpy.random.multivariate_normal</code>	<code>cupy.random.multivariate_normal</code>
<code>numpy.random.negative_binomial</code>	<code>cupy.random.negative_binomial</code>
<code>numpy.random.noncentral_chisquare</code>	<code>cupy.random.noncentral_chisquare</code>
<code>numpy.random.noncentral_f</code>	<code>cupy.random.noncentral_f</code>
<code>numpy.random.normal</code>	<code>cupy.random.normal</code>
<code>numpy.random.pareto</code>	<code>cupy.random.pareto</code>
<code>numpy.random.permutation</code>	<code>cupy.random.permutation</code>
<code>numpy.random.poisson</code>	<code>cupy.random.poisson</code>
<code>numpy.random.power</code>	<code>cupy.random.power</code>
<code>numpy.random.rand</code>	<code>cupy.random.rand</code>
<code>numpy.random.randint</code>	<code>cupy.random.randint</code>
<code>numpy.random.randn</code>	<code>cupy.random.randn</code>
<code>numpy.random.random</code>	<code>cupy.random.random</code>
<code>numpy.random.random_integers</code>	<code>cupy.random.random_integers</code>
<code>numpy.random.random_sample</code>	<code>cupy.random.random_sample</code>
<code>numpy.random.ranf</code>	<code>cupy.random.ranf</code>
<code>numpy.random.rayleigh</code>	<code>cupy.random.rayleigh</code>
<code>numpy.random.sample</code>	<code>cupy.random.sample</code>
<code>numpy.random.seed</code>	<code>cupy.random.seed</code>
<code>numpy.random.set_bit_generator</code>	-
<code>numpy.random.set_state</code>	-
<code>numpy.random.shuffle</code>	<code>cupy.random.shuffle</code>
<code>numpy.random.standard_cauchy</code>	<code>cupy.random.standard_cauchy</code>
<code>numpy.random.standard_exponential</code>	<code>cupy.random.standard_exponential</code>
<code>numpy.random.standard_gamma</code>	<code>cupy.random.standard_gamma</code>
<code>numpy.random.standard_normal</code>	<code>cupy.random.standard_normal</code>
<code>numpy.random.standard_t</code>	<code>cupy.random.standard_t</code>
<code>numpy.random.triangular</code>	<code>cupy.random.triangular</code>
<code>numpy.random.uniform</code>	<code>cupy.random.uniform</code>
<code>numpy.random.vonmises</code>	<code>cupy.random.vonmises</code>
<code>numpy.random.wald</code>	<code>cupy.random.wald</code>
<code>numpy.random.weibull</code>	<code>cupy.random.weibull</code>
<code>numpy.random.zipf</code>	<code>cupy.random.zipf</code>

## Polynomials

NumPy	CuPy
<code>numpy.polynomial.Chebyshev</code>	-
<code>numpy.polynomial.Hermite</code>	-
<code>numpy.polynomial.HermiteE</code>	-
<code>numpy.polynomial.Laguerre</code>	-
<code>numpy.polynomial.Legendre</code>	-
<code>numpy.polynomial.Polynomial</code>	-
<code>numpy.polynomial.set_default_printstyle</code>	-



## Power Series

NumPy	CuPy
<code>numpy.polynomial.polynomial.ABCPolyBase</code>	-
<code>numpy.polynomial.polynomial.Polynomial</code>	-
<code>numpy.polynomial.polynomial.normalize_axis_index</code>	-
<code>numpy.polynomial.polynomial.polyadd</code>	-
<code>numpy.polynomial.polynomial.polycompanion</code>	<code>cupy.polynomial.polynomial.polycompanion</code>
<code>numpy.polynomial.polynomial.polyder</code>	-
<code>numpy.polynomial.polynomial.polydiv</code>	-
<code>numpy.polynomial.polynomial.polydomain</code>	-
<code>numpy.polynomial.polynomial.polyfit</code>	-
<code>numpy.polynomial.polynomial.polyfromroots</code>	-
<code>numpy.polynomial.polynomial.polygrid2d</code>	-
<code>numpy.polynomial.polynomial.polygrid3d</code>	-
<code>numpy.polynomial.polynomial.polyint</code>	-
<code>numpy.polynomial.polynomial.polyline</code>	-
<code>numpy.polynomial.polynomial.polymul</code>	-
<code>numpy.polynomial.polynomial.polymulx</code>	-
<code>numpy.polynomial.polynomial.polyone</code>	-
<code>numpy.polynomial.polynomial.polypow</code>	-
<code>numpy.polynomial.polynomial.polyroots</code>	-
<code>numpy.polynomial.polynomial.polysub</code>	-
<code>numpy.polynomial.polynomial.polytrim</code>	-
<code>numpy.polynomial.polynomial.polyval</code>	<code>cupy.polynomial.polynomial.polyval</code>
<code>numpy.polynomial.polynomial.polyval2d</code>	-
<code>numpy.polynomial.polynomial.polyval3d</code>	-
<code>numpy.polynomial.polynomial.polyvalfromroots</code>	<code>cupy.polynomial.polynomial.polyvalfromroots</code>
<code>numpy.polynomial.polynomial.polyvander</code>	<code>cupy.polynomial.polynomial.polyvander</code>
<code>numpy.polynomial.polynomial.polyvander2d</code>	-
<code>numpy.polynomial.polynomial.polyvander3d</code>	-
<code>numpy.polynomial.polynomial.polyx</code>	-
<code>numpy.polynomial.polynomial.polyzero</code>	-

## Polyutils

NumPy	CuPy
<code>numpy.polynomial.polyutils.absolute</code>	-
<code>numpy.polynomial.polyutils.as_series</code>	<code>cupy.polynomial.polyutils.as_series</code>
<code>numpy.polynomial.polyutils.dragon4_positional</code>	-
<code>numpy.polynomial.polyutils.dragon4_scientific</code>	-
<code>numpy.polynomial.polyutils.format_float</code>	-
<code>numpy.polynomial.polyutils.getdomain</code>	-
<code>numpy.polynomial.polyutils.mapdomain</code>	-
<code>numpy.polynomial.polyutils.mapparms</code>	-
<code>numpy.polynomial.polyutils.trimcoef</code>	<code>cupy.polynomial.polyutils.trimcoef</code>
<code>numpy.polynomial.polyutils.trimseq</code>	<code>cupy.polynomial.polyutils.trimseq</code>

## 5.10.2 SciPy / CuPy APIs

### Discrete Fourier Transform

SciPy	CuPy
<code>scipy.fft.dct</code>	<code>cupyx.scipy.fft.dct</code>
<code>scipy.fft.dctn</code>	<code>cupyx.scipy.fft.dctn</code>
<code>scipy.fft.dst</code>	<code>cupyx.scipy.fft.dst</code>
<code>scipy.fft.dstn</code>	<code>cupyx.scipy.fft.dstn</code>
<code>scipy.fft.fft</code>	<code>cupyx.scipy.fft.fft</code>
<code>scipy.fft.fft2</code>	<code>cupyx.scipy.fft.fft2</code>
<code>scipy.fft.fftfreq</code>	<code>cupyx.scipy.fft.fftfreq</code>
<code>scipy.fft.fftn</code>	<code>cupyx.scipy.fft.fftn</code>
<code>scipy.fft.fftshift</code>	<code>cupyx.scipy.fft.fftshift</code>
<code>scipy.fft.fht</code>	<code>cupyx.scipy.fft.fht</code>
<code>scipy.fft.fhtoffset</code>	-
<code>scipy.fft.get_workers</code>	-
<code>scipy.fft.hfft</code>	<code>cupyx.scipy.fft.hfft</code>
<code>scipy.fft.hfft2</code>	<code>cupyx.scipy.fft.hfft2</code>
<code>scipy.fft.hfftn</code>	<code>cupyx.scipy.fft.hfftn</code>
<code>scipy.fft.idct</code>	<code>cupyx.scipy.fft.idct</code>
<code>scipy.fft.idctn</code>	<code>cupyx.scipy.fft.idctn</code>
<code>scipy.fft.idst</code>	<code>cupyx.scipy.fft.idst</code>
<code>scipy.fft.idstn</code>	<code>cupyx.scipy.fft.idstn</code>
<code>scipy.fft.iffft</code>	<code>cupyx.scipy.fft.iffft</code>
<code>scipy.fft.iffft2</code>	<code>cupyx.scipy.fft.iffft2</code>
<code>scipy.fft.ifftn</code>	<code>cupyx.scipy.fft.ifftn</code>
<code>scipy.fft.iffshift</code>	<code>cupyx.scipy.fft.iffshift</code>
<code>scipy.fft.ifht</code>	<code>cupyx.scipy.fft.ifht</code>
<code>scipy.fft.ihfft</code>	<code>cupyx.scipy.fft.ihfft</code>
<code>scipy.fft.ihfft2</code>	<code>cupyx.scipy.fft.ihfft2</code>
<code>scipy.fft.ihfftn</code>	<code>cupyx.scipy.fft.ihfftn</code>
<code>scipy.fft.irfft</code>	<code>cupyx.scipy.fft.irfft</code>
<code>scipy.fft.irfft2</code>	<code>cupyx.scipy.fft.irfft2</code>
<code>scipy.fft.irfftn</code>	<code>cupyx.scipy.fft.irfftn</code>
<code>scipy.fft.next_fast_len</code>	<code>cupyx.scipy.fft.next_fast_len</code>
<code>scipy.fft.register_backend</code>	-
<code>scipy.fft.rfft</code>	<code>cupyx.scipy.fft.rfft</code>
<code>scipy.fft.rfft2</code>	<code>cupyx.scipy.fft.rfft2</code>
<code>scipy.fft.rfftfreq</code>	<code>cupyx.scipy.fft.rfftfreq</code>
<code>scipy.fft.rfftn</code>	<code>cupyx.scipy.fft.rfftn</code>
<code>scipy.fft.set_backend</code>	-
<code>scipy.fft.set_global_backend</code>	-
<code>scipy.fft.set_workers</code>	-
<code>scipy.fft.skip_backend</code>	-

## Legacy Discrete Fourier Transform

SciPy	CuPy
<code>scipy.fftpack.cc_diff</code>	-
<code>scipy.fftpack.cs_diff</code>	-
<code>scipy.fftpack.dct</code>	-
<code>scipy.fftpack.dctn</code>	-
<code>scipy.fftpack.diff</code>	-
<code>scipy.fftpack.dst</code>	-
<code>scipy.fftpack.dstn</code>	-
<code>scipy.fftpack.fft</code>	<code>cupyx.scipy.fftpack.fft</code>
<code>scipy.fftpack.fft2</code>	<code>cupyx.scipy.fftpack.fft2</code>
<code>scipy.fftpack.fftfreq</code>	-
<code>scipy.fftpack.fftn</code>	<code>cupyx.scipy.fftpack.fftn</code>
<code>scipy.fftpack.fftshift</code>	-
<code>scipy.fftpack.hilbert</code>	-
<code>scipy.fftpack.idct</code>	-
<code>scipy.fftpack.idctn</code>	-
<code>scipy.fftpack.idst</code>	-
<code>scipy.fftpack.idstn</code>	-
<code>scipy.fftpack.iff</code>	<code>cupyx.scipy.fftpack.iff</code>
<code>scipy.fftpack.iff2</code>	<code>cupyx.scipy.fftpack.iff2</code>
<code>scipy.fftpack.iffn</code>	<code>cupyx.scipy.fftpack.iffn</code>
<code>scipy.fftpack.iffshift</code>	-
<code>scipy.fftpack.ihilbert</code>	-
<code>scipy.fftpack.irfft</code>	<code>cupyx.scipy.fftpack.irfft</code>
<code>scipy.fftpack.itolbert</code>	-
<code>scipy.fftpack.next_fast_len</code>	-
<code>scipy.fftpack.rfft</code>	<code>cupyx.scipy.fftpack.rfft</code>
<code>scipy.fftpack.rfftfreq</code>	-
<code>scipy.fftpack.sc_diff</code>	-
<code>scipy.fftpack.shift</code>	-
<code>scipy.fftpack.ss_diff</code>	-
<code>scipy.fftpack.tilbert</code>	-

## Interpolation

SciPy	CuPy
<code>scipy.interpolate.Akima1DInterpolator</code>	<code>cupyx.scipy.interpolate.Akima1DInterpolator</code>
<code>scipy.interpolate.BPoly</code>	<code>cupyx.scipy.interpolate.BPoly</code>
<code>scipy.interpolate.BSpline</code>	<code>cupyx.scipy.interpolate.BSpline</code>
<code>scipy.interpolate.BarycentricInterpolator</code>	<code>cupyx.scipy.interpolate.BarycentricInterpolator</code>
<code>scipy.interpolate.BivariateSpline</code>	-
<code>scipy.interpolate.CloughTocher2DInterpolator</code>	-
<code>scipy.interpolate.CubicHermiteSpline</code>	<code>cupyx.scipy.interpolate.CubicHermiteSpline</code>
<code>scipy.interpolate.CubicSpline</code>	-
<code>scipy.interpolate.InterpolatedUnivariateSpline</code>	-
<code>scipy.interpolate.KroghInterpolator</code>	<code>cupyx.scipy.interpolate.KroghInterpolator</code>
<code>scipy.interpolate.LSQBivariateSpline</code>	-

continues on next page

Table 11 – continued from previous page

SciPy	CuPy
<code>scipy.interpolate.LSQSphereBivariateSpline</code>	-
<code>scipy.interpolate.LSQUnivariateSpline</code>	-
<code>scipy.interpolate.LinearNDInterpolator</code>	-
<code>scipy.interpolate.NdPPoly</code>	<code>cupyx.scipy.interpolate.NdPPoly</code>
<code>scipy.interpolate.NearestNDInterpolator</code>	-
<code>scipy.interpolate.PPoly</code>	<code>cupyx.scipy.interpolate.PPoly</code>
<code>scipy.interpolate.PchipInterpolator</code>	<code>cupyx.scipy.interpolate.PchipInterpolator</code>
<code>scipy.interpolate.RBFInterpolator</code>	<code>cupyx.scipy.interpolate.RBFInterpolator</code>
<code>scipy.interpolate.Rbf</code>	-
<code>scipy.interpolate.RectBivariateSpline</code>	-
<code>scipy.interpolate.RectSphereBivariateSpline</code>	-
<code>scipy.interpolate.RegularGridInterpolator</code>	<code>cupyx.scipy.interpolate.RegularGridInterpolator</code>
<code>scipy.interpolate.SmoothBivariateSpline</code>	-
<code>scipy.interpolate.SmoothSphereBivariateSpline</code>	-
<code>scipy.interpolate.UnivariateSpline</code>	-
<code>scipy.interpolate.approximate_taylor_polynomial</code>	-
<code>scipy.interpolate.barycentric_interpolate</code>	<code>cupyx.scipy.interpolate.barycentric_interpolate</code>
<code>scipy.interpolate.bisplev</code>	-
<code>scipy.interpolate.bisplrep</code>	-
<code>scipy.interpolate.griddata</code>	-
<code>scipy.interpolate.insert</code>	-
<code>scipy.interpolate.interp1d</code>	-
<code>scipy.interpolate.interp2d</code>	-
<code>scipy.interpolate.interpn</code>	<code>cupyx.scipy.interpolate.interpn</code>
<code>scipy.interpolate.krogh_interpolate</code>	<code>cupyx.scipy.interpolate.krogh_interpolate</code>
<code>scipy.interpolate.lagrange</code>	-
<code>scipy.interpolate.make_interp_spline</code>	<code>cupyx.scipy.interpolate.make_interp_spline</code>
<code>scipy.interpolate.make_lsq_spline</code>	-
<code>scipy.interpolate.make_smoothing_spline</code>	-
<code>scipy.interpolate.pade</code>	-
<code>scipy.interpolate.pchip</code>	<code>cupyx.scipy.interpolate.pchip</code>
<code>scipy.interpolate.pchip_interpolate</code>	<code>cupyx.scipy.interpolate.pchip_interpolate</code>
<code>scipy.interpolate.spalde</code>	-
<code>scipy.interpolate.splantider</code>	<code>cupyx.scipy.interpolate.splantider</code>
<code>scipy.interpolate.splder</code>	<code>cupyx.scipy.interpolate.splder</code>
<code>scipy.interpolate.splev</code>	-
<code>scipy.interpolate.splint</code>	-
<code>scipy.interpolate.splprep</code>	-
<code>scipy.interpolate.splrep</code>	-
<code>scipy.interpolate.sproot</code>	-

## Advanced Linear Algebra

SciPy	CuPy
<code>scipy.linalg.bandwidth</code>	<code>cupyx.scipy.linalg.bandwidth</code>
<code>scipy.linalg.block_diag</code>	<code>cupyx.scipy.linalg.block_diag</code>
<code>scipy.linalg.cdf2rdf</code>	-
<code>scipy.linalg.cho_factor</code>	-
<code>scipy.linalg.cho_solve</code>	-
<code>scipy.linalg.cho_solve_banded</code>	-
<code>scipy.linalg.cholesky_banded</code>	-
<code>scipy.linalg.circulant</code>	<code>cupyx.scipy.linalg.circulant</code>
<code>scipy.linalg.clarkson_woodruff_transform</code>	-
<code>scipy.linalg.companion</code>	<code>cupyx.scipy.linalg.companion</code>
<code>scipy.linalg.convolution_matrix</code>	<code>cupyx.scipy.linalg.convolution_matrix</code>
<code>scipy.linalg.coshm</code>	-
<code>scipy.linalg.cosm</code>	-
<code>scipy.linalg.cossin</code>	-
<code>scipy.linalg.dft</code>	<code>cupyx.scipy.linalg.dft</code>
<code>scipy.linalg.diagsvd</code>	-
<code>scipy.linalg.eig_banded</code>	-
<code>scipy.linalg.eigh_tridiagonal</code>	-
<code>scipy.linalg.eigvals_banded</code>	-
<code>scipy.linalg.eigvalsh_tridiagonal</code>	-
<code>scipy.linalg.expm</code>	<code>cupyx.scipy.linalg.expm</code>
<code>scipy.linalg.expm_cond</code>	-
<code>scipy.linalg.expm_frechet</code>	-
<code>scipy.linalg.fiedler</code>	<code>cupyx.scipy.linalg.fiedler</code>
<code>scipy.linalg.fiedler_companion</code>	<code>cupyx.scipy.linalg.fiedler_companion</code>
<code>scipy.linalg.find_best_blas_type</code>	-
<code>scipy.linalg.fractional_matrix_power</code>	-
<code>scipy.linalg.funm</code>	-
<code>scipy.linalg.get_blas_funcs</code>	-
<code>scipy.linalg.get_lapack_funcs</code>	-
<code>scipy.linalg.hadamard</code>	<code>cupyx.scipy.linalg.hadamard</code>
<code>scipy.linalg.hankel</code>	<code>cupyx.scipy.linalg.hankel</code>
<code>scipy.linalg.helmert</code>	<code>cupyx.scipy.linalg.helmert</code>
<code>scipy.linalg.hessenberg</code>	-
<code>scipy.linalg.hilbert</code>	<code>cupyx.scipy.linalg.hilbert</code>
<code>scipy.linalg.invhilbert</code>	-
<code>scipy.linalg.invpascal</code>	-
<code>scipy.linalg.ishermitian</code>	-
<code>scipy.linalg.issymmetric</code>	-
<code>scipy.linalg.khatri_rao</code>	<code>cupyx.scipy.linalg.khatri_rao</code>
<code>scipy.linalg.kron</code>	<code>cupyx.scipy.linalg.kron</code>
<code>scipy.linalg.lda</code>	-
<code>scipy.linalg.leslie</code>	<code>cupyx.scipy.linalg.leslie</code>
<code>scipy.linalg.logm</code>	-
<code>scipy.linalg.lu</code>	<code>cupyx.scipy.linalg.lu</code>
<code>scipy.linalg.lu_factor</code>	<code>cupyx.scipy.linalg.lu_factor</code>
<code>scipy.linalg.lu_solve</code>	<code>cupyx.scipy.linalg.lu_solve</code>

continues on next page

Table 12 – continued from previous page

SciPy	CuPy
<code>scipy.linalg.matmul_toeplitz</code>	-
<code>scipy.linalg.matrix_balance</code>	-
<code>scipy.linalg.null_space</code>	-
<code>scipy.linalg.ordqz</code>	-
<code>scipy.linalg.orth</code>	-
<code>scipy.linalg.orthogonal_procrustes</code>	-
<code>scipy.linalg.pascal</code>	-
<code>scipy.linalg.pinvh</code>	-
<code>scipy.linalg.polar</code>	-
<code>scipy.linalg.qr_delete</code>	-
<code>scipy.linalg.qr_insert</code>	-
<code>scipy.linalg.qr_multiply</code>	-
<code>scipy.linalg.qr_update</code>	-
<code>scipy.linalg.qz</code>	-
<code>scipy.linalg.rq</code>	-
<code>scipy.linalg.rsf2csf</code>	-
<code>scipy.linalg.schur</code>	-
<code>scipy.linalg.signm</code>	-
<code>scipy.linalg.sinhm</code>	-
<code>scipy.linalg.sinm</code>	-
<code>scipy.linalg.solve_banded</code>	-
<code>scipy.linalg.solve_circulant</code>	-
<code>scipy.linalg.solve_continuous_are</code>	-
<code>scipy.linalg.solve_continuous_lyapunov</code>	-
<code>scipy.linalg.solve_discrete_are</code>	-
<code>scipy.linalg.solve_discrete_lyapunov</code>	-
<code>scipy.linalg.solve_lyapunov</code>	-
<code>scipy.linalg.solve_sylvester</code>	-
<code>scipy.linalg.solve_toeplitz</code>	-
<code>scipy.linalg.solve_triangular</code>	<code>cupyx.scipy.linalg.solve_triangular</code>
<code>scipy.linalg.solveh_banded</code>	-
<code>scipy.linalg.sqrth</code>	-
<code>scipy.linalg.subspace_angles</code>	-
<code>scipy.linalg.svdvals</code>	-
<code>scipy.linalg.tanhm</code>	-
<code>scipy.linalg.tanm</code>	-
<code>scipy.linalg.toeplitz</code>	<code>cupyx.scipy.linalg.toeplitz</code>
<code>scipy.linalg.tri</code>	<code>cupyx.scipy.linalg.tri</code>
<code>scipy.linalg.tril</code>	<code>cupyx.scipy.linalg.tril</code>
<code>scipy.linalg.triu</code>	<code>cupyx.scipy.linalg.triu</code>

## Multidimensional Image Processing

SciPy	CuPy
<code>scipy.ndimage.affine_transform</code>	<code>cupyx.scipy.ndimage.affine_transform</code>
<code>scipy.ndimage.binary_closing</code>	<code>cupyx.scipy.ndimage.binary_closing</code>
<code>scipy.ndimage.binary_dilation</code>	<code>cupyx.scipy.ndimage.binary_dilation</code>
<code>scipy.ndimage.binary_erosion</code>	<code>cupyx.scipy.ndimage.binary_erosion</code>
<code>scipy.ndimage.binary_fill_holes</code>	<code>cupyx.scipy.ndimage.binary_fill_holes</code>
<code>scipy.ndimage.binary_hit_or_miss</code>	<code>cupyx.scipy.ndimage.binary_hit_or_miss</code>
<code>scipy.ndimage.binary_opening</code>	<code>cupyx.scipy.ndimage.binary_opening</code>
<code>scipy.ndimage.binary_propagation</code>	<code>cupyx.scipy.ndimage.binary_propagation</code>
<code>scipy.ndimage.black_tophat</code>	<code>cupyx.scipy.ndimage.black_tophat</code>
<code>scipy.ndimage.center_of_mass</code>	<code>cupyx.scipy.ndimage.center_of_mass</code>
<code>scipy.ndimage.convolve</code>	<code>cupyx.scipy.ndimage.convolve</code>
<code>scipy.ndimage.convolve1d</code>	<code>cupyx.scipy.ndimage.convolve1d</code>
<code>scipy.ndimage.correlate</code>	<code>cupyx.scipy.ndimage.correlate</code>
<code>scipy.ndimage.correlate1d</code>	<code>cupyx.scipy.ndimage.correlate1d</code>
<code>scipy.ndimage.distance_transform_bf</code>	-
<code>scipy.ndimage.distance_transform_cdt</code>	-
<code>scipy.ndimage.distance_transform_edt</code>	<code>cupyx.scipy.ndimage.distance_transform_edt</code>
<code>scipy.ndimage.extrema</code>	<code>cupyx.scipy.ndimage.extrema</code>
<code>scipy.ndimage.find_objects</code>	-
<code>scipy.ndimage.fourier_ellipsoid</code>	<code>cupyx.scipy.ndimage.fourier_ellipsoid</code>
<code>scipy.ndimage.fourier_gaussian</code>	<code>cupyx.scipy.ndimage.fourier_gaussian</code>
<code>scipy.ndimage.fourier_shift</code>	<code>cupyx.scipy.ndimage.fourier_shift</code>
<code>scipy.ndimage.fourier_uniform</code>	<code>cupyx.scipy.ndimage.fourier_uniform</code>
<code>scipy.ndimage.gaussian_filter</code>	<code>cupyx.scipy.ndimage.gaussian_filter</code>
<code>scipy.ndimage.gaussian_filter1d</code>	<code>cupyx.scipy.ndimage.gaussian_filter1d</code>
<code>scipy.ndimage.gaussian_gradient_magnitude</code>	<code>cupyx.scipy.ndimage.gaussian_gradient_magnitude</code>
<code>scipy.ndimage.gaussian_laplace</code>	<code>cupyx.scipy.ndimage.gaussian_laplace</code>
<code>scipy.ndimage.generate_binary_structure</code>	<code>cupyx.scipy.ndimage.generate_binary_structure</code>
<code>scipy.ndimage.generic_filter</code>	<code>cupyx.scipy.ndimage.generic_filter</code>
<code>scipy.ndimage.generic_filter1d</code>	<code>cupyx.scipy.ndimage.generic_filter1d</code>
<code>scipy.ndimage.generic_gradient_magnitude</code>	<code>cupyx.scipy.ndimage.generic_gradient_magnitude</code>
<code>scipy.ndimage.generic_laplace</code>	<code>cupyx.scipy.ndimage.generic_laplace</code>
<code>scipy.ndimage.geometric_transform</code>	-
<code>scipy.ndimage.grey_closing</code>	<code>cupyx.scipy.ndimage.grey_closing</code>
<code>scipy.ndimage.grey_dilation</code>	<code>cupyx.scipy.ndimage.grey_dilation</code>
<code>scipy.ndimage.grey_erosion</code>	<code>cupyx.scipy.ndimage.grey_erosion</code>
<code>scipy.ndimage.grey_opening</code>	<code>cupyx.scipy.ndimage.grey_opening</code>
<code>scipy.ndimage.histogram</code>	<code>cupyx.scipy.ndimage.histogram</code>
<code>scipy.ndimage.iterate_structure</code>	<code>cupyx.scipy.ndimage.iterate_structure</code>
<code>scipy.ndimage.label</code>	<code>cupyx.scipy.ndimage.label</code>
<code>scipy.ndimage.labeled_comprehension</code>	<code>cupyx.scipy.ndimage.labeled_comprehension</code>
<code>scipy.ndimage.laplace</code>	<code>cupyx.scipy.ndimage.laplace</code>
<code>scipy.ndimage.map_coordinates</code>	<code>cupyx.scipy.ndimage.map_coordinates</code>
<code>scipy.ndimage.maximum</code>	<code>cupyx.scipy.ndimage.maximum</code>
<code>scipy.ndimage.maximum_filter</code>	<code>cupyx.scipy.ndimage.maximum_filter</code>
<code>scipy.ndimage.maximum_filter1d</code>	<code>cupyx.scipy.ndimage.maximum_filter1d</code>
<code>scipy.ndimage.maximum_position</code>	<code>cupyx.scipy.ndimage.maximum_position</code>

continues on next page



Table 13 – continued from previous page

SciPy	CuPy
<code>scipy.ndimage.mean</code>	<code>cupyx.scipy.ndimage.mean</code>
<code>scipy.ndimage.median</code>	<code>cupyx.scipy.ndimage.median</code>
<code>scipy.ndimage.median_filter</code>	<code>cupyx.scipy.ndimage.median_filter</code>
<code>scipy.ndimage.minimum</code>	<code>cupyx.scipy.ndimage.minimum</code>
<code>scipy.ndimage.minimum_filter</code>	<code>cupyx.scipy.ndimage.minimum_filter</code>
<code>scipy.ndimage.minimum_filter1d</code>	<code>cupyx.scipy.ndimage.minimum_filter1d</code>
<code>scipy.ndimage.minimum_position</code>	<code>cupyx.scipy.ndimage.minimum_position</code>
<code>scipy.ndimage.morphological_gradient</code>	<code>cupyx.scipy.ndimage.morphological_gradient</code>
<code>scipy.ndimage.morphological_laplace</code>	<code>cupyx.scipy.ndimage.morphological_laplace</code>
<code>scipy.ndimage.percentile_filter</code>	<code>cupyx.scipy.ndimage.percentile_filter</code>
<code>scipy.ndimage.prewitt</code>	<code>cupyx.scipy.ndimage.prewitt</code>
<code>scipy.ndimage.rank_filter</code>	<code>cupyx.scipy.ndimage.rank_filter</code>
<code>scipy.ndimage.rotate</code>	<code>cupyx.scipy.ndimage.rotate</code>
<code>scipy.ndimage.shift</code>	<code>cupyx.scipy.ndimage.shift</code>
<code>scipy.ndimage.sobel</code>	<code>cupyx.scipy.ndimage.sobel</code>
<code>scipy.ndimage.spline_filter</code>	<code>cupyx.scipy.ndimage.spline_filter</code>
<code>scipy.ndimage.spline_filter1d</code>	<code>cupyx.scipy.ndimage.spline_filter1d</code>
<code>scipy.ndimage.standard_deviation</code>	<code>cupyx.scipy.ndimage.standard_deviation</code>
<code>scipy.ndimage.sum</code>	<code>cupyx.scipy.ndimage.sum</code>
<code>scipy.ndimage.sum_labels</code>	<code>cupyx.scipy.ndimage.sum_labels</code>
<code>scipy.ndimage.uniform_filter</code>	<code>cupyx.scipy.ndimage.uniform_filter</code>
<code>scipy.ndimage.uniform_filter1d</code>	<code>cupyx.scipy.ndimage.uniform_filter1d</code>
<code>scipy.ndimage.value_indices</code>	<code>cupyx.scipy.ndimage.value_indices</code>
<code>scipy.ndimage.variance</code>	<code>cupyx.scipy.ndimage.variance</code>
<code>scipy.ndimage.watershed_ift</code>	-
<code>scipy.ndimage.white_tophat</code>	<code>cupyx.scipy.ndimage.white_tophat</code>
<code>scipy.ndimage.zoom</code>	<code>cupyx.scipy.ndimage.zoom</code>

## Signal processing

SciPy	CuPy
<code>scipy.signal.CZT</code>	<code>cupyx.scipy.signal.CZT</code>
<code>scipy.signal.StateSpace</code>	<code>cupyx.scipy.signal.StateSpace</code>
<code>scipy.signal.TransferFunction</code>	<code>cupyx.scipy.signal.TransferFunction</code>
<code>scipy.signal.ZerosPolesGain</code>	<code>cupyx.scipy.signal.ZerosPolesGain</code>
<code>scipy.signal.ZoomFFT</code>	<code>cupyx.scipy.signal.ZoomFFT</code>
<code>scipy.signal.abcd_normalize</code>	<code>cupyx.scipy.signal.abcd_normalize</code>
<code>scipy.signal.argrextrema</code>	<code>cupyx.scipy.signal.argrextrema</code>
<code>scipy.signal.argrelmax</code>	<code>cupyx.scipy.signal.argrelmax</code>
<code>scipy.signal.argrelmin</code>	<code>cupyx.scipy.signal.argrelmin</code>
<code>scipy.signal.band_stop_obj</code>	<code>cupyx.scipy.signal.band_stop_obj</code>
<code>scipy.signal.barthann</code>	-
<code>scipy.signal.bartlett</code>	-
<code>scipy.signal.bessel</code>	-
<code>scipy.signal.besselap</code>	-
<code>scipy.signal.bilinear</code>	<code>cupyx.scipy.signal.bilinear</code>
<code>scipy.signal.bilinear_zpk</code>	<code>cupyx.scipy.signal.bilinear_zpk</code>
<code>scipy.signal.blackman</code>	-

continues on next page



Table 14 – continued from previous page

SciPy	CuPy
<code>scipy.signal.blackmanharris</code>	-
<code>scipy.signal.bode</code>	<code>cupyx.scipy.signal.bode</code>
<code>scipy.signal.bohman</code>	-
<code>scipy.signal.boxcar</code>	-
<code>scipy.signal.bspline</code>	-
<code>scipy.signal.buttap</code>	<code>cupyx.scipy.signal.buttap</code>
<code>scipy.signal.butter</code>	<code>cupyx.scipy.signal.butter</code>
<code>scipy.signal.buttord</code>	<code>cupyx.scipy.signal.buttord</code>
<code>scipy.signal.cascade</code>	-
<code>scipy.signal.chb1ap</code>	<code>cupyx.scipy.signal.chb1ap</code>
<code>scipy.signal.chb1ord</code>	<code>cupyx.scipy.signal.chb1ord</code>
<code>scipy.signal.chb2ap</code>	<code>cupyx.scipy.signal.chb2ap</code>
<code>scipy.signal.chb2ord</code>	<code>cupyx.scipy.signal.chb2ord</code>
<code>scipy.signal.chewin</code>	-
<code>scipy.signal.cheby1</code>	<code>cupyx.scipy.signal.cheby1</code>
<code>scipy.signal.cheby2</code>	<code>cupyx.scipy.signal.cheby2</code>
<code>scipy.signal.check_COLA</code>	<code>cupyx.scipy.signal.check_COLA</code>
<code>scipy.signal.check_NOLA</code>	<code>cupyx.scipy.signal.check_NOLA</code>
<code>scipy.signal.chirp</code>	<code>cupyx.scipy.signal.chirp</code>
<code>scipy.signal.choose_conv_method</code>	<code>cupyx.scipy.signal.choose_conv_method</code>
<code>scipy.signal.cmplx_sort</code>	-
<code>scipy.signal.coherence</code>	<code>cupyx.scipy.signal.coherence</code>
<code>scipy.signal.cont2discrete</code>	<code>cupyx.scipy.signal.cont2discrete</code>
<code>scipy.signal.convolve</code>	<code>cupyx.scipy.signal.convolve</code>
<code>scipy.signal.convolve2d</code>	<code>cupyx.scipy.signal.convolve2d</code>
<code>scipy.signal.correlate</code>	<code>cupyx.scipy.signal.correlate</code>
<code>scipy.signal.correlate2d</code>	<code>cupyx.scipy.signal.correlate2d</code>
<code>scipy.signal.correlation_lags</code>	<code>cupyx.scipy.signal.correlation_lags</code>
<code>scipy.signal.cosine</code>	-
<code>scipy.signal.csd</code>	<code>cupyx.scipy.signal.csd</code>
<code>scipy.signal.cspline1d</code>	<code>cupyx.scipy.signal.cspline1d</code>
<code>scipy.signal.cspline1d_eval</code>	<code>cupyx.scipy.signal.cspline1d_eval</code>
<code>scipy.signal.cspline2d</code>	<code>cupyx.scipy.signal.cspline2d</code>
<code>scipy.signal.cubic</code>	-
<code>scipy.signal.cwt</code>	<code>cupyx.scipy.signal.cwt</code>
<code>scipy.signal.czt</code>	<code>cupyx.scipy.signal.czt</code>
<code>scipy.signal.czt_points</code>	<code>cupyx.scipy.signal.czt_points</code>
<code>scipy.signal.daub</code>	-
<code>scipy.signal.dbode</code>	<code>cupyx.scipy.signal.dbode</code>
<code>scipy.signal.decimate</code>	<code>cupyx.scipy.signal.decimate</code>
<code>scipy.signal.deconvolve</code>	<code>cupyx.scipy.signal.deconvolve</code>
<code>scipy.signal.detrend</code>	<code>cupyx.scipy.signal.detrend</code>
<code>scipy.signal.dfreqresp</code>	<code>cupyx.scipy.signal.dfreqresp</code>
<code>scipy.signal.dimpulse</code>	<code>cupyx.scipy.signal.dimpulse</code>
<code>scipy.signal.dlsim</code>	<code>cupyx.scipy.signal.dlsim</code>
<code>scipy.signal.dlti</code>	<code>cupyx.scipy.signal.dlti</code>
<code>scipy.signal.dstep</code>	<code>cupyx.scipy.signal.dstep</code>
<code>scipy.signal.ellip</code>	<code>cupyx.scipy.signal.ellip</code>
<code>scipy.signal.ellipap</code>	<code>cupyx.scipy.signal.ellipap</code>

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
<code>scipy.signal.ellipord</code>	<code>cupyx.scipy.signal.ellipord</code>
<code>scipy.signal.exponential</code>	-
<code>scipy.signal.fftconvolve</code>	<code>cupyx.scipy.signal.fftconvolve</code>
<code>scipy.signal.filtfilt</code>	<code>cupyx.scipy.signal.filtfilt</code>
<code>scipy.signal.find_peaks</code>	<code>cupyx.scipy.signal.find_peaks</code>
<code>scipy.signal.find_peaks_cwt</code>	-
<code>scipy.signal.findfreqs</code>	<code>cupyx.scipy.signal.findfreqs</code>
<code>scipy.signal.firls</code>	<code>cupyx.scipy.signal.firls</code>
<code>scipy.signal.firwin</code>	<code>cupyx.scipy.signal.firwin</code>
<code>scipy.signal.firwin2</code>	<code>cupyx.scipy.signal.firwin2</code>
<code>scipy.signal.flattop</code>	-
<code>scipy.signal.freqresp</code>	<code>cupyx.scipy.signal.freqresp</code>
<code>scipy.signal.freqs</code>	<code>cupyx.scipy.signal.freqs</code>
<code>scipy.signal.freqs_zpk</code>	<code>cupyx.scipy.signal.freqs_zpk</code>
<code>scipy.signal.freqz</code>	<code>cupyx.scipy.signal.freqz</code>
<code>scipy.signal.freqz_zpk</code>	<code>cupyx.scipy.signal.freqz_zpk</code>
<code>scipy.signal.gammatone</code>	<code>cupyx.scipy.signal.gammatone</code>
<code>scipy.signal.gauss_spline</code>	<code>cupyx.scipy.signal.gauss_spline</code>
<code>scipy.signal.gaussian</code>	-
<code>scipy.signal.gausspulse</code>	<code>cupyx.scipy.signal.gausspulse</code>
<code>scipy.signal.general_gaussian</code>	-
<code>scipy.signal.get_window</code>	<code>cupyx.scipy.signal.get_window</code>
<code>scipy.signal.group_delay</code>	<code>cupyx.scipy.signal.group_delay</code>
<code>scipy.signal.hamming</code>	-
<code>scipy.signal.hann</code>	-
<code>scipy.signal.hilbert</code>	<code>cupyx.scipy.signal.hilbert</code>
<code>scipy.signal.hilbert2</code>	<code>cupyx.scipy.signal.hilbert2</code>
<code>scipy.signal.iircomb</code>	<code>cupyx.scipy.signal.iircomb</code>
<code>scipy.signal.iirdesign</code>	<code>cupyx.scipy.signal.iirdesign</code>
<code>scipy.signal.iirfilter</code>	<code>cupyx.scipy.signal.iirfilter</code>
<code>scipy.signal.iirnotch</code>	<code>cupyx.scipy.signal.iirnotch</code>
<code>scipy.signal.iirpeak</code>	<code>cupyx.scipy.signal.iirpeak</code>
<code>scipy.signal.impulse</code>	<code>cupyx.scipy.signal.impulse</code>
<code>scipy.signal.impulse2</code>	-
<code>scipy.signal.invres</code>	<code>cupyx.scipy.signal.invres</code>
<code>scipy.signal.invresz</code>	<code>cupyx.scipy.signal.invresz</code>
<code>scipy.signal.istft</code>	<code>cupyx.scipy.signal.istft</code>
<code>scipy.signal.kaiser</code>	-
<code>scipy.signal.kaiser_atten</code>	<code>cupyx.scipy.signal.kaiser_atten</code>
<code>scipy.signal.kaiser_beta</code>	<code>cupyx.scipy.signal.kaiser_beta</code>
<code>scipy.signal.kaiserord</code>	<code>cupyx.scipy.signal.kaiserord</code>
<code>scipy.signal.lfilter</code>	<code>cupyx.scipy.signal.lfilter</code>
<code>scipy.signal.lfilter_zi</code>	<code>cupyx.scipy.signal.lfilter_zi</code>
<code>scipy.signal.lfiltic</code>	<code>cupyx.scipy.signal.lfiltic</code>
<code>scipy.signal.lombscargle</code>	<code>cupyx.scipy.signal.lombscargle</code>
<code>scipy.signal.lp2bp</code>	<code>cupyx.scipy.signal.lp2bp</code>
<code>scipy.signal.lp2bp_zpk</code>	<code>cupyx.scipy.signal.lp2bp_zpk</code>
<code>scipy.signal.lp2bs</code>	<code>cupyx.scipy.signal.lp2bs</code>
<code>scipy.signal.lp2bs_zpk</code>	<code>cupyx.scipy.signal.lp2bs_zpk</code>

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
<code>scipy.signal.lp2hp</code>	<code>cupyx.scipy.signal.lp2hp</code>
<code>scipy.signal.lp2hp_zpk</code>	<code>cupyx.scipy.signal.lp2hp_zpk</code>
<code>scipy.signal.lp2lp</code>	<code>cupyx.scipy.signal.lp2lp</code>
<code>scipy.signal.lp2lp_zpk</code>	<code>cupyx.scipy.signal.lp2lp_zpk</code>
<code>scipy.signal.lsim</code>	<code>cupyx.scipy.signal.lsim</code>
<code>scipy.signal.lsim2</code>	-
<code>scipy.signal.lti</code>	<code>cupyx.scipy.signal.lti</code>
<code>scipy.signal.max_len_seq</code>	<code>cupyx.scipy.signal.max_len_seq</code>
<code>scipy.signal.medfilt</code>	<code>cupyx.scipy.signal.medfilt</code>
<code>scipy.signal.medfilt2d</code>	<code>cupyx.scipy.signal.medfilt2d</code>
<code>scipy.signal.minimum_phase</code>	<code>cupyx.scipy.signal.minimum_phase</code>
<code>scipy.signal.morlet</code>	<code>cupyx.scipy.signal.morlet</code>
<code>scipy.signal.morlet2</code>	<code>cupyx.scipy.signal.morlet2</code>
<code>scipy.signal.normalize</code>	<code>cupyx.scipy.signal.normalize</code>
<code>scipy.signal.nuttall</code>	-
<code>scipy.signal.oaconvolve</code>	<code>cupyx.scipy.signal.oaconvolve</code>
<code>scipy.signal.order_filter</code>	<code>cupyx.scipy.signal.order_filter</code>
<code>scipy.signal.parzen</code>	-
<code>scipy.signal.peak_prominences</code>	<code>cupyx.scipy.signal.peak_prominences</code>
<code>scipy.signal.peak_widths</code>	<code>cupyx.scipy.signal.peak_widths</code>
<code>scipy.signal.periodogram</code>	<code>cupyx.scipy.signal.periodogram</code>
<code>scipy.signal.place_poles</code>	<code>cupyx.scipy.signal.place_poles</code>
<code>scipy.signal.qmf</code>	<code>cupyx.scipy.signal.qmf</code>
<code>scipy.signal.qspline1d</code>	<code>cupyx.scipy.signal.qspline1d</code>
<code>scipy.signal.qspline1d_eval</code>	<code>cupyx.scipy.signal.qspline1d_eval</code>
<code>scipy.signal.qspline2d</code>	<code>cupyx.scipy.signal.qspline2d</code>
<code>scipy.signal.quadratic</code>	-
<code>scipy.signal.remez</code>	-
<code>scipy.signal.resample</code>	<code>cupyx.scipy.signal.resample</code>
<code>scipy.signal.resample_poly</code>	<code>cupyx.scipy.signal.resample_poly</code>
<code>scipy.signal.residue</code>	<code>cupyx.scipy.signal.residue</code>
<code>scipy.signal.residuez</code>	<code>cupyx.scipy.signal.residuez</code>
<code>scipy.signal.ricker</code>	<code>cupyx.scipy.signal.ricker</code>
<code>scipy.signal.savgol_coeffs</code>	<code>cupyx.scipy.signal.savgol_coeffs</code>
<code>scipy.signal.savgol_filter</code>	<code>cupyx.scipy.signal.savgol_filter</code>
<code>scipy.signal.sawtooth</code>	<code>cupyx.scipy.signal.sawtooth</code>
<code>scipy.signal.sepfir2d</code>	<code>cupyx.scipy.signal.sepfir2d</code>
<code>scipy.signal.sos2tf</code>	<code>cupyx.scipy.signal.sos2tf</code>
<code>scipy.signal.sos2zpk</code>	<code>cupyx.scipy.signal.sos2zpk</code>
<code>scipy.signal.sosfilt</code>	<code>cupyx.scipy.signal.sosfilt</code>
<code>scipy.signal.sosfilt_zi</code>	<code>cupyx.scipy.signal.sosfilt_zi</code>
<code>scipy.signal.sosfiltfilt</code>	<code>cupyx.scipy.signal.sosfiltfilt</code>
<code>scipy.signal.sosfreqz</code>	<code>cupyx.scipy.signal.sosfreqz</code>
<code>scipy.signal.spectrogram</code>	<code>cupyx.scipy.signal.spectrogram</code>
<code>scipy.signal.spline_filter</code>	<code>cupyx.scipy.signal.spline_filter</code>
<code>scipy.signal.square</code>	<code>cupyx.scipy.signal.square</code>
<code>scipy.signal.ss2tf</code>	<code>cupyx.scipy.signal.ss2tf</code>
<code>scipy.signal.ss2zpk</code>	<code>cupyx.scipy.signal.ss2zpk</code>
<code>scipy.signal.step</code>	<code>cupyx.scipy.signal.step</code>

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
<code>scipy.signal.step2</code>	-
<code>scipy.signal.stft</code>	<code>cupyx.scipy.signal.stft</code>
<code>scipy.signal.sweep_poly</code>	-
<code>scipy.signal.symiirorder1</code>	<code>cupyx.scipy.signal.symiirorder1</code>
<code>scipy.signal.symiirorder2</code>	<code>cupyx.scipy.signal.symiirorder2</code>
<code>scipy.signal.tf2sos</code>	<code>cupyx.scipy.signal.tf2sos</code>
<code>scipy.signal.tf2ss</code>	<code>cupyx.scipy.signal.tf2ss</code>
<code>scipy.signal.tf2zpk</code>	<code>cupyx.scipy.signal.tf2zpk</code>
<code>scipy.signal.triang</code>	-
<code>scipy.signal.tukey</code>	-
<code>scipy.signal.unique_roots</code>	<code>cupyx.scipy.signal.unique_roots</code>
<code>scipy.signal.unit_impulse</code>	<code>cupyx.scipy.signal.unit_impulse</code>
<code>scipy.signal.upfirdn</code>	<code>cupyx.scipy.signal.upfirdn</code>
<code>scipy.signal.vectorstrength</code>	<code>cupyx.scipy.signal.vectorstrength</code>
<code>scipy.signal.welch</code>	<code>cupyx.scipy.signal.welch</code>
<code>scipy.signal.wiener</code>	<code>cupyx.scipy.signal.wiener</code>
<code>scipy.signal.zoom_fft</code>	<code>cupyx.scipy.signal.zoom_fft</code>
<code>scipy.signal.zpk2sos</code>	<code>cupyx.scipy.signal.zpk2sos</code>
<code>scipy.signal.zpk2ss</code>	<code>cupyx.scipy.signal.zpk2ss</code>
<code>scipy.signal.zpk2tf</code>	<code>cupyx.scipy.signal.zpk2tf</code>

## Sparse Matrices

SciPy	CuPy
<code>scipy.sparse.block_diag</code>	-
<code>scipy.sparse.bmat</code>	<code>cupyx.scipy.sparse.bmat</code>
<code>scipy.sparse.bsr_array</code>	-
<code>scipy.sparse.bsr_matrix</code>	-
<code>scipy.sparse.coo_array</code>	-
<code>scipy.sparse.coo_matrix</code>	<code>cupyx.scipy.sparse.coo_matrix</code>
<code>scipy.sparse.csc_array</code>	-
<code>scipy.sparse.csc_matrix</code>	<code>cupyx.scipy.sparse.csc_matrix</code>
<code>scipy.sparse.csr_array</code>	-
<code>scipy.sparse.csr_matrix</code>	<code>cupyx.scipy.sparse.csr_matrix</code>
<code>scipy.sparse.dia_array</code>	-
<code>scipy.sparse.dia_matrix</code>	<code>cupyx.scipy.sparse.dia_matrix</code>
<code>scipy.sparse.diags</code>	<code>cupyx.scipy.sparse.diags</code>
<code>scipy.sparse.dok_array</code>	-
<code>scipy.sparse.dok_matrix</code>	-
<code>scipy.sparse.eye</code>	<code>cupyx.scipy.sparse.eye</code>
<code>scipy.sparse.find</code>	<code>cupyx.scipy.sparse.find</code>
<code>scipy.sparse.hstack</code>	<code>cupyx.scipy.sparse.hstack</code>
<code>scipy.sparse.identity</code>	<code>cupyx.scipy.sparse.identity</code>
<code>scipy.sparse.issparse</code>	<code>cupyx.scipy.sparse.issparse</code>
<code>scipy.sparse.isspmatrix</code>	<code>cupyx.scipy.sparse.isspmatrix</code>
<code>scipy.sparse.isspmatrix_bsr</code>	-
<code>scipy.sparse.isspmatrix_coo</code>	<code>cupyx.scipy.sparse.isspmatrix_coo</code>
<code>scipy.sparse.isspmatrix_csc</code>	<code>cupyx.scipy.sparse.isspmatrix_csc</code>

continues on next page

Table 15 – continued from previous page

SciPy	CuPy
<code>scipy.sparse.isspmatrix_csr</code>	<code>cupyx.scipy.sparse.isspmatrix_csr</code>
<code>scipy.sparse.isspmatrix_dia</code>	<code>cupyx.scipy.sparse.isspmatrix_dia</code>
<code>scipy.sparse.isspmatrix_dok</code>	-
<code>scipy.sparse.isspmatrix_lil</code>	-
<code>scipy.sparse.kron</code>	<code>cupyx.scipy.sparse.kron</code>
<code>scipy.sparse.kronsum</code>	<code>cupyx.scipy.sparse.kronsum</code>
<code>scipy.sparse.lil_array</code>	-
<code>scipy.sparse.lil_matrix</code>	-
<code>scipy.sparse.load_npz</code>	-
<code>scipy.sparse.rand</code>	<code>cupyx.scipy.sparse.rand</code>
<code>scipy.sparse.random</code>	<code>cupyx.scipy.sparse.random</code>
<code>scipy.sparse.save_npz</code>	-
<code>scipy.sparse.sparray</code>	-
<code>scipy.sparse.spdiags</code>	<code>cupyx.scipy.sparse.spdiags</code>
<code>scipy.sparse.spmatrix</code>	<code>cupyx.scipy.sparse.spmatrix</code>
<code>scipy.sparse.tril</code>	<code>cupyx.scipy.sparse.tril</code>
<code>scipy.sparse.triu</code>	<code>cupyx.scipy.sparse.triu</code>
<code>scipy.sparse.vstack</code>	<code>cupyx.scipy.sparse.vstack</code>

## Sparse Linear Algebra

SciPy	CuPy
<code>scipy.sparse.linalg.LinearOperator</code>	<code>cupyx.scipy.sparse.linalg.LinearOperator</code>
<code>scipy.sparse.linalg.SuperLU</code>	<code>cupyx.scipy.sparse.linalg.SuperLU</code>
<code>scipy.sparse.linalg.aslinearoperator</code>	<code>cupyx.scipy.sparse.linalg.aslinearoperator</code>
<code>scipy.sparse.linalg.bicg</code>	-
<code>scipy.sparse.linalg.bicgstab</code>	-
<code>scipy.sparse.linalg.cg</code>	<code>cupyx.scipy.sparse.linalg.cg</code>
<code>scipy.sparse.linalg.cgs</code>	<code>cupyx.scipy.sparse.linalg.cgs</code>
<code>scipy.sparse.linalg.eigs</code>	-
<code>scipy.sparse.linalg.eigsh</code>	<code>cupyx.scipy.sparse.linalg.eigsh</code>
<code>scipy.sparse.linalg.expm</code>	-
<code>scipy.sparse.linalg.expm_multiply</code>	-
<code>scipy.sparse.linalg.factorized</code>	<code>cupyx.scipy.sparse.linalg.factorized</code>
<code>scipy.sparse.linalg.gcrotmk</code>	-
<code>scipy.sparse.linalg.gmres</code>	<code>cupyx.scipy.sparse.linalg.gmres</code>
<code>scipy.sparse.linalg.inv</code>	-
<code>scipy.sparse.linalg.lgmres</code>	-
<code>scipy.sparse.linalg.lobpcg</code>	<code>cupyx.scipy.sparse.linalg.lobpcg</code>
<code>scipy.sparse.linalg.lsmr</code>	<code>cupyx.scipy.sparse.linalg.lsmr</code>
<code>scipy.sparse.linalg.lsqr</code>	<code>cupyx.scipy.sparse.linalg.lsqr</code>
<code>scipy.sparse.linalg.minres</code>	<code>cupyx.scipy.sparse.linalg.minres</code>
<code>scipy.sparse.linalg.norm</code>	<code>cupyx.scipy.sparse.linalg.norm</code>
<code>scipy.sparse.linalg.onenormest</code>	-
<code>scipy.sparse.linalg.qmr</code>	-
<code>scipy.sparse.linalg.spilu</code>	<code>cupyx.scipy.sparse.linalg.spilu</code>
<code>scipy.sparse.linalg.splu</code>	<code>cupyx.scipy.sparse.linalg.splu</code>
<code>scipy.sparse.linalg.spsolve</code>	<code>cupyx.scipy.sparse.linalg.spsolve</code>

continues on next page

Table 16 – continued from previous page

SciPy	CuPy
<code>scipy.sparse.linalg.spsolve_triangular</code>	<code>cupyx.scipy.sparse.linalg.spsolve_triangular</code>
<code>scipy.sparse.linalg.svds</code>	<code>cupyx.scipy.sparse.linalg.svds</code>
<code>scipy.sparse.linalg.tfqmr</code>	-
<code>scipy.sparse.linalg.use_solver</code>	-

### Compressed sparse graph routines

SciPy	CuPy
<code>scipy.sparse.csgraph.bellman_ford</code>	-
<code>scipy.sparse.csgraph.breadth_first_order</code>	-
<code>scipy.sparse.csgraph.breadth_first_tree</code>	-
<code>scipy.sparse.csgraph.connected_components</code>	<code>cupyx.scipy.sparse.csgraph. connected_components</code>
<code>scipy.sparse.csgraph.construct_dist_matrix</code>	-
<code>scipy.sparse.csgraph.csgraph_from_dense</code>	-
<code>scipy.sparse.csgraph.csgraph_from_masked</code>	-
<code>scipy.sparse.csgraph. csgraph_masked_from_dense</code>	-
<code>scipy.sparse.csgraph.csgraph_to_dense</code>	-
<code>scipy.sparse.csgraph.csgraph_to_masked</code>	-
<code>scipy.sparse.csgraph.depth_first_order</code>	-
<code>scipy.sparse.csgraph.depth_first_tree</code>	-
<code>scipy.sparse.csgraph.dijkstra</code>	-
<code>scipy.sparse.csgraph.floyd_warshall</code>	-
<code>scipy.sparse.csgraph.johnson</code>	-
<code>scipy.sparse.csgraph.laplacian</code>	-
<code>scipy.sparse.csgraph. maximum_bipartite_matching</code>	-
<code>scipy.sparse.csgraph.maximum_flow</code>	-
<code>scipy.sparse.csgraph. min_weight_full_bipartite_matching</code>	-
<code>scipy.sparse.csgraph.minimum_spanning_tree</code>	-
<code>scipy.sparse.csgraph.reconstruct_path</code>	-
<code>scipy.sparse.csgraph.reverse_cuthill_mckee</code>	-
<code>scipy.sparse.csgraph.shortest_path</code>	-
<code>scipy.sparse.csgraph.structural_rank</code>	-

### Special Functions

SciPy	CuPy
<code>scipy.special.agm</code>	-
<code>scipy.special.ai_zeros</code>	-
<code>scipy.special.airy</code>	-
<code>scipy.special.airye</code>	-
<code>scipy.special.assoc_laguerre</code>	-
<code>scipy.special.bdtr</code>	<code>cupyx.scipy.special.bdtr</code>

continues on next page



Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.bdtrc</code>	<code>cupyx.scipy.special.bdtrc</code>
<code>scipy.special.bdtri</code>	<code>cupyx.scipy.special.bdtri</code>
<code>scipy.special.bdtrik</code>	-
<code>scipy.special.bdtrin</code>	-
<code>scipy.special.bei</code>	-
<code>scipy.special.bei_zeros</code>	-
<code>scipy.special.beip</code>	-
<code>scipy.special.beip_zeros</code>	-
<code>scipy.special.ber</code>	-
<code>scipy.special.ber_zeros</code>	-
<code>scipy.special.bernoulli</code>	-
<code>scipy.special.berp</code>	-
<code>scipy.special.berp_zeros</code>	-
<code>scipy.special.besselpoly</code>	-
<code>scipy.special.beta</code>	<code>cupyx.scipy.special.beta</code>
<code>scipy.special.betainc</code>	<code>cupyx.scipy.special.betainc</code>
<code>scipy.special.betaincinv</code>	<code>cupyx.scipy.special.betaincinv</code>
<code>scipy.special.betaln</code>	<code>cupyx.scipy.special.betaln</code>
<code>scipy.special.bi_zeros</code>	-
<code>scipy.special.binom</code>	<code>cupyx.scipy.special.binom</code>
<code>scipy.special.boxcox</code>	<code>cupyx.scipy.special.boxcox</code>
<code>scipy.special.boxcox1p</code>	<code>cupyx.scipy.special.boxcox1p</code>
<code>scipy.special.btdtr</code>	<code>cupyx.scipy.special.btdtr</code>
<code>scipy.special.btdtri</code>	<code>cupyx.scipy.special.btdtri</code>
<code>scipy.special.btdtria</code>	-
<code>scipy.special.btdtrib</code>	-
<code>scipy.special.c_roots</code>	-
<code>scipy.special.cbirt</code>	<code>cupyx.scipy.special.cbirt</code>
<code>scipy.special.cg_roots</code>	-
<code>scipy.special.chdtr</code>	<code>cupyx.scipy.special.chdtr</code>
<code>scipy.special.chdtrc</code>	<code>cupyx.scipy.special.chdtrc</code>
<code>scipy.special.chdtri</code>	<code>cupyx.scipy.special.chdtri</code>
<code>scipy.special.chdtriv</code>	-
<code>scipy.special.chebyc</code>	-
<code>scipy.special.chebys</code>	-
<code>scipy.special.chebyt</code>	-
<code>scipy.special.chebyu</code>	-
<code>scipy.special.chndtr</code>	-
<code>scipy.special.chndtridf</code>	-
<code>scipy.special.chndtrinc</code>	-
<code>scipy.special.chndtrix</code>	-
<code>scipy.special.clpmn</code>	-
<code>scipy.special.comb</code>	-
<code>scipy.special.cosdg</code>	<code>cupyx.scipy.special.cosdg</code>
<code>scipy.special.cosml</code>	<code>cupyx.scipy.special.cosml</code>
<code>scipy.special.cotdg</code>	<code>cupyx.scipy.special.cotdg</code>
<code>scipy.special.dawsn</code>	-
<code>scipy.special.digamma</code>	<code>cupyx.scipy.special.digamma</code>
<code>scipy.special.diric</code>	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.ellip_harm</code>	-
<code>scipy.special.ellip_harm_2</code>	-
<code>scipy.special.ellip_normal</code>	-
<code>scipy.special.ellipe</code>	-
<code>scipy.special.ellipeinc</code>	-
<code>scipy.special.ellipj</code>	<code>cupyx.scipy.special.ellipj</code>
<code>scipy.special.ellipk</code>	<code>cupyx.scipy.special.ellipk</code>
<code>scipy.special.ellipkinc</code>	-
<code>scipy.special.ellipkm1</code>	<code>cupyx.scipy.special.ellipkm1</code>
<code>scipy.special.elliprc</code>	-
<code>scipy.special.elliprd</code>	-
<code>scipy.special.elliprf</code>	-
<code>scipy.special.elliprg</code>	-
<code>scipy.special.elliprj</code>	-
<code>scipy.special.entr</code>	<code>cupyx.scipy.special.entr</code>
<code>scipy.special.erf</code>	<code>cupyx.scipy.special.erf</code>
<code>scipy.special.erf_zeros</code>	-
<code>scipy.special.erfc</code>	<code>cupyx.scipy.special.erfc</code>
<code>scipy.special.erfcinv</code>	<code>cupyx.scipy.special.erfcinv</code>
<code>scipy.special.erfcx</code>	<code>cupyx.scipy.special.erfcx</code>
<code>scipy.special.erfi</code>	-
<code>scipy.special.erfinv</code>	<code>cupyx.scipy.special.erfinv</code>
<code>scipy.special.errstate</code>	-
<code>scipy.special.euler</code>	-
<code>scipy.special.eval_chebyc</code>	-
<code>scipy.special.eval_chebys</code>	-
<code>scipy.special.eval_chebyt</code>	-
<code>scipy.special.eval_chebyu</code>	-
<code>scipy.special.eval_gegenbauer</code>	-
<code>scipy.special.eval_genlaguerre</code>	-
<code>scipy.special.eval_hermite</code>	-
<code>scipy.special.eval_hermitenorm</code>	-
<code>scipy.special.eval_jacobi</code>	-
<code>scipy.special.eval_laguerre</code>	-
<code>scipy.special.eval_legendre</code>	-
<code>scipy.special.eval_sh_chebyt</code>	-
<code>scipy.special.eval_sh_chebyu</code>	-
<code>scipy.special.eval_sh_jacobi</code>	-
<code>scipy.special.eval_sh_legendre</code>	-
<code>scipy.special.exp1</code>	<code>cupyx.scipy.special.exp1</code>
<code>scipy.special.exp10</code>	<code>cupyx.scipy.special.exp10</code>
<code>scipy.special.exp2</code>	<code>cupyx.scipy.special.exp2</code>
<code>scipy.special.expi</code>	<code>cupyx.scipy.special.expi</code>
<code>scipy.special.expit</code>	<code>cupyx.scipy.special.expit</code>
<code>scipy.special.expm1</code>	<code>cupyx.scipy.special.expm1</code>
<code>scipy.special.expn</code>	<code>cupyx.scipy.special.expn</code>
<code>scipy.special.exprel</code>	<code>cupyx.scipy.special.exprel</code>
<code>scipy.special.factorial</code>	-
<code>scipy.special.factorial2</code>	-

continues on next page



Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.factorialk</code>	-
<code>scipy.special.fdtr</code>	<code>cupyx.scipy.special.fdtr</code>
<code>scipy.special.fdtrc</code>	<code>cupyx.scipy.special.fdtrc</code>
<code>scipy.special.fdtri</code>	<code>cupyx.scipy.special.fdtri</code>
<code>scipy.special.fdtroidf</code>	-
<code>scipy.special.fresnel</code>	-
<code>scipy.special.fresnel_zeros</code>	-
<code>scipy.special.fresnelc_zeros</code>	-
<code>scipy.special.fresnels_zeros</code>	-
<code>scipy.special.gamma</code>	<code>cupyx.scipy.special.gamma</code>
<code>scipy.special.gammainc</code>	<code>cupyx.scipy.special.gammainc</code>
<code>scipy.special.gammaincc</code>	<code>cupyx.scipy.special.gammaincc</code>
<code>scipy.special.gammainccinv</code>	<code>cupyx.scipy.special.gammainccinv</code>
<code>scipy.special.gammaincinv</code>	<code>cupyx.scipy.special.gammaincinv</code>
<code>scipy.special.gammaln</code>	<code>cupyx.scipy.special.gammaln</code>
<code>scipy.special.gammasgn</code>	<code>cupyx.scipy.special.gammasgn</code>
<code>scipy.special.gdtr</code>	<code>cupyx.scipy.special.gdtr</code>
<code>scipy.special.gdtrc</code>	<code>cupyx.scipy.special.gdtrc</code>
<code>scipy.special.gdtria</code>	-
<code>scipy.special.gdtrib</code>	-
<code>scipy.special.gdtrix</code>	-
<code>scipy.special.gegenbauer</code>	-
<code>scipy.special.genlaguerre</code>	-
<code>scipy.special.geterr</code>	-
<code>scipy.special.h1vp</code>	-
<code>scipy.special.h2vp</code>	-
<code>scipy.special.h_roots</code>	-
<code>scipy.special.hankel1</code>	-
<code>scipy.special.hankel1e</code>	-
<code>scipy.special.hankel2</code>	-
<code>scipy.special.hankel2e</code>	-
<code>scipy.special.he_roots</code>	-
<code>scipy.special.hermite</code>	-
<code>scipy.special.hermitenorm</code>	-
<code>scipy.special.huber</code>	<code>cupyx.scipy.special.huber</code>
<code>scipy.special.hyp0f1</code>	-
<code>scipy.special.hyp1f1</code>	-
<code>scipy.special.hyp2f1</code>	-
<code>scipy.special.hyperu</code>	-
<code>scipy.special.i0</code>	<code>cupyx.scipy.special.i0</code>
<code>scipy.special.i0e</code>	<code>cupyx.scipy.special.i0e</code>
<code>scipy.special.i1</code>	<code>cupyx.scipy.special.i1</code>
<code>scipy.special.i1e</code>	<code>cupyx.scipy.special.i1e</code>
<code>scipy.special.inv_boxcox</code>	<code>cupyx.scipy.special.inv_boxcox</code>
<code>scipy.special.inv_boxcox1p</code>	<code>cupyx.scipy.special.inv_boxcox1p</code>
<code>scipy.special.it2i0k0</code>	-
<code>scipy.special.it2j0y0</code>	-
<code>scipy.special.it2struve0</code>	-
<code>scipy.special.itairy</code>	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.iti0k0</code>	-
<code>scipy.special.itj0y0</code>	-
<code>scipy.special.itmodstruve0</code>	-
<code>scipy.special.itstruve0</code>	-
<code>scipy.special.iv</code>	-
<code>scipy.special.ive</code>	-
<code>scipy.special.ivp</code>	-
<code>scipy.special.j0</code>	<code>cupyx.scipy.special.j0</code>
<code>scipy.special.j1</code>	<code>cupyx.scipy.special.j1</code>
<code>scipy.special.j_roots</code>	-
<code>scipy.special.jacobi</code>	-
<code>scipy.special.jn</code>	-
<code>scipy.special.jn_zeros</code>	-
<code>scipy.special.jn_jnp_zeros</code>	-
<code>scipy.special.jnp_zeros</code>	-
<code>scipy.special.jnyn_zeros</code>	-
<code>scipy.special.js_roots</code>	-
<code>scipy.special.jv</code>	-
<code>scipy.special.jve</code>	-
<code>scipy.special.jvp</code>	-
<code>scipy.special.k0</code>	<code>cupyx.scipy.special.k0</code>
<code>scipy.special.k0e</code>	<code>cupyx.scipy.special.k0e</code>
<code>scipy.special.k1</code>	<code>cupyx.scipy.special.k1</code>
<code>scipy.special.k1e</code>	<code>cupyx.scipy.special.k1e</code>
<code>scipy.special.kei</code>	-
<code>scipy.special.kei_zeros</code>	-
<code>scipy.special.keip</code>	-
<code>scipy.special.keip_zeros</code>	-
<code>scipy.special.kelvin</code>	-
<code>scipy.special.kelvin_zeros</code>	-
<code>scipy.special.ker</code>	-
<code>scipy.special.ker_zeros</code>	-
<code>scipy.special.kerp</code>	-
<code>scipy.special.kerp_zeros</code>	-
<code>scipy.special.kl_div</code>	<code>cupyx.scipy.special.kl_div</code>
<code>scipy.special.kn</code>	-
<code>scipy.special.kolmogi</code>	-
<code>scipy.special.kolmogorov</code>	-
<code>scipy.special.kv</code>	-
<code>scipy.special.kve</code>	-
<code>scipy.special.kvp</code>	-
<code>scipy.special.l_roots</code>	-
<code>scipy.special.la_roots</code>	-
<code>scipy.special.laguerre</code>	-
<code>scipy.special.lambertw</code>	-
<code>scipy.special.legendre</code>	-
<code>scipy.special.lmbda</code>	-
<code>scipy.special.log1p</code>	<code>cupyx.scipy.special.log1p</code>
<code>scipy.special.log_expit</code>	<code>cupyx.scipy.special.log_expit</code>

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.log_ndtr</code>	<code>cupyx.scipy.special.log_ndtr</code>
<code>scipy.special.log_softmax</code>	<code>cupyx.scipy.special.log_softmax</code>
<code>scipy.special.loggamma</code>	<code>cupyx.scipy.special.loggamma</code>
<code>scipy.special.logit</code>	<code>cupyx.scipy.special.logit</code>
<code>scipy.special.logsumexp</code>	<code>cupyx.scipy.special.logsumexp</code>
<code>scipy.special.lpmn</code>	-
<code>scipy.special.lpmv</code>	<code>cupyx.scipy.special.lpmv</code>
<code>scipy.special.lpn</code>	-
<code>scipy.special.lqmn</code>	-
<code>scipy.special.lqn</code>	-
<code>scipy.special.mathieu_a</code>	-
<code>scipy.special.mathieu_b</code>	-
<code>scipy.special.mathieu_cem</code>	-
<code>scipy.special.mathieu_even_coef</code>	-
<code>scipy.special.mathieu_modcem1</code>	-
<code>scipy.special.mathieu_modcem2</code>	-
<code>scipy.special.mathieu_modsem1</code>	-
<code>scipy.special.mathieu_modsem2</code>	-
<code>scipy.special.mathieu_odd_coef</code>	-
<code>scipy.special.mathieu_sem</code>	-
<code>scipy.special.modfresnelm</code>	-
<code>scipy.special.modfresnelp</code>	-
<code>scipy.special.modstruve</code>	-
<code>scipy.special.multigammaln</code>	<code>cupyx.scipy.special.multigammaln</code>
<code>scipy.special.nbdtr</code>	<code>cupyx.scipy.special.nbdtr</code>
<code>scipy.special.nbdtrc</code>	<code>cupyx.scipy.special.nbdtrc</code>
<code>scipy.special.nbdtri</code>	<code>cupyx.scipy.special.nbdtri</code>
<code>scipy.special.nbdtrik</code>	-
<code>scipy.special.nbdtrin</code>	-
<code>scipy.special.ncfdtr</code>	-
<code>scipy.special.ncfdtri</code>	-
<code>scipy.special.ncfdtridfd</code>	-
<code>scipy.special.ncfdtridfn</code>	-
<code>scipy.special.ncfdtrinc</code>	-
<code>scipy.special.nctdtr</code>	-
<code>scipy.special.nctdtridf</code>	-
<code>scipy.special.nctdtrinc</code>	-
<code>scipy.special.nctdtrit</code>	-
<code>scipy.special.ndtr</code>	<code>cupyx.scipy.special.ndtr</code>
<code>scipy.special.ndtri</code>	<code>cupyx.scipy.special.ndtri</code>
<code>scipy.special.ndtri_exp</code>	-
<code>scipy.special.nrdtrimn</code>	-
<code>scipy.special.nrdtrisd</code>	-
<code>scipy.special.obl_ang1</code>	-
<code>scipy.special.obl_ang1_cv</code>	-
<code>scipy.special.obl_cv</code>	-
<code>scipy.special.obl_cv_seq</code>	-
<code>scipy.special.obl_rad1</code>	-
<code>scipy.special.obl_rad1_cv</code>	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.obl_rad2</code>	-
<code>scipy.special.obl_rad2_cv</code>	-
<code>scipy.special.owens_t</code>	-
<code>scipy.special.p_roots</code>	-
<code>scipy.special.pbdn_seq</code>	-
<code>scipy.special.pbdv</code>	-
<code>scipy.special.pbdv_seq</code>	-
<code>scipy.special.pbvv</code>	-
<code>scipy.special.pbvv_seq</code>	-
<code>scipy.special.pbwa</code>	-
<code>scipy.special.pdtr</code>	<code>cupyx.scipy.special.pdtr</code>
<code>scipy.special.pdtrc</code>	<code>cupyx.scipy.special.pdtrc</code>
<code>scipy.special.pdtri</code>	<code>cupyx.scipy.special.pdtri</code>
<code>scipy.special.pdtrik</code>	-
<code>scipy.special.perm</code>	-
<code>scipy.special.poch</code>	<code>cupyx.scipy.special.poch</code>
<code>scipy.special.polygamma</code>	<code>cupyx.scipy.special.polygamma</code>
<code>scipy.special.powm1</code>	-
<code>scipy.special.pro_ang1</code>	-
<code>scipy.special.pro_ang1_cv</code>	-
<code>scipy.special.pro_cv</code>	-
<code>scipy.special.pro_cv_seq</code>	-
<code>scipy.special.pro_rad1</code>	-
<code>scipy.special.pro_rad1_cv</code>	-
<code>scipy.special.pro_rad2</code>	-
<code>scipy.special.pro_rad2_cv</code>	-
<code>scipy.special.ps_roots</code>	-
<code>scipy.special.pseudo_huber</code>	<code>cupyx.scipy.special.pseudo_huber</code>
<code>scipy.special.psi</code>	<code>cupyx.scipy.special.psi</code>
<code>scipy.special.radian</code>	<code>cupyx.scipy.special.radian</code>
<code>scipy.special.rel_entr</code>	<code>cupyx.scipy.special.rel_entr</code>
<code>scipy.special.rgamma</code>	<code>cupyx.scipy.special.rgamma</code>
<code>scipy.special.riccati_jn</code>	-
<code>scipy.special.riccati_yn</code>	-
<code>scipy.special.roots_chebyc</code>	-
<code>scipy.special.roots_chebys</code>	-
<code>scipy.special.roots_chebyt</code>	-
<code>scipy.special.roots_chebyu</code>	-
<code>scipy.special.roots_gegenbauer</code>	-
<code>scipy.special.roots_genlaguerre</code>	-
<code>scipy.special.roots_hermite</code>	-
<code>scipy.special.roots_hermitenorm</code>	-
<code>scipy.special.roots_jacobi</code>	-
<code>scipy.special.roots_laguerre</code>	-
<code>scipy.special.roots_legendre</code>	-
<code>scipy.special.roots_sh_chebyt</code>	-
<code>scipy.special.roots_sh_chebyu</code>	-
<code>scipy.special.roots_sh_jacobi</code>	-
<code>scipy.special.roots_sh_legendre</code>	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
<code>scipy.special.round</code>	<code>cupyx.scipy.special.round</code>
<code>scipy.special.s_roots</code>	-
<code>scipy.special.seterr</code>	-
<code>scipy.special.sh_chebyt</code>	-
<code>scipy.special.sh_chebyu</code>	-
<code>scipy.special.sh_jacobi</code>	-
<code>scipy.special.sh_legendre</code>	-
<code>scipy.special.shichi</code>	-
<code>scipy.special.sici</code>	-
<code>scipy.special.sinc</code>	<code>cupyx.scipy.special.sinc</code>
<code>scipy.special.sindg</code>	<code>cupyx.scipy.special.sindg</code>
<code>scipy.special.smirnov</code>	-
<code>scipy.special.smirnovi</code>	-
<code>scipy.special.softmax</code>	<code>cupyx.scipy.special.softmax</code>
<code>scipy.special.spence</code>	-
<code>scipy.special.sph_harm</code>	<code>cupyx.scipy.special.sph_harm</code>
<code>scipy.special.spherical_in</code>	-
<code>scipy.special.spherical_jn</code>	-
<code>scipy.special.spherical_kn</code>	-
<code>scipy.special.spherical_yn</code>	<code>cupyx.scipy.special.spherical_yn</code>
<code>scipy.special.stdtr</code>	-
<code>scipy.special.stdtridf</code>	-
<code>scipy.special.stdtrit</code>	-
<code>scipy.special.struve</code>	-
<code>scipy.special.t_roots</code>	-
<code>scipy.special.tandg</code>	<code>cupyx.scipy.special.tandg</code>
<code>scipy.special.tklmbda</code>	-
<code>scipy.special.ts_roots</code>	-
<code>scipy.special.u_roots</code>	-
<code>scipy.special.us_roots</code>	-
<code>scipy.special.voigt_profile</code>	-
<code>scipy.special.wofz</code>	-
<code>scipy.special.wright_bessel</code>	-
<code>scipy.special.wrightomega</code>	-
<code>scipy.special.xlog1py</code>	<code>cupyx.scipy.special.xlog1py</code>
<code>scipy.special.xlogy</code>	<code>cupyx.scipy.special.xlogy</code>
<code>scipy.special.y0</code>	<code>cupyx.scipy.special.y0</code>
<code>scipy.special.y0_zeros</code>	-
<code>scipy.special.y1</code>	<code>cupyx.scipy.special.y1</code>
<code>scipy.special.y1_zeros</code>	-
<code>scipy.special.y1p_zeros</code>	-
<code>scipy.special.yn</code>	<code>cupyx.scipy.special.yn</code>
<code>scipy.special.yn_zeros</code>	-
<code>scipy.special.ynp_zeros</code>	-
<code>scipy.special.yv</code>	-
<code>scipy.special.yve</code>	-
<code>scipy.special.yvp</code>	-
<code>scipy.special.zeta</code>	<code>cupyx.scipy.special.zeta</code>
<code>scipy.special.zetac</code>	<code>cupyx.scipy.special.zetac</code>

## Statistical Functions

SciPy	CuPy
<code>scipy.stats.BootstrapMethod</code>	-
<code>scipy.stats.CensoredData</code>	-
<code>scipy.stats.Covariance</code>	-
<code>scipy.stats.MonteCarloMethod</code>	-
<code>scipy.stats.PermutationMethod</code>	-
<code>scipy.stats.alexandergovern</code>	-
<code>scipy.stats.alpha</code>	-
<code>scipy.stats.anderson</code>	-
<code>scipy.stats.anderson_ksamp</code>	-
<code>scipy.stats.anglit</code>	-
<code>scipy.stats.ansari</code>	-
<code>scipy.stats.arcsine</code>	-
<code>scipy.stats.argus</code>	-
<code>scipy.stats.barnard_exact</code>	-
<code>scipy.stats.bartlett</code>	-
<code>scipy.stats.bayes_mvs</code>	-
<code>scipy.stats.bernoulli</code>	-
<code>scipy.stats.beta</code>	-
<code>scipy.stats.betabinom</code>	-
<code>scipy.stats.betaprime</code>	-
<code>scipy.stats.binned_statistic</code>	-
<code>scipy.stats.binned_statistic_2d</code>	-
<code>scipy.stats.binned_statistic_dd</code>	-
<code>scipy.stats.binom</code>	-
<code>scipy.stats.binom_test</code>	-
<code>scipy.stats.binomtest</code>	-
<code>scipy.stats.boltzmann</code>	-
<code>scipy.stats.bootstrap</code>	-
<code>scipy.stats.boschloo_exact</code>	-
<code>scipy.stats.boxcox</code>	-
<code>scipy.stats.boxcox_llf</code>	<code>cupyx.scipy.stats.boxcox_llf</code>
<code>scipy.stats.boxcox_normmax</code>	-
<code>scipy.stats.boxcox_normplot</code>	-
<code>scipy.stats.bradford</code>	-
<code>scipy.stats.brunnermunzel</code>	-
<code>scipy.stats.burr</code>	-
<code>scipy.stats.burr12</code>	-
<code>scipy.stats.cauchy</code>	-
<code>scipy.stats.chi</code>	-
<code>scipy.stats.chi2</code>	-
<code>scipy.stats.chi2_contingency</code>	-
<code>scipy.stats.chisquare</code>	-
<code>scipy.stats.circmean</code>	-
<code>scipy.stats.circstd</code>	-
<code>scipy.stats.circvar</code>	-
<code>scipy.stats.combine_pvalues</code>	-
<code>scipy.stats.cosine</code>	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
<code>scipy.stats.cramervonmises</code>	-
<code>scipy.stats.cramervonmises_2samp</code>	-
<code>scipy.stats.crystalball</code>	-
<code>scipy.stats.cumfreq</code>	-
<code>scipy.stats.describe</code>	-
<code>scipy.stats.dgamma</code>	-
<code>scipy.stats.differential_entropy</code>	-
<code>scipy.stats.directional_stats</code>	-
<code>scipy.stats.dirichlet</code>	-
<code>scipy.stats.dirichlet_multinomial</code>	-
<code>scipy.stats.dlaplace</code>	-
<code>scipy.stats.dunnett</code>	-
<code>scipy.stats.dweibull</code>	-
<code>scipy.stats.ecdf</code>	-
<code>scipy.stats.energy_distance</code>	-
<code>scipy.stats.entropy</code>	<code>cupyx.scipy.stats.entropy</code>
<code>scipy.stats.epps_singleton_2samp</code>	-
<code>scipy.stats.erlang</code>	-
<code>scipy.stats.expectile</code>	-
<code>scipy.stats.expon</code>	-
<code>scipy.stats.exponnorm</code>	-
<code>scipy.stats.exponpow</code>	-
<code>scipy.stats.exponweib</code>	-
<code>scipy.stats.f</code>	-
<code>scipy.stats.f_oneway</code>	-
<code>scipy.stats.false_discovery_control</code>	-
<code>scipy.stats.fatiguelife</code>	-
<code>scipy.stats.find_repeats</code>	-
<code>scipy.stats.fisher_exact</code>	-
<code>scipy.stats.fisk</code>	-
<code>scipy.stats.fit</code>	-
<code>scipy.stats.fligner</code>	-
<code>scipy.stats.foldcauchy</code>	-
<code>scipy.stats.foldnorm</code>	-
<code>scipy.stats.friedmanchisquare</code>	-
<code>scipy.stats.gamma</code>	-
<code>scipy.stats.gausshyper</code>	-
<code>scipy.stats.gaussian_kde</code>	-
<code>scipy.stats.genexpon</code>	-
<code>scipy.stats.genextreme</code>	-
<code>scipy.stats.gengamma</code>	-
<code>scipy.stats.genhalflogistic</code>	-
<code>scipy.stats.genhyperbolic</code>	-
<code>scipy.stats.geninvgauss</code>	-
<code>scipy.stats.genlogistic</code>	-
<code>scipy.stats.gennorm</code>	-
<code>scipy.stats.genpareto</code>	-
<code>scipy.stats.geom</code>	-
<code>scipy.stats.gibrat</code>	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
<code>scipy.stats.gmean</code>	-
<code>scipy.stats.gompertz</code>	-
<code>scipy.stats.goodness_of_fit</code>	-
<code>scipy.stats.gstd</code>	-
<code>scipy.stats.gumbel_l</code>	-
<code>scipy.stats.gumbel_r</code>	-
<code>scipy.stats.gzscore</code>	-
<code>scipy.stats.halfcauchy</code>	-
<code>scipy.stats.halfgennorm</code>	-
<code>scipy.stats.halflogistic</code>	-
<code>scipy.stats.halfnorm</code>	-
<code>scipy.stats.hmean</code>	-
<code>scipy.stats.hypergeom</code>	-
<code>scipy.stats.hypsecant</code>	-
<code>scipy.stats.invgamma</code>	-
<code>scipy.stats.invgauss</code>	-
<code>scipy.stats.invweibull</code>	-
<code>scipy.stats.invwishart</code>	-
<code>scipy.stats.iqr</code>	-
<code>scipy.stats.jarque_bera</code>	-
<code>scipy.stats.johnsonsb</code>	-
<code>scipy.stats.johnsonsu</code>	-
<code>scipy.stats.kappa3</code>	-
<code>scipy.stats.kappa4</code>	-
<code>scipy.stats.kendalltau</code>	-
<code>scipy.stats.kruskal</code>	-
<code>scipy.stats.ks_1samp</code>	-
<code>scipy.stats.ks_2samp</code>	-
<code>scipy.stats.ksone</code>	-
<code>scipy.stats.kstat</code>	-
<code>scipy.stats.kstatvar</code>	-
<code>scipy.stats.kstest</code>	-
<code>scipy.stats.kstwo</code>	-
<code>scipy.stats.kstwobign</code>	-
<code>scipy.stats.kurtosis</code>	-
<code>scipy.stats.kurtosistest</code>	-
<code>scipy.stats.laplace</code>	-
<code>scipy.stats.laplace_asymmetric</code>	-
<code>scipy.stats.levene</code>	-
<code>scipy.stats.levy</code>	-
<code>scipy.stats.levy_l</code>	-
<code>scipy.stats.levy_stable</code>	-
<code>scipy.stats.linregress</code>	-
<code>scipy.stats.loggamma</code>	-
<code>scipy.stats.logistic</code>	-
<code>scipy.stats.loglaplace</code>	-
<code>scipy.stats.lognorm</code>	-
<code>scipy.stats.logrank</code>	-
<code>scipy.stats.logser</code>	-

continues on next page



Table 18 – continued from previous page

SciPy	CuPy
<code>scipy.stats.loguniform</code>	-
<code>scipy.stats.lomax</code>	-
<code>scipy.stats.mannwhitneyu</code>	-
<code>scipy.stats.matrix_normal</code>	-
<code>scipy.stats.maxwell</code>	-
<code>scipy.stats.median_abs_deviation</code>	-
<code>scipy.stats.median_test</code>	-
<code>scipy.stats.mielke</code>	-
<code>scipy.stats.mode</code>	-
<code>scipy.stats.moment</code>	-
<code>scipy.stats.monte_carlo_test</code>	-
<code>scipy.stats.mood</code>	-
<code>scipy.stats.moyal</code>	-
<code>scipy.stats.multinomial</code>	-
<code>scipy.stats.multiscale_graphcorr</code>	-
<code>scipy.stats.multivariate_hypergeom</code>	-
<code>scipy.stats.multivariate_normal</code>	-
<code>scipy.stats.multivariate_t</code>	-
<code>scipy.stats.mvsdist</code>	-
<code>scipy.stats.nakagami</code>	-
<code>scipy.stats.nbinom</code>	-
<code>scipy.stats.ncf</code>	-
<code>scipy.stats.nchypergeom_fisher</code>	-
<code>scipy.stats.nchypergeom_wallenius</code>	-
<code>scipy.stats.nct</code>	-
<code>scipy.stats.ncx2</code>	-
<code>scipy.stats.nhypergeom</code>	-
<code>scipy.stats.norm</code>	-
<code>scipy.stats.normaltest</code>	-
<code>scipy.stats.norminvgauss</code>	-
<code>scipy.stats.obrientransform</code>	-
<code>scipy.stats.ortho_group</code>	-
<code>scipy.stats.page_trend_test</code>	-
<code>scipy.stats.pareto</code>	-
<code>scipy.stats.pearson3</code>	-
<code>scipy.stats.pearsonr</code>	-
<code>scipy.stats.percentileofscore</code>	-
<code>scipy.stats.permutation_test</code>	-
<code>scipy.stats.planck</code>	-
<code>scipy.stats.pmean</code>	-
<code>scipy.stats.pointbiserialr</code>	-
<code>scipy.stats.poisson</code>	-
<code>scipy.stats.poisson_means_test</code>	-
<code>scipy.stats.power_divergence</code>	-
<code>scipy.stats.powerlaw</code>	-
<code>scipy.stats.powerlognorm</code>	-
<code>scipy.stats.powernorm</code>	-
<code>scipy.stats.ppcc_max</code>	-
<code>scipy.stats.ppcc_plot</code>	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
<code>scipy.stats.probplot</code>	-
<code>scipy.stats.randint</code>	-
<code>scipy.stats.random_correlation</code>	-
<code>scipy.stats.random_table</code>	-
<code>scipy.stats.rankdata</code>	-
<code>scipy.stats.ranksums</code>	-
<code>scipy.stats.rayleigh</code>	-
<code>scipy.stats.rdist</code>	-
<code>scipy.stats.recipinvgauss</code>	-
<code>scipy.stats.reciprocal</code>	-
<code>scipy.stats.rel_breitwigner</code>	-
<code>scipy.stats.relfreq</code>	-
<code>scipy.stats.rice</code>	-
<code>scipy.stats.rv_continuous</code>	-
<code>scipy.stats.rv_discrete</code>	-
<code>scipy.stats.rv_histogram</code>	-
<code>scipy.stats.rvs_ratio_uniforms</code>	-
<code>scipy.stats.scoreatpercentile</code>	-
<code>scipy.stats.sem</code>	-
<code>scipy.stats.semicircular</code>	-
<code>scipy.stats.shapiro</code>	-
<code>scipy.stats.siegelslopes</code>	-
<code>scipy.stats.sigmaclip</code>	-
<code>scipy.stats.skellam</code>	-
<code>scipy.stats.skew</code>	-
<code>scipy.stats.skewcauchy</code>	-
<code>scipy.stats.skewnorm</code>	-
<code>scipy.stats.skewtest</code>	-
<code>scipy.stats.sobol_indices</code>	-
<code>scipy.stats.somersd</code>	-
<code>scipy.stats.spearmanr</code>	-
<code>scipy.stats.special_ortho_group</code>	-
<code>scipy.stats.studentized_range</code>	-
<code>scipy.stats.t</code>	-
<code>scipy.stats.theilslopes</code>	-
<code>scipy.stats.tiecorrect</code>	-
<code>scipy.stats.tmax</code>	-
<code>scipy.stats.tmean</code>	-
<code>scipy.stats.tmin</code>	-
<code>scipy.stats.trapezoid</code>	-
<code>scipy.stats.trapz</code>	-
<code>scipy.stats.triang</code>	-
<code>scipy.stats.trim1</code>	-
<code>scipy.stats.trim_mean</code>	<code>cupyx.scipy.stats.trim_mean</code>
<code>scipy.stats.trimboth</code>	-
<code>scipy.stats.truncexpon</code>	-
<code>scipy.stats.truncnorm</code>	-
<code>scipy.stats.truncpareto</code>	-
<code>scipy.stats.truncweibull_min</code>	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
<code>scipy.stats.tsem</code>	-
<code>scipy.stats.tstd</code>	-
<code>scipy.stats.ttest_1samp</code>	-
<code>scipy.stats.ttest_ind</code>	-
<code>scipy.stats.ttest_ind_from_stats</code>	-
<code>scipy.stats.ttest_rel</code>	-
<code>scipy.stats.tukey_hsd</code>	-
<code>scipy.stats.tukeylambda</code>	-
<code>scipy.stats.tvar</code>	-
<code>scipy.stats.uniform</code>	-
<code>scipy.stats.uniform_direction</code>	-
<code>scipy.stats.unitary_group</code>	-
<code>scipy.stats.variation</code>	-
<code>scipy.stats.vonmises</code>	-
<code>scipy.stats.vonmises_fisher</code>	-
<code>scipy.stats.vonmises_line</code>	-
<code>scipy.stats.wald</code>	-
<code>scipy.stats.wasserstein_distance</code>	-
<code>scipy.stats.weibull_max</code>	-
<code>scipy.stats.weibull_min</code>	-
<code>scipy.stats.weightedtau</code>	-
<code>scipy.stats.wilcoxon</code>	-
<code>scipy.stats.wishart</code>	-
<code>scipy.stats.wrapcauchy</code>	-
<code>scipy.stats.yeojohnson</code>	-
<code>scipy.stats.yeojohnson_llf</code>	-
<code>scipy.stats.yeojohnson_normmax</code>	-
<code>scipy.stats.yeojohnson_normplot</code>	-
<code>scipy.stats.yulesimon</code>	-
<code>scipy.stats.zipf</code>	-
<code>scipy.stats.zipfian</code>	-
<code>scipy.stats.zmap</code>	<code>cupyx.scipy.stats.zmap</code>
<code>scipy.stats.zscore</code>	<code>cupyx.scipy.stats.zscore</code>

## 5.11 Python Array API Support

The [Python array API standard](#) aims to provide a coherent set of APIs for array and tensor libraries developed by the community to build upon. This solves the API fragmentation issue across the community by offering concrete function signatures, semantics and scopes of coverage, enabling writing backend-agnostic codes for better portability.

CuPy provides **experimental** support based on NumPy’s [NEP-47](#), which is in turn based on the v2021 standard. All of the functionalities can be accessed through the `cupy.array_api` namespace.

NumPy’s [Array API Standard Compatibility](#) is an excellent starting point to understand better the differences between the APIs under the main namespace and the `array_api` namespace. Keep in mind, however, that the key difference between NumPy and CuPy is that we are a GPU-only library, therefore CuPy users should be aware of potential [device management](#) issues. Same as in regular CuPy codes, the GPU-to-use can be specified via the `Device` objects, see [Device management](#). GPU-related semantics (e.g. streams, asynchronicity, etc) are also respected. Finally, remember there are already [differences between NumPy and CuPy](#), although some of which are rectified in the standard.

### 5.11.1 Array API Functions

This section is a full list of implemented APIs. For the detailed documentation, see the [array API specification](#).

`cupy.array_api.abs(x, /)`

Array API compatible wrapper for `np.abs`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.acos(x, /)`

Array API compatible wrapper for `np.arccos`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.acosh(x, /)`

Array API compatible wrapper for `np.arccosh`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.add(x1, x2, /)`

Array API compatible wrapper for `np.add`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.all(x, /, *, axis=None, keepdims=False)`

Array API compatible wrapper for `np.all`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.any(x, /, *, axis=None, keepdims=False)`

Array API compatible wrapper for `np.any`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.arange(start, /, stop=None, step=1, *, dtype=None, device=None)`

Array API compatible wrapper for `np.arange`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.argmax(x, /, *, axis=None, keepdims=False)`

Array API compatible wrapper for `np.argmax`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.argmin(x, /, *, axis=None, keepdims=False)`

Array API compatible wrapper for `np.argmin`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.argsort(x, /, *, axis=-1, descending=False, stable=True)`

Array API compatible wrapper for `np.argsort`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.asarray(obj, /, *, dtype=None, device=None, copy=None)`

Array API compatible wrapper for `np.asarray`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.asin(x, /)`

Array API compatible wrapper for `np.arcsin`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.asinh(x, /)`

Array API compatible wrapper for `np.arcsinh`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.atan(x, /)`

Array API compatible wrapper for `np.arctan`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.atan2(x1, x2, /)`

Array API compatible wrapper for `np.arctan2`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.atanh(x, /)`

Array API compatible wrapper for `np.arctanh`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_and(x1, x2, /)`

Array API compatible wrapper for `np.bitwise_and`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_invert(x, /)`

Array API compatible wrapper for `np.invert`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_left_shift(x1, x2, /)`

Array API compatible wrapper for `np.left_shift`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_or(x1, x2, /)`

Array API compatible wrapper for `np.bitwise_or`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_right_shift(x1, x2, /)`

Array API compatible wrapper for `np.right_shift`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.bitwise_xor(x1, x2, /)`

Array API compatible wrapper for `np.bitwise_xor`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.broadcast_arrays(*arrays)`

Array API compatible wrapper for `np.broadcast_arrays`.

See its docstring for more information.

**Return type**

[List](#)[[Array](#)]

`cupy.array_api.broadcast_to(x, /, shape)`

Array API compatible wrapper for `np.broadcast_to`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.can_cast(from_, to, /)`

Array API compatible wrapper for `np.can_cast`.

See its docstring for more information.

**Return type**

`bool`

`cupy.array_api.ceil(x, /)`

Array API compatible wrapper for `np.ceil`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.concat(arrays, /, *, axis=0)`

Array API compatible wrapper for `np.concatenate`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.cos(x, /)`

Array API compatible wrapper for `np.cos`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.cosh(x, /)`

Array API compatible wrapper for `np.cosh`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.divide(x1, x2, /)`

Array API compatible wrapper for `np.divide`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.empty(shape, *, dtype=None, device=None)`

Array API compatible wrapper for `np.empty`.

See its docstring for more information.

**Return type**

`Array`

`cupy.array_api.empty_like(x, /, *, dtype=None, device=None)`

Array API compatible wrapper for `np.empty_like`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.equal(x1, x2, /)`

Array API compatible wrapper for `np.equal`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.exp(x, /)`

Array API compatible wrapper for `np.exp`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.expand_dims(x, /, *, axis)`

Array API compatible wrapper for `np.expand_dims`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.expm1(x, /)`

Array API compatible wrapper for `np.expm1`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.eye(n_rows, n_cols=None, /, *, k=0, dtype=None, device=None)`

Array API compatible wrapper for `np.eye`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.finfo(type, /)`

Array API compatible wrapper for `np.finfo`.

See its docstring for more information.

**Return type**

`finfo_object`

`cupy.array_api.flip(x, /, *, axis=None)`

Array API compatible wrapper for `np.flip`.

See its docstring for more information.

**Return type**

*Array*



`cupy.array_api.floor(x, /)`

Array API compatible wrapper for `np.floor`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.floor_divide(x1, x2, /)`

Array API compatible wrapper for `np.floor_divide`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.from_dlpack(x, /)`

Array API compatible wrapper for `np.from_dlpack`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.full(shape, fill_value, *, dtype=None, device=None)`

Array API compatible wrapper for `np.full`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.full_like(x, /, fill_value, *, dtype=None, device=None)`

Array API compatible wrapper for `np.full_like`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.greater(x1, x2, /)`

Array API compatible wrapper for `np.greater`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.greater_equal(x1, x2, /)`

Array API compatible wrapper for `np.greater_equal`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.iinfo(type, /)`

Array API compatible wrapper for `np.iinfo`.

See its docstring for more information.

**Return type**

`iinfo_object`

`cupy.array_api.isfinite(x, /)`

Array API compatible wrapper for `np.isfinite`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.isinf(x, /)`

Array API compatible wrapper for `np.isinf`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.isnan(x, /)`

Array API compatible wrapper for `np.isnan`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.less(x1, x2, /)`

Array API compatible wrapper for `np.less`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.less_equal(x1, x2, /)`

Array API compatible wrapper for `np.less_equal`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.linspace(start, stop, /, num, *, dtype=None, device=None, endpoint=True)`

Array API compatible wrapper for [np.linspace](#).

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.log(x, /)`

Array API compatible wrapper for `np.log`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.log10(x, /)`

Array API compatible wrapper for `np.log10`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.log1p(x, /)`

Array API compatible wrapper for `np.log1p`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.log2(x, /)`

Array API compatible wrapper for `np.log2`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.logaddexp(x1, x2)`

Array API compatible wrapper for `np.logaddexp`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.logical_and(x1, x2, /)`

Array API compatible wrapper for `np.logical_and`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.logical_not(x, /)`

Array API compatible wrapper for `np.logical_not`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.logical_or(x1, x2, /)`

Array API compatible wrapper for `np.logical_or`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.logical_xor(x1, x2, /)`

Array API compatible wrapper for `np.logical_xor`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.matmul(x1, x2, /)`

Array API compatible wrapper for `np.matmul`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.meshgrid(*arrays, indexing='xy')`  
Array API compatible wrapper for `np.meshgrid`.  
See its docstring for more information.

**Return type**  
`List[Array]`

`cupy.array_api.multiply(x1, x2, /)`  
Array API compatible wrapper for `np.multiply`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.negative(x, /)`  
Array API compatible wrapper for `np.negative`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.nonzero(x, /)`  
Array API compatible wrapper for `np.nonzero`.  
See its docstring for more information.

**Return type**  
`Tuple[Array, ...]`

`cupy.array_api.not_equal(x1, x2, /)`  
Array API compatible wrapper for `np.not_equal`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.ones(shape, *, dtype=None, device=None)`  
Array API compatible wrapper for `np.ones`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.ones_like(x, /, *, dtype=None, device=None)`  
Array API compatible wrapper for `np.ones_like`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.permute_dims(x, /, axes)`  
Array API compatible wrapper for `np.transpose`.  
See its docstring for more information.

**Return type**  
`Array`

`cupy.array_api.positive(x, /)`

Array API compatible wrapper for `np.positive`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.pow(x1, x2, /)`

Array API compatible wrapper for `np.power`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.remainder(x1, x2, /)`

Array API compatible wrapper for `np.remainder`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.reshape(x, /, shape)`

Array API compatible wrapper for `np.reshape`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.result_type(*arrays_and_dtypes)`

Array API compatible wrapper for `np.result_type`.

See its docstring for more information.

**Return type**

[Dtype](#)

`cupy.array_api.roll(x, /, shift, *, axis=None)`

Array API compatible wrapper for `np.roll`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.round(x, /)`

Array API compatible wrapper for `np.round`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.sign(x, /)`

Array API compatible wrapper for `np.sign`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.sin(x, /)`

Array API compatible wrapper for `np.sin`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.sinh(x, /)`

Array API compatible wrapper for `np.sinh`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.sort(x, /, *, axis=-1, descending=False, stable=True)`

Array API compatible wrapper for `np.sort`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.sqrt(x, /)`

Array API compatible wrapper for `np.sqrt`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.square(x, /)`

Array API compatible wrapper for `np.square`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.squeeze(x, /, axis)`

Array API compatible wrapper for `np.squeeze`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.stack(arrays, /, *, axis=0)`

Array API compatible wrapper for `np.stack`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.subtract(x1, x2, /)`

Array API compatible wrapper for `np.subtract`.

See its docstring for more information.

**Return type**

[Array](#)

`cupy.array_api.take(x, indices, /, *, axis)`

Array API compatible wrapper for `np.take`. See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.tan(x, /)`

Array API compatible wrapper for `np.tan`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.tanh(x, /)`

Array API compatible wrapper for `np.tanh`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.tril(x, /, *, k=0)`

Array API compatible wrapper for `np.tril`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.triu(x, /, *, k=0)`

Array API compatible wrapper for `np.triu`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.trunc(x, /)`

Array API compatible wrapper for `np.trunc`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.unique_all(x, /)`

Array API compatible wrapper for `np.unique`.

See its docstring for more information.

**Return type**

*UniqueAllResult*

`cupy.array_api.unique_inverse(x, /)`

Array API compatible wrapper for `np.unique`.

See its docstring for more information.

**Return type**

*UniqueInverseResult*

`cupy.array_api.unique_values(x, /)`

Array API compatible wrapper for `np.unique`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.where(condition, x1, x2, /)`

Array API compatible wrapper for `np.where`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.zeros(shape, *, dtype=None, device=None)`

Array API compatible wrapper for `np.zeros`.

See its docstring for more information.

**Return type**

*Array*

`cupy.array_api.zeros_like(x, /, *, dtype=None, device=None)`

Array API compatible wrapper for `np.zeros_like`.

See its docstring for more information.

**Return type**

*Array*

## 5.11.2 Array API Compliant Object

*Array* is a wrapper class built upon `cupy.ndarray` to enforce strict compliance with the array API standard. See the [documentation](#) for detail.

This object should not be constructed directly. Rather, use one of the [creation functions](#), such as `cupy.array_api.asarray()`.

---

<code>Array(*args, **kwargs)</code>
-------------------------------------

n-d array object for the array API namespace.
---

---

### `cupy.array_api._array_object.Array`

**class** `cupy.array_api._array_object.Array(*args, **kwargs)`

n-d array object for the array API namespace.

See the docstring of `np.ndarray` for more information.

This is a wrapper around `numpy.ndarray` that restricts the usage to only those things that are required by the array API namespace. Note, attributes on this object that start with a single underscore are not part of the API specification and should only be used internally. This object should not be constructed directly. Rather, use one of the creation functions, such as `asarray()`.



## Methods

**`__getitem__`**(*key*, /)

Performs the operation `__getitem__`.

**Return type**

*Array*

**`__setitem__`**(*key*, *value*, /)

Performs the operation `__setitem__`.

**`to_device`**(*device*, /, *stream*=None)

**Return type**

*Array*

**`__eq__`**(*other*, /)

Performs the operation `__eq__`.

**Return type**

*Array*

**`__ne__`**(*other*, /)

Performs the operation `__ne__`.

**Return type**

*Array*

**`__lt__`**(*other*, /)

Performs the operation `__lt__`.

**Return type**

*Array*

**`__le__`**(*other*, /)

Performs the operation `__le__`.

**Return type**

*Array*

**`__gt__`**(*other*, /)

Performs the operation `__gt__`.

**Return type**

*Array*

**`__ge__`**(*other*, /)

Performs the operation `__ge__`.

**Return type**

*Array*

**`__bool__`**()

Performs the operation `__bool__`.

**Return type**

*bool*

## Attributes

**T**

Array API compatible wrapper for `np.ndarray.T`.

See its docstring for more information.

**device**

**dtype**

Array API compatible wrapper for `np.ndarray.dtype`.

See its docstring for more information.

**mT**

**ndim**

Array API compatible wrapper for `np.ndarray.ndim`.

See its docstring for more information.

**shape**

Array API compatible wrapper for `np.ndarray.shape`.

See its docstring for more information.

**size**

Array API compatible wrapper for `np.ndarray.size`.

See its docstring for more information.

## CONTRIBUTION GUIDE

This is a guide for all contributions to CuPy. The development of CuPy is running on [the official repository at GitHub](#). Anyone that wants to register an issue or to send a pull request should read through this document.

### 6.1 Classification of Contributions

There are several ways to contribute to CuPy community:

1. Registering an issue
2. Sending a pull request (PR)
3. Sending a question to [CuPy's Gitter channel](#), [CuPy User Group](#), or [StackOverflow](#)
4. Open-sourcing an external example
5. Writing a post about CuPy

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

### 6.2 Development Cycle

This section explains the development process of CuPy. Before contributing to CuPy, it is strongly recommended to understand the development cycle.

#### 6.2.1 Versioning

The versioning of CuPy follows [PEP 440](#) and a part of [Semantic versioning](#). The version number consists of three or four parts: `X.Y.Zw` where `X` denotes the **major version**, `Y` denotes the **minor version**, `Z` denotes the **revision number**, and the optional `w` denotes the pre-release suffix. While the major, minor, and revision numbers follow the rule of semantic versioning, the pre-release suffix follows PEP 440 so that the version string is much friendly with Python eco-system.

**Note that a major update basically does not contain compatibility-breaking changes from the last release candidate (RC).** This is not a strict rule, though; if there is a critical API bug that we have to fix for the major version, we may add breaking changes to the major version up.

As for the backward compatibility, see [API Compatibility Policy](#).

## 6.2.2 Release Cycle

The first one is the track of **stable versions**, which is a series of revision updates for the latest major version. The second one is the track of **development versions**, which is a series of pre-releases for the upcoming major version.

Consider that  $X.0.0$  is the latest major version and  $Y.0.0$ ,  $Z.0.0$  are the succeeding major versions. Then, the timeline of the updates is depicted by the following table.

Date	ver X	ver Y	ver Z
0 weeks	$X.0.0rc1$	–	–
4 weeks	$X.0.0$	$Y.0.0a1$	–
8 weeks	$X.1.0^*$	$Y.0.0b1$	–
12 weeks	$X.2.0^*$	$Y.0.0rc1$	–
16 weeks	–	$Y.0.0$	$Z.0.0a1$

(\* These might be revision releases)

The dates shown in the left-most column are relative to the release of  $X.0.0rc1$ . In particular, each revision/minor release is made four weeks after the previous one of the same major version, and the pre-release of the upcoming major version is made at the same time. Whether these releases are revision or minor is determined based on the contents of each update.

Note that there are only three stable releases for the versions  $X.x.x$ . During the parallel development of  $Y.0.0$  and  $Z.0.0a1$ , the version Y is treated as an **almost-stable version** and Z is treated as a development version.

If there is a critical bug found in  $X.x.x$  after stopping the development of version X, we may release a hot-fix for this version at any time.

We create a milestone for each upcoming release at GitHub. The GitHub milestone is basically used for collecting the issues and PRs resolved in the release.

## 6.2.3 Git Branches

The **main** branch is used to develop pre-release versions. It means that **alpha, beta, and RC updates are developed at the main branch**. This branch contains the most up-to-date source tree that includes features newly added after the latest major version.

The stable version is developed at the individual branch named as  $vN$  where “N” reflects the version number (we call it a *versioned branch*). For example,  $v1.0.0$ ,  $v1.0.1$ , and  $v1.0.2$  will be developed at the  $v1$  branch.

**Notes for contributors:** When you send a pull request, you basically have to send it to the **main** branch. If the change can also be applied to the stable version, a core team member will apply the same change to the stable version so that the change is also included in the next revision update.

If the change is only applicable to the stable version and not to the **main** branch, please send it to the versioned branch. We basically only accept changes to the latest versioned branch (where the stable version is developed) unless the fix is critical.

If you want to make a new feature of the **main** branch available in the current stable version, please send a *backport PR* to the stable version (the latest  $vN$  branch). See the next section for details.

*Note: a change that can be applied to both branches should be sent to the **main** branch. Each release of the stable version is also merged to the development version so that the change is also reflected to the next major version.*

### 6.2.4 Feature Backport PRs

We basically do not backport any new features of the development version to the stable versions. If you desire to include the feature to the current stable version and you can work on the backport work, we welcome such a contribution. In such a case, you have to send a backport PR to the latest vN branch. **Note that we do not accept any feature backport PRs to older versions because we are not running quality assurance workflows (e.g. CI) for older versions so that we cannot ensure that the PR is correctly ported.**

There are some rules on sending a backport PR.

- Start the PR title from the prefix **[backport]**.
- Clarify the original PR number in the PR description (something like “This is a backport of #XXXX”).
- (optional) Write to the PR description the motivation of backporting the feature to the stable version.

Please follow these rules when you create a feature backport PR.

Note: PRs that do not include any changes/additions to APIs (e.g. bug fixes, documentation improvements) are usually backported by core dev members. It is also appreciated to make such a backport PR by any contributors, though, so that the overall development proceeds more smoothly!

## 6.3 Issues and Pull Requests

In this section, we explain how to send pull requests (PRs).

### 6.3.1 How to Send a Pull Request

If you can write code to fix an issue, we encourage to send a PR.

First of all, before starting to write any code, do not forget to confirm the following points.

- Read through the *Coding Guidelines* and *Unit Testing*.
- Check the appropriate branch that you should send the PR following *Git Branches*. If you do not have any idea about selecting a branch, please choose the `main` branch.

In particular, **check the branch before writing any code**. The current source tree of the chosen branch is the starting point of your change.

After writing your code (**including unit tests and hopefully documentations!**), send a PR on GitHub. You have to write a precise explanation of **what** and **how** you fix; it is the first documentation of your code that developers read, which is a very important part of your PR.

Once you send a PR, it is automatically tested on GitHub Actions. After the automatic test passes, core developers will start reviewing your code. Note that this automatic PR test only includes CPU tests.

---

**Note:** We are also running continuous integration with GPU tests for the `main` branch and the versioned branch of the latest major version. Since this service is currently running on our internal server, we do not use it for automatic PR tests to keep the server secure.

---

If you are planning to add a new feature or modify existing APIs, **it is recommended to open an issue and discuss the design first**. The design discussion needs lower cost for the core developers than code review. Following the consequences of the discussions, you can send a PR that is smoothly reviewed in a shorter time.

Even if your code is not complete, you can send a pull request as a *work-in-progress PR* by putting the [WIP] prefix to the PR title. If you write a precise explanation about the PR, core developers and other contributors can join the discussion about how to proceed the PR. WIP PR is also useful to have discussions based on a concrete code.

## 6.4 Coding Guidelines

---

**Note:** Coding guidelines are updated at v5.0. Those who have contributed to older versions should read the guidelines again.

---

We use [PEP8](#) and a part of [OpenStack Style Guidelines](#) related to general coding style as our basic style guidelines.

You can use `autopep8` and `flake8` commands to check your code.

In order to avoid confusion from using different tool versions, we pin the versions of those tools. Install them with the following command (from within the top directory of CuPy repository):

```
$ pip install -e '[stylecheck]'
```

And check your code with:

```
$ autopep8 path/to/your/code.py
$ flake8 path/to/your/code.py
```

To check Cython code, use `.flake8.cython` configuration file:

```
$ flake8 --config=.flake8.cython path/to/your/cython/code.pyx
```

The `autopep8` supports automatically correct Python code to conform to the PEP 8 style guide:

```
$ autopep8 --in-place path/to/your/code.py
```

The `flake8` command lets you know the part of your code not obeying our style guidelines. Before sending a pull request, be sure to check that your code passes the `flake8` checking.

Note that `flake8` command is not perfect. It does not check some of the style guidelines. Here is a (not-complete) list of the rules that `flake8` cannot check.

- Relative imports are prohibited. [H304]
- Importing non-module symbols is prohibited.
- Import statements must be organized into three parts: standard libraries, third-party libraries, and internal imports. [H306]

In addition, we restrict the usage of *shortcut symbols* in our code base. They are symbols imported by packages and sub-packages of `cupy`. For example, `cupy.cuda.Device` is a shortcut of `cupy.cuda.device.Device`. **It is not allowed to use such shortcuts in the ``cupy`` library implementation.** Note that you can still use them in `tests` and `examples` directories.

Once you send a pull request, your coding style is automatically checked by *GitHub Actions*. The reviewing process starts after the check passes.

The CuPy is designed based on NumPy's API design. CuPy's source code and documents contain the original NumPy ones. Please note the followings when writing the document.

- In order to identify overlapping parts, it is preferable to add some remarks that this document is just copied or altered from the original one. It is also preferable to briefly explain the specification of the function in a short paragraph, and refer to the corresponding function in NumPy so that users can read the detailed document. However, it is possible to include a complete copy of the document with such a remark if users cannot summarize in such a way.
- If a function in CuPy only implements a limited amount of features in the original one, users should explicitly describe only what is implemented in the document.

For changes that modify or add new Cython files, please make sure the pointer types follow these guidelines ([#1913](#)).

- Pointers should be `void*` if only used within Cython, or `intptr_t` if exposed to the Python space.
- Memory sizes should be `size_t`.
- Memory offsets should be `ptrdiff_t`.

---

**Note:** We are incrementally enforcing the above rules, so some existing code may not follow the above guidelines, but please ensure all new contributions do.

---

## 6.5 Unit Testing

Testing is one of the most important part of your code. You must write test cases and verify your implementation by following our testing guide.

Note that we are using `pytest` and `mock` package for testing, so install them before writing your code:

```
$ pip install pytest mock
```

### 6.5.1 How to Run Tests

In order to run unit tests at the repository root, you first have to build Cython files in place by running the following command:

```
$ pip install -e .
```

---

**Note:** When you modify `*.pxd` files, before running `pip install -e .`, you must clean `*.cpp` and `*.so` files once with the following command, because Cython does not automatically rebuild those files nicely:

```
$ git clean -fdx
```

---

Once Cython modules are built, you can run unit tests by running the following command at the repository root:

```
$ python -m pytest
```

CUDA must be installed to run unit tests.

Some GPU tests require cuDNN to run. In order to skip unit tests that require cuDNN, specify `-m='not cudnn'` option:

```
$ python -m pytest path/to/your/test.py -m='not cudnn'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs to `CUPY_TEST_GPU_LIMIT`. For example, if you have only one GPU, launch `pytest` by the following command to skip multi-GPU tests:

```
$ export CUPY_TEST_GPU_LIMIT=1
$ python -m pytest path/to/gpu/test.py
```

Following this naming convention, you can run all the tests by running the following command at the repository root:

```
$ python -m pytest
```

Or you can also specify a root directory to search test scripts from:

```
$ python -m pytest tests/cupy_tests      # to just run tests of CuPy
$ python -m pytest tests/install_tests  # to just run tests of installation modules
```

If you modify the code related to existing unit tests, you must run appropriate commands.

## 6.5.2 Test File and Directory Naming Conventions

Tests are put into the `tests/cupy_tests` directory. In order to enable test runner to find test scripts correctly, we are using special naming convention for the test subdirectories and the test scripts.

- The name of each subdirectory of `tests` must end with the `_tests` suffix.
- The name of each test script must start with the `test_` prefix.

When we write a test for a module, we use the appropriate path and file name for the test script whose correspondence to the tested module is clear. For example, if you want to write a test for a module `cupy.x.y.z`, the test script must be located at `tests/cupy_tests/x_tests/y_tests/test_z.py`.

## 6.5.3 How to Write Tests

There are many examples of unit tests under the `tests` directory, so reading some of them is a good and recommended way to learn how to write tests for CuPy. They simply use the `unittest` package of the standard library, while some tests are using utilities from `cupy.testing`.

In addition to the *Coding Guidelines* mentioned above, the following rules are applied to the test code:

- All test classes must inherit from `unittest.TestCase`.
- Use `unittest` features to write tests, except for the following cases:
  - Use `assert` statement instead of `self.assert*` methods (e.g., write `assert x == 1` instead of `self.assertEqual(x, 1)`).
  - Use `with pytest.raises(...):` instead of with `self.assertRaises(...):`.

---

**Note:** We are incrementally applying the above style. Some existing tests may be using the old style (`self.assertRaises`, etc.), but all newly written tests should follow the above style.

---

In order to write tests for multiple GPUs, use `cupy.testing.multi_gpu()` decorators instead:



```
import unittest
from cupy import testing

class TestMyFunc(unittest.TestCase):
    ...

    @testing.multi_gpu(2) # specify the number of required GPUs here
    def test_my_two_gpu_func(self):
        ...
```

If your test requires too much time, add `cupy.testing.slow` decorator. The test functions decorated by `slow` are skipped if `-m='not slow'` is given:

```
import unittest
from cupy import testing

class TestMyFunc(unittest.TestCase):
    ...

    @testing.slow
    def test_my_slow_func(self):
        ...
```

Once you send a pull request, GitHub Actions automatically checks if your code meets our coding guidelines described above. Since GitHub Actions does not support CUDA, we cannot run unit tests automatically. The reviewing process starts after the automatic check passes. Note that reviewers will test your code without the option to check CUDA-related code.

---

**Note:** Some of numerically unstable tests might cause errors irrelevant to your changes. In such a case, we ignore the failures and go on to the review process, so do not worry about it!

---

## 6.6 Documentation

When adding a new feature to the framework, you also need to document it in the reference.

---

**Note:** If you are unsure about how to fix the documentation, you can submit a pull request without doing so. Reviewers will help you fix the documentation appropriately.

---

The documentation source is stored under `docs` directory and written in `reStructuredText` format.

To build the documentation, you need to install `Sphinx`:

```
$ pip install -r docs/requirements.txt
```

Then you can build the documentation in HTML format locally:

```
$ cd docs
$ make html
```

HTML files are generated under `build/html` directory. Open `index.html` with the browser and see if it is rendered as expected.

---

**Note:** Docstrings (documentation comments in the source code) are collected from the installed CuPy module. If you modified docstrings, make sure to install the module (e.g., using `pip install -e .`) before building the documentation.

---

## 6.7 Tips for Developers

Here are some tips for developers hacking CuPy source code.

### 6.7.1 Install as Editable

During the development we recommend using `pip` with `-e` option to install as editable mode:

```
$ pip install -e .
```

Please note that even with `-e`, you will have to rerun `pip install -e .` to regenerate C++ sources using Cython if you modified Cython source files (e.g., `*.pyx` files).

### 6.7.2 Use ccache

NVCC environment variable can be specified at the build time to use the custom command instead of `nvcc`. You can speed up the rebuild using `ccache` (v3.4 or later) by:

```
$ export NVCC='ccache nvcc'
```

### 6.7.3 Limit Architecture

Use `CUPY_NVCC_GENERATE_CODE` environment variable to reduce the build time by limiting the target CUDA architectures. For example, if you only run your CuPy build with NVIDIA P100 and V100, you can use:

```
$ export CUPY_NVCC_GENERATE_CODE=arch=compute_60,code=sm_60;arch=compute_70,code=sm_70
```

See [Environment variables](#) for the description.

## UPGRADE GUIDE

This page covers changes introduced in each major version that users should know when migrating from older releases. Please see also the [Compatibility Matrix](#) for supported environments of each major version.

### 7.1 CuPy v13

#### 7.1.1 Modernized CCCL support and requirement

NVIDIA's CUDA C++ Core Libraries (CCCL) is the new home for the inter-dependent C++ libraries Thrust, CUB, and libcu++ that are shipped with CUDA Toolkit 11.0+. To better serve our users with the latest CCCL features, improvements, and bug fixes, starting CuPy v13 we bundle CCCL in the source and binary (pip/conda) releases of CuPy. The same version of CCCL is used at both build-time (for building CuPy) and run-time (for JIT-compiling kernels). This ensures uniform behavior, avoids surprises, and allows dual CUDA support as promised by CCCL (currently CUDA 11 & 12), but this change leads to the following consequences distinct from the past releases:

- after the upgrade, the very first time of executing certain CuPy features may take longer than usual;
- the CCCL from any local CUDA installation is now ignored on purpose, either at build- or run- time;
- adventurous users who want to experiment with local CCCL changes need to update the CCCL submodule and build CuPy from source;

As a result of this movement, CuPy now follows the same compiler requirement as CCCL (and, in turn, CUDA Toolkit) and requires C++11 as the lowest C++ standard. CCCL expects to move to C++17 in the near future.

#### 7.1.2 Requirement Changes

The following versions are no longer supported in CuPy v13.

- CUDA 11.1 or earlier
- cuDNN 8.7 or earlier
- **cuTENSOR 1.x**
  - Support for cuTENSOR 2.0 is added starting with CuPy v13, and support for cuTENSOR 1.x will be dropped. This is because there are significant API changes from cuTENSOR 1.x to 2.0, and from the maintenance perspective, it is not practical to support both cuTENSOR 1.x and 2.0 APIs simultaneously.
- Python 3.8 or earlier
- NumPy 1.21 or earlier
- Ubuntu 18.04

### 7.1.3 NumPy/SciPy Baseline API Update

Baseline API has been bumped from NumPy 1.24 and SciPy 1.9 to NumPy 1.26 and SciPy 1.11. CuPy v13 will follow the upstream products' specifications of these baseline versions.

### 7.1.4 Change in `cupy.asnumpy()/cupy.ndarray.get()` Behavior

When transferring a CuPy array from GPU to CPU (as a NumPy array), previously the transfer could be nonblocking and not properly ordered when a non-default stream is in use, leading to potential data race if the resulting array is modified on host immediately after the copy starts. In CuPy v13, the default behavior is changed to be always blocking, with a new optional argument `blocking` added to allow the previous nonblocking behavior if set to `False`, in which case users are responsible for ensuring proper stream order.

### 7.1.5 Change in `cupy.array()/cupy.asarray()/cupy.asanyarray()` Behavior

When transferring a NumPy array from CPU to GPU, previously the transfer was always blocking even if the source array is backed by pinned memory. In CuPy v13, the default behavior is changed to be asynchronous if the source array is allocated as pinned to improve the performance.

A new optional argument `blocking` has been added to allow the previous blocking behavior if set to `True`. You might want to set this option in case there is a possibility of overwriting the source array on CPU before the transfer completes.

### 7.1.6 Removal of `cupy-wheel` package

The `cupy-wheel` package, which aimed to serve as a “meta” package that chooses and installs the right CuPy binary packages for the users' environment, has been removed in CuPy v13. This is because the recent Pip no longer allows changing requirements dynamically. See [#7628](#) for the details.

### 7.1.7 API Changes

- An *internal and undocumented* API `cupy.cuda.compile_with_cache()`, which was marked deprecated in CuPy v10, has been removed. We encourage downstream libraries and users to migrate to use public APIs, such as [RawModule](#) (added in CuPy v7) or [RawKernel](#) (added in CuPy v5). See [User-Defined Kernels](#) for their tutorials.

### 7.1.8 CUDA Runtime API is now statically linked

CuPy is now shipped with CUDA Runtime statically linked. Due to this, `cupy.cuda.runtime.runtimeGetVersion()` always returns the version of CUDA Runtime that CuPy is built with, regardless of the version of CUDA Runtime installed locally. If you need to retrieve the version of CUDA Runtime shared library installed locally, use `cupy.cuda.get_local_runtime_version()` instead.

## 7.1.9 Update of Docker Images

CuPy official Docker images (see *Installation* for details) are now updated to use CUDA 12.2.

## 7.2 CuPy v12

### 7.2.1 Change in `cupy.cuda.Device` Behavior

The CUDA current device (set via `cupy.cuda.Device.use()` or `cudaSetDevice()`) will be reactivated when exiting a device context manager. This reverts the *change introduced in CuPy v10*, making the behavior identical to the one in CuPy v9 or earlier.

This decision was made for better interoperability with other libraries that might mutate the current CUDA device. Suppose the following code:

```
def do_preprocess_cupy():
    with cupy.cuda.Device(2):
        # ...
        pass

torch.cuda.set_device(1)
do_preprocess_cupy()
print(torch.cuda.get_device()) # -> ???
```

In CuPy v10 and v11, the code prints 0, which can be surprising for users. In CuPy v12, the code now prints 1, making it easy for both users and library developers to maintain the current device where multiple devices are involved.

### 7.2.2 Deprecation of `cupy.ndarray.scatter_{add,max,min}`

These APIs have been marked as deprecated as `cupy.{add,maximum,minimum}.at` ufunc methods have been implemented, which behave as equivalent and NumPy-compatible.

### 7.2.3 Requirement Changes

The following versions are no longer supported in CuPy v12.

- Python 3.7 or earlier
- NumPy 1.20 or earlier
- SciPy 1.6 or earlier

## 7.2.4 Baseline API Update

Baseline API has been bumped from NumPy 1.23 and SciPy 1.8 to NumPy 1.24 and SciPy 1.9. CuPy v12 will follow the upstream products' specifications of these baseline versions.

## 7.2.5 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 11.8.

# 7.3 CuPy v11

## 7.3.1 Unified Binary Package for CUDA 11.2+

CuPy v11 provides a unified binary package named `cupy-cuda11x` that supports all CUDA 11.2+ releases. This replaces per-CUDA version binary packages (`cupy-cuda112 ~ cupy-cuda117`).

Note that CUDA 11.1 or earlier still requires per-CUDA version binary packages. `cupy-cuda102`, `cupy-cuda110`, and `cupy-cuda111` will be provided for CUDA 10.2, 11.0, and 11.1, respectively.

## 7.3.2 Requirement Changes

The following versions are no longer supported in CuPy v11.

- ROCm 4.2 or earlier
- NumPy 1.19 or earlier
- SciPy 1.5 or earlier

## 7.3.3 CUB Enabled by Default

CuPy v11 accelerates the computation with CUB by default. In case needed, you can turn it off by setting `CUPY_ACCELERATORS` environment variable to `""`.

## 7.3.4 Baseline API Update

Baseline API has been bumped from NumPy 1.21 and SciPy 1.7 to NumPy 1.23 and SciPy 1.8. CuPy v11 will follow the upstream products' specifications of these baseline versions.

## 7.3.5 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 11.7 and ROCm 5.0.

## 7.4 CuPy v10

### 7.4.1 Dropping CUDA 9.2 / 10.0 / 10.1 Support

CUDA 10.1 or earlier is no longer supported. Use CUDA 10.2 or later.

### 7.4.2 Dropping NCCL v2.4 / v2.6 / v2.7 Support

NCCL v2.4, v2.6, and v2.7 are no longer supported.

### 7.4.3 Dropping Python 3.6 Support

Python 3.6 is no longer supported.

### 7.4.4 Dropping NumPy 1.17 Support

NumPy 1.17 is no longer supported.

### 7.4.5 Change in `cupy.cuda.Device` Behavior

**Current device set via `use()` will not be honored by the `with Device` block**

---

**Note:** This change has been reverted in CuPy v12. See **CuPy v12** section above for details.

---

The current device set via `cupy.cuda.Device.use()` will not be reactivated when exiting a device context manager. An existing code mixing `with device:` block and `device.use()` may get different results between CuPy v10 and v9.

```
cupy.cuda.Device(1).use()
with cupy.cuda.Device(0):
    pass
cupy.cuda.Device() # -> CuPy v10 returns device 0 instead of device 1
```

This decision was made to serve CuPy *users* better, but it could lead to surprises to downstream *developers* depending on CuPy, as essentially CuPy's *Device* context manager no longer respects the CUDA `cudaSetDevice()` API. Mixing device management functionalities (especially using context manager) from different libraries is highly discouraged.

For downstream libraries that still wish to respect the `cudaGetDevice()/cudaSetDevice()` APIs, you should avoid managing current devices using the `with Device` context manager, and instead calling these APIs explicitly, see for example [cupy/cupy#5963](#).

## 7.4.6 Changes in `cupy.cuda.Stream` Behavior

### Stream is now managed per-device

Previously, it was users' responsibility to keep the current stream consistent with the current CUDA device. For example, the following code raises an error in CuPy v9 or earlier:

```
import cupy

with cupy.cuda.Device(0):
    # Create a stream on device 0.
    s0 = cupy.cuda.Stream()

with cupy.cuda.Device(1):
    with s0:
        # Try to use the stream on device 1
        cupy.arange(10)  # -> CUDA_ERROR_INVALID_HANDLE: invalid resource handle
```

CuPy v10 manages the current stream per-device, thus eliminating the need of switching the stream every time the active device is changed. When using CuPy v10, the above example behaves differently because whenever a stream is created, it is automatically associated with the current device and will be ignored when switching devices. In early versions, trying to use `s0` in device 1 raises an error because `s0` is associated with device 0. However, in v10, this `s0` is ignored and the default stream for device 1 will be used instead.

### Current stream set via `use()` will not be restored when exiting with block

Samely as the change of `cupy.cuda.Device` above, the current stream set via `cupy.cuda.Stream.use()` will not be reactivated when exiting a stream context manager. An existing code mixing with `stream: block` and `stream.use()` may get different results between CuPy v10 and v9.

```
s1 = cupy.cuda.Stream()
s2 = cupy.cuda.Stream()
s3 = cupy.cuda.Stream()
with s1:
    s2.use()
    with s3:
        pass
    cupy.cuda.get_current_stream()  # -> CuPy v10 returns `s1` instead of `s2`.
```

### Streams can now be shared between threads

The same `cupy.cuda.Stream` instance can now safely be shared between multiple threads.

To achieve this, CuPy v10 will not destroy the stream (`cudaStreamDestroy`) if the stream is the current stream of any thread.



### 7.4.7 Big-Endian Arrays Automatically Converted to Little-Endian

`cupy.array()`, `cupy.asarray()` and its variants now always transfer the data to GPU in little-endian byte order.

Previously CuPy was copying the given `numpy.ndarray` to GPU as-is, regardless of the endianness. In CuPy v10, big-endian arrays are converted to little-endian before the transfer, which is the native byte order on GPUs. This change eliminates the need to manually change the array endianness before creating the CuPy array.

### 7.4.8 Baseline API Update

Baseline API has been bumped from NumPy 1.20 and SciPy 1.6 to NumPy 1.21 and SciPy 1.7. CuPy v10 will follow the upstream products' specifications of these baseline versions.

### 7.4.9 API Changes

- Device synchronize detection APIs (`cupyx.allow_synchronize()` and `cupyx.DeviceSynchronized`), introduced as an experimental feature in CuPy v8, have been marked as deprecated because it is impossible to detect synchronizations reliably.
- An *internal* API `cupy.cuda.compile_with_cache()` has been marked as deprecated as there are better alternatives (see [RawModule](#) added since CuPy v7 and [RawKernel](#) since v5). While it has a longstanding history, this API has never been meant to be public. We encourage downstream libraries and users to migrate to the aforementioned public APIs. See [User-Defined Kernels](#) for their tutorials.
- The DLPack routine `cupy.fromDlpack()` is deprecated in favor of `cupy.from_dlpack()`, which addresses potential data race issues.
- A new module `cupyx.profiler` is added to host all profiling related APIs in CuPy. Accordingly, the following APIs are relocated to this module as follows. The old routines are deprecated.
  - `cupy.prof.TimeRangeDecorator()` -> `cupyx.profiler.time_range()`
  - `cupy.prof.time_range()` -> `cupyx.profiler.time_range()`
  - `cupy.cuda.profile()` -> `cupyx.profiler.profile()`
  - `cupyx.time.repeat()` -> `cupyx.profiler.benchmark()`
- `cupy.ndarray.__pos__()` now returns a copy (samely as `cupy.positive()`) instead of returning `self`.

Note that deprecated APIs may be removed in the future CuPy releases.

### 7.4.10 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 11.4 and ROCm 4.3.

## 7.5 CuPy v9

### 7.5.1 Dropping Support of CUDA 9.0

CUDA 9.0 is no longer supported. Use CUDA 9.2 or later.

### 7.5.2 Dropping Support of cuDNN v7.5 and NCCL v2.3

cuDNN v7.5 (or earlier) and NCCL v2.3 (or earlier) are no longer supported.

### 7.5.3 Dropping Support of NumPy 1.16 and SciPy 1.3

NumPy 1.16 and SciPy 1.3 are no longer supported.

### 7.5.4 Dropping Support of Python 3.5

Python 3.5 is no longer supported in CuPy v9.

### 7.5.5 NCCL and cuDNN No Longer Included in Wheels

NCCL and cuDNN shared libraires are no longer included in wheels (see [#4850](#) for discussions). You can manually install them after installing wheel if you don't have a previous installation; see [Installation](#) for details.

### 7.5.6 cuTENSOR Enabled in Wheels

cuTENSOR can now be used when installing CuPy via wheels.

### 7.5.7 `cupy.cuda.{nccl,cudnn}` Modules Needs Explicit Import

Previously `cupy.cuda.nccl` and `cupy.cuda.cudnn` modules were automatically imported. Since CuPy v9, these modules need to be explicitly imported (i.e., `import cupy.cuda.nccl` / `import cupy.cuda.cudnn`.)

### 7.5.8 Baseline API Update

Baseline API has been bumped from NumPy 1.19 and SciPy 1.5 to NumPy 1.20 and SciPy 1.6. CuPy v9 will follow the upstream products' specifications of these baseline versions.

Following NumPy 1.20, aliases for the Python scalar types (`cupy.bool`, `cupy.int`, `cupy.float`, and `cupy.complex`) are now deprecated. `cupy.bool_`, `cupy.int_`, `cupy.float_` and `cupy.complex_` should be used instead when required.

### 7.5.9 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 11.2 and Python 3.8.

## 7.6 CuPy v8

### 7.6.1 Dropping Support of CUDA 8.0 and 9.1

CUDA 8.0 and 9.1 are no longer supported. Use CUDA 9.0, 9.2, 10.0, or later.

### 7.6.2 Dropping Support of NumPy 1.15 and SciPy 1.2

NumPy 1.15 (or earlier) and SciPy 1.2 (or earlier) are no longer supported.

### 7.6.3 Update of Docker Images

- CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 10.2 and Python 3.6.
- SciPy and Optuna are now pre-installed.

### 7.6.4 CUB Support and Compiler Requirement

CUB module is now built by default. You can enable the use of CUB by setting `CUPY_ACCELERATORS="cub"` (see [CUPY\\_ACCELERATORS](#) for details).

Due to this change, g++-6 or later is required when building CuPy from the source. See [Installation](#) for details.

The following environment variables are no longer effective:

- `CUB_DISABLED`: Use [CUPY\\_ACCELERATORS](#) as aforementioned.
- `CUB_PATH`: No longer required as CuPy uses either the CUB source bundled with CUDA (only when using CUDA 11.0 or later) or the one in the CuPy distribution.

### 7.6.5 API Changes

- `cupy.scatter_add`, which was deprecated in CuPy v4, has been removed. Use `cupyx.scatter_add()` instead.
- `cupy.sparse` module has been deprecated and will be removed in future releases. Use `cupyx.scipy.sparse` instead.
- `dtype` argument of `cupy.ndarray.min()` and `cupy.ndarray.max()` has been removed to align with the NumPy specification.
- `cupy.allclose()` now returns the result as 0-dim GPU array instead of Python bool to avoid device synchronization.
- `cupy.RawModule` now delays the compilation to the time of the first call to align the behavior with `cupy.RawKernel`.
- `cupy.cuda.*_enabled` flags (`nccl_enabled`, `nvtx_enabled`, etc.) has been deprecated. Use `cupy.cuda.*.available` flag (`cupy.cuda.nccl.available`, `cupy.cuda.nvtx.available`, etc.) instead.

- CHAINER\_SEED environment variable is no longer effective. Use CUPY\_SEED instead.

## 7.7 CuPy v7

### 7.7.1 Dropping Support of Python 2.7 and 3.4

Starting from CuPy v7, Python 2.7 and 3.4 are no longer supported as it reaches its end-of-life (EOL) in January 2020 (2.7) and March 2019 (3.4). Python 3.5.1 is the minimum Python version supported by CuPy v7. Please upgrade the Python version if you are using affected versions of Python to any later versions listed under [Installation](#).

## 7.8 CuPy v6

### 7.8.1 Binary Packages Ignore LD\_LIBRARY\_PATH

Prior to CuPy v6, LD\_LIBRARY\_PATH environment variable can be used to override cuDNN / NCCL libraries bundled in the binary distribution (also known as wheels). In CuPy v6, LD\_LIBRARY\_PATH will be ignored during discovery of cuDNN / NCCL; CuPy binary distributions always use libraries that comes with the package to avoid errors caused by unexpected override.

## 7.9 CuPy v5

### 7.9.1 cupyx.scipy Namespace

`cupyx.scipy` namespace has been introduced to provide CUDA-enabled SciPy functions. `cupy.sparse` module has been renamed to `cupyx.scipy.sparse`; `cupy.sparse` will be kept as an alias for backward compatibility.

### 7.9.2 Dropped Support for CUDA 7.0 / 7.5

CuPy v5 no longer supports CUDA 7.0 / 7.5.

### 7.9.3 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 9.2 and cuDNN 7.

To use these images, you may need to upgrade the NVIDIA driver on your host. See [Requirements of nvidia-docker](#) for details.

## 7.10 CuPy v4

**Note:** The version number has been bumped from v2 to v4 to align with the versioning of Chainer. Therefore, CuPy v3 does not exist.

### 7.10.1 Default Memory Pool

Prior to CuPy v4, memory pool was only enabled by default when CuPy is used with Chainer. In CuPy v4, memory pool is now enabled by default, even when you use CuPy without Chainer. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

**Attention:** When you monitor GPU memory usage (e.g., using `nvidia-smi`), you may notice that GPU memory not being freed even after the array instance become out of scope. This is expected behavior, as the default memory pool “caches” the allocated memory blocks.

To access the default memory pool instance, use `get_default_memory_pool()` and `get_default_pinned_memory_pool()`. You can access the statistics and free all unused memory blocks “cached” in the memory pool.

```
import cupy
a = cupy.ndarray(100, dtype=cupy.float32)
mempool = cupy.get_default_memory_pool()

# For performance, the size of actual allocation may become larger than the requested
# array size.
print(mempool.used_bytes()) # 512
print(mempool.total_bytes()) # 512

# Even if the array goes out of scope, its memory block is kept in the pool.
a = None
print(mempool.used_bytes()) # 0
print(mempool.total_bytes()) # 512

# You can clear the memory block by calling `free_all_blocks`.
mempool.free_all_blocks()
print(mempool.used_bytes()) # 0
print(mempool.total_bytes()) # 0
```

You can even disable the default memory pool by the code below. Be sure to do this before any other CuPy operations.

```
import cupy
cupy.cuda.set_allocator(None)
cupy.cuda.set_pinned_memory_allocator(None)
```

## 7.10.2 Compute Capability

CuPy v4 now requires NVIDIA GPU with Compute Capability 3.0 or larger. See the [List of CUDA GPUs](#) to check if your GPU supports Compute Capability 3.0.

## 7.10.3 CUDA Stream

As CUDA Stream is fully supported in CuPy v4, `cupy.cuda.RandomState.set_stream`, the function to change the stream used by the random number generator, has been removed. Please use `cupy.cuda.Stream.use()` instead.

See the discussion in [#306](#) for more details.

## 7.10.4 cupyx Namespace

`cupyx` namespace has been introduced to provide features specific to CuPy (i.e., features not provided in NumPy) while avoiding collision in future. See *CuPy-specific functions* for the list of such functions.

For this rule, `cupy.scatter_add()` has been moved to `cupyx.scatter_add()`. `cupy.scatter_add()` is still available as an alias, but it is encouraged to use `cupyx.scatter_add()` instead.

## 7.10.5 Update of Docker Images

CuPy official Docker images (see [Installation](#) for details) are now updated to use CUDA 8.0 and cuDNN 6.0. This change was introduced because CUDA 7.5 does not support NVIDIA Pascal GPUs.

To use these images, you may need to upgrade the NVIDIA driver on your host. See [Requirements of nvidia-docker](#) for details.

# 7.11 CuPy v2

## 7.11.1 Changed Behavior of `count_nonzero` Function

For performance reasons, `cupy.count_nonzero()` has been changed to return zero-dimensional ndarray instead of `int` when `axis=None`. See the discussion in [#154](#) for more details.

## 7.12 Compatibility Matrix

CuPy	CC <sup>1</sup>	CUDA	ROCm	cuTENSOR	NCCL	cuDNN	Python	NumPy	SciPy	Baseline API Spec.	Docs
v13	3.5~	11.2~	4.3~	2.0~	2.16~	8.8~	3.9~	1.22~	1.7~	NumPy 1.26 & SciPy 1.11	latest
v12	3.0~9.0	10.2~12.x	4.3 & 5.0	1.4~1.7	2.8~2.17	7.6~8.8	3.8~3.12	1.21~1.26	1.1~1.11	NumPy 1.24 & SciPy 1.9	stable
v11	3.0~9.0	10.2~12.0	4.3 & 5.0	1.4~1.6	2.8~2.16	7.6~8.7	3.7~3.11	1.20~1.24	1.0~1.9	NumPy 1.23 & SciPy 1.8	v11.6.0
v10	3.0~8.x	10.2~11.7	4.0 & 4.2 & 4.3 & 5.0	1.3~1.5	2.8~2.11	7.6~8.4	3.7~3.10	1.18~1.22	1.0~1.8	NumPy 1.21 & SciPy 1.7	v10.6.0
v9	3.0~8.x	9.2~11.5	3.5~4.3	1.2~1.3	2.4 & 2.6~2.11	7.6~8.3	3.6~3.9	1.17~1.21	1.0~1.7	NumPy 1.20 & SciPy 1.6	v9.6.0
v8	3.0~8.x	9.0 & 9.2~11.2	3.x <sup>2</sup>	1.2	2.0~2.8	7.0~8.3	3.5~3.9	1.16~1.20	1.0~1.6	NumPy 1.19 & SciPy 1.5	v8.6.0
v7	3.0~8.x	8.0~11.0	2.x <sup>2</sup>	1.0	1.3~2.7	5.0~8.0	3.5~3.8	1.9~1.19	not specified	(not specified)	v7.8.0
v6	3.0~7.x	8.0~10.1	n/a	n/a	1.3~2.4	5.0~7.5	2.7 & 3.4~3.8	1.9~1.17	not specified	(not specified)	v6.7.0
v5	3.0~7.x	8.0~10.1	n/a	n/a	1.3~2.4	5.0~7.5	2.7 & 3.4~3.7	1.9~1.16	not specified	(not specified)	v5.4.0
v4	3.0~7.x	7.0~9.2	n/a	n/a	1.3~2.2	4.0~7.1	2.7 & 3.4~3.6	1.9~1.14	not specified	(not specified)	v4.5.0

<sup>1</sup> CUDA Compute Capability

<sup>2</sup> Highly experimental support with limited features.





## LICENSE

Copyright (c) 2015 Preferred Infrastructure, Inc.

Copyright (c) 2015 Preferred Networks, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 8.1 NumPy

The CuPy is designed based on NumPy’s API. CuPy’s source code and documents contain the original NumPy ones.

Copyright (c) 2005-2016, NumPy Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 8.2 SciPy

The CuPy is designed based on SciPy's API. CuPy's source code and documents contain the original SciPy ones.

Copyright (c) 2001, 2002 Enthought, Inc.

All rights reserved.

Copyright (c) 2003-2016 SciPy Developers.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of Enthought nor the names of the SciPy Developers may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 8.3 cuSignal

Part of CuPy's signal processing routines and their documentation are ported from [RAPIDS cuSignal](#).

Copyright (c) 2019-2023 NVIDIA CORPORATION & AFFILIATES. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## PYTHON MODULE INDEX

### C

- `cupy`, ??
- `cupy.array_api`, 896
- `cupy.fft`, 138
- `cupy.linalg`, 173
- `cupy.polynomial.polynomial`, 217
- `cupy.polynomial.polyutils`, 220
- `cupy.random`, 227
- `cupy.testing`, 301
- `cupyx.distributed`, 836
- `cupyx.distributed.array`, 841
- `cupyx.optimizing`, 747
- `cupyx.scipy`, 318
- `cupyx.scipy.fft`, 318
- `cupyx.scipy.fftpack`, 340
- `cupyx.scipy.interpolate`, 347
- `cupyx.scipy.linalg`, 392
- `cupyx.scipy.ndimage`, 405
- `cupyx.scipy.signal`, 455
- `cupyx.scipy.signal.windows`, 596
- `cupyx.scipy.sparse`, 624
- `cupyx.scipy.sparse.csgraph`, 690
- `cupyx.scipy.sparse.linalg`, 676
- `cupyx.scipy.spatial`, 691
- `cupyx.scipy.spatial.distance`, 692
- `cupyx.scipy.special`, 700
- `cupyx.scipy.stats`, 733



## Symbols

- `_JitRawKernel` (class in `cupyx.jit._interface`), 834
- `__bool__()` (`cupy.array_api._array_object.Array` method), 909
- `__bool__()` (`cupy.ndarray` method), 68
- `__bool__()` (`cupyx.distributed.array.DistributedArray` method), 846
- `__bool__()` (`cupyx.scipy.sparse.coo_matrix` method), 633
- `__bool__()` (`cupyx.scipy.sparse.csc_matrix` method), 643
- `__bool__()` (`cupyx.scipy.sparse.csr_matrix` method), 653
- `__bool__()` (`cupyx.scipy.sparse.dia_matrix` method), 660
- `__bool__()` (`cupyx.scipy.sparse.spmatrix` method), 665
- `__call__()` (`cupy.ElementwiseKernel` method), 814
- `__call__()` (`cupy.RawKernel` method), 818
- `__call__()` (`cupy.ReductionKernel` method), 816
- `__call__()` (`cupy.poly1d` method), 222
- `__call__()` (`cupy.ufunc` method), 73
- `__call__()` (`cupy.vectorize` method), 151
- `__call__()` (`cupyx.GeneralizedUFunc` method), 97
- `__call__()` (`cupyx.jit._interface._JitRawKernel` method), 834
- `__call__()` (`cupyx.profiler.time_range` method), 746
- `__call__()` (`cupyx.scipy.interpolate.Akima1DInterpolator` method), 363
- `__call__()` (`cupyx.scipy.interpolate.BPoly` method), 372
- `__call__()` (`cupyx.scipy.interpolate.BSpline` method), 377
- `__call__()` (`cupyx.scipy.interpolate.BarycentricInterpolator` method), 349
- `__call__()` (`cupyx.scipy.interpolate.CubicHermiteSpline` method), 354
- `__call__()` (`cupyx.scipy.interpolate.KroghInterpolator` method), 350
- `__call__()` (`cupyx.scipy.interpolate.NdPPoly` method), 390
- `__call__()` (`cupyx.scipy.interpolate.PPoly` method), 367
- `__call__()` (`cupyx.scipy.interpolate.PchipInterpolator` method), 358
- `__call__()` (`cupyx.scipy.interpolate.RBFInterpolator` method), 383
- `__call__()` (`cupyx.scipy.interpolate.RegularGridInterpolator` method), 388
- `__call__()` (`cupyx.scipy.signal.CZT` method), 593
- `__call__()` (`cupyx.scipy.signal.ZoomFFT` method), 595
- `__call__()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677
- `__copy__()` (`cupy.ndarray` method), 59
- `__copy__()` (`cupyx.distributed.array.DistributedArray` method), 842
- `__enter__()` (`cupy.cuda.Device` method), 749
- `__enter__()` (`cupy.cuda.ExternalStream` method), 783
- `__enter__()` (`cupy.cuda.MemoryHook` method), 773
- `__enter__()` (`cupy.cuda.Stream` method), 780
- `__enter__()` (`cupy.cuda.memory_hooks.DebugPrintHook` method), 775
- `__enter__()` (`cupy.cuda.memory_hooks.LineProfileHook` method), 777
- `__enter__()` (`cupy.fft.config.set_cufft_callbacks` method), 149
- `__enter__()` (`cupyx.profiler.time_range` method), 746
- `__eq__()` (`cupy.ElementwiseKernel` method), 814
- `__eq__()` (`cupy.RawKernel` method), 818
- `__eq__()` (`cupy.RawModule` method), 822
- `__eq__()` (`cupy.ReductionKernel` method), 816
- `__eq__()` (`cupy.array_api._array_object.Array` method), 909
- `__eq__()` (`cupy.broadcast` method), 119
- `__eq__()` (`cupy.cuda.CFunctionAllocator` method), 771
- `__eq__()` (`cupy.cuda.Device` method), 749
- `__eq__()` (`cupy.cuda.Event` method), 787
- `__eq__()` (`cupy.cuda.ExternalStream` method), 785
- `__eq__()` (`cupy.cuda.Graph` method), 788
- `__eq__()` (`cupy.cuda.ManagedMemory` method), 755
- `__eq__()` (`cupy.cuda.Memory` method), 752
- `__eq__()` (`cupy.cuda.MemoryAsync` method), 753
- `__eq__()` (`cupy.cuda.MemoryAsyncPool` method), 768
- `__eq__()` (`cupy.cuda.MemoryHook` method), 774
- `__eq__()` (`cupy.cuda.MemoryPointer` method), 760

- `__eq__()` (*cupy.cuda.MemoryPool* method), 766
- `__eq__()` (*cupy.cuda.PinnedMemory* method), 757
- `__eq__()` (*cupy.cuda.PinnedMemoryPointer* method), 761
- `__eq__()` (*cupy.cuda.PinnedMemoryPool* method), 769
- `__eq__()` (*cupy.cuda.PythonFunctionAllocator* method), 770
- `__eq__()` (*cupy.cuda.Stream* method), 782
- `__eq__()` (*cupy.cuda.UnownedMemory* method), 756
- `__eq__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 776
- `__eq__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 778
- `__eq__()` (*cupy.cuda.nccl.NcclCommunicator* method), 799
- `__eq__()` (*cupy.cuda.texture.CUDAarray* method), 791
- `__eq__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 789
- `__eq__()` (*cupy.cuda.texture.ResourceDescriptor* method), 792
- `__eq__()` (*cupy.cuda.texture.SurfaceObject* method), 795
- `__eq__()` (*cupy.cuda.texture.TextureDescriptor* method), 794
- `__eq__()` (*cupy.cuda.texture.TextureObject* method), 794
- `__eq__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 149
- `__eq__()` (*cupy.flatiter* method), 168
- `__eq__()` (*cupy.ndarray* method), 68
- `__eq__()` (*cupy.poly1d* method), 222
- `__eq__()` (*cupy.random.BitGenerator* method), 237
- `__eq__()` (*cupy.random.Generator* method), 236
- `__eq__()` (*cupy.random.MRG32k3a* method), 239
- `__eq__()` (*cupy.random.Philox4x3210* method), 241
- `__eq__()` (*cupy.random.RandomState* method), 250
- `__eq__()` (*cupy.random.XORWOW* method), 238
- `__eq__()` (*cupy.ufunc* method), 74
- `__eq__()` (*cupy.vectorize* method), 151
- `__eq__()` (*cupyx.GeneralizedUFunc* method), 97
- `__eq__()` (*cupyx.distributed.NCCLBackend* method), 840
- `__eq__()` (*cupyx.distributed.array.DistributedArray* method), 846
- `__eq__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 835
- `__eq__()` (*cupyx.profiler.time\_range* method), 746
- `__eq__()` (*cupyx.scipy.interpolate.Akima1DInterpolator* method), 366
- `__eq__()` (*cupyx.scipy.interpolate.BPoly* method), 375
- `__eq__()` (*cupyx.scipy.interpolate.BSpline* method), 379
- `__eq__()` (*cupyx.scipy.interpolate.BarycentricInterpolator* method), 349
- `__eq__()` (*cupyx.scipy.interpolate.CubicHermiteSpline* method), 357
- `__eq__()` (*cupyx.scipy.interpolate.KroghInterpolator* method), 351
- `__eq__()` (*cupyx.scipy.interpolate.NdPPoly* method), 392
- `__eq__()` (*cupyx.scipy.interpolate.PPoly* method), 370
- `__eq__()` (*cupyx.scipy.interpolate.PchipInterpolator* method), 361
- `__eq__()` (*cupyx.scipy.interpolate.RBFInterpolator* method), 383
- `__eq__()` (*cupyx.scipy.interpolate.RegularGridInterpolator* method), 389
- `__eq__()` (*cupyx.scipy.signal.CZT* method), 593
- `__eq__()` (*cupyx.scipy.signal.StateSpace* method), 540
- `__eq__()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `__eq__()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `__eq__()` (*cupyx.scipy.signal.ZoomFFT* method), 595
- `__eq__()` (*cupyx.scipy.signal.dlti* method), 550
- `__eq__()` (*cupyx.scipy.signal.lti* method), 539
- `__eq__()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `__eq__()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `__eq__()` (*cupyx.scipy.sparse.csr\_matrix* method), 653
- `__eq__()` (*cupyx.scipy.sparse.dia\_matrix* method), 660
- `__eq__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 677
- `__eq__()` (*cupyx.scipy.sparse.linalg.SuperLU* method), 690
- `__eq__()` (*cupyx.scipy.sparse.spmatrix* method), 665
- `__exit__()` (*cupy.cuda.Device* method), 749
- `__exit__()` (*cupy.cuda.ExternalStream* method), 783
- `__exit__()` (*cupy.cuda.MemoryHook* method), 773
- `__exit__()` (*cupy.cuda.Stream* method), 780
- `__exit__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 775
- `__exit__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 777
- `__exit__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 149
- `__exit__()` (*cupyx.profiler.time\_range* method), 746
- `__ge__()` (*cupy.ElementwiseKernel* method), 814
- `__ge__()` (*cupy.RawKernel* method), 818
- `__ge__()` (*cupy.RawModule* method), 822
- `__ge__()` (*cupy.ReductionKernel* method), 816
- `__ge__()` (*cupy.array\_api.\_array\_object.Array* method), 909
- `__ge__()` (*cupy.broadcast* method), 119
- `__ge__()` (*cupy.cuda.CFunctionAllocator* method), 771
- `__ge__()` (*cupy.cuda.Device* method), 750
- `__ge__()` (*cupy.cuda.Event* method), 787
- `__ge__()` (*cupy.cuda.ExternalStream* method), 785
- `__ge__()` (*cupy.cuda.Graph* method), 788
- `__ge__()` (*cupy.cuda.ManagedMemory* method), 755



- `__ge__()` (*cupy.cuda.Memory* method), 753
- `__ge__()` (*cupy.cuda.MemoryAsync* method), 754
- `__ge__()` (*cupy.cuda.MemoryAsyncPool* method), 769
- `__ge__()` (*cupy.cuda.MemoryHook* method), 774
- `__ge__()` (*cupy.cuda.MemoryPointer* method), 760
- `__ge__()` (*cupy.cuda.MemoryPool* method), 766
- `__ge__()` (*cupy.cuda.PinnedMemory* method), 757
- `__ge__()` (*cupy.cuda.PinnedMemoryPointer* method), 761
- `__ge__()` (*cupy.cuda.PinnedMemoryPool* method), 770
- `__ge__()` (*cupy.cuda.PythonFunctionAllocator* method), 770
- `__ge__()` (*cupy.cuda.Stream* method), 782
- `__ge__()` (*cupy.cuda.UnownedMemory* method), 756
- `__ge__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 776
- `__ge__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 779
- `__ge__()` (*cupy.cuda.nccl.NcclCommunicator* method), 799
- `__ge__()` (*cupy.cuda.texture.CUDAarray* method), 791
- `__ge__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 790
- `__ge__()` (*cupy.cuda.texture.ResourceDescriptor* method), 793
- `__ge__()` (*cupy.cuda.texture.SurfaceObject* method), 795
- `__ge__()` (*cupy.cuda.texture.TextureDescriptor* method), 794
- `__ge__()` (*cupy.cuda.texture.TextureObject* method), 795
- `__ge__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 149
- `__ge__()` (*cupy.flatiter* method), 168
- `__ge__()` (*cupy.ndarray* method), 68
- `__ge__()` (*cupy.poly1d* method), 223
- `__ge__()` (*cupy.random.BitGenerator* method), 237
- `__ge__()` (*cupy.random.Generator* method), 236
- `__ge__()` (*cupy.random.MRG32k3a* method), 240
- `__ge__()` (*cupy.random.Philox4x3210* method), 241
- `__ge__()` (*cupy.random.RandomState* method), 250
- `__ge__()` (*cupy.random.XORWOW* method), 239
- `__ge__()` (*cupy.ufunc* method), 74
- `__ge__()` (*cupy.vectorize* method), 152
- `__ge__()` (*cupyx.GeneralizedUFunc* method), 98
- `__ge__()` (*cupyx.distributed.NCCLBackend* method), 840
- `__ge__()` (*cupyx.distributed.array.DistributedArray* method), 846
- `__ge__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 835
- `__ge__()` (*cupyx.profiler.time\_range* method), 746
- `__ge__()` (*cupyx.scipy.interpolate.Akima1DInterpolator* method), 366
- `__ge__()` (*cupyx.scipy.interpolate.BPoly* method), 375
- `__ge__()` (*cupyx.scipy.interpolate.BSpline* method), 379
- `__ge__()` (*cupyx.scipy.interpolate.BarycentricInterpolator* method), 350
- `__ge__()` (*cupyx.scipy.interpolate.CubicHermiteSpline* method), 357
- `__ge__()` (*cupyx.scipy.interpolate.KroghInterpolator* method), 351
- `__ge__()` (*cupyx.scipy.interpolate.NdPPoly* method), 392
- `__ge__()` (*cupyx.scipy.interpolate.PPoly* method), 371
- `__ge__()` (*cupyx.scipy.interpolate.PchipInterpolator* method), 362
- `__ge__()` (*cupyx.scipy.interpolate.RBFInterpolator* method), 384
- `__ge__()` (*cupyx.scipy.interpolate.RegularGridInterpolator* method), 389
- `__ge__()` (*cupyx.scipy.signal.CZT* method), 594
- `__ge__()` (*cupyx.scipy.signal.StateSpace* method), 541
- `__ge__()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `__ge__()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `__ge__()` (*cupyx.scipy.signal.ZoomFFT* method), 595
- `__ge__()` (*cupyx.scipy.signal.dlti* method), 550
- `__ge__()` (*cupyx.scipy.signal.lti* method), 539
- `__ge__()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `__ge__()` (*cupyx.scipy.sparse.csc\_matrix* method), 643
- `__ge__()` (*cupyx.scipy.sparse.csr\_matrix* method), 653
- `__ge__()` (*cupyx.scipy.sparse.dia\_matrix* method), 660
- `__ge__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 677
- `__ge__()` (*cupyx.scipy.sparse.linalg.SuperLU* method), 690
- `__ge__()` (*cupyx.scipy.sparse.spmatrix* method), 665
- `__getitem__()` (*cupy.array\_api.\_array\_object.Array* method), 909
- `__getitem__()` (*cupy.flatiter* method), 168
- `__getitem__()` (*cupy.ndarray* method), 58
- `__getitem__()` (*cupy.poly1d* method), 222
- `__getitem__()` (*cupyx.distributed.array.DistributedArray* method), 842
- `__getitem__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 834
- `__getitem__()` (*cupyx.scipy.sparse.csc\_matrix* method), 634
- `__getitem__()` (*cupyx.scipy.sparse.csr\_matrix* method), 644
- `__gt__()` (*cupy.ElementwiseKernel* method), 814
- `__gt__()` (*cupy.RawKernel* method), 818
- `__gt__()` (*cupy.RawModule* method), 822
- `__gt__()` (*cupy.ReductionKernel* method), 816
- `__gt__()` (*cupy.array\_api.\_array\_object.Array* method), 909

- `__gt__()` (*cupy.broadcast method*), 119
- `__gt__()` (*cupy.cuda.CFunctionAllocator method*), 771
- `__gt__()` (*cupy.cuda.Device method*), 750
- `__gt__()` (*cupy.cuda.Event method*), 787
- `__gt__()` (*cupy.cuda.ExternalStream method*), 785
- `__gt__()` (*cupy.cuda.Graph method*), 788
- `__gt__()` (*cupy.cuda.ManagedMemory method*), 755
- `__gt__()` (*cupy.cuda.Memory method*), 753
- `__gt__()` (*cupy.cuda.MemoryAsync method*), 754
- `__gt__()` (*cupy.cuda.MemoryAsyncPool method*), 769
- `__gt__()` (*cupy.cuda.MemoryHook method*), 774
- `__gt__()` (*cupy.cuda.MemoryPointer method*), 760
- `__gt__()` (*cupy.cuda.MemoryPool method*), 766
- `__gt__()` (*cupy.cuda.PinnedMemory method*), 757
- `__gt__()` (*cupy.cuda.PinnedMemoryPointer method*), 761
- `__gt__()` (*cupy.cuda.PinnedMemoryPool method*), 770
- `__gt__()` (*cupy.cuda.PythonFunctionAllocator method*), 770
- `__gt__()` (*cupy.cuda.Stream method*), 782
- `__gt__()` (*cupy.cuda.UnownedMemory method*), 756
- `__gt__()` (*cupy.cuda.memory\_hooks.DebugPrintHook method*), 776
- `__gt__()` (*cupy.cuda.memory\_hooks.LineProfileHook method*), 779
- `__gt__()` (*cupy.cuda.nccl.NcclCommunicator method*), 799
- `__gt__()` (*cupy.cuda.texture.CUDAarray method*), 791
- `__gt__()` (*cupy.cuda.texture.ChannelFormatDescriptor method*), 789
- `__gt__()` (*cupy.cuda.texture.ResourceDescriptor method*), 793
- `__gt__()` (*cupy.cuda.texture.SurfaceObject method*), 795
- `__gt__()` (*cupy.cuda.texture.TextureDescriptor method*), 794
- `__gt__()` (*cupy.cuda.texture.TextureObject method*), 795
- `__gt__()` (*cupy.fft.config.set\_cufft\_callbacks method*), 149
- `__gt__()` (*cupy.flatiter method*), 168
- `__gt__()` (*cupy.ndarray method*), 68
- `__gt__()` (*cupy.poly1d method*), 223
- `__gt__()` (*cupy.random.BitGenerator method*), 237
- `__gt__()` (*cupy.random.Generator method*), 236
- `__gt__()` (*cupy.random.MRG32k3a method*), 240
- `__gt__()` (*cupy.random.Philox4x3210 method*), 241
- `__gt__()` (*cupy.random.RandomState method*), 250
- `__gt__()` (*cupy.random.XORWOW method*), 238
- `__gt__()` (*cupy.ufunc method*), 74
- `__gt__()` (*cupy.vectorize method*), 152
- `__gt__()` (*cupyx.GeneralizedUFunc method*), 98
- `__gt__()` (*cupyx.distributed.NCCLBackend method*), 840
- `__gt__()` (*cupyx.distributed.array.DistributedArray method*), 846
- `__gt__()` (*cupyx.jit.\_interface.\_JitRawKernel method*), 835
- `__gt__()` (*cupyx.profiler.time\_range method*), 746
- `__gt__()` (*cupyx.scipy.interpolate.Akima1DInterpolator method*), 366
- `__gt__()` (*cupyx.scipy.interpolate.BPoly method*), 375
- `__gt__()` (*cupyx.scipy.interpolate.BSpline method*), 379
- `__gt__()` (*cupyx.scipy.interpolate.BarycentricInterpolator method*), 349
- `__gt__()` (*cupyx.scipy.interpolate.CubicHermiteSpline method*), 357
- `__gt__()` (*cupyx.scipy.interpolate.KroghInterpolator method*), 351
- `__gt__()` (*cupyx.scipy.interpolate.NdPPoly method*), 392
- `__gt__()` (*cupyx.scipy.interpolate.PPoly method*), 370
- `__gt__()` (*cupyx.scipy.interpolate.PchipInterpolator method*), 361
- `__gt__()` (*cupyx.scipy.interpolate.RBFInterpolator method*), 384
- `__gt__()` (*cupyx.scipy.interpolate.RegularGridInterpolator method*), 389
- `__gt__()` (*cupyx.scipy.signal.CZT method*), 594
- `__gt__()` (*cupyx.scipy.signal.StateSpace method*), 540
- `__gt__()` (*cupyx.scipy.signal.TransferFunction method*), 542
- `__gt__()` (*cupyx.scipy.signal.ZerosPolesGain method*), 544
- `__gt__()` (*cupyx.scipy.signal.ZoomFFT method*), 595
- `__gt__()` (*cupyx.scipy.signal.dlti method*), 550
- `__gt__()` (*cupyx.scipy.signal.lti method*), 539
- `__gt__()` (*cupyx.scipy.sparse.coo\_matrix method*), 632
- `__gt__()` (*cupyx.scipy.sparse.csc\_matrix method*), 642
- `__gt__()` (*cupyx.scipy.sparse.csr\_matrix method*), 653
- `__gt__()` (*cupyx.scipy.sparse.dia\_matrix method*), 660
- `__gt__()` (*cupyx.scipy.sparse.linalg.LinearOperator method*), 677
- `__gt__()` (*cupyx.scipy.sparse.linalg.SuperLU method*), 690
- `__gt__()` (*cupyx.scipy.sparse.spmatrix method*), 665
- `__iter__()` (*cupy.flatiter method*), 168
- `__iter__()` (*cupy.ndarray method*), 59
- `__iter__()` (*cupy.poly1d method*), 222
- `__iter__()` (*cupyx.distributed.array.DistributedArray method*), 842
- `__iter__()` (*cupyx.scipy.sparse.coo\_matrix method*), 626
- `__iter__()` (*cupyx.scipy.sparse.csc\_matrix method*), 634
- `__iter__()` (*cupyx.scipy.sparse.csr\_matrix method*), 644
- `__iter__()` (*cupyx.scipy.sparse.dia\_matrix method*),

- 654
- `__iter__()` (*cupyx.scipy.sparse.spmatrix* method), 661
  - `__le__()` (*cupy.ElementwiseKernel* method), 814
  - `__le__()` (*cupy.RawKernel* method), 818
  - `__le__()` (*cupy.RawModule* method), 822
  - `__le__()` (*cupy.ReductionKernel* method), 816
  - `__le__()` (*cupy.array\_api.\_array\_object.Array* method), 909
  - `__le__()` (*cupy.broadcast* method), 119
  - `__le__()` (*cupy.cuda.CFunctionAllocator* method), 771
  - `__le__()` (*cupy.cuda.Device* method), 749
  - `__le__()` (*cupy.cuda.Event* method), 787
  - `__le__()` (*cupy.cuda.ExternalStream* method), 785
  - `__le__()` (*cupy.cuda.Graph* method), 788
  - `__le__()` (*cupy.cuda.ManagedMemory* method), 755
  - `__le__()` (*cupy.cuda.Memory* method), 752
  - `__le__()` (*cupy.cuda.MemoryAsync* method), 753
  - `__le__()` (*cupy.cuda.MemoryAsyncPool* method), 769
  - `__le__()` (*cupy.cuda.MemoryHook* method), 774
  - `__le__()` (*cupy.cuda.MemoryPointer* method), 760
  - `__le__()` (*cupy.cuda.MemoryPool* method), 766
  - `__le__()` (*cupy.cuda.PinnedMemory* method), 757
  - `__le__()` (*cupy.cuda.PinnedMemoryPointer* method), 761
  - `__le__()` (*cupy.cuda.PinnedMemoryPool* method), 770
  - `__le__()` (*cupy.cuda.PythonFunctionAllocator* method), 770
  - `__le__()` (*cupy.cuda.Stream* method), 782
  - `__le__()` (*cupy.cuda.UnownedMemory* method), 756
  - `__le__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 776
  - `__le__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 779
  - `__le__()` (*cupy.cuda.nccl.NcclCommunicator* method), 799
  - `__le__()` (*cupy.cuda.texture.CUDAarray* method), 791
  - `__le__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 789
  - `__le__()` (*cupy.cuda.texture.ResourceDescriptor* method), 792
  - `__le__()` (*cupy.cuda.texture.SurfaceObject* method), 795
  - `__le__()` (*cupy.cuda.texture.TextureDescriptor* method), 794
  - `__le__()` (*cupy.cuda.texture.TextureObject* method), 794
  - `__le__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 149
  - `__le__()` (*cupy.flatiter* method), 168
  - `__le__()` (*cupy.ndarray* method), 68
  - `__le__()` (*cupy.poly1d* method), 222
  - `__le__()` (*cupy.random.BitGenerator* method), 237
  - `__le__()` (*cupy.random.Generator* method), 236
  - `__le__()` (*cupy.random.MRG32k3a* method), 240
  - `__le__()` (*cupy.random.Philox4x3210* method), 241
  - `__le__()` (*cupy.random.RandomState* method), 250
  - `__le__()` (*cupy.random.XORWOW* method), 238
  - `__le__()` (*cupy.ufunc* method), 74
  - `__le__()` (*cupy.vectorize* method), 152
  - `__le__()` (*cupyx.GeneralizedUFunc* method), 98
  - `__le__()` (*cupyx.distributed.NCCLBackend* method), 840
  - `__le__()` (*cupyx.distributed.array.DistributedArray* method), 846
  - `__le__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 835
  - `__le__()` (*cupyx.profiler.time\_range* method), 746
  - `__le__()` (*cupyx.scipy.interpolate.Akima1DInterpolator* method), 366
  - `__le__()` (*cupyx.scipy.interpolate.BPoly* method), 375
  - `__le__()` (*cupyx.scipy.interpolate.BSpline* method), 379
  - `__le__()` (*cupyx.scipy.interpolate.BarycentricInterpolator* method), 349
  - `__le__()` (*cupyx.scipy.interpolate.CubicHermiteSpline* method), 357
  - `__le__()` (*cupyx.scipy.interpolate.KroghInterpolator* method), 351
  - `__le__()` (*cupyx.scipy.interpolate.NdPPoly* method), 392
  - `__le__()` (*cupyx.scipy.interpolate.PPoly* method), 370
  - `__le__()` (*cupyx.scipy.interpolate.PchipInterpolator* method), 361
  - `__le__()` (*cupyx.scipy.interpolate.RBFInterpolator* method), 384
  - `__le__()` (*cupyx.scipy.interpolate.RegularGridInterpolator* method), 389
  - `__le__()` (*cupyx.scipy.signal.CZT* method), 593
  - `__le__()` (*cupyx.scipy.signal.StateSpace* method), 540
  - `__le__()` (*cupyx.scipy.signal.TransferFunction* method), 542
  - `__le__()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
  - `__le__()` (*cupyx.scipy.signal.ZoomFFT* method), 595
  - `__le__()` (*cupyx.scipy.signal.dlti* method), 550
  - `__le__()` (*cupyx.scipy.signal.lti* method), 539
  - `__le__()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
  - `__le__()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
  - `__le__()` (*cupyx.scipy.sparse.csr\_matrix* method), 653
  - `__le__()` (*cupyx.scipy.sparse.dia\_matrix* method), 660
  - `__le__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 677
  - `__le__()` (*cupyx.scipy.sparse.linalg.SuperLU* method), 690
  - `__le__()` (*cupyx.scipy.sparse.spmatrix* method), 665
  - `__len__()` (*cupy.flatiter* method), 168
  - `__len__()` (*cupy.ndarray* method), 59
  - `__len__()` (*cupy.poly1d* method), 222
  - `__len__()` (*cupyx.distributed.array.DistributedArray* method), 846

- method*), 842
- `__len__()` (*cupyx.scipy.sparse.coo\_matrix method*), 626
- `__len__()` (*cupyx.scipy.sparse.csc\_matrix method*), 634
- `__len__()` (*cupyx.scipy.sparse.csr\_matrix method*), 644
- `__len__()` (*cupyx.scipy.sparse.dia\_matrix method*), 654
- `__len__()` (*cupyx.scipy.sparse.spmatrix method*), 661
- `__lt__()` (*cupy.ElementwiseKernel method*), 814
- `__lt__()` (*cupy.RawKernel method*), 818
- `__lt__()` (*cupy.RawModule method*), 822
- `__lt__()` (*cupy.ReductionKernel method*), 816
- `__lt__()` (*cupy.array\_api.\_array\_object.Array method*), 909
- `__lt__()` (*cupy.broadcast method*), 119
- `__lt__()` (*cupy.cuda.CFunctionAllocator method*), 771
- `__lt__()` (*cupy.cuda.Device method*), 749
- `__lt__()` (*cupy.cuda.Event method*), 787
- `__lt__()` (*cupy.cuda.ExternalStream method*), 785
- `__lt__()` (*cupy.cuda.Graph method*), 788
- `__lt__()` (*cupy.cuda.ManagedMemory method*), 755
- `__lt__()` (*cupy.cuda.Memory method*), 752
- `__lt__()` (*cupy.cuda.MemoryAsync method*), 753
- `__lt__()` (*cupy.cuda.MemoryAsyncPool method*), 769
- `__lt__()` (*cupy.cuda.MemoryHook method*), 774
- `__lt__()` (*cupy.cuda.MemoryPointer method*), 760
- `__lt__()` (*cupy.cuda.MemoryPool method*), 766
- `__lt__()` (*cupy.cuda.PinnedMemory method*), 757
- `__lt__()` (*cupy.cuda.PinnedMemoryPointer method*), 761
- `__lt__()` (*cupy.cuda.PinnedMemoryPool method*), 769
- `__lt__()` (*cupy.cuda.PythonFunctionAllocator method*), 770
- `__lt__()` (*cupy.cuda.Stream method*), 782
- `__lt__()` (*cupy.cuda.UnownedMemory method*), 756
- `__lt__()` (*cupy.cuda.memory\_hooks.DebugPrintHook method*), 776
- `__lt__()` (*cupy.cuda.memory\_hooks.LineProfileHook method*), 779
- `__lt__()` (*cupy.cuda.nccl.NcclCommunicator method*), 799
- `__lt__()` (*cupy.cuda.texture.CUDAArray method*), 791
- `__lt__()` (*cupy.cuda.texture.ChannelFormatDescriptor method*), 789
- `__lt__()` (*cupy.cuda.texture.ResourceDescriptor method*), 792
- `__lt__()` (*cupy.cuda.texture.SurfaceObject method*), 795
- `__lt__()` (*cupy.cuda.texture.TextureDescriptor method*), 794
- `__lt__()` (*cupy.cuda.texture.TextureObject method*), 794
- `__lt__()` (*cupy.fft.config.set\_cufft\_callbacks method*), 149
- `__lt__()` (*cupy.flatiter method*), 168
- `__lt__()` (*cupy.ndarray method*), 68
- `__lt__()` (*cupy.poly1d method*), 222
- `__lt__()` (*cupy.random.BitGenerator method*), 237
- `__lt__()` (*cupy.random.Generator method*), 236
- `__lt__()` (*cupy.random.MRG32k3a method*), 240
- `__lt__()` (*cupy.random.Philox4x3210 method*), 241
- `__lt__()` (*cupy.random.RandomState method*), 250
- `__lt__()` (*cupy.random.XORWOW method*), 238
- `__lt__()` (*cupy.ufunc method*), 74
- `__lt__()` (*cupy.vectorize method*), 151
- `__lt__()` (*cupyx.GeneralizedUFunc method*), 98
- `__lt__()` (*cupyx.distributed.NCCLBackend method*), 840
- `__lt__()` (*cupyx.distributed.array.DistributedArray method*), 846
- `__lt__()` (*cupyx.jit.\_interface.\_JitRawKernel method*), 835
- `__lt__()` (*cupyx.profiler.time\_range method*), 746
- `__lt__()` (*cupyx.scipy.interpolate.Akima1DInterpolator method*), 366
- `__lt__()` (*cupyx.scipy.interpolate.BPoly method*), 375
- `__lt__()` (*cupyx.scipy.interpolate.BSpline method*), 379
- `__lt__()` (*cupyx.scipy.interpolate.BarycentricInterpolator method*), 349
- `__lt__()` (*cupyx.scipy.interpolate.CubicHermiteSpline method*), 357
- `__lt__()` (*cupyx.scipy.interpolate.KroghInterpolator method*), 351
- `__lt__()` (*cupyx.scipy.interpolate.NdPPoly method*), 392
- `__lt__()` (*cupyx.scipy.interpolate.PPoly method*), 370
- `__lt__()` (*cupyx.scipy.interpolate.PchipInterpolator method*), 361
- `__lt__()` (*cupyx.scipy.interpolate.RBFInterpolator method*), 384
- `__lt__()` (*cupyx.scipy.interpolate.RegularGridInterpolator method*), 389
- `__lt__()` (*cupyx.scipy.signal.CZT method*), 593
- `__lt__()` (*cupyx.scipy.signal.StateSpace method*), 540
- `__lt__()` (*cupyx.scipy.signal.TransferFunction method*), 542
- `__lt__()` (*cupyx.scipy.signal.ZerosPolesGain method*), 544
- `__lt__()` (*cupyx.scipy.signal.ZoomFFT method*), 595
- `__lt__()` (*cupyx.scipy.signal.dlti method*), 550
- `__lt__()` (*cupyx.scipy.signal.lti method*), 539
- `__lt__()` (*cupyx.scipy.sparse.coo\_matrix method*), 632
- `__lt__()` (*cupyx.scipy.sparse.csc\_matrix method*), 642
- `__lt__()` (*cupyx.scipy.sparse.csr\_matrix method*), 653
- `__lt__()` (*cupyx.scipy.sparse.dia\_matrix method*), 660
- `__lt__()` (*cupyx.scipy.sparse.linalg.LinearOperator method*), 677
- `__lt__()` (*cupyx.scipy.sparse.linalg.SuperLU method*), 690
- `__lt__()` (*cupyx.scipy.sparse.spmatrix method*), 665



- `__ne__()` (*cupy.ElementwiseKernel* method), 814
- `__ne__()` (*cupy.RawKernel* method), 818
- `__ne__()` (*cupy.RawModule* method), 822
- `__ne__()` (*cupy.ReductionKernel* method), 816
- `__ne__()` (*cupy.array\_api.\_array\_object.Array* method), 909
- `__ne__()` (*cupy.broadcast* method), 119
- `__ne__()` (*cupy.cuda.CFunctionAllocator* method), 771
- `__ne__()` (*cupy.cuda.Device* method), 749
- `__ne__()` (*cupy.cuda.Event* method), 787
- `__ne__()` (*cupy.cuda.ExternalStream* method), 785
- `__ne__()` (*cupy.cuda.Graph* method), 788
- `__ne__()` (*cupy.cuda.ManagedMemory* method), 755
- `__ne__()` (*cupy.cuda.Memory* method), 752
- `__ne__()` (*cupy.cuda.MemoryAsync* method), 753
- `__ne__()` (*cupy.cuda.MemoryAsyncPool* method), 768
- `__ne__()` (*cupy.cuda.MemoryHook* method), 774
- `__ne__()` (*cupy.cuda.MemoryPointer* method), 760
- `__ne__()` (*cupy.cuda.MemoryPool* method), 766
- `__ne__()` (*cupy.cuda.PinnedMemory* method), 757
- `__ne__()` (*cupy.cuda.PinnedMemoryPointer* method), 761
- `__ne__()` (*cupy.cuda.PinnedMemoryPool* method), 769
- `__ne__()` (*cupy.cuda.PythonFunctionAllocator* method), 770
- `__ne__()` (*cupy.cuda.Stream* method), 782
- `__ne__()` (*cupy.cuda.UnownedMemory* method), 756
- `__ne__()` (*cupy.cuda.memory\_hooks.DebugPrintHook* method), 776
- `__ne__()` (*cupy.cuda.memory\_hooks.LineProfileHook* method), 778
- `__ne__()` (*cupy.cuda.nccl.NcclCommunicator* method), 799
- `__ne__()` (*cupy.cuda.texture.CUDAarray* method), 791
- `__ne__()` (*cupy.cuda.texture.ChannelFormatDescriptor* method), 789
- `__ne__()` (*cupy.cuda.texture.ResourceDescriptor* method), 792
- `__ne__()` (*cupy.cuda.texture.SurfaceObject* method), 795
- `__ne__()` (*cupy.cuda.texture.TextureDescriptor* method), 794
- `__ne__()` (*cupy.cuda.texture.TextureObject* method), 794
- `__ne__()` (*cupy.fft.config.set\_cufft\_callbacks* method), 149
- `__ne__()` (*cupy.flatiter* method), 168
- `__ne__()` (*cupy.ndarray* method), 68
- `__ne__()` (*cupy.poly1d* method), 222
- `__ne__()` (*cupy.random.BitGenerator* method), 237
- `__ne__()` (*cupy.random.Generator* method), 236
- `__ne__()` (*cupy.random.MRG32k3a* method), 240
- `__ne__()` (*cupy.random.Philox4x3210* method), 241
- `__ne__()` (*cupy.random.RandomState* method), 250
- `__ne__()` (*cupy.random.XORWOW* method), 238
- `__ne__()` (*cupy.ufunc* method), 74
- `__ne__()` (*cupy.vectorize* method), 151
- `__ne__()` (*cupyx.GeneralizedUFunc* method), 97
- `__ne__()` (*cupyx.distributed.NCCLBackend* method), 840
- `__ne__()` (*cupyx.distributed.array.DistributedArray* method), 846
- `__ne__()` (*cupyx.jit.\_interface.\_JitRawKernel* method), 835
- `__ne__()` (*cupyx.profiler.time\_range* method), 746
- `__ne__()` (*cupyx.scipy.interpolate.Akima1DInterpolator* method), 366
- `__ne__()` (*cupyx.scipy.interpolate.BPoly* method), 375
- `__ne__()` (*cupyx.scipy.interpolate.BSpline* method), 379
- `__ne__()` (*cupyx.scipy.interpolate.BarycentricInterpolator* method), 349
- `__ne__()` (*cupyx.scipy.interpolate.CubicHermiteSpline* method), 357
- `__ne__()` (*cupyx.scipy.interpolate.KroghInterpolator* method), 351
- `__ne__()` (*cupyx.scipy.interpolate.NdPPoly* method), 392
- `__ne__()` (*cupyx.scipy.interpolate.PPoly* method), 370
- `__ne__()` (*cupyx.scipy.interpolate.PchipInterpolator* method), 361
- `__ne__()` (*cupyx.scipy.interpolate.RBFInterpolator* method), 384
- `__ne__()` (*cupyx.scipy.interpolate.RegularGridInterpolator* method), 389
- `__ne__()` (*cupyx.scipy.signal.CZT* method), 593
- `__ne__()` (*cupyx.scipy.signal.StateSpace* method), 540
- `__ne__()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `__ne__()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `__ne__()` (*cupyx.scipy.signal.ZoomFFT* method), 595
- `__ne__()` (*cupyx.scipy.signal.dlti* method), 550
- `__ne__()` (*cupyx.scipy.signal.lti* method), 539
- `__ne__()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `__ne__()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `__ne__()` (*cupyx.scipy.sparse.csr\_matrix* method), 653
- `__ne__()` (*cupyx.scipy.sparse.dia\_matrix* method), 660
- `__ne__()` (*cupyx.scipy.sparse.linalg.LinearOperator* method), 677
- `__ne__()` (*cupyx.scipy.sparse.linalg.SuperLU* method), 690
- `__ne__()` (*cupyx.scipy.sparse.spmatrix* method), 665
- `__next__()` (*cupy.flatiter* method), 168
- `__nonzero__()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `__nonzero__()` (*cupyx.scipy.sparse.csc\_matrix* method), 643
- `__nonzero__()` (*cupyx.scipy.sparse.csr\_matrix* method), 653

method), 653  
 \_\_nonzero\_\_() (cupyx.scipy.sparse.dia\_matrix method), 660  
 \_\_nonzero\_\_() (cupyx.scipy.sparse.spmatrix method), 665  
 \_\_setitem\_\_() (cupy.array\_api.\_array\_object.Array method), 909  
 \_\_setitem\_\_() (cupy.flatiter method), 168  
 \_\_setitem\_\_() (cupy.ndarray method), 58  
 \_\_setitem\_\_() (cupy.poly1d method), 222  
 \_\_setitem\_\_() (cupyx.distributed.array.DistributedArray method), 842  
 \_\_setitem\_\_() (cupyx.scipy.sparse.csc\_matrix method), 634  
 \_\_setitem\_\_() (cupyx.scipy.sparse.csr\_matrix method), 644

## A

A (cupyx.scipy.signal.StateSpace attribute), 541  
 A (cupyx.scipy.sparse.coo\_matrix attribute), 633  
 A (cupyx.scipy.sparse.csc\_matrix attribute), 643  
 A (cupyx.scipy.sparse.csr\_matrix attribute), 653  
 A (cupyx.scipy.sparse.dia\_matrix attribute), 660  
 A (cupyx.scipy.sparse.spmatrix attribute), 665  
 abcd\_normalize() (in module cupyx.scipy.signal), 521  
 abort() (cupy.cuda.nccl.NcclCommunicator method), 798  
 abs() (in module cupy.array\_api), 896  
 absolute() (in module cupy), 80  
 accumulate() (cupy.ufunc method), 74  
 acos() (in module cupy.array\_api), 896  
 acosh() (in module cupy.array\_api), 896  
 add() (in module cupy), 76  
 add() (in module cupy.array\_api), 896  
 add\_callback() (cupy.cuda.ExternalStream method), 783  
 add\_callback() (cupy.cuda.Stream method), 780  
 add\_xi() (cupyx.scipy.interpolate.BarycentricInterpolator method), 349  
 adjoint() (cupyx.scipy.sparse.linalg.LinearOperator method), 677  
 advise() (cupy.cuda.ManagedMemory method), 754  
 affine\_transform() (in module cupyx.scipy.ndimage), 424  
 Akima1DInterpolator (class in cupyx.scipy.interpolate), 362  
 all() (cupy.ndarray method), 59  
 all() (cupyx.distributed.array.DistributedArray method), 842  
 all() (in module cupy), 187  
 all() (in module cupy.array\_api), 896  
 all\_chunks() (cupyx.distributed.array.DistributedArray method), 842

all\_gather() (cupyx.distributed.NCCLBackend method), 838  
 all\_reduce() (cupyx.distributed.NCCLBackend method), 838  
 all\_to\_all() (cupyx.distributed.NCCLBackend method), 838  
 allclose() (in module cupy), 194  
 allGather() (cupy.cuda.nccl.NcclCommunicator method), 798  
 alloc() (in module cupy.cuda), 762  
 alloc\_pinned\_memory() (in module cupy.cuda), 762  
 alloc\_postprocess() (cupy.cuda.memory\_hooks.DebugPrintHook method), 775  
 alloc\_postprocess() (cupy.cuda.memory\_hooks.LineProfileHook method), 777  
 alloc\_postprocess() (cupy.cuda.MemoryHook method), 773  
 alloc\_preprocess() (cupy.cuda.memory\_hooks.DebugPrintHook method), 775  
 alloc\_preprocess() (cupy.cuda.memory\_hooks.LineProfileHook method), 777  
 alloc\_preprocess() (cupy.cuda.MemoryHook method), 773  
 allReduce() (cupy.cuda.nccl.NcclCommunicator method), 798  
 amax() (in module cupy), 288  
 amin() (in module cupy), 287  
 angle() (in module cupy), 208  
 antiderivative() (cupyx.scipy.interpolate.Akima1DInterpolator method), 363  
 antiderivative() (cupyx.scipy.interpolate.BPoly method), 372  
 antiderivative() (cupyx.scipy.interpolate.BSpline method), 377  
 antiderivative() (cupyx.scipy.interpolate.CubicHermiteSpline method), 354  
 antiderivative() (cupyx.scipy.interpolate.NdPPoly method), 390  
 antiderivative() (cupyx.scipy.interpolate.PchipInterpolator method), 359  
 antiderivative() (cupyx.scipy.interpolate.PPoly method), 368  
 any() (cupy.ndarray method), 59  
 any() (cupyx.distributed.array.DistributedArray method), 842  
 any() (in module cupy), 187  
 any() (in module cupy.array\_api), 896  
 append() (in module cupy), 129  
 apply\_along\_axis() (in module cupy), 151

- `arange()` (in module `cupy`), 107
- `arange()` (in module `cupy.array_api`), 896
- `arccos()` (in module `cupy`), 85
- `arccosh()` (in module `cupy`), 87
- `arcsin()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `arcsin()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `arcsin()` (`cupyx.scipy.sparse.csr_matrix` method), 644
- `arcsin()` (`cupyx.scipy.sparse.dia_matrix` method), 654
- `arcsin()` (in module `cupy`), 85
- `arcsinh()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `arcsinh()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `arcsinh()` (`cupyx.scipy.sparse.csr_matrix` method), 644
- `arcsinh()` (`cupyx.scipy.sparse.dia_matrix` method), 654
- `arcsinh()` (in module `cupy`), 87
- `arctan()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `arctan()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `arctan()` (`cupyx.scipy.sparse.csr_matrix` method), 644
- `arctan()` (`cupyx.scipy.sparse.dia_matrix` method), 654
- `arctan()` (in module `cupy`), 85
- `arctan2()` (in module `cupy`), 86
- `arctanh()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `arctanh()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `arctanh()` (`cupyx.scipy.sparse.csr_matrix` method), 645
- `arctanh()` (`cupyx.scipy.sparse.dia_matrix` method), 655
- `arctanh()` (in module `cupy`), 87
- `argmax()` (`cupy.ndarray` method), 59
- `argmax()` (`cupyx.distributed.array.DistributedArray` method), 842
- `argmax()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `argmax()` (`cupyx.scipy.sparse.csr_matrix` method), 645
- `argmax()` (in module `cupy`), 282
- `argmax()` (in module `cupy.array_api`), 896
- `argmin()` (`cupy.ndarray` method), 59
- `argmin()` (`cupyx.distributed.array.DistributedArray` method), 842
- `argmin()` (`cupyx.scipy.sparse.csc_matrix` method), 634
- `argmin()` (`cupyx.scipy.sparse.csr_matrix` method), 645
- `argmin()` (in module `cupy`), 283
- `argmin()` (in module `cupy.array_api`), 897
- `argpartition()` (`cupy.ndarray` method), 60
- `argpartition()` (`cupyx.distributed.array.DistributedArray` method), 843
- `argpartition()` (in module `cupy`), 281
- `argrelextrema()` (in module `cupyx.scipy.signal`), 567
- `argrelmax()` (in module `cupyx.scipy.signal`), 566
- `argrelmin()` (in module `cupyx.scipy.signal`), 565
- `argsort()` (`cupy.ndarray` method), 60
- `argsort()` (`cupyx.distributed.array.DistributedArray` method), 843
- `argsort()` (in module `cupy`), 279
- `argsort()` (in module `cupy.array_api`), 897
- `argwhere()` (in module `cupy`), 284
- `around()` (in module `cupy`), 198
- `arr` (`cupy.cuda.texture.ResourceDescriptor` attribute), 793
- `Array` (class in `cupy.array_api._array_object`), 908
- `array()` (in module `cupy`), 70
- `array2string()` (in module `cupy`), 171
- `array_equal()` (in module `cupy`), 195
- `array_equiv()` (in module `cupy`), 196
- `array_repr()` (in module `cupy`), 172
- `array_split()` (in module `cupy`), 127
- `array_str()` (in module `cupy`), 172
- `as_series()` (in module `cupy.polynomial.polyutils`), 220
- `as_strided()` (in module `cupy.lib.stride_tricks`), 164
- `asanyarray()` (in module `cupy`), 104
- `asarray()` (in module `cupy`), 71
- `asarray()` (in module `cupy.array_api`), 897
- `asarray_chkfinite()` (in module `cupy`), 122
- `ascontiguousarray()` (in module `cupy`), 104
- `asfarray()` (in module `cupy`), 121
- `asformat()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `asformat()` (`cupyx.scipy.sparse.csc_matrix` method), 635
- `asformat()` (`cupyx.scipy.sparse.csr_matrix` method), 645
- `asformat()` (`cupyx.scipy.sparse.dia_matrix` method), 655
- `asformat()` (`cupyx.scipy.sparse.spmatrix` method), 661
- `asfortranarray()` (in module `cupy`), 121
- `asfptype()` (`cupyx.scipy.sparse.coo_matrix` method), 626
- `asfptype()` (`cupyx.scipy.sparse.csc_matrix` method), 635
- `asfptype()` (`cupyx.scipy.sparse.csr_matrix` method), 645
- `asfptype()` (`cupyx.scipy.sparse.dia_matrix` method), 655
- `asfptype()` (`cupyx.scipy.sparse.spmatrix` method), 661
- `asin()` (in module `cupy.array_api`), 897
- `asinh()` (in module `cupy.array_api`), 897
- `aslinearoperator()` (in module `cupyx.scipy.sparse.linalg`), 678
- `asnumpy()` (in module `cupy`), 71
- `assert_allclose()` (in module `cupy.testing`), 303
- `assert_array_almost_equal()` (in module `cupy.testing`), 302
- `assert_array_almost_equal_nulp()` (in module `cupy.testing`), 303
- `assert_array_equal()` (in module `cupy.testing`), 304
- `assert_array_less()` (in module `cupy.testing`), 304
- `assert_array_list_equal()` (in module `cupy.testing`), 305
- `assert_array_max_ulp()` (in module `cupy.testing`), 303
- `astype()` (`cupy.ndarray` method), 60

- `astype()` (*cupyx.distributed.array.DistributedArray* method), 843  
`astype()` (*cupyx.scipy.sparse.coo\_matrix* method), 626  
`astype()` (*cupyx.scipy.sparse.csc\_matrix* method), 635  
`astype()` (*cupyx.scipy.sparse.csr\_matrix* method), 646  
`astype()` (*cupyx.scipy.sparse.dia\_matrix* method), 655  
`astype()` (*cupyx.scipy.sparse.spmatrix* method), 661  
`at()` (*cupy.ufunc* method), 74  
`atan()` (*in module cupy.array\_api*), 897  
`atan2()` (*in module cupy.array\_api*), 897  
`atanh()` (*in module cupy.array\_api*), 897  
`atleast_1d()` (*in module cupy*), 117  
`atleast_2d()` (*in module cupy*), 117  
`atleast_3d()` (*in module cupy*), 118  
`atomic_add` (*in module cupyx.jit*), 829  
`atomic_and` (*in module cupyx.jit*), 832  
`atomic_cas` (*in module cupyx.jit*), 831  
`atomic_dec` (*in module cupyx.jit*), 831  
`atomic_exch` (*in module cupyx.jit*), 829  
`atomic_inc` (*in module cupyx.jit*), 831  
`atomic_max` (*in module cupyx.jit*), 830  
`atomic_min` (*in module cupyx.jit*), 830  
`atomic_or` (*in module cupyx.jit*), 832  
`atomic_sub` (*in module cupyx.jit*), 829  
`atomic_xor` (*in module cupyx.jit*), 832  
`attributes` (*cupy.cuda.Device* attribute), 750  
`attributes` (*cupy.RawKernel* attribute), 819  
`average()` (*in module cupy*), 292  
`axis` (*cupyx.scipy.interpolate.Akima1DInterpolator* attribute), 366  
`axis` (*cupyx.scipy.interpolate.BPoly* attribute), 375  
`axis` (*cupyx.scipy.interpolate.CubicHermiteSpline* attribute), 357  
`axis` (*cupyx.scipy.interpolate.PchipInterpolator* attribute), 362  
`axis` (*cupyx.scipy.interpolate.PPoly* attribute), 371
- ## B
- `B` (*cupyx.scipy.signal.StateSpace* attribute), 541  
`backend` (*cupy.RawKernel* attribute), 819  
`backend` (*cupy.RawModule* attribute), 823  
`BadCoefficients`, 509  
`band_stop_obj()` (*in module cupyx.scipy.signal*), 522  
`barrier()` (*cupyx.distributed.NCCLBackend* method), 838  
`barthann()` (*in module cupyx.scipy.signal.windows*), 599  
`bartlett()` (*in module cupy*), 316  
`bartlett()` (*in module cupyx.scipy.signal.windows*), 599  
`barycentric_interpolate()` (*in module cupyx.scipy.interpolate*), 351  
`BarycentricInterpolator` (*class in cupyx.scipy.interpolate*), 348  
`base` (*cupy.flatiter* attribute), 168  
`base` (*cupy.ndarray* attribute), 68  
`base` (*cupyx.distributed.array.DistributedArray* attribute), 847  
`base_repr()` (*in module cupy*), 173  
`basis_element()` (*cupyx.scipy.interpolate.BSpline* class method), 377  
`bcast()` (*cupy.cuda.nccl.NcclCommunicator* method), 798  
`bdtr()` (*in module cupyx.scipy.special*), 705  
`bdtrc()` (*in module cupyx.scipy.special*), 706  
`bdtri()` (*in module cupyx.scipy.special*), 706  
`begin_capture()` (*cupy.cuda.ExternalStream* method), 783  
`begin_capture()` (*cupy.cuda.Stream* method), 780  
`benchmark()` (*in module cupyx.profiler*), 744  
`beta()` (*cupy.random.Generator* method), 228  
`beta()` (*cupy.random.RandomState* method), 242  
`beta()` (*in module cupy.random*), 251  
`beta()` (*in module cupyx.scipy.special*), 720  
`betainc()` (*in module cupyx.scipy.special*), 721  
`betaincinv()` (*in module cupyx.scipy.special*), 721  
`betaln()` (*in module cupyx.scipy.special*), 720  
`bilinear()` (*in module cupyx.scipy.signal*), 485  
`bilinear_zpk()` (*in module cupyx.scipy.signal*), 486  
`binary_closing()` (*in module cupyx.scipy.ndimage*), 440  
`binary_dilation()` (*in module cupyx.scipy.ndimage*), 441  
`binary_erosion()` (*in module cupyx.scipy.ndimage*), 442  
`binary_fill_holes()` (*in module cupyx.scipy.ndimage*), 443  
`binary_hit_or_miss()` (*in module cupyx.scipy.ndimage*), 443  
`binary_opening()` (*in module cupyx.scipy.ndimage*), 444  
`binary_propagation()` (*in module cupyx.scipy.ndimage*), 445  
`binary_repr()` (*in module cupy*), 136  
`binary_version` (*cupy.RawKernel* attribute), 819  
`bincount()` (*in module cupy*), 301  
`binomial()` (*cupy.random.Generator* method), 229  
`binomial()` (*cupy.random.RandomState* method), 242  
`binomial()` (*in module cupy.random*), 252  
`BitGenerator` (*class in cupy.random*), 237  
`bitwise_and()` (*in module cupy*), 88  
`bitwise_and()` (*in module cupy.array\_api*), 898  
`bitwise_invert()` (*in module cupy.array\_api*), 898  
`bitwise_left_shift()` (*in module cupy.array\_api*), 898  
`bitwise_or()` (*in module cupy*), 88  
`bitwise_or()` (*in module cupy.array\_api*), 898



- `bitwise_right_shift()` (in module `cupy.array_api`), 898
  - `bitwise_xor()` (in module `cupy`), 89
  - `bitwise_xor()` (in module `cupy.array_api`), 898
  - `black_tophat()` (in module `cupyx.scipy.ndimage`), 446
  - `blackman()` (in module `cupy`), 316
  - `blackman()` (in module `cupyx.scipy.signal.windows`), 601
  - `blackmanharris()` (in module `cupyx.scipy.signal.windows`), 602
  - `block_diag()` (in module `cupyx.scipy.linalg`), 397
  - `blockDim` (in module `cupyx.jit`), 825
  - `blockIdx` (in module `cupyx.jit`), 825
  - `bmat()` (in module `cupyx.scipy.sparse`), 670
  - `bode()` (`cupyx.scipy.signal.dlti` method), 549
  - `bode()` (`cupyx.scipy.signal.lti` method), 538
  - `bode()` (in module `cupyx.scipy.signal`), 548
  - `bohman()` (in module `cupyx.scipy.signal.windows`), 603
  - `boxcar()` (in module `cupyx.scipy.signal.windows`), 604
  - `boxcox()` (in module `cupyx.scipy.special`), 715
  - `boxcox1p()` (in module `cupyx.scipy.special`), 715
  - `boxcox_1lf()` (in module `cupyx.scipy.stats`), 735
  - `BPoly` (class in `cupyx.scipy.interpolate`), 371
  - `broadcast` (class in `cupy`), 118
  - `broadcast()` (`cupy.cuda.nccl.NcclCommunicator` method), 798
  - `broadcast()` (`cupyx.distributed.NCCLBackend` method), 838
  - `broadcast_arrays()` (in module `cupy`), 120
  - `broadcast_arrays()` (in module `cupy.array_api`), 898
  - `broadcast_to()` (in module `cupy`), 119
  - `broadcast_to()` (in module `cupy.array_api`), 898
  - `BSpline` (class in `cupyx.scipy.interpolate`), 375
  - `btdtr()` (in module `cupyx.scipy.special`), 707
  - `btdtri()` (in module `cupyx.scipy.special`), 707
  - `buttap()` (in module `cupyx.scipy.signal`), 522
  - `butter()` (in module `cupyx.scipy.signal`), 509
  - `buttord()` (in module `cupyx.scipy.signal`), 510
  - `byte_bounds()` (in module `cupy`), 212
  - `bytes()` (in module `cupy.random`), 252
- ## C
- `c` (`cupy.poly1d` attribute), 223
  - `c` (`cupyx.scipy.interpolate.Akima1DInterpolator` attribute), 366
  - `c` (`cupyx.scipy.interpolate.BPoly` attribute), 375
  - `c` (`cupyx.scipy.interpolate.CubicHermiteSpline` attribute), 357
  - `c` (`cupyx.scipy.interpolate.PchipInterpolator` attribute), 362
  - `c` (`cupyx.scipy.interpolate.PPoly` attribute), 371
  - `C` (`cupyx.scipy.signal.StateSpace` attribute), 541
  - `c_` (in module `cupy`), 153
  - `ca_cfar()` (in module `cupyx.signal`), 744
  - `cache_mode_ca` (`cupy.RawKernel` attribute), 819
  - `cached_code` (`cupy.ElementwiseKernel` attribute), 814
  - `cached_code` (`cupy.ReductionKernel` attribute), 816
  - `cached_code` (`cupyx.jit._interface._JitRawKernel` attribute), 835
  - `cached_codes` (`cupy.ElementwiseKernel` attribute), 814
  - `cached_codes` (`cupy.ReductionKernel` attribute), 816
  - `cached_codes` (`cupyx.jit._interface._JitRawKernel` attribute), 835
  - `can_cast()` (in module `cupy`), 136
  - `can_cast()` (in module `cupy.array_api`), 899
  - `canberra()` (in module `cupyx.scipy.spatial.distance`), 696
  - `cbrt()` (in module `cupy`), 83
  - `cbrt()` (in module `cupyx.scipy.special`), 729
  - `cdist()` (in module `cupyx.scipy.spatial.distance`), 693
  - `ceil()` (`cupyx.scipy.sparse.coo_matrix` method), 626
  - `ceil()` (`cupyx.scipy.sparse.csc_matrix` method), 635
  - `ceil()` (`cupyx.scipy.sparse.csr_matrix` method), 646
  - `ceil()` (`cupyx.scipy.sparse.dia_matrix` method), 655
  - `ceil()` (in module `cupy`), 95
  - `ceil()` (in module `cupy.array_api`), 899
  - `center_of_mass()` (in module `cupyx.scipy.ndimage`), 431
  - `cfar_alpha()` (in module `cupyx.signal`), 743
  - `CFunctionAllocator` (class in `cupy.cuda`), 771
  - `cg()` (in module `cupyx.scipy.sparse.linalg`), 680
  - `cgs()` (in module `cupyx.scipy.sparse.linalg`), 682
  - `change_mode()` (`cupyx.distributed.array.DistributedArray` method), 843
  - `ChannelFormatDescriptor` (class in `cupy.cuda.texture`), 789
  - `channelize_poly()` (in module `cupyx.signal`), 741
  - `chDesc` (`cupy.cuda.texture.ResourceDescriptor` attribute), 793
  - `chdtr()` (in module `cupyx.scipy.special`), 712
  - `chdtrc()` (in module `cupyx.scipy.special`), 712
  - `chdtri()` (in module `cupyx.scipy.special`), 713
  - `cheb1ap()` (in module `cupyx.scipy.signal`), 522
  - `cheb1ord()` (in module `cupyx.scipy.signal`), 515
  - `cheb2ap()` (in module `cupyx.scipy.signal`), 523
  - `cheb2ord()` (in module `cupyx.scipy.signal`), 517
  - `chebwin()` (in module `cupyx.scipy.signal.windows`), 605
  - `cheby1()` (in module `cupyx.scipy.signal`), 514
  - `cheby2()` (in module `cupyx.scipy.signal`), 516
  - `chebyshev()` (in module `cupyx.scipy.spatial.distance`), 696
  - `check_async_error()` (`cupy.cuda.nccl.NcclCommunicator` method), 798
  - `check_COLA()` (in module `cupyx.scipy.signal`), 588
  - `check_NOLA()` (in module `cupyx.scipy.signal`), 589
  - `chirp()` (in module `cupyx.scipy.signal`), 554
  - `chisquare()` (`cupy.random.Generator` method), 229

- `chisquare()` (*cupy.random.RandomState* method), 242
- `chisquare()` (in module *cupy.random*), 253
- `choice()` (*cupy.random.RandomState* method), 242
- `choice()` (in module *cupy.random*), 253
- `cholesky()` (in module *cupy.linalg*), 177
- `choose()` (*cupy.ndarray* method), 61
- `choose()` (*cupyx.distributed.array.DistributedArray* method), 843
- `choose()` (in module *cupy*), 163
- `choose_conv_method()` (in module *cupyx.scipy.signal*), 461
- `circulant()` (in module *cupyx.scipy.linalg*), 398
- `cityblock()` (in module *cupyx.scipy.spatial.distance*), 696
- `clear_memo()` (in module *cupy*), 836
- `clip()` (*cupy.ndarray* method), 61
- `clip()` (*cupyx.distributed.array.DistributedArray* method), 843
- `clip()` (in module *cupy*), 210
- `code` (*cupy.RawKernel* attribute), 819
- `code` (*cupy.RawModule* attribute), 823
- `coef` (*cupy.poly1d* attribute), 223
- `coefficients` (*cupy.poly1d* attribute), 223
- `coffs` (*cupy.poly1d* attribute), 223
- `coherence()` (in module *cupyx.scipy.signal*), 577
- `column_stack()` (in module *cupy*), 125
- `comm` (*cupy.cuda.nccl.NcclCommunicator* attribute), 800
- `common_type()` (in module *cupy*), 137
- `companion()` (in module *cupyx.scipy.linalg*), 398
- `compile()` (*cupy.RawKernel* method), 818
- `compile()` (*cupy.RawModule* method), 821
- `compress()` (*cupy.ndarray* method), 61
- `compress()` (*cupyx.distributed.array.DistributedArray* method), 843
- `compress()` (in module *cupy*), 163
- `compute_capability` (*cupy.cuda.Device* attribute), 750
- `concat()` (in module *cupy.array\_api*), 899
- `concatenate()` (in module *cupy*), 123
- `conj()` (*cupy.ndarray* method), 61
- `conj()` (*cupyx.distributed.array.DistributedArray* method), 843
- `conj()` (*cupyx.scipy.sparse.coo\_matrix* method), 626
- `conj()` (*cupyx.scipy.sparse.csc\_matrix* method), 635
- `conj()` (*cupyx.scipy.sparse.csr\_matrix* method), 646
- `conj()` (*cupyx.scipy.sparse.dia\_matrix* method), 655
- `conj()` (*cupyx.scipy.sparse.spmatrix* method), 661
- `conj()` (in module *cupy*), 81
- `conjugate()` (*cupy.ndarray* method), 61
- `conjugate()` (*cupyx.distributed.array.DistributedArray* method), 843
- `conjugate()` (*cupyx.scipy.sparse.coo\_matrix* method), 627
- `conjugate()` (*cupyx.scipy.sparse.csc\_matrix* method), 635
- `conjugate()` (*cupyx.scipy.sparse.csr\_matrix* method), 646
- `conjugate()` (*cupyx.scipy.sparse.dia\_matrix* method), 655
- `conjugate()` (*cupyx.scipy.sparse.spmatrix* method), 661
- `conjugate()` (in module *cupy*), 81
- `connected_components()` (in module *cupyx.scipy.sparse.csgraph*), 691
- `const_size_bytes` (*cupy.RawKernel* attribute), 819
- `construct_fast()` (*cupyx.scipy.interpolate.Akima1DInterpolator* class method), 364
- `construct_fast()` (*cupyx.scipy.interpolate.BPoly* class method), 373
- `construct_fast()` (*cupyx.scipy.interpolate.BSpline* class method), 378
- `construct_fast()` (*cupyx.scipy.interpolate.CubicHermiteSpline* class method), 355
- `construct_fast()` (*cupyx.scipy.interpolate.NdPPoly* class method), 391
- `construct_fast()` (*cupyx.scipy.interpolate.PchipInterpolator* class method), 359
- `construct_fast()` (*cupyx.scipy.interpolate.PPoly* class method), 368
- `cont2discrete()` (in module *cupyx.scipy.signal*), 534
- `convolution_matrix()` (in module *cupyx.scipy.linalg*), 399
- `convolve()` (in module *cupy*), 209
- `convolve()` (in module *cupyx.scipy.ndimage*), 406
- `convolve()` (in module *cupyx.scipy.signal*), 455
- `convolve1d()` (in module *cupyx.scipy.ndimage*), 406
- `convolve1d3o()` (in module *cupyx.signal*), 742
- `convolve2d()` (in module *cupyx.scipy.signal*), 459
- `coo_matrix` (class in *cupyx.scipy.sparse*), 625
- `copy()` (*cupy.flatiter* method), 168
- `copy()` (*cupy.ndarray* method), 61
- `copy()` (*cupyx.distributed.array.DistributedArray* method), 843
- `copy()` (*cupyx.scipy.sparse.coo\_matrix* method), 627
- `copy()` (*cupyx.scipy.sparse.csc\_matrix* method), 636
- `copy()` (*cupyx.scipy.sparse.csr\_matrix* method), 646
- `copy()` (*cupyx.scipy.sparse.dia\_matrix* method), 656
- `copy()` (*cupyx.scipy.sparse.spmatrix* method), 662
- `copy()` (in module *cupy*), 105
- `copy_from()` (*cupy.cuda.MemoryPointer* method), 757
- `copy_from()` (*cupy.cuda.texture.CUDAarray* method), 790
- `copy_from_async()` (*cupy.cuda.MemoryPointer* method), 758
- `copy_from_device()` (*cupy.cuda.MemoryPointer* method), 758
- `copy_from_device_async()`

- `(cupy.cuda.MemoryPointer method)`, 758
- `copy_from_host()` (`cupy.cuda.MemoryPointer method`), 758
- `copy_from_host_async()` (`cupy.cuda.MemoryPointer method`), 758
- `copy_to()` (`cupy.cuda.texture.CUDAArray method`), 791
- `copy_to_host()` (`cupy.cuda.MemoryPointer method`), 759
- `copy_to_host_async()` (`cupy.cuda.MemoryPointer method`), 759
- `copysign()` (in module `cupy`), 94
- `copyto()` (in module `cupy`), 113
- `corrcoef()` (in module `cupy`), 296
- `correlate()` (in module `cupy`), 297
- `correlate()` (in module `cupyx.scipy.ndimage`), 407
- `correlate()` (in module `cupyx.scipy.signal`), 456
- `correlate1d()` (in module `cupyx.scipy.ndimage`), 408
- `correlate2d()` (in module `cupyx.scipy.signal`), 460
- `correlation()` (in module `cupyx.scipy.spatial.distance`), 697
- `correlation_lags()` (in module `cupyx.scipy.signal`), 462
- `cos()` (in module `cupy`), 85
- `cos()` (in module `cupy.array_api`), 899
- `cosdg()` (in module `cupyx.scipy.special`), 730
- `cosh()` (in module `cupy`), 86
- `cosh()` (in module `cupy.array_api`), 899
- `cosine()` (in module `cupyx.scipy.signal.windows`), 606
- `cosine()` (in module `cupyx.scipy.spatial.distance`), 697
- `cosml()` (in module `cupyx.scipy.special`), 731
- `cotdg()` (in module `cupyx.scipy.special`), 730
- `count_nonzero()` (`cupyx.scipy.sparse.coo_matrix method`), 627
- `count_nonzero()` (`cupyx.scipy.sparse.csc_matrix method`), 636
- `count_nonzero()` (`cupyx.scipy.sparse.csr_matrix method`), 646
- `count_nonzero()` (`cupyx.scipy.sparse.dia_matrix method`), 656
- `count_nonzero()` (`cupyx.scipy.sparse.spmatrix method`), 662
- `count_nonzero()` (in module `cupy`), 286
- `cov()` (in module `cupy`), 297
- `cross()` (in module `cupy`), 205
- `csc_matrix` (class in `cupyx.scipy.sparse`), 633
- `csd()` (in module `cupyx.scipy.signal`), 575
- `cspline1d()` (in module `cupyx.scipy.signal`), 463
- `cspline1d_eval()` (in module `cupyx.scipy.signal`), 465
- `cspline2d()` (in module `cupyx.scipy.signal`), 464
- `csr_matrix` (class in `cupyx.scipy.sparse`), 644
- `cstruct` (`cupy.ndarray attribute`), 68
- `cstruct` (`cupyx.distributed.array.DistributedArray attribute`), 847
- `cuArr` (`cupy.cuda.texture.ResourceDescriptor attribute`), 793
- `CubicHermiteSpline` (class in `cupyx.scipy.interpolate`), 353
- `cublas_handle` (`cupy.cuda.Device attribute`), 750
- `CUDA_PATH`, 853
- `CUDAArray` (class in `cupy.cuda.texture`), 790
- `cumprod()` (`cupy.ndarray method`), 61
- `cumprod()` (`cupyx.distributed.array.DistributedArray method`), 843
- `cumprod()` (in module `cupy`), 201
- `cumsum()` (`cupy.ndarray method`), 61
- `cumsum()` (`cupyx.distributed.array.DistributedArray method`), 843
- `cumsum()` (in module `cupy`), 202
- `cupy`
  - module, 1
- `cupy.array_api`
  - module, 896
- `cupy.fft`
  - module, 138
- `cupy.linalg`
  - module, 173
- `cupy.polynomial.polynomial`
  - module, 217
- `cupy.polynomial.polyutils`
  - module, 220
- `cupy.random`
  - module, 227
- `cupy.testing`
  - module, 301
- `CUPY_ACCELERATORS`, 40, 176, 853, 922, 927
- `CUPY_CACHE_DIR`, 40, 851
- `CUPY_CACHE_SAVE_CUDA_SOURCE`, 851
- `CUPY_CUDA_ARRAY_INTERFACE_EXPORT_VERSION`, 43
- `CUPY_CUDA_ARRAY_INTERFACE_SYNC`, 43
- `CUPY_DLPACK_EXPORT_VERSION`, 48
- `cupyx.distributed`
  - module, 836
- `cupyx.distributed.array`
  - module, 841
- `cupyx.optimizing`
  - module, 747
- `cupyx.scipy`
  - module, 318
- `cupyx.scipy.fft`
  - module, 318
- `cupyx.scipy.fftpack`
  - module, 340
- `cupyx.scipy.interpolate`
  - module, 347
- `cupyx.scipy.linalg`
  - module, 392
- `cupyx.scipy.ndimage`

- module, 405
- cupyx.scipy.signal
  - module, 455
- cupyx.scipy.signal.windows
  - module, 596
- cupyx.scipy.sparse
  - module, 624
- cupyx.scipy.sparse.csgraph
  - module, 690
- cupyx.scipy.sparse.linalg
  - module, 676
- cupyx.scipy.spatial
  - module, 691
- cupyx.scipy.spatial.distance
  - module, 692
- cupyx.scipy.special
  - module, 700
- cupyx.scipy.stats
  - module, 733
- cusolver\_handle (cupy.cuda.Device attribute), 750
- cusolver\_sp\_handle (cupy.cuda.Device attribute), 750
- cusparse\_handle (cupy.cuda.Device attribute), 750
- cwt() (in module cupyx.scipy.signal), 564
- CZT (class in cupyx.scipy.signal), 592
- czt() (in module cupyx.scipy.signal), 590
- czt\_points() (in module cupyx.scipy.signal), 596

## D

- D (cupyx.scipy.signal.StateSpace attribute), 541
- data (cupy.ndarray attribute), 68
- data (cupyx.distributed.array.DistributedArray attribute), 847
- dbode() (in module cupyx.scipy.signal), 553
- dct() (in module cupyx.scipy.fft), 330
- dctn() (in module cupyx.scipy.fft), 332
- DebugPrintHook (class in cupy.cuda.memory\_hooks), 774
- decimate() (in module cupyx.scipy.signal), 479
- deconvolve() (in module cupyx.scipy.signal), 476
- default\_rng() (in module cupy.random), 228
- deg2rad() (cupyx.scipy.sparse.coo\_matrix method), 627
- deg2rad() (cupyx.scipy.sparse.csc\_matrix method), 636
- deg2rad() (cupyx.scipy.sparse.csr\_matrix method), 646
- deg2rad() (cupyx.scipy.sparse.dia\_matrix method), 656
- deg2rad() (in module cupy), 88
- degrees() (in module cupy), 87
- delete() (in module cupy), 129
- den (cupyx.scipy.signal.TransferFunction attribute), 543
- depth (cupy.cuda.texture.CUDAArray attribute), 791
- deriv() (cupy.poly1d method), 222
- derivative() (cupyx.scipy.interpolate.Akima1DInterpolator method), 364
- derivative() (cupyx.scipy.interpolate.BPoly method), 373
- derivative() (cupyx.scipy.interpolate.BSpline method), 378
- derivative() (cupyx.scipy.interpolate.CubicHermiteSpline method), 355
- derivative() (cupyx.scipy.interpolate.KroghInterpolator method), 350
- derivative() (cupyx.scipy.interpolate.NdPPoly method), 391
- derivative() (cupyx.scipy.interpolate.PchipInterpolator method), 359
- derivative() (cupyx.scipy.interpolate.PPoly method), 368
- derivatives() (cupyx.scipy.interpolate.KroghInterpolator method), 351
- desc (cupy.cuda.texture.CUDAArray attribute), 791
- design\_matrix() (cupyx.scipy.interpolate.BSpline class method), 378
- destroy() (cupy.cuda.nccl.NcclCommunicator method), 798
- det() (in module cupy.linalg), 181
- detrend() (in module cupyx.scipy.signal), 480
- Device (class in cupy.cuda), 749
- device (cupy.array\_api.\_array\_object.Array attribute), 910
- device (cupy.cuda.ManagedMemory attribute), 755
- device (cupy.cuda.Memory attribute), 753
- device (cupy.cuda.MemoryAsync attribute), 754
- device (cupy.cuda.MemoryPointer attribute), 760
- device (cupy.cuda.UnownedMemory attribute), 756
- device (cupy.ndarray attribute), 68
- device (cupyx.distributed.array.DistributedArray attribute), 847
- device (cupyx.scipy.sparse.coo\_matrix attribute), 633
- device (cupyx.scipy.sparse.csc\_matrix attribute), 643
- device (cupyx.scipy.sparse.csr\_matrix attribute), 653
- device (cupyx.scipy.sparse.dia\_matrix attribute), 660
- device (cupyx.scipy.sparse.spmatrix attribute), 665
- device\_id (cupy.cuda.ManagedMemory attribute), 755
- device\_id (cupy.cuda.Memory attribute), 753
- device\_id (cupy.cuda.MemoryAsync attribute), 754
- device\_id (cupy.cuda.MemoryPointer attribute), 760
- device\_id (cupy.cuda.UnownedMemory attribute), 756
- device\_id() (cupy.cuda.nccl.NcclCommunicator method), 798
- deviceCanAccessPeer() (in module cupy.cuda.runtime), 806
- deviceEnablePeerAccess() (in module cupy.cuda.runtime), 806
- deviceGetAttribute() (in module cupy.cuda.runtime), 805
- deviceGetByPCIBusId() (in module cupy.cuda.runtime), 805
- deviceGetDefaultMemPool() (in module cupy.cuda.runtime), 805



[deviceGetLimit\(\)](#) (in module `cupy.cuda.runtime`), 807  
[deviceGetMemPool\(\)](#) (in module `cupy.cuda.runtime`), 805  
[deviceGetPCIBusId\(\)](#) (in module `cupy.cuda.runtime`), 805  
[devices](#) (`cupyx.distributed.array.DistributedArray` attribute), 847  
[deviceSetLimit\(\)](#) (in module `cupy.cuda.runtime`), 807  
[deviceSetMemPool\(\)](#) (in module `cupy.cuda.runtime`), 806  
[deviceSynchronize\(\)](#) (in module `cupy.cuda.runtime`), 806  
[dfreqresp\(\)](#) (in module `cupyx.scipy.signal`), 552  
[dft\(\)](#) (in module `cupyx.scipy.linalg`), 399  
[dia\\_matrix](#) (class in `cupyx.scipy.sparse`), 654  
[diag\(\)](#) (in module `cupy`), 110  
[diag\\_indices\(\)](#) (in module `cupy`), 160  
[diag\\_indices\\_from\(\)](#) (in module `cupy`), 161  
[diagflat\(\)](#) (in module `cupy`), 111  
[diagonal\(\)](#) (`cupy.ndarray` method), 61  
[diagonal\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 843  
[diagonal\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 627  
[diagonal\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 636  
[diagonal\(\)](#) (`cupyx.scipy.sparse.csr_matrix` method), 647  
[diagonal\(\)](#) (`cupyx.scipy.sparse.dia_matrix` method), 656  
[diagonal\(\)](#) (`cupyx.scipy.sparse.spmatrix` method), 662  
[diagonal\(\)](#) (in module `cupy`), 163  
[diags\(\)](#) (in module `cupyx.scipy.sparse`), 668  
[diff\(\)](#) (in module `cupy`), 203  
[digamma\(\)](#) (in module `cupyx.scipy.special`), 723  
[digitize\(\)](#) (in module `cupy`), 301  
[dimpulse\(\)](#) (in module `cupyx.scipy.signal`), 551  
[dirichlet\(\)](#) (`cupy.random.Generator` method), 230  
[dirichlet\(\)](#) (`cupy.random.RandomState` method), 243  
[dirichlet\(\)](#) (in module `cupy.random`), 254  
[distance\\_matrix\(\)](#) (in module `cupyx.scipy.spatial`), 692  
[distance\\_matrix\(\)](#) (in module `cupyx.scipy.spatial.distance`), 694  
[distance\\_transform\\_edt\(\)](#) (in module `cupyx.scipy.ndimage`), 446  
[distributed\\_array\(\)](#) (in module `cupyx.distributed.array`), 841  
[DistributedArray](#) (class in `cupyx.distributed.array`), 841  
[divide\(\)](#) (in module `cupy`), 77  
[divide\(\)](#) (in module `cupy.array_api`), 899  
[divmod\(\)](#) (in module `cupy`), 80  
[dlsim\(\)](#) (in module `cupyx.scipy.signal`), 550  
[dlti](#) (class in `cupyx.scipy.signal`), 549  
[done](#) (`cupy.cuda.Event` attribute), 787  
[done](#) (`cupy.cuda.ExternalStream` attribute), 786  
[done](#) (`cupy.cuda.Stream` attribute), 782  
[dot\(\)](#) (`cupy.ndarray` method), 61  
[dot\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 843  
[dot\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 628  
[dot\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 636  
[dot\(\)](#) (`cupyx.scipy.sparse.csr_matrix` method), 647  
[dot\(\)](#) (`cupyx.scipy.sparse.dia_matrix` method), 656  
[dot\(\)](#) (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677  
[dot\(\)](#) (`cupyx.scipy.sparse.spmatrix` method), 662  
[dot\(\)](#) (in module `cupy`), 174  
[driverGetVersion\(\)](#) (in module `cupy.cuda.runtime`), 804  
[dsplit\(\)](#) (in module `cupy`), 127  
[dst\(\)](#) (in module `cupyx.scipy.fft`), 334  
[dstack\(\)](#) (in module `cupy`), 125  
[dstep\(\)](#) (in module `cupyx.scipy.signal`), 551  
[dstn\(\)](#) (in module `cupyx.scipy.fft`), 335  
[dt](#) (`cupyx.scipy.signal.dlti` attribute), 550  
[dt](#) (`cupyx.scipy.signal.lti` attribute), 539  
[dt](#) (`cupyx.scipy.signal.StateSpace` attribute), 541  
[dt](#) (`cupyx.scipy.signal.TransferFunction` attribute), 543  
[dt](#) (`cupyx.scipy.signal.ZerosPolesGain` attribute), 544  
[dtype](#) (`cupy.array_api._array_object.Array` attribute), 910  
[dtype](#) (`cupy.ndarray` attribute), 68  
[dtype](#) (`cupyx.distributed.array.DistributedArray` attribute), 847  
[dtype](#) (`cupyx.scipy.sparse.coo_matrix` attribute), 633  
[dtype](#) (`cupyx.scipy.sparse.csc_matrix` attribute), 643  
[dtype](#) (`cupyx.scipy.sparse.csr_matrix` attribute), 653  
[dtype](#) (`cupyx.scipy.sparse.dia_matrix` attribute), 660  
[dump\(\)](#) (`cupy.ndarray` method), 62  
[dump\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 843  
[dumps\(\)](#) (`cupy.ndarray` method), 62  
[dumps\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 843

## E

[ediff1d\(\)](#) (in module `cupy`), 204  
[eigh\(\)](#) (in module `cupy.linalg`), 180  
[eigsh\(\)](#) (in module `cupyx.scipy.sparse.linalg`), 686  
[eigvalsh\(\)](#) (in module `cupy.linalg`), 180  
[einsum\(\)](#) (in module `cupy`), 176  
[ElementwiseKernel](#) (class in `cupy`), 813  
[eliminate\\_zeros\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 628  
[eliminate\\_zeros\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 636

`eliminate_zeros()` (*cupyx.scipy.sparse.csr\_matrix* method), 647  
`ellip()` (*in module cupyx.scipy.signal*), 511  
`ellipap()` (*in module cupyx.scipy.signal*), 523  
`ellipord()` (*in module cupyx.scipy.signal*), 512  
`empty()` (*in module cupy*), 99  
`empty()` (*in module cupy.array\_api*), 899  
`empty_like()` (*in module cupy*), 99  
`empty_like()` (*in module cupy.array\_api*), 899  
`empty_like_pinned()` (*in module cupyx*), 739  
`empty_pinned()` (*in module cupyx*), 739  
`enable_cooperative_groups` (*cupy.RawKernel* attribute), 819  
`enable_cooperative_groups` (*cupy.RawModule* attribute), 823  
`end_capture()` (*cupy.cuda.ExternalStream* method), 784  
`end_capture()` (*cupy.cuda.Stream* method), 781  
`entr()` (*in module cupyx.scipy.special*), 717  
`entropy()` (*in module cupyx.scipy.stats*), 734  
environment variable  
  CUDA\_PATH, 851, 853  
  CUPY\_ACCELERATORS, 40, 176, 852, 853, 922, 927  
  CUPY\_CACHE\_DIR, 40, 851  
  CUPY\_CACHE\_IN\_MEMORY, 851  
  CUPY\_CACHE\_SAVE\_CUDA\_SOURCE, 851  
  CUPY\_COMPILE\_WITH\_PTX, 852  
  CUPY\_CUDA\_ARRAY\_INTERFACE\_EXPORT\_VERSION, 43, 852  
  CUPY\_CUDA\_ARRAY\_INTERFACE\_SYNC, 43, 852  
  CUPY\_CUDA\_COMPILE\_WITH\_DEBUG, 851  
  CUPY\_CUDA\_PER\_THREAD\_DEFAULT\_STREAM, 852  
  CUPY\_DISABLE\_JITIFY\_CACHE, 851  
  CUPY\_DLPACK\_EXPORT\_VERSION, 48, 852  
  CUPY\_DUMP\_CUDA\_SOURCE\_ON\_ERROR, 851  
  CUPY\_EXPERIMENTAL\_SLICE\_COPY, 851  
  CUPY\_GPU\_MEMORY\_LIMIT, 851  
  CUPY\_INSTALL\_USE\_HIP, 853  
  CUPY\_NUM\_BUILD\_JOBS, 853  
  CUPY\_NUM\_NVCC\_THREADS, 853  
  CUPY\_NVCC\_GENERATE\_CODE, 853  
  CUPY\_SEED, 851  
  CUPY\_TF32, 852  
  CUPY\_USE\_CUDA\_PYTHON, 853  
  CUTENSOR\_PATH, 853  
  NVCC, 852, 853  
`equal()` (*in module cupy*), 91  
`equal()` (*in module cupy.array\_api*), 900  
`erf()` (*in module cupyx.scipy.special*), 724  
`erfc()` (*in module cupyx.scipy.special*), 724  
`erfcinv()` (*in module cupyx.scipy.special*), 725  
`erfcx()` (*in module cupyx.scipy.special*), 724  
`erfinv()` (*in module cupyx.scipy.special*), 724  
`euclidean()` (*in module cupyx.scipy.spatial.distance*), 698  
Event (class *in cupy.cuda*), 786  
`eventCreate()` (*in module cupy.cuda.runtime*), 811  
`eventCreateWithFlags()` (*in module cupy.cuda.runtime*), 811  
`eventDestroy()` (*in module cupy.cuda.runtime*), 811  
`eventElapsedTime()` (*in module cupy.cuda.runtime*), 811  
`eventQuery()` (*in module cupy.cuda.runtime*), 811  
`eventRecord()` (*in module cupy.cuda.runtime*), 811  
`eventSynchronize()` (*in module cupy.cuda.runtime*), 812  
`exp()` (*in module cupy*), 81  
`exp()` (*in module cupy.array\_api*), 900  
`exp1()` (*in module cupyx.scipy.special*), 726  
`exp10()` (*in module cupyx.scipy.special*), 729  
`exp2()` (*in module cupy*), 81  
`exp2()` (*in module cupyx.scipy.special*), 729  
`expand_dims()` (*in module cupy*), 120  
`expand_dims()` (*in module cupy.array\_api*), 900  
`expi()` (*in module cupyx.scipy.special*), 726  
`expit()` (*in module cupyx.scipy.special*), 714  
`expm()` (*in module cupyx.scipy.linalg*), 394  
`expm1()` (*cupyx.scipy.sparse.coo\_matrix* method), 628  
`expm1()` (*cupyx.scipy.sparse.csc\_matrix* method), 636  
`expm1()` (*cupyx.scipy.sparse.csr\_matrix* method), 647  
`expm1()` (*cupyx.scipy.sparse.dia\_matrix* method), 656  
`expm1()` (*in module cupy*), 82  
`expm1()` (*in module cupy.array\_api*), 900  
`expm1()` (*in module cupyx.scipy.special*), 731  
`expn()` (*in module cupyx.scipy.special*), 726  
`exponential()` (*cupy.random.Generator* method), 230  
`exponential()` (*cupy.random.RandomState* method), 243  
`exponential()` (*in module cupy.random*), 254  
`exponential()` (*in module cupyx.scipy.signal.windows*), 607  
`exprel()` (*in module cupyx.scipy.special*), 727  
`extend()` (*cupyx.scipy.interpolate.Akima1DInterpolator* method), 364  
`extend()` (*cupyx.scipy.interpolate.BPoly* method), 373  
`extend()` (*cupyx.scipy.interpolate.CubicHermiteSpline* method), 355  
`extend()` (*cupyx.scipy.interpolate.PchipInterpolator* method), 360  
`extend()` (*cupyx.scipy.interpolate.PPoly* method), 369  
ExternalStream (class *in cupy.cuda*), 783  
`extract()` (*in module cupy*), 285  
`extrapolate` (*cupyx.scipy.interpolate.Akima1DInterpolator* attribute), 366  
`extrapolate` (*cupyx.scipy.interpolate.BPoly* attribute), 375

- `extrapolate` (`cupyx.scipy.interpolate.CubicHermiteSpline` attribute), 357
- `extrapolate` (`cupyx.scipy.interpolate.PchipInterpolator` attribute), 362
- `extrapolate` (`cupyx.scipy.interpolate.PPoly` attribute), 371
- `extrema()` (in module `cupyx.scipy.ndimage`), 431
- `eye()` (in module `cupy`), 100
- `eye()` (in module `cupy.array_api`), 900
- `eye()` (in module `cupyx.scipy.sparse`), 666
- ## F
- `f()` (`cupy.random.Generator` method), 230
- `f()` (`cupy.random.RandomState` method), 243
- `f()` (in module `cupy.random`), 255
- `fabs()` (in module `cupy`), 80
- `factorized()` (in module `cupyx.scipy.sparse.linalg`), 680
- `fdtr()` (in module `cupyx.scipy.special`), 707
- `fdtrc()` (in module `cupyx.scipy.special`), 708
- `fdtri()` (in module `cupyx.scipy.special`), 708
- `fft()` (in module `cupy.fft`), 138
- `fft()` (in module `cupyx.scipy.fft`), 319
- `fft()` (in module `cupyx.scipy.fftpack`), 341
- `fft2()` (in module `cupy.fft`), 139
- `fft2()` (in module `cupyx.scipy.fft`), 320
- `fft2()` (in module `cupyx.scipy.fftpack`), 342
- `fftconvolve()` (in module `cupyx.scipy.signal`), 457
- `fftfreq()` (in module `cupy.fft`), 145
- `fftfreq()` (in module `cupyx.scipy.fft`), 339
- `fftn()` (in module `cupy.fft`), 140
- `fftn()` (in module `cupyx.scipy.fft`), 322
- `fftn()` (in module `cupyx.scipy.fftpack`), 344
- `fftshift()` (in module `cupy.fft`), 146
- `fftshift()` (in module `cupyx.scipy.fft`), 338
- `fht()` (in module `cupyx.scipy.fft`), 337
- `fiedler()` (in module `cupyx.scipy.linalg`), 400
- `fiedler_companion()` (in module `cupyx.scipy.linalg`), 400
- `file_path` (`cupy.RawKernel` attribute), 819
- `file_path` (`cupy.RawModule` attribute), 823
- `fill()` (`cupy.ndarray` method), 62
- `fill()` (`cupyx.distributed.array.DistributedArray` method), 843
- `fill_diagonal()` (in module `cupy`), 167
- `filtfilt()` (in module `cupyx.scipy.signal`), 473
- `find()` (in module `cupyx.scipy.sparse`), 674
- `find_peaks()` (in module `cupyx.scipy.signal`), 568
- `findfreqs()` (in module `cupyx.scipy.signal`), 486
- `finfo()` (in module `cupy.array_api`), 900
- `firls()` (in module `cupyx.scipy.signal`), 494
- `firwin()` (in module `cupyx.scipy.signal`), 491
- `firwin2()` (in module `cupyx.scipy.signal`), 493
- `fix()` (in module `cupy`), 198
- `flags` (`cupy.cuda.texture.CUDAArray` attribute), 791
- `flags` (`cupy.ndarray` attribute), 68
- `flags` (`cupyx.distributed.array.DistributedArray` attribute), 847
- `flat` (`cupy.ndarray` attribute), 68
- `flat` (`cupyx.distributed.array.DistributedArray` attribute), 847
- `flatiter` (class in `cupy`), 167
- `flatnonzero()` (in module `cupy`), 284
- `flatten()` (`cupy.ndarray` method), 62
- `flatten()` (`cupyx.distributed.array.DistributedArray` method), 844
- `flattop()` (in module `cupyx.scipy.signal.windows`), 609
- `flip()` (in module `cupy`), 132
- `flip()` (in module `cupy.array_api`), 900
- `fliplr()` (in module `cupy`), 132
- `flipud()` (in module `cupy`), 133
- `float_power()` (in module `cupy`), 79
- `floor()` (`cupyx.scipy.sparse.coo_matrix` method), 628
- `floor()` (`cupyx.scipy.sparse.csc_matrix` method), 636
- `floor()` (`cupyx.scipy.sparse.csr_matrix` method), 647
- `floor()` (`cupyx.scipy.sparse.dia_matrix` method), 656
- `floor()` (in module `cupy`), 95
- `floor()` (in module `cupy.array_api`), 900
- `floor_divide()` (in module `cupy`), 78
- `floor_divide()` (in module `cupy.array_api`), 901
- `fmax()` (in module `cupy`), 92
- `fmin()` (in module `cupy`), 93
- `fmod()` (in module `cupy`), 80
- `for_all_dtypes()` (in module `cupy.testing`), 311
- `for_all_dtypes_combination()` (in module `cupy.testing`), 313
- `for_CF_orders()` (in module `cupy.testing`), 315
- `for_complex_dtypes()` (in module `cupy.testing`), 313
- `for_dtypes()` (in module `cupy.testing`), 310
- `for_dtypes_combination()` (in module `cupy.testing`), 313
- `for_float_dtypes()` (in module `cupy.testing`), 312
- `for_int_dtypes()` (in module `cupy.testing`), 312
- `for_int_dtypes_combination()` (in module `cupy.testing`), 314
- `for_orders()` (in module `cupy.testing`), 315
- `for_signed_dtypes()` (in module `cupy.testing`), 312
- `for_signed_dtypes_combination()` (in module `cupy.testing`), 314
- `for_unsigned_dtypes()` (in module `cupy.testing`), 312
- `for_unsigned_dtypes_combination()` (in module `cupy.testing`), 314
- `format` (`cupyx.scipy.sparse.coo_matrix` attribute), 633
- `format` (`cupyx.scipy.sparse.csc_matrix` attribute), 643
- `format` (`cupyx.scipy.sparse.csr_matrix` attribute), 653
- `format` (`cupyx.scipy.sparse.dia_matrix` attribute), 660
- `format_float_positional()` (in module `cupy`), 172
- `format_float_scientific()` (in module `cupy`), 173

[fourier\\_ellipsoid\(\)](#) (in module `cupyx.scipy.ndimage`), 422  
[fourier\\_gaussian\(\)](#) (in module `cupyx.scipy.ndimage`), 423  
[fourier\\_shift\(\)](#) (in module `cupyx.scipy.ndimage`), 423  
[fourier\\_uniform\(\)](#) (in module `cupyx.scipy.ndimage`), 424  
[free\(\)](#) (`cupy.cuda.PinnedMemoryPool` method), 769  
[free\(\)](#) (in module `cupy.cuda.runtime`), 808  
[free\\_all\\_blocks\(\)](#) (`cupy.cuda.MemoryAsyncPool` method), 767  
[free\\_all\\_blocks\(\)](#) (`cupy.cuda.MemoryPool` method), 764  
[free\\_all\\_blocks\(\)](#) (`cupy.cuda.PinnedMemoryPool` method), 769  
[free\\_all\\_free\(\)](#) (`cupy.cuda.MemoryPool` method), 764  
[free\\_bytes\(\)](#) (`cupy.cuda.MemoryAsyncPool` method), 767  
[free\\_bytes\(\)](#) (`cupy.cuda.MemoryPool` method), 764  
[free\\_postprocess\(\)](#) (`cupy.cuda.memory_hooks.DebugPrintHook` method), 775  
[free\\_postprocess\(\)](#) (`cupy.cuda.memory_hooks.LineProfilerHook` method), 778  
[free\\_postprocess\(\)](#) (`cupy.cuda.MemoryHook` method), 773  
[free\\_preprocess\(\)](#) (`cupy.cuda.memory_hooks.DebugPrintHook` method), 775  
[free\\_preprocess\(\)](#) (`cupy.cuda.memory_hooks.LineProfilerHook` method), 778  
[free\\_preprocess\(\)](#) (`cupy.cuda.MemoryHook` method), 773  
[freeArray\(\)](#) (in module `cupy.cuda.runtime`), 808  
[freeAsync\(\)](#) (in module `cupy.cuda.runtime`), 808  
[freeHost\(\)](#) (in module `cupy.cuda.runtime`), 808  
[freq\\_shift\(\)](#) (in module `cupyx.signal`), 744  
[freqresp\(\)](#) (`cupyx.scipy.signal.dlti` method), 549  
[freqresp\(\)](#) (`cupyx.scipy.signal.lti` method), 538  
[freqresp\(\)](#) (in module `cupyx.scipy.signal`), 547  
[freqs\(\)](#) (in module `cupyx.scipy.signal`), 487  
[freqs\\_zpk\(\)](#) (in module `cupyx.scipy.signal`), 488  
[freqz\(\)](#) (in module `cupyx.scipy.signal`), 488  
[freqz\\_zpk\(\)](#) (in module `cupyx.scipy.signal`), 490  
[frexp\(\)](#) (in module `cupy`), 95  
[from\\_bernstein\\_basis\(\)](#) (`cupyx.scipy.interpolate.Akima1DInterpolator` class method), 364  
[from\\_bernstein\\_basis\(\)](#) (`cupyx.scipy.interpolate.CubicHermiteSpline` class method), 355  
[from\\_bernstein\\_basis\(\)](#) (`cupyx.scipy.interpolate.PchipInterpolator` class method), 360  
[from\\_bernstein\\_basis\(\)](#) (`cupyx.scipy.interpolate.PPoly` class method), 369  
[from\\_derivatives\(\)](#) (`cupyx.scipy.interpolate.BPoly` class method), 373  
[from\\_dlpack\(\)](#) (in module `cupy`), 747  
[from\\_dlpack\(\)](#) (in module `cupy.array_api`), 901  
[from\\_pci\\_bus\\_id\(\)](#) (`cupy.cuda.Device` method), 749  
[from\\_power\\_basis\(\)](#) (`cupyx.scipy.interpolate.BPoly` class method), 374  
[from\\_spline\(\)](#) (`cupyx.scipy.interpolate.Akima1DInterpolator` class method), 364  
[from\\_spline\(\)](#) (`cupyx.scipy.interpolate.CubicHermiteSpline` class method), 355  
[from\\_spline\(\)](#) (`cupyx.scipy.interpolate.PchipInterpolator` class method), 360  
[from\\_spline\(\)](#) (`cupyx.scipy.interpolate.PPoly` class method), 369  
[frombuffer\(\)](#) (in module `cupy`), 105  
[fromfile\(\)](#) (in module `cupy`), 105  
[fromfunction\(\)](#) (in module `cupy`), 106  
[fromhook\(\)](#) (in module `cupy`), 106  
[fromstring\(\)](#) (in module `cupy`), 106  
[full\(\)](#) (in module `cupy`), 103  
[full\(\)](#) (in module `cupy.array_api`), 901  
[full\\_like\(\)](#) (in module `cupy`), 103  
[full\\_like\(\)](#) (in module `cupy.array_api`), 901  
[fuse\(\)](#) (in module `cupy`), 823  
[gain](#) (`cupyx.scipy.signal.ZerosPolesGain` attribute), 544  
[gamma\(\)](#) (`cupy.random.Generator` method), 231  
[gamma\(\)](#) (`cupy.random.RandomState` method), 243  
[gamma\(\)](#) (in module `cupy.random`), 255  
[gamma\(\)](#) (in module `cupyx.scipy.special`), 718  
[gammaln\(\)](#) (in module `cupyx.scipy.special`), 719  
[gammalncc\(\)](#) (in module `cupyx.scipy.special`), 720  
[gammalnccinv\(\)](#) (in module `cupyx.scipy.special`), 720  
[gammalnccinv\(\)](#) (in module `cupyx.scipy.special`), 719  
[gammaln\(\)](#) (in module `cupyx.scipy.special`), 718  
[gammatone\(\)](#) (in module `cupyx.scipy.signal`), 497  
[gather\(\)](#) (`cupyx.distributed.NCCLBackend` method), 838  
[gauss\\_spline\(\)](#) (in module `cupyx.scipy.signal`), 463  
[gaussian\(\)](#) (in module `cupyx.scipy.signal.windows`), 610  
[gaussian\\_filter\(\)](#) (in module `cupyx.scipy.ndimage`), 408  
[gaussian\\_filter1d\(\)](#) (in module `cupyx.scipy.ndimage`), 409  
[gaussian\\_gradient\\_magnitude\(\)](#) (in module `cupyx.scipy.ndimage`), 410  
[gaussian\\_laplace\(\)](#) (in module `cupyx.scipy.ndimage`), 410  
[gausspulse\(\)](#) (in module `cupyx.scipy.signal`), 555



- [gcd\(\)](#) (in module *cupy*), 83  
[gdtr\(\)](#) (in module *cupyx.scipy.special*), 709  
[gdtrc\(\)](#) (in module *cupyx.scipy.special*), 709  
[general\\_cosine\(\)](#) (in module *cupyx.scipy.signal.windows*), 611  
[general\\_gaussian\(\)](#) (in module *cupyx.scipy.signal.windows*), 612  
[general\\_hamming\(\)](#) (in module *cupyx.scipy.signal.windows*), 613  
[GeneralizedUNFunc](#) (class in *cupyx*), 96  
[generate\\_binary\\_structure\(\)](#) (in module *cupyx.scipy.ndimage*), 449  
[Generator](#) (class in *cupy.random*), 228  
[generator](#) (*cupy.random.MRG32k3a* attribute), 240  
[generator](#) (*cupy.random.Philox4x3210* attribute), 241  
[generator](#) (*cupy.random.XORWOW* attribute), 239  
[generic\\_filter\(\)](#) (in module *cupyx.scipy.ndimage*), 411  
[generic\\_filter1d\(\)](#) (in module *cupyx.scipy.ndimage*), 412  
[generic\\_gradient\\_magnitude\(\)](#) (in module *cupyx.scipy.ndimage*), 413  
[generic\\_laplace\(\)](#) (in module *cupyx.scipy.ndimage*), 414  
[genfromtxt\(\)](#) (in module *cupy*), 171  
[geometric\(\)](#) (*cupy.random.Generator* method), 231  
[geometric\(\)](#) (*cupy.random.RandomState* method), 243  
[geometric\(\)](#) (in module *cupy.random*), 256  
[get\(\)](#) (*cupy.ndarray* method), 62  
[get\(\)](#) (*cupy.poly1d* method), 222  
[get\(\)](#) (*cupyx.distributed.array.DistributedArray* method), 844  
[get\(\)](#) (*cupyx.scipy.sparse.coo\_matrix* method), 628  
[get\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 636  
[get\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 647  
[get\(\)](#) (*cupyx.scipy.sparse.dia\_matrix* method), 656  
[get\(\)](#) (*cupyx.scipy.sparse.spmatrix* method), 662  
[get\\_allocator\(\)](#) (in module *cupy.cuda*), 763  
[get\\_array\\_module\(\)](#) (in module *cupy*), 72  
[get\\_array\\_module\(\)](#) (in module *cupyx.scipy*), 72  
[get\\_build\\_version\(\)](#) (in module *cupy.cuda.nccl*), 800  
[get\\_channel\\_format\(\)](#) (*cupy.cuda.texture.ChannelFormatDescriptor* method), 789  
[get\\_current\\_stream\(\)](#) (in module *cupy.cuda*), 786  
[get\\_default\\_memory\\_pool\(\)](#) (in module *cupy*), 752  
[get\\_default\\_pinned\\_memory\\_pool\(\)](#) (in module *cupy*), 752  
[get\\_elapsed\\_time\(\)](#) (in module *cupy.cuda*), 787  
[get\\_fft\\_plan\(\)](#) (in module *cupyx.scipy.fftpack*), 346  
[get\\_function\(\)](#) (*cupy.RawModule* method), 821  
[get\\_global\(\)](#) (*cupy.RawModule* method), 822  
[get\\_limit\(\)](#) (*cupy.cuda.MemoryAsyncPool* method), 767  
[get\\_limit\(\)](#) (*cupy.cuda.MemoryPool* method), 764  
[get\\_local\\_runtime\\_version\(\)](#) (in module *cupy.cuda*), 801  
[get\\_plan\\_cache\(\)](#) (in module *cupy.fft.config*), 149  
[get\\_random\\_state\(\)](#) (in module *cupy.random*), 275  
[get\\_resource\\_desc\(\)](#) (*cupy.cuda.texture.ResourceDescriptor* method), 792  
[get\\_shape\(\)](#) (*cupyx.scipy.sparse.coo\_matrix* method), 628  
[get\\_shape\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 637  
[get\\_shape\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 647  
[get\\_shape\(\)](#) (*cupyx.scipy.sparse.dia\_matrix* method), 657  
[get\\_shape\(\)](#) (*cupyx.scipy.sparse.spmatrix* method), 662  
[get\\_texture\\_desc\(\)](#) (*cupy.cuda.texture.TextureDescriptor* method), 794  
[get\\_unique\\_id\(\)](#) (in module *cupy.cuda.nccl*), 800  
[get\\_version\(\)](#) (in module *cupy.cuda.nccl*), 800  
[get\\_window\(\)](#) (in module *cupyx.scipy.signal*), 559  
[get\\_window\(\)](#) (in module *cupyx.scipy.signal.windows*), 597  
[getcol\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 637  
[getcol\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 647  
[getDevice\(\)](#) (in module *cupy.cuda.runtime*), 805  
[getDeviceCount\(\)](#) (in module *cupy.cuda.runtime*), 806  
[getDeviceProperties\(\)](#) (in module *cupy.cuda.runtime*), 805  
[getformat\(\)](#) (*cupyx.scipy.sparse.coo\_matrix* method), 628  
[getformat\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 637  
[getformat\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 648  
[getformat\(\)](#) (*cupyx.scipy.sparse.dia\_matrix* method), 657  
[getformat\(\)](#) (*cupyx.scipy.sparse.spmatrix* method), 662  
[getH\(\)](#) (*cupyx.scipy.sparse.coo\_matrix* method), 628  
[getH\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 637  
[getH\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 647  
[getH\(\)](#) (*cupyx.scipy.sparse.dia\_matrix* method), 656  
[getH\(\)](#) (*cupyx.scipy.sparse.spmatrix* method), 662  
[getmaxprint\(\)](#) (*cupyx.scipy.sparse.coo\_matrix* method), 628  
[getmaxprint\(\)](#) (*cupyx.scipy.sparse.csc\_matrix* method), 637  
[getmaxprint\(\)](#) (*cupyx.scipy.sparse.csr\_matrix* method), 648  
[getmaxprint\(\)](#) (*cupyx.scipy.sparse.dia\_matrix* method), 657  
[getmaxprint\(\)](#) (*cupyx.scipy.sparse.spmatrix* method), 662

`getnnz()` (*cupyx.scipy.sparse.coo\_matrix* method), 628  
`getnnz()` (*cupyx.scipy.sparse.csc\_matrix* method), 637  
`getnnz()` (*cupyx.scipy.sparse.csr\_matrix* method), 648  
`getnnz()` (*cupyx.scipy.sparse.dia\_matrix* method), 657  
`getnnz()` (*cupyx.scipy.sparse.spmatrix* method), 663  
`getrow()` (*cupyx.scipy.sparse.csc\_matrix* method), 637  
`getrow()` (*cupyx.scipy.sparse.csr\_matrix* method), 648  
`gmres()` (*in module cupyx.scipy.sparse.linalg*), 681  
`gradient()` (*in module cupy*), 204  
`Graph` (*class in cupy.cuda*), 788  
`graph` (*cupy.cuda.Graph* attribute), 789  
`graphExec` (*cupy.cuda.Graph* attribute), 789  
`greater()` (*in module cupy*), 90  
`greater()` (*in module cupy.array\_api*), 901  
`greater_equal()` (*in module cupy*), 90  
`greater_equal()` (*in module cupy.array\_api*), 901  
`grey_closing()` (*in module cupyx.scipy.ndimage*), 449  
`grey_dilation()` (*in module cupyx.scipy.ndimage*), 450  
`grey_erosion()` (*in module cupyx.scipy.ndimage*), 450  
`grey_opening()` (*in module cupyx.scipy.ndimage*), 451  
`grid` (*in module cupyx.jit*), 826  
`gridDim` (*in module cupyx.jit*), 826  
`gridsize` (*in module cupyx.jit*), 826  
`group_delay()` (*in module cupyx.scipy.signal*), 498  
`groupEnd()` (*in module cupy.cuda.nccl*), 801  
`groupStart()` (*in module cupy.cuda.nccl*), 800  
`gumbel()` (*cupy.random.RandomState* method), 243  
`gumbel()` (*in module cupy.random*), 256

## H

`H` (*cupyx.scipy.sparse.coo\_matrix* attribute), 633  
`H` (*cupyx.scipy.sparse.csc\_matrix* attribute), 643  
`H` (*cupyx.scipy.sparse.csr\_matrix* attribute), 653  
`H` (*cupyx.scipy.sparse.dia\_matrix* attribute), 660  
`H` (*cupyx.scipy.sparse.linalg.LinearOperator* attribute), 678  
`H` (*cupyx.scipy.sparse.spmatrix* attribute), 665  
`hadamard()` (*in module cupyx.scipy.linalg*), 401  
`hamming()` (*in module cupy*), 317  
`hamming()` (*in module cupyx.scipy.signal.windows*), 615  
`hamming()` (*in module cupyx.scipy.spatial.distance*), 698  
`hankel()` (*in module cupyx.scipy.linalg*), 401  
`hann()` (*in module cupyx.scipy.signal.windows*), 616  
`hanning()` (*in module cupy*), 317  
`has_canonical_format` (*cupyx.scipy.sparse.csc\_matrix* attribute), 643  
`has_canonical_format` (*cupyx.scipy.sparse.csr\_matrix* attribute), 653  
`has_sorted_indices` (*cupyx.scipy.sparse.csc\_matrix* attribute), 643  
`has_sorted_indices` (*cupyx.scipy.sparse.csr\_matrix* attribute), 654  
`heaviside()` (*in module cupy*), 81  
`height` (*cupy.cuda.texture.CUDAarray* attribute), 791

`hellinger()` (*in module cupyx.scipy.spatial.distance*), 700  
`helmert()` (*in module cupyx.scipy.linalg*), 402  
`hfft()` (*in module cupy.fft*), 144  
`hfft()` (*in module cupyx.scipy.fft*), 327  
`hfft2()` (*in module cupyx.scipy.fft*), 328  
`hfftn()` (*in module cupyx.scipy.fft*), 329  
`hilbert()` (*in module cupyx.scipy.linalg*), 402  
`hilbert()` (*in module cupyx.scipy.signal*), 478  
`hilbert2()` (*in module cupyx.scipy.signal*), 479  
`histogram()` (*in module cupy*), 298  
`histogram()` (*in module cupyx.scipy.ndimage*), 432  
`histogram2d()` (*in module cupy*), 299  
`histogramdd()` (*in module cupy*), 300  
`hostAlloc()` (*in module cupy.cuda.runtime*), 807  
`hostRegister()` (*in module cupy.cuda.runtime*), 808  
`hostUnregister()` (*in module cupy.cuda.runtime*), 808  
`hsplit()` (*in module cupy*), 127  
`hstack()` (*in module cupy*), 124  
`hstack()` (*in module cupyx.scipy.sparse*), 671  
`huber()` (*in module cupyx.scipy.special*), 717  
`hypergeometric()` (*cupy.random.Generator* method), 232  
`hypergeometric()` (*cupy.random.RandomState* method), 244  
`hypergeometric()` (*in module cupy.random*), 257  
`hypot()` (*in module cupy*), 86

## I

`i0()` (*in module cupy*), 206  
`i0()` (*in module cupyx.scipy.special*), 704  
`i0e()` (*in module cupyx.scipy.special*), 704  
`i1()` (*in module cupyx.scipy.special*), 704  
`i1e()` (*in module cupyx.scipy.special*), 704  
`id` (*cupy.cuda.Device* attribute), 750  
`idct()` (*in module cupyx.scipy.fft*), 331  
`idctn()` (*in module cupyx.scipy.fft*), 333  
`identity` (*cupy.ReductionKernel* attribute), 816  
`identity()` (*in module cupy*), 100  
`identity()` (*in module cupyx.scipy.sparse*), 667  
`idst()` (*in module cupyx.scipy.fft*), 334  
`idstn()` (*in module cupyx.scipy.fft*), 336  
`ifft()` (*in module cupy.fft*), 139  
`ifft()` (*in module cupyx.scipy.fft*), 320  
`ifft()` (*in module cupyx.scipy.fftpack*), 342  
`ifft2()` (*in module cupy.fft*), 140  
`ifft2()` (*in module cupyx.scipy.fft*), 321  
`ifft2()` (*in module cupyx.scipy.fftpack*), 343  
`ifftn()` (*in module cupy.fft*), 141  
`ifftn()` (*in module cupyx.scipy.fft*), 322  
`ifftn()` (*in module cupyx.scipy.fftpack*), 344  
`ifftshift()` (*in module cupy.fft*), 147  
`ifftshift()` (*in module cupyx.scipy.fft*), 338  
`ifht()` (*in module cupyx.scipy.fft*), 337

- `ihfft()` (in module `cupy.fft`), 145
- `ihfft()` (in module `cupyx.scipy.fft`), 328
- `ihfft2()` (in module `cupyx.scipy.fft`), 329
- `ihfftn()` (in module `cupyx.scipy.fft`), 330
- `iinfo()` (in module `cupy.array_api`), 901
- `iircomb()` (in module `cupyx.scipy.signal`), 518
- `iirdesign()` (in module `cupyx.scipy.signal`), 499
- `iirfilter()` (in module `cupyx.scipy.signal`), 501
- `iirnotch()` (in module `cupyx.scipy.signal`), 519
- `iirpeak()` (in module `cupyx.scipy.signal`), 520
- `imag` (`cupy.ndarray` attribute), 69
- `imag` (`cupyx.distributed.array.DistributedArray` attribute), 847
- `imag()` (in module `cupy`), 208
- `impulse()` (`cupyx.scipy.signal.dlti` method), 549
- `impulse()` (`cupyx.scipy.signal.lti` method), 538
- `impulse()` (in module `cupyx.scipy.signal`), 546
- `in1d()` (in module `cupy`), 276
- `in_params` (`cupy.ElementwiseKernel` attribute), 814
- `in_params` (`cupy.ReductionKernel` attribute), 817
- `index_map` (`cupyx.distributed.array.DistributedArray` attribute), 847
- `indices()` (in module `cupy`), 154
- `init_process_group()` (in module `cupyx.distributed`), 836
- `initAll()` (`cupy.cuda.nccl.NcclCommunicator` static method), 799
- `inner()` (in module `cupy`), 174
- `integ()` (`cupy.poly1d` method), 222
- `integers()` (`cupy.random.Generator` method), 232
- `integrate()` (`cupyx.scipy.interpolate.Akima1DInterpolator` method), 365
- `integrate()` (`cupyx.scipy.interpolate.BPoly` method), 374
- `integrate()` (`cupyx.scipy.interpolate.BSpline` method), 379
- `integrate()` (`cupyx.scipy.interpolate.CubicHermiteSpline` method), 356
- `integrate()` (`cupyx.scipy.interpolate.NdPPoly` method), 391
- `integrate()` (`cupyx.scipy.interpolate.PchipInterpolator` method), 360
- `integrate()` (`cupyx.scipy.interpolate.PPoly` method), 369
- `integrate_1d()` (`cupyx.scipy.interpolate.NdPPoly` method), 391
- `interp()` (in module `cupy`), 211
- `interp_n()` (in module `cupyx.scipy.interpolate`), 384
- `intersect1d()` (in module `cupy`), 276
- `inv()` (in module `cupy.linalg`), 185
- `inv_boxcox()` (in module `cupyx.scipy.special`), 716
- `inv_boxcox1p()` (in module `cupyx.scipy.special`), 716
- `invert()` (in module `cupy`), 89
- `invres()` (in module `cupyx.scipy.signal`), 507
- `invresz()` (in module `cupyx.scipy.signal`), 508
- `ipcCloseMemHandle()` (in module `cupy.cuda.runtime`), 812
- `ipcGetEventHandle()` (in module `cupy.cuda.runtime`), 812
- `ipcGetMemHandle()` (in module `cupy.cuda.runtime`), 812
- `ipcOpenEventHandle()` (in module `cupy.cuda.runtime`), 812
- `ipcOpenMemHandle()` (in module `cupy.cuda.runtime`), 812
- `irfft()` (in module `cupy.fft`), 142
- `irfft()` (in module `cupyx.scipy.fft`), 324
- `irfft()` (in module `cupyx.scipy.fftpack`), 346
- `irfft2()` (in module `cupy.fft`), 143
- `irfft2()` (in module `cupyx.scipy.fft`), 325
- `irfftn()` (in module `cupy.fft`), 144
- `irfftn()` (in module `cupyx.scipy.fft`), 326
- `is_capturing()` (`cupy.cuda.ExternalStream` method), 784
- `is_capturing()` (`cupy.cuda.Stream` method), 781
- `isclose()` (in module `cupy`), 195
- `iscomplex()` (in module `cupy`), 190
- `iscomplexobj()` (in module `cupy`), 190
- `isfinite()` (in module `cupy`), 93
- `isfinite()` (in module `cupy.array_api`), 901
- `isfortran()` (in module `cupy`), 191
- `isin()` (in module `cupy`), 277
- `isinf()` (in module `cupy`), 94
- `isinf()` (in module `cupy.array_api`), 902
- `isnan()` (in module `cupy`), 94
- `isnan()` (in module `cupy.array_api`), 902
- `isneginf()` (in module `cupy`), 188
- `isposinf()` (in module `cupy`), 189
- `isreal()` (in module `cupy`), 192
- `isrealobj()` (in module `cupy`), 193
- `isscalar()` (in module `cupy`), 193
- `issparse()` (in module `cupyx.scipy.sparse`), 674
- `isspmatrix()` (in module `cupyx.scipy.sparse`), 674
- `isspmatrix_coo()` (in module `cupyx.scipy.sparse`), 675
- `isspmatrix_csc()` (in module `cupyx.scipy.sparse`), 675
- `isspmatrix_csr()` (in module `cupyx.scipy.sparse`), 675
- `isspmatrix_dia()` (in module `cupyx.scipy.sparse`), 675
- `istft()` (in module `cupyx.scipy.signal`), 585
- `item()` (`cupy.ndarray` method), 63
- `item()` (`cupyx.distributed.array.DistributedArray` method), 844
- `itemsize` (`cupy.ndarray` attribute), 69
- `itemsize` (`cupyx.distributed.array.DistributedArray` attribute), 847
- `iterate_structure()` (in module `cupyx.scipy.ndimage`), 452
- `ix_()` (in module `cupy`), 157

## J

`j0()` (in module `cupyx.scipy.special`), 701  
`j1()` (in module `cupyx.scipy.special`), 701  
`jensenshannon()` (in module `cupyx.scipy.spatial.distance`), 698

## K

`k0()` (in module `cupyx.scipy.special`), 701  
`k0e()` (in module `cupyx.scipy.special`), 702  
`k1()` (in module `cupyx.scipy.special`), 702  
`k1e()` (in module `cupyx.scipy.special`), 702  
`kaiser()` (in module `cupy`), 317  
`kaiser()` (in module `cupyx.scipy.signal.windows`), 617  
`kaiser_atten()` (in module `cupyx.scipy.signal`), 502  
`kaiser_beta()` (in module `cupyx.scipy.signal`), 503  
`kaiserord()` (in module `cupyx.scipy.signal`), 503  
`kernel` (`cupy.RawKernel` attribute), 819  
`kl_div()` (in module `cupyx.scipy.special`), 717  
`kl_divergence()` (in module `cupyx.scipy.spatial.distance`), 700  
`krogh_interpolate()` (in module `cupyx.scipy.interpolate`), 352  
`KroghInterpolator` (class in `cupyx.scipy.interpolate`), 350  
`kron()` (in module `cupy`), 177  
`kron()` (in module `cupyx.scipy.linalg`), 403  
`kron()` (in module `cupyx.scipy.sparse`), 667  
`kronsum()` (in module `cupyx.scipy.sparse`), 668  
`kwargs` (`cupy.ElementwiseKernel` attribute), 814

## L

`label()` (in module `cupyx.scipy.ndimage`), 432  
`labeled_comprehension()` (in module `cupyx.scipy.ndimage`), 433  
`laneid` (in module `cupyx.jit`), 827  
`laplace()` (`cupy.random.RandomState` method), 244  
`laplace()` (in module `cupy.random`), 257  
`laplace()` (in module `cupyx.scipy.ndimage`), 414  
`launch()` (`cupy.cuda.Graph` method), 788  
`launch_host_func()` (`cupy.cuda.ExternalStream` method), 784  
`launch_host_func()` (`cupy.cuda.Stream` method), 781  
`launchHostFunc()` (in module `cupy.cuda.runtime`), 811  
`lcm()` (in module `cupy`), 84  
`ldexp()` (in module `cupy`), 95  
`left_shift()` (in module `cupy`), 89  
`leslie()` (in module `cupyx.scipy.linalg`), 403  
`less()` (in module `cupy`), 90  
`less()` (in module `cupy.array_api`), 902  
`less_equal()` (in module `cupy`), 91  
`less_equal()` (in module `cupy.array_api`), 902  
`lexsort()` (in module `cupy`), 279  
`lfilter()` (in module `cupyx.scipy.signal`), 470

`lfilter_zi()` (in module `cupyx.scipy.signal`), 473  
`lfiltic()` (in module `cupyx.scipy.signal`), 472  
`LinearOperator` (class in `cupyx.scipy.sparse.linalg`), 676  
`LineProfileHook` (class in `cupy.cuda.memory_hooks`), 777  
`linspace()` (in module `cupy`), 107  
`linspace()` (in module `cupy.array_api`), 902  
`load()` (in module `cupy`), 169  
`loadtxt()` (in module `cupy`), 106  
`lobpcg()` (in module `cupyx.scipy.sparse.linalg`), 686  
`local_size_bytes` (`cupy.RawKernel` attribute), 819  
`log()` (in module `cupy`), 82  
`log()` (in module `cupy.array_api`), 902  
`log10()` (in module `cupy`), 82  
`log10()` (in module `cupy.array_api`), 902  
`loglp()` (`cupyx.scipy.sparse.coo_matrix` method), 628  
`loglp()` (`cupyx.scipy.sparse.csc_matrix` method), 638  
`loglp()` (`cupyx.scipy.sparse.csr_matrix` method), 648  
`loglp()` (`cupyx.scipy.sparse.dia_matrix` method), 657  
`loglp()` (in module `cupy`), 82  
`loglp()` (in module `cupy.array_api`), 902  
`loglp()` (in module `cupyx.scipy.special`), 731  
`log2()` (in module `cupy`), 82  
`log2()` (in module `cupy.array_api`), 903  
`log_expit()` (in module `cupyx.scipy.special`), 715  
`log_ndtr()` (in module `cupyx.scipy.special`), 713  
`log_softmax()` (in module `cupyx.scipy.special`), 727  
`logaddexp()` (in module `cupy`), 78  
`logaddexp()` (in module `cupy.array_api`), 903  
`logaddexp2()` (in module `cupy`), 78  
`loggamma()` (in module `cupyx.scipy.special`), 719  
`logical_and()` (in module `cupy`), 91  
`logical_and()` (in module `cupy.array_api`), 903  
`logical_not()` (in module `cupy`), 92  
`logical_not()` (in module `cupy.array_api`), 903  
`logical_or()` (in module `cupy`), 91  
`logical_or()` (in module `cupy.array_api`), 903  
`logical_xor()` (in module `cupy`), 92  
`logical_xor()` (in module `cupy.array_api`), 903  
`logistic()` (`cupy.random.RandomState` method), 244  
`logistic()` (in module `cupy.random`), 258  
`logit()` (in module `cupyx.scipy.special`), 714  
`lognormal()` (`cupy.random.RandomState` method), 244  
`lognormal()` (in module `cupy.random`), 258  
`logseries()` (`cupy.random.Generator` method), 233  
`logseries()` (`cupy.random.RandomState` method), 244  
`logseries()` (in module `cupy.random`), 259  
`logspace()` (in module `cupy`), 108  
`logsumexp()` (in module `cupyx.scipy.special`), 732  
`lombscargle()` (in module `cupyx.scipy.signal`), 581  
`lp2bp()` (in module `cupyx.scipy.signal`), 523  
`lp2bp_zpk()` (in module `cupyx.scipy.signal`), 524  
`lp2bs()` (in module `cupyx.scipy.signal`), 525



lp2bs\_zpk() (in module `cupyx.scipy.signal`), 525  
 lp2hp() (in module `cupyx.scipy.signal`), 526  
 lp2hp\_zpk() (in module `cupyx.scipy.signal`), 527  
 lp2lp() (in module `cupyx.scipy.signal`), 527  
 lp2lp\_zpk() (in module `cupyx.scipy.signal`), 528  
 lpmv() (in module `cupyx.scipy.special`), 725  
 lsim() (in module `cupyx.scipy.signal`), 545  
 lsqr() (in module `cupyx.scipy.sparse.linalg`), 684  
 lsqr() (in module `cupyx.scipy.sparse.linalg`), 683  
 lstsq() (in module `cupy.linalg`), 184  
 lti (class in `cupyx.scipy.signal`), 538  
 lu() (in module `cupyx.scipy.linalg`), 395  
 lu\_factor() (in module `cupyx.scipy.linalg`), 395  
 lu\_solve() (in module `cupyx.scipy.linalg`), 396

## M

make\_2d\_index\_map() (in module `cupyx.distributed.array`), 849  
 make\_interp\_spline() (in module `cupyx.scipy.interpolate`), 379  
 malloc() (`cupy.cuda.CFunctionAllocator` method), 771  
 malloc() (`cupy.cuda.MemoryAsyncPool` method), 767  
 malloc() (`cupy.cuda.MemoryPool` method), 765  
 malloc() (`cupy.cuda.PinnedMemoryPool` method), 769  
 malloc() (`cupy.cuda.PythonFunctionAllocator` method), 770  
 malloc() (in module `cupy.cuda.runtime`), 807  
 malloc3DArray() (in module `cupy.cuda.runtime`), 807  
 malloc\_async() (in module `cupy.cuda`), 762  
 malloc\_managed() (in module `cupy.cuda`), 761  
 malloc\_postprocess() (`cupy.cuda.memory_hooks.DebugPrintHook` method), 776  
 malloc\_postprocess() (`cupy.cuda.memory_hooks.LineProfileHook` method), 778  
 malloc\_postprocess() (`cupy.cuda.MemoryHook` method), 773  
 malloc\_preprocess() (`cupy.cuda.memory_hooks.DebugPrintHook` method), 776  
 malloc\_preprocess() (`cupy.cuda.memory_hooks.LineProfileHook` method), 778  
 malloc\_preprocess() (`cupy.cuda.MemoryHook` method), 773  
 mallocArray() (in module `cupy.cuda.runtime`), 807  
 mallocAsync() (in module `cupy.cuda.runtime`), 807  
 mallocFromPoolAsync() (in module `cupy.cuda.runtime`), 807  
 mallocManaged() (in module `cupy.cuda.runtime`), 807  
 ManagedMemory (class in `cupy.cuda`), 754  
 map\_coordinates() (in module `cupyx.scipy.ndimage`), 426

map\_expr (`cupy.ReductionKernel` attribute), 817  
 Mark() (in module `cupy.cuda.nvtx`), 796  
 MarkC() (in module `cupy.cuda.nvtx`), 796  
 mask\_indices() (in module `cupy`), 155  
 matmat() (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677  
 matmul (in module `cupy`), 77  
 matmul() (in module `cupy.array_api`), 903  
 matmul() (in module `cupyx.distributed.array`), 850  
 matrix\_power() (in module `cupy.linalg`), 177  
 matrix\_rank() (in module `cupy.linalg`), 182  
 matvec() (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677  
 max() (`cupy.ndarray` method), 63  
 max() (`cupyx.distributed.array.DistributedArray` method), 844  
 max() (`cupyx.scipy.sparse.csc_matrix` method), 638  
 max() (`cupyx.scipy.sparse.csr_matrix` method), 648  
 max\_dynamic\_shared\_size\_bytes (`cupy.RawKernel` attribute), 819  
 max\_len\_seq() (in module `cupyx.scipy.signal`), 556  
 max\_threads\_per\_block (`cupy.RawKernel` attribute), 819  
 maximum() (`cupyx.scipy.sparse.coo_matrix` method), 628  
 maximum() (`cupyx.scipy.sparse.csc_matrix` method), 638  
 maximum() (`cupyx.scipy.sparse.csr_matrix` method), 649  
 maximum() (`cupyx.scipy.sparse.dia_matrix` method), 657  
 maximum() (`cupyx.scipy.sparse.spmatrix` method), 663  
 maximum() (in module `cupy`), 92  
 maximum() (in module `cupyx.scipy.ndimage`), 433  
 maximum\_filter() (in module `cupyx.scipy.ndimage`), 415  
 maximum\_filter1d() (in module `cupyx.scipy.ndimage`), 416  
 maximum\_position() (in module `cupyx.scipy.ndimage`), 434  
 may\_share\_memory() (in module `cupy`), 212  
 mean() (`cupy.ndarray` method), 63  
 mean() (`cupyx.distributed.array.DistributedArray` method), 844  
 mean() (`cupyx.scipy.sparse.coo_matrix` method), 628  
 mean() (`cupyx.scipy.sparse.csc_matrix` method), 638  
 mean() (`cupyx.scipy.sparse.csr_matrix` method), 649  
 mean() (`cupyx.scipy.sparse.dia_matrix` method), 657  
 mean() (`cupyx.scipy.sparse.spmatrix` method), 663  
 mean() (in module `cupy`), 293  
 mean() (in module `cupyx.scipy.ndimage`), 435  
 medfilt() (in module `cupyx.scipy.signal`), 468  
 medfilt2d() (in module `cupyx.scipy.signal`), 468  
 median() (in module `cupy`), 292  
 median() (in module `cupyx.scipy.ndimage`), 435  
 median\_filter() (in module `cupyx.scipy.ndimage`), 416  
 mem (`cupy.cuda.MemoryPointer` attribute), 760  
 mem (`cupy.cuda.PinnedMemoryPointer` attribute), 761

- `mem_info` (*cupy.cuda.Device* attribute), 750
- `memAdvise()` (in module *cupy.cuda.runtime*), 810
- `memcpy()` (in module *cupy.cuda.runtime*), 808
- `memcpy2D()` (in module *cupy.cuda.runtime*), 809
- `memcpy2DAsync()` (in module *cupy.cuda.runtime*), 809
- `memcpy2DFromArray()` (in module *cupy.cuda.runtime*), 809
- `memcpy2DFromArrayAsync()` (in module *cupy.cuda.runtime*), 809
- `memcpy2DToArray()` (in module *cupy.cuda.runtime*), 809
- `memcpy2DToArrayAsync()` (in module *cupy.cuda.runtime*), 809
- `memcpy3D()` (in module *cupy.cuda.runtime*), 809
- `memcpy3DAsync()` (in module *cupy.cuda.runtime*), 810
- `memcpy_async` (in module *cupyx.jit.cg*), 833
- `memcpyAsync()` (in module *cupy.cuda.runtime*), 808
- `memcpyPeer()` (in module *cupy.cuda.runtime*), 808
- `memcpyPeerAsync()` (in module *cupy.cuda.runtime*), 809
- `memGetInfo()` (in module *cupy.cuda.runtime*), 808
- `memoize()` (in module *cupy*), 836
- `Memory` (class in *cupy.cuda*), 752
- `MemoryAsync` (class in *cupy.cuda*), 753
- `memoryAsyncHasStat` (*cupy.cuda.MemoryAsyncPool* attribute), 769
- `MemoryAsyncPool` (class in *cupy.cuda*), 766
- `MemoryHook` (class in *cupy.cuda*), 772
- `MemoryPointer` (class in *cupy.cuda*), 757
- `MemoryPool` (class in *cupy.cuda*), 764
- `memPoolCreate()` (in module *cupy.cuda.runtime*), 806
- `memPoolDestroy()` (in module *cupy.cuda.runtime*), 806
- `memPoolTrimTo()` (in module *cupy.cuda.runtime*), 806
- `memPrefetchAsync()` (in module *cupy.cuda.runtime*), 810
- `memset()` (*cupy.cuda.MemoryPointer* method), 759
- `memset()` (in module *cupy.cuda.runtime*), 810
- `memset_async()` (*cupy.cuda.MemoryPointer* method), 759
- `memsetAsync()` (in module *cupy.cuda.runtime*), 810
- `meshgrid()` (in module *cupy*), 109
- `meshgrid()` (in module *cupy.array\_api*), 903
- `mgrid` (in module *cupy*), 109
- `min()` (*cupy.ndarray* method), 63
- `min()` (*cupyx.distributed.array.DistributedArray* method), 844
- `min()` (*cupyx.scipy.sparse.csc\_matrix* method), 638
- `min()` (*cupyx.scipy.sparse.csr\_matrix* method), 649
- `min_scalar_type()` (in module *cupy*), 136
- `minimum()` (*cupyx.scipy.sparse.coo\_matrix* method), 629
- `minimum()` (*cupyx.scipy.sparse.csc\_matrix* method), 639
- `minimum()` (*cupyx.scipy.sparse.csr\_matrix* method), 649
- `minimum()` (*cupyx.scipy.sparse.dia\_matrix* method), 657
- `minimum()` (*cupyx.scipy.sparse.spmatrix* method), 663
- `minimum()` (in module *cupy*), 92
- `minimum()` (in module *cupyx.scipy.ndimage*), 436
- `minimum_filter()` (in module *cupyx.scipy.ndimage*), 417
- `minimum_filter1d()` (in module *cupyx.scipy.ndimage*), 418
- `minimum_phase()` (in module *cupyx.scipy.signal*), 495
- `minimum_position()` (in module *cupyx.scipy.ndimage*), 436
- `minkowski()` (in module *cupyx.scipy.spatial.distance*), 695
- `minres()` (in module *cupyx.scipy.sparse.linalg*), 683
- `mod()` (in module *cupy*), 79
- `mode` (*cupyx.distributed.array.DistributedArray* attribute), 847
- `modf()` (in module *cupy*), 95
- `module`
  - `cupy`, 1
  - `cupy.array_api`, 896
  - `cupy.fft`, 138
  - `cupy.linalg`, 173
  - `cupy.polynomial.polynomial`, 217
  - `cupy.polynomial.polyutils`, 220
  - `cupy.random`, 227
  - `cupy.testing`, 301
  - `cupyx.distributed`, 836
  - `cupyx.distributed.array`, 841
  - `cupyx.optimizing`, 747
  - `cupyx.scipy`, 318
  - `cupyx.scipy.fft`, 318
  - `cupyx.scipy.fftpack`, 340
  - `cupyx.scipy.interpolate`, 347
  - `cupyx.scipy.linalg`, 392
  - `cupyx.scipy.ndimage`, 405
  - `cupyx.scipy.signal`, 455
  - `cupyx.scipy.signal.windows`, 596
  - `cupyx.scipy.sparse`, 624
  - `cupyx.scipy.sparse.csgraph`, 690
  - `cupyx.scipy.sparse.linalg`, 676
  - `cupyx.scipy.spatial`, 691
  - `cupyx.scipy.spatial.distance`, 692
  - `cupyx.scipy.special`, 700
  - `cupyx.scipy.stats`, 733
- `module` (*cupy.RawModule* attribute), 823
- `morlet()` (in module *cupyx.scipy.signal*), 561
- `morlet2()` (in module *cupyx.scipy.signal*), 563
- `morphological_gradient()` (in module *cupyx.scipy.ndimage*), 452
- `morphological_laplace()` (in module *cupyx.scipy.ndimage*), 453
- `moveaxis()` (in module *cupy*), 115
- `MRG32k3a` (class in *cupy.random*), 239
- `msort()` (in module *cupy*), 280
- `mT` (*cupy.array\_api.\_array\_object.Array* attribute), 910

- [multigammaln\(\)](#) (in module `cupyx.scipy.special`), 723  
[multinomial\(\)](#) (in module `cupy.random`), 259  
[multiply\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 629  
[multiply\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 639  
[multiply\(\)](#) (`cupyx.scipy.sparse.csr_matrix` method), 649  
[multiply\(\)](#) (`cupyx.scipy.sparse.dia_matrix` method), 657  
[multiply\(\)](#) (`cupyx.scipy.sparse.spmatrix` method), 663  
[multiply\(\)](#) (in module `cupy`), 77  
[multiply\(\)](#) (in module `cupy.array_api`), 904  
[multivariate\\_normal\(\)](#) (`cupy.random.RandomState` method), 244  
[multivariate\\_normal\(\)](#) (in module `cupy.random`), 260
- ## N
- [n\\_free\\_blocks\(\)](#) (`cupy.cuda.MemoryAsyncPool` method), 768  
[n\\_free\\_blocks\(\)](#) (`cupy.cuda.MemoryPool` method), 765  
[n\\_free\\_blocks\(\)](#) (`cupy.cuda.PinnedMemoryPool` method), 769  
[name](#) (`cupy.cuda.memory_hooks.DebugPrintHook` attribute), 776  
[name](#) (`cupy.cuda.memory_hooks.LineProfileHook` attribute), 779  
[name](#) (`cupy.cuda.MemoryHook` attribute), 774  
[name](#) (`cupy.ElementwiseKernel` attribute), 814  
[name](#) (`cupy.RawKernel` attribute), 819  
[name](#) (`cupy.ReductionKernel` attribute), 817  
[name](#) (`cupy.ufunc` attribute), 75  
[name\\_expressions](#) (`cupy.RawModule` attribute), 823  
[nan\\_to\\_num\(\)](#) (in module `cupy`), 210  
[nanargmax\(\)](#) (in module `cupy`), 283  
[nanargmin\(\)](#) (in module `cupy`), 284  
[nancumprod\(\)](#) (in module `cupy`), 202  
[nancumsum\(\)](#) (in module `cupy`), 203  
[nanmax\(\)](#) (in module `cupy`), 289  
[nanmean\(\)](#) (in module `cupy`), 295  
[nanmedian\(\)](#) (in module `cupy`), 294  
[nanmin\(\)](#) (in module `cupy`), 288  
[nanprod\(\)](#) (in module `cupy`), 200  
[nanstd\(\)](#) (in module `cupy`), 295  
[nansum\(\)](#) (in module `cupy`), 201  
[nanvar\(\)](#) (in module `cupy`), 296  
[nargs](#) (`cupy.ElementwiseKernel` attribute), 815  
[nargs](#) (`cupy.ReductionKernel` attribute), 817  
[nargs](#) (`cupy.ufunc` attribute), 75  
[nbdtr\(\)](#) (in module `cupyx.scipy.special`), 710  
[nbdtrc\(\)](#) (in module `cupyx.scipy.special`), 710  
[nbdtri\(\)](#) (in module `cupyx.scipy.special`), 710  
[nbytes](#) (`cupy.ndarray` attribute), 69  
[nbytes](#) (`cupyx.distributed.array.DistributedArray` attribute), 848  
[NCCLBackend](#) (class in `cupyx.distributed`), 837  
[NcclCommunicator](#) (class in `cupy.cuda.nccl`), 798  
[nd](#) (`cupy.broadcast` attribute), 119  
[ndarray](#) (class in `cupy`), 57  
[ndim](#) (`cupy.array_api._array_object.Array` attribute), 910  
[ndim](#) (`cupy.cuda.texture.CUDAArray` attribute), 791  
[ndim](#) (`cupy.ndarray` attribute), 69  
[ndim](#) (`cupyx.distributed.array.DistributedArray` attribute), 848  
[ndim](#) (`cupyx.scipy.sparse.coo_matrix` attribute), 633  
[ndim](#) (`cupyx.scipy.sparse.csc_matrix` attribute), 643  
[ndim](#) (`cupyx.scipy.sparse.csr_matrix` attribute), 654  
[ndim](#) (`cupyx.scipy.sparse.dia_matrix` attribute), 660  
[ndim](#) (`cupyx.scipy.sparse.linalg.LinearOperator` attribute), 678  
[ndim](#) (`cupyx.scipy.sparse.spmatrix` attribute), 665  
[NdPPoly](#) (class in `cupyx.scipy.interpolate`), 389  
[ndtr\(\)](#) (in module `cupyx.scipy.special`), 713  
[ndtri\(\)](#) (in module `cupyx.scipy.special`), 714  
[negative\(\)](#) (in module `cupy`), 78  
[negative\(\)](#) (in module `cupy.array_api`), 904  
[negative\\_binomial\(\)](#) (`cupy.random.RandomState` method), 245  
[negative\\_binomial\(\)](#) (in module `cupy.random`), 261  
[next\\_fast\\_len\(\)](#) (in module `cupyx.scipy.fft`), 340  
[nextafter\(\)](#) (in module `cupy`), 94  
[nin](#) (`cupy.ElementwiseKernel` attribute), 815  
[nin](#) (`cupy.ReductionKernel` attribute), 817  
[nin](#) (`cupy.ufunc` attribute), 75  
[nnz](#) (`cupyx.scipy.sparse.coo_matrix` attribute), 633  
[nnz](#) (`cupyx.scipy.sparse.csc_matrix` attribute), 643  
[nnz](#) (`cupyx.scipy.sparse.csr_matrix` attribute), 654  
[nnz](#) (`cupyx.scipy.sparse.dia_matrix` attribute), 660  
[nnz](#) (`cupyx.scipy.sparse.spmatrix` attribute), 665  
[no\\_return](#) (`cupy.ElementwiseKernel` attribute), 815  
[noncentral\\_chisquare\(\)](#) (`cupy.random.RandomState` method), 245  
[noncentral\\_chisquare\(\)](#) (in module `cupy.random`), 261  
[noncentral\\_f\(\)](#) (`cupy.random.RandomState` method), 245  
[noncentral\\_f\(\)](#) (in module `cupy.random`), 262  
[nonzero\(\)](#) (`cupy.ndarray` method), 63  
[nonzero\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 844  
[nonzero\(\)](#) (in module `cupy`), 153  
[nonzero\(\)](#) (in module `cupy.array_api`), 904  
[norm\(\)](#) (in module `cupy.linalg`), 181  
[norm\(\)](#) (in module `cupyx.scipy.sparse.linalg`), 678  
[normal\(\)](#) (`cupy.random.RandomState` method), 245

[normal\(\)](#) (in module `cupy.random`), 262  
[normalize\(\)](#) (in module `cupyx.scipy.signal`), 528  
[not\\_equal\(\)](#) (in module `cupy`), 91  
[not\\_equal\(\)](#) (in module `cupy.array_api`), 904  
[nout](#) (`cupy.ElementwiseKernel` attribute), 815  
[nout](#) (`cupy.ReductionKernel` attribute), 817  
[nout](#) (`cupy.ufunc` attribute), 75  
[null](#) (`cupy.cuda.Stream` attribute), 782  
[num](#) (`cupyx.scipy.signal.TransferFunction` attribute), 543  
[num\\_regs](#) (`cupy.RawKernel` attribute), 819  
[numpy\\_cupy\\_allclose\(\)](#) (in module `cupy.testing`), 305  
[numpy\\_cupy\\_array\\_almost\\_equal\(\)](#) (in module `cupy.testing`), 306  
[numpy\\_cupy\\_array\\_almost\\_equal\\_nulp\(\)](#) (in module `cupy.testing`), 307  
[numpy\\_cupy\\_array\\_equal\(\)](#) (in module `cupy.testing`), 308  
[numpy\\_cupy\\_array\\_less\(\)](#) (in module `cupy.testing`), 309  
[numpy\\_cupy\\_array\\_list\\_equal\(\)](#) (in module `cupy.testing`), 309  
[numpy\\_cupy\\_array\\_max\\_ulp\(\)](#) (in module `cupy.testing`), 308  
[nuttall\(\)](#) (in module `cupyx.scipy.signal.windows`), 619  
[NVCC](#), 853

## O

[o](#) (`cupy.poly1d` attribute), 223  
[oconvolve\(\)](#) (in module `cupyx.scipy.signal`), 458  
[ogrid](#) (in module `cupy`), 110  
[ones\(\)](#) (in module `cupy`), 101  
[ones\(\)](#) (in module `cupy.array_api`), 904  
[ones\\_like\(\)](#) (in module `cupy`), 101  
[ones\\_like\(\)](#) (in module `cupy.array_api`), 904  
[operation](#) (`cupy.ElementwiseKernel` attribute), 815  
[optimize\(\)](#) (in module `cupyx.optimizing`), 748  
[options](#) (`cupy.RawKernel` attribute), 819  
[options](#) (`cupy.RawModule` attribute), 823  
[options](#) (`cupy.ReductionKernel` attribute), 817  
[order](#) (`cupy.poly1d` attribute), 223  
[order\\_filter\(\)](#) (in module `cupyx.scipy.signal`), 467  
[out\\_params](#) (`cupy.ElementwiseKernel` attribute), 815  
[out\\_params](#) (`cupy.ReductionKernel` attribute), 817  
[outer\(\)](#) (`cupy.ufunc` method), 74  
[outer\(\)](#) (in module `cupy`), 175  
[output\(\)](#) (`cupyx.scipy.signal.dlti` method), 549  
[output\(\)](#) (`cupyx.scipy.signal.lti` method), 538

## P

[packbits\(\)](#) (in module `cupy`), 135  
[pad\(\)](#) (in module `cupy`), 214  
[params](#) (`cupy.ElementwiseKernel` attribute), 815  
[params](#) (`cupy.ReductionKernel` attribute), 817  
[pareto\(\)](#) (`cupy.random.RandomState` method), 246

[pareto\(\)](#) (in module `cupy.random`), 263  
[partition\(\)](#) (`cupy.ndarray` method), 63  
[partition\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 844  
[partition\(\)](#) (in module `cupy`), 280  
[parzen\(\)](#) (in module `cupyx.scipy.signal.windows`), 620  
[pchip\\_interpolate\(\)](#) (in module `cupyx.scipy.interpolate`), 352  
[PchipInterpolator](#) (class in `cupyx.scipy.interpolate`), 357  
[pci\\_bus\\_id](#) (`cupy.cuda.Device` attribute), 751  
[pdist\(\)](#) (in module `cupyx.scipy.spatial.distance`), 693  
[pdist\(\)](#) (in module `cupyx.scipy.special`), 711  
[pdtrc\(\)](#) (in module `cupyx.scipy.special`), 711  
[pdtri\(\)](#) (in module `cupyx.scipy.special`), 712  
[peak\\_prominences\(\)](#) (in module `cupyx.scipy.signal`), 570  
[peak\\_widths\(\)](#) (in module `cupyx.scipy.signal`), 571  
[percentile\(\)](#) (in module `cupy`), 290  
[percentile\\_filter\(\)](#) (in module `cupyx.scipy.ndimage`), 418  
[periodogram\(\)](#) (in module `cupyx.scipy.signal`), 573  
[permutation\(\)](#) (`cupy.random.RandomState` method), 246  
[permutation\(\)](#) (in module `cupy.random`), 263  
[permute\\_dims\(\)](#) (in module `cupy.array_api`), 904  
[Philox4x3210](#) (class in `cupy.random`), 240  
[piecewise\(\)](#) (in module `cupy`), 152  
[PinnedMemory](#) (class in `cupy.cuda`), 756  
[PinnedMemoryPointer](#) (class in `cupy.cuda`), 760  
[PinnedMemoryPool](#) (class in `cupy.cuda`), 769  
[pinv\(\)](#) (in module `cupy.linalg`), 186  
[place\(\)](#) (in module `cupy`), 165  
[place\\_poles\(\)](#) (in module `cupyx.scipy.signal`), 535  
[poch\(\)](#) (in module `cupyx.scipy.special`), 723  
[pointerGetAttributes\(\)](#) (in module `cupy.cuda.runtime`), 810  
[points\(\)](#) (`cupyx.scipy.signal.CZT` method), 593  
[points\(\)](#) (`cupyx.scipy.signal.ZoomFFT` method), 595  
[poisson\(\)](#) (`cupy.random.Generator` method), 233  
[poisson\(\)](#) (`cupy.random.RandomState` method), 246  
[poisson\(\)](#) (in module `cupy.random`), 263  
[poles](#) (`cupyx.scipy.signal.dlti` attribute), 550  
[poles](#) (`cupyx.scipy.signal.lti` attribute), 539  
[poles](#) (`cupyx.scipy.signal.StateSpace` attribute), 541  
[poles](#) (`cupyx.scipy.signal.TransferFunction` attribute), 543  
[poles](#) (`cupyx.scipy.signal.ZerosPolesGain` attribute), 544  
[poly\(\)](#) (in module `cupy`), 223  
[poly1d](#) (class in `cupy`), 221  
[polyadd\(\)](#) (in module `cupy`), 226  
[polycompanion\(\)](#) (in module `cupy.polynomial.polynomial`), 218  
[polyfit\(\)](#) (in module `cupy`), 225



- polygamma() (in module `cupyx.scipy.special`), 722  
 polymul() (in module `cupy`), 227  
 polysub() (in module `cupy`), 226  
 polyval() (in module `cupy`), 224  
 polyval() (in module `cupy.polynomial.polynomial`), 218  
 polyvalfromroots() (in module `cupy.polynomial.polynomial`), 219  
 polyvander() (in module `cupy.polynomial.polynomial`), 217  
 positive() (in module `cupy`), 79  
 positive() (in module `cupy.array_api`), 904  
 post\_map\_expr (cupy.ReductionKernel attribute), 817  
 pow() (in module `cupy.array_api`), 905  
 power() (cupy.random.Generator method), 233  
 power() (cupy.random.RandomState method), 246  
 power() (cupyx.scipy.sparse.coo\_matrix method), 629  
 power() (cupyx.scipy.sparse.csc\_matrix method), 639  
 power() (cupyx.scipy.sparse.csr\_matrix method), 649  
 power() (cupyx.scipy.sparse.dia\_matrix method), 657  
 power() (cupyx.scipy.sparse.spmatrix method), 663  
 power() (in module `cupy`), 79  
 power() (in module `cupy.random`), 264  
 PPoly (class in `cupyx.scipy.interpolate`), 366  
 preamble (cupy.ElementwiseKernel attribute), 815  
 preamble (cupy.ReductionKernel attribute), 817  
 preferred\_shared\_memory\_carveout (cupy.RawKernel attribute), 819  
 prefetch() (cupy.cuda.ManagedMemory method), 754  
 prewitt() (in module `cupyx.scipy.ndimage`), 419  
 print\_report() (cupy.cuda.memory\_hooks.LineProfileHook method), 778  
 prod() (cupy.ndarray method), 64  
 prod() (cupyx.distributed.array.DistributedArray method), 844  
 prod() (in module `cupy`), 199  
 profile() (in module `cupyx.profiler`), 746  
 profilerStart() (in module `cupy.cuda.runtime`), 812  
 profilerStop() (in module `cupy.cuda.runtime`), 812  
 pseudo\_huber() (in module `cupyx.scipy.special`), 717  
 psi() (in module `cupyx.scipy.special`), 722  
 ptids (cupy.cuda.Stream attribute), 782  
 ptp() (cupy.ndarray method), 64  
 ptp() (cupyx.distributed.array.DistributedArray method), 844  
 ptp() (in module `cupy`), 289  
 ptr (cupy.cuda.ManagedMemory attribute), 755  
 ptr (cupy.cuda.Memory attribute), 753  
 ptr (cupy.cuda.MemoryAsync attribute), 754  
 ptr (cupy.cuda.MemoryPointer attribute), 760  
 ptr (cupy.cuda.PinnedMemoryPointer attribute), 761  
 ptr (cupy.cuda.texture.ChannelFormatDescriptor attribute), 790  
 ptr (cupy.cuda.texture.CUDAarray attribute), 791  
 ptr (cupy.cuda.texture.ResourceDescriptor attribute), 793  
 ptr (cupy.cuda.texture.SurfaceObject attribute), 796  
 ptr (cupy.cuda.texture.TextureDescriptor attribute), 794  
 ptr (cupy.cuda.texture.TextureObject attribute), 795  
 ptr (cupy.cuda.UnownedMemory attribute), 756  
 ptx\_version (cupy.RawKernel attribute), 820  
 pulse\_compression() (in module `cupyx.signal`), 743  
 pulse\_doppler() (in module `cupyx.signal`), 743  
 put() (cupy.ndarray method), 64  
 put() (cupyx.distributed.array.DistributedArray method), 844  
 put() (in module `cupy`), 166  
 putmask() (in module `cupy`), 166  
 PythonFunctionAllocator (class in `cupy.cuda`), 770
- ## Q
- qmf() (in module `cupyx.scipy.signal`), 562  
 qr() (in module `cupy.linalg`), 178  
 qspline1d() (in module `cupyx.scipy.signal`), 464  
 qspline1d\_eval() (in module `cupyx.scipy.signal`), 466  
 qspline2d() (in module `cupyx.scipy.signal`), 465  
 quantile() (in module `cupy`), 291
- ## R
- r (cupy.poly1d attribute), 223  
 r\_ (in module `cupy`), 153  
 rad2deg() (cupyx.scipy.sparse.coo\_matrix method), 629  
 rad2deg() (cupyx.scipy.sparse.csc\_matrix method), 639  
 rad2deg() (cupyx.scipy.sparse.csr\_matrix method), 650  
 rad2deg() (cupyx.scipy.sparse.dia\_matrix method), 657  
 rad2deg() (in module `cupy`), 88  
 radian() (in module `cupyx.scipy.special`), 730  
 radians() (in module `cupy`), 87  
 rand() (cupy.random.RandomState method), 246  
 rand() (in module `cupy.random`), 264  
 rand() (in module `cupyx.scipy.sparse`), 672  
 randint() (cupy.random.RandomState method), 246  
 randint() (in module `cupy.random`), 265  
 randn() (cupy.random.RandomState method), 247  
 randn() (in module `cupy.random`), 265  
 random() (cupy.random.Generator method), 234  
 random() (in module `cupy.random`), 266  
 random() (in module `cupyx.scipy.sparse`), 673  
 random\_integers() (in module `cupy.random`), 267  
 random\_raw() (cupy.random.BitGenerator method), 237  
 random\_raw() (cupy.random.MRG32k3a method), 239  
 random\_raw() (cupy.random.Philox4x3210 method), 240  
 random\_raw() (cupy.random.XORWOW method), 238  
 random\_sample() (cupy.random.RandomState method), 247  
 random\_sample() (in module `cupy.random`), 267  
 RandomState (class in `cupy.random`), 242

`ranf()` (in module `cupy.random`), 267  
`range` (in module `cupyx.jit`), 827  
`RangePop()` (in module `cupy.cuda.nvtx`), 797  
`RangePush()` (in module `cupy.cuda.nvtx`), 796  
`RangePushC()` (in module `cupy.cuda.nvtx`), 797  
`rank_filter()` (in module `cupyx.scipy.ndimage`), 419  
`rank_id()` (`cupy.cuda.nccl.NcclCommunicator` method), 799  
`ravel()` (`cupy.ndarray` method), 64  
`ravel()` (`cupyx.distributed.array.DistributedArray` method), 845  
`ravel()` (in module `cupy`), 114  
`ravel_multi_index()` (in module `cupy`), 158  
`RawKernel` (class in `cupy`), 817  
`rawkernel()` (in module `cupyx.jit`), 825  
`RawModule` (class in `cupy`), 820  
`rayleigh()` (`cupy.random.RandomState` method), 247  
`rayleigh()` (in module `cupy.random`), 268  
`RBFInterpolator` (class in `cupyx.scipy.interpolate`), 382  
`real` (`cupy.ndarray` attribute), 69  
`real` (`cupyx.distributed.array.DistributedArray` attribute), 848  
`real()` (in module `cupy`), 208  
`real_if_close()` (in module `cupy`), 210  
`reciprocal()` (in module `cupy`), 83  
`record()` (`cupy.cuda.Event` method), 786  
`record()` (`cupy.cuda.ExternalStream` method), 785  
`record()` (`cupy.cuda.Stream` method), 781  
`recv()` (`cupy.cuda.nccl.NcclCommunicator` method), 799  
`recv()` (`cupyx.distributed.NCCLBackend` method), 839  
`reduce()` (`cupy.cuda.nccl.NcclCommunicator` method), 799  
`reduce()` (`cupy.ufunc` method), 74  
`reduce()` (`cupyx.distributed.NCCLBackend` method), 839  
`reduce_dims` (`cupy.ElementwiseKernel` attribute), 815  
`reduce_dims` (`cupy.ReductionKernel` attribute), 817  
`reduce_expr` (`cupy.ReductionKernel` attribute), 817  
`reduce_scatter()` (`cupyx.distributed.NCCLBackend` method), 839  
`reduce_type` (`cupy.ReductionKernel` attribute), 817  
`reduceat()` (`cupy.ufunc` method), 74  
`reduced_view()` (`cupy.ndarray` method), 64  
`reduced_view()` (`cupyx.distributed.array.DistributedArray` method), 845  
`reduceScatter()` (`cupy.cuda.nccl.NcclCommunicator` method), 799  
`ReductionKernel` (class in `cupy`), 815  
`RegularGridInterpolator` (class in `cupyx.scipy.interpolate`), 385  
`rel ENTR()` (in module `cupyx.scipy.special`), 717  
`remainder()` (in module `cupy`), 79  
`remainder()` (in module `cupy.array_api`), 905  
`repeat()` (`cupy.ndarray` method), 64  
`repeat()` (`cupyx.distributed.array.DistributedArray` method), 845  
`repeat()` (in module `cupy`), 128  
`require()` (in module `cupy`), 122  
`resample()` (in module `cupyx.scipy.signal`), 481  
`resample_poly()` (in module `cupyx.scipy.signal`), 482  
`ResDesc` (`cupy.cuda.texture.SurfaceObject` attribute), 796  
`ResDesc` (`cupy.cuda.texture.TextureObject` attribute), 795  
`reshape()` (`cupy.ndarray` method), 64  
`reshape()` (`cupyx.distributed.array.DistributedArray` method), 845  
`reshape()` (`cupyx.scipy.sparse.coo_matrix` method), 629  
`reshape()` (`cupyx.scipy.sparse.csc_matrix` method), 639  
`reshape()` (`cupyx.scipy.sparse.csr_matrix` method), 650  
`reshape()` (`cupyx.scipy.sparse.dia_matrix` method), 658  
`reshape()` (`cupyx.scipy.sparse.spmatrix` method), 663  
`reshape()` (in module `cupy`), 114  
`reshape()` (in module `cupy.array_api`), 905  
`reshard()` (`cupyx.distributed.array.DistributedArray` method), 845  
`residue()` (in module `cupyx.scipy.signal`), 504  
`residuez()` (in module `cupyx.scipy.signal`), 506  
`resize()` (in module `cupy`), 130  
`ResourceDescriptor` (class in `cupy.cuda.texture`), 792  
`result_type()` (in module `cupy`), 137  
`result_type()` (in module `cupy.array_api`), 905  
`return_tuple` (`cupy.ElementwiseKernel` attribute), 815  
`rfft()` (in module `cupy.fft`), 141  
`rfft()` (in module `cupyx.scipy.fft`), 323  
`rfft()` (in module `cupyx.scipy.fftpack`), 345  
`rfft2()` (in module `cupy.fft`), 142  
`rfft2()` (in module `cupyx.scipy.fft`), 324  
`rfftfreq()` (in module `cupy.fft`), 146  
`rfftfreq()` (in module `cupyx.scipy.fft`), 339  
`rfftn()` (in module `cupy.fft`), 143  
`rfftn()` (in module `cupyx.scipy.fft`), 326  
`rgamma()` (in module `cupyx.scipy.special`), 722  
`ricker()` (in module `cupyx.scipy.signal`), 562  
`right_shift()` (in module `cupy`), 89  
`rint()` (`cupyx.scipy.sparse.coo_matrix` method), 629  
`rint()` (`cupyx.scipy.sparse.csc_matrix` method), 639  
`rint()` (`cupyx.scipy.sparse.csr_matrix` method), 650  
`rint()` (`cupyx.scipy.sparse.dia_matrix` method), 658  
`rint()` (in module `cupy`), 80  
`rmatmat()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677  
`rmatvec()` (`cupyx.scipy.sparse.linalg.LinearOperator` method), 677  
`roll()` (in module `cupy`), 133  
`roll()` (in module `cupy.array_api`), 905  
`rollaxis()` (in module `cupy`), 116

[roots \(cupy.poly1d attribute\)](#), 223  
[roots\(\) \(cupyx.scipy.interpolate.Akima1DInterpolator method\)](#), 365  
[roots\(\) \(cupyx.scipy.interpolate.CubicHermiteSpline method\)](#), 356  
[roots\(\) \(cupyx.scipy.interpolate.PchipInterpolator method\)](#), 360  
[roots\(\) \(cupyx.scipy.interpolate.PPoly method\)](#), 369  
[roots\(\) \(in module cupy\)](#), 224  
[rot90\(\) \(in module cupy\)](#), 134  
[rotate\(\) \(in module cupyx.scipy.ndimage\)](#), 426  
[round\(\) \(cupy.ndarray method\)](#), 64  
[round\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[round\(\) \(in module cupy.array\\_api\)](#), 905  
[round\(\) \(in module cupyx.scipy.special\)](#), 731  
[round\\_\(\) \(in module cupy\)](#), 198  
[row\\_stack\(\) \(in module cupy\)](#), 125  
[rsqrt\(\) \(in module cupyx\)](#), 737  
[runtimeGetVersion\(\) \(in module cupy.cuda.runtime\)](#), 805  
[russellrao\(\) \(in module cupyx.scipy.spatial.distance\)](#), 699

## S

[sample\(\) \(in module cupy.random\)](#), 268  
[save\(\) \(in module cupy\)](#), 169  
[savetxt\(\) \(in module cupy\)](#), 171  
[savez\(\) \(in module cupy\)](#), 170  
[savez\\_compressed\(\) \(in module cupy\)](#), 170  
[savgol\\_coeffs\(\) \(in module cupyx.scipy.signal\)](#), 496  
[savgol\\_filter\(\) \(in module cupyx.scipy.signal\)](#), 474  
[sawtooth\(\) \(in module cupyx.scipy.signal\)](#), 557  
[scatter\(\) \(cupyx.distributed.NCCLBackend method\)](#), 839  
[scatter\\_add\(\) \(cupy.ndarray method\)](#), 65  
[scatter\\_add\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[scatter\\_add\(\) \(in module cupyx\)](#), 737  
[scatter\\_max\(\) \(cupy.ndarray method\)](#), 65  
[scatter\\_max\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[scatter\\_max\(\) \(in module cupyx\)](#), 738  
[scatter\\_min\(\) \(cupy.ndarray method\)](#), 65  
[scatter\\_min\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[scatter\\_min\(\) \(in module cupyx\)](#), 738  
[searchsorted\(\) \(cupy.ndarray method\)](#), 65  
[searchsorted\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[searchsorted\(\) \(in module cupy\)](#), 285  
[seed\(\) \(cupy.random.RandomState method\)](#), 247  
[seed\(\) \(in module cupy.random\)](#), 269  
[select\(\) \(in module cupy\)](#), 164  
[send\(\) \(cupy.cuda.nccl.NcclCommunicator method\)](#), 799  
[send\(\) \(cupyx.distributed.NCCLBackend method\)](#), 840  
[send\\_recv\(\) \(cupyx.distributed.NCCLBackend method\)](#), 840  
[sepfir2d\(\) \(in module cupyx.scipy.signal\)](#), 461  
[set\(\) \(cupy.ndarray method\)](#), 65  
[set\(\) \(cupy.poly1d method\)](#), 222  
[set\(\) \(cupyx.distributed.array.DistributedArray method\)](#), 845  
[set\\_allocator\(\) \(in module cupy.cuda\)](#), 763  
[set\\_cufft\\_callbacks \(class in cupy.fft.config\)](#), 147  
[set\\_cufft\\_gpus\(\) \(in module cupy.fft.config\)](#), 149  
[set\\_limit\(\) \(cupy.cuda.MemoryAsyncPool method\)](#), 768  
[set\\_limit\(\) \(cupy.cuda.MemoryPool method\)](#), 765  
[set\\_pinned\\_memory\\_allocator\(\) \(in module cupy.cuda\)](#), 763  
[set\\_random\\_state\(\) \(in module cupy.random\)](#), 275  
[set\\_shape\(\) \(cupyx.scipy.sparse.coo\\_matrix method\)](#), 629  
[set\\_shape\(\) \(cupyx.scipy.sparse.csc\\_matrix method\)](#), 639  
[set\\_shape\(\) \(cupyx.scipy.sparse.csr\\_matrix method\)](#), 650  
[set\\_shape\(\) \(cupyx.scipy.sparse.dia\\_matrix method\)](#), 658  
[set\\_shape\(\) \(cupyx.scipy.sparse.spmatrix method\)](#), 664  
[set\\_yi\(\) \(cupyx.scipy.interpolate.BarycentricInterpolator method\)](#), 349  
[setDevice\(\) \(in module cupy.cuda.runtime\)](#), 806  
[setdiag\(\) \(cupyx.scipy.sparse.coo\\_matrix method\)](#), 629  
[setdiag\(\) \(cupyx.scipy.sparse.csc\\_matrix method\)](#), 639  
[setdiag\(\) \(cupyx.scipy.sparse.csr\\_matrix method\)](#), 650  
[setdiag\(\) \(cupyx.scipy.sparse.dia\\_matrix method\)](#), 658  
[setdiag\(\) \(cupyx.scipy.sparse.spmatrix method\)](#), 664  
[setdiff1d\(\) \(in module cupy\)](#), 277  
[setxor1d\(\) \(in module cupy\)](#), 278  
[shape \(cupy.array\\_api.\\_array\\_object.Array attribute\)](#), 910  
[shape \(cupy.broadcast attribute\)](#), 119  
[shape \(cupy.ndarray attribute\)](#), 69  
[shape \(cupyx.distributed.array.DistributedArray attribute\)](#), 849  
[shape \(cupyx.scipy.sparse.coo\\_matrix attribute\)](#), 633  
[shape \(cupyx.scipy.sparse.csc\\_matrix attribute\)](#), 644  
[shape \(cupyx.scipy.sparse.csr\\_matrix attribute\)](#), 654  
[shape \(cupyx.scipy.sparse.dia\\_matrix attribute\)](#), 660  
[shape \(cupyx.scipy.sparse.spmatrix attribute\)](#), 665  
[shape\(\) \(in module cupy\)](#), 113  
[shared\\_memory \(in module cupyx.jit\)](#), 828  
[shared\\_size\\_bytes \(cupy.RawKernel attribute\)](#), 820  
[shares\\_memory\(\) \(in module cupy\)](#), 212  
[shfl\\_down\\_sync \(in module cupyx.jit\)](#), 828



- `shfl_sync` (in module `cupyx.jit`), 828
- `shfl_up_sync` (in module `cupyx.jit`), 828
- `shfl_xor_sync` (in module `cupyx.jit`), 828
- `shift()` (in module `cupyx.scipy.ndimage`), 427
- `show_config()` (in module `cupy`), 212
- `show_plan_cache_info()` (in module `cupy.fft.config`), 150
- `shuffle()` (`cupy.random.RandomState` method), 247
- `shuffle()` (in module `cupy.random`), 269
- `sign()` (`cupyx.scipy.sparse.coo_matrix` method), 629
- `sign()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sign()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sign()` (`cupyx.scipy.sparse.dia_matrix` method), 658
- `sign()` (in module `cupy`), 80
- `sign()` (in module `cupy.array_api`), 905
- `signbit()` (in module `cupy`), 94
- `sin()` (`cupyx.scipy.sparse.coo_matrix` method), 629
- `sin()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sin()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sin()` (`cupyx.scipy.sparse.dia_matrix` method), 658
- `sin()` (in module `cupy`), 84
- `sin()` (in module `cupy.array_api`), 905
- `sinc()` (in module `cupy`), 206
- `sinc()` (in module `cupyx.scipy.special`), 733
- `sindg()` (in module `cupyx.scipy.special`), 730
- `sinh()` (`cupyx.scipy.sparse.coo_matrix` method), 630
- `sinh()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sinh()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sinh()` (`cupyx.scipy.sparse.dia_matrix` method), 658
- `sinh()` (in module `cupy`), 86
- `sinh()` (in module `cupy.array_api`), 906
- `size` (`cupy.array_api._array_object.Array` attribute), 910
- `size` (`cupy.broadcast` attribute), 119
- `size` (`cupy.cuda.ManagedMemory` attribute), 755
- `size` (`cupy.cuda.Memory` attribute), 753
- `size` (`cupy.cuda.MemoryAsync` attribute), 754
- `size` (`cupy.cuda.UnownedMemory` attribute), 756
- `size` (`cupy.ndarray` attribute), 69
- `size` (`cupyx.distributed.array.DistributedArray` attribute), 849
- `size` (`cupyx.scipy.sparse.coo_matrix` attribute), 633
- `size` (`cupyx.scipy.sparse.csc_matrix` attribute), 644
- `size` (`cupyx.scipy.sparse.csr_matrix` attribute), 654
- `size` (`cupyx.scipy.sparse.dia_matrix` attribute), 660
- `size` (`cupyx.scipy.sparse.spmatrix` attribute), 665
- `size()` (`cupy.cuda.nccl.NcclCommunicator` method), 799
- `size()` (`cupy.cuda.PinnedMemoryPointer` method), 761
- `slogdet()` (in module `cupy.linalg`), 182
- `sobel()` (in module `cupyx.scipy.ndimage`), 420
- `softmax()` (in module `cupyx.scipy.special`), 727
- `solve()` (`cupyx.scipy.interpolate.Akima1DInterpolator` method), 365
- `solve()` (`cupyx.scipy.interpolate.CubicHermiteSpline` method), 356
- `solve()` (`cupyx.scipy.interpolate.PchipInterpolator` method), 361
- `solve()` (`cupyx.scipy.interpolate.PPoly` method), 370
- `solve()` (`cupyx.scipy.sparse.linalg.SuperLU` method), 690
- `solve()` (in module `cupy.linalg`), 183
- `solve_triangular()` (in module `cupyx.scipy.linalg`), 393
- `sort()` (`cupy.ndarray` method), 65
- `sort()` (`cupyx.distributed.array.DistributedArray` method), 845
- `sort()` (in module `cupy`), 278
- `sort()` (in module `cupy.array_api`), 906
- `sort_complex()` (in module `cupy`), 280
- `sort_indices()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sort_indices()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sorted_indices()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sorted_indices()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sos2tf()` (in module `cupyx.scipy.signal`), 534
- `sos2zpk()` (in module `cupyx.scipy.signal`), 534
- `sosfilt()` (in module `cupyx.scipy.signal`), 477
- `sosfilt_zi()` (in module `cupyx.scipy.signal`), 477
- `sosfiltfilt()` (in module `cupyx.scipy.signal`), 478
- `sosfreqz()` (in module `cupyx.scipy.signal`), 491
- `spdiags()` (in module `cupyx.scipy.sparse`), 669
- `spectrogram()` (in module `cupyx.scipy.signal`), 578
- `sph_harm()` (in module `cupyx.scipy.special`), 725
- `spherical_yn()` (in module `cupyx.scipy.special`), 704
- `spilu()` (in module `cupyx.scipy.sparse.linalg`), 689
- `splantider()` (in module `cupyx.scipy.interpolate`), 381
- `splder()` (in module `cupyx.scipy.interpolate`), 380
- `spline_filter()` (in module `cupyx.scipy.ndimage`), 428
- `spline_filter()` (in module `cupyx.scipy.signal`), 466
- `spline_filter1d()` (in module `cupyx.scipy.ndimage`), 428
- `split()` (in module `cupy`), 126
- `splu()` (in module `cupyx.scipy.sparse.linalg`), 689
- `spmatrix` (class in `cupyx.scipy.sparse`), 661
- `spsolve()` (in module `cupyx.scipy.sparse.linalg`), 679
- `spsolve_triangular()` (in module `cupyx.scipy.sparse.linalg`), 679
- `squeuclidean()` (in module `cupyx.scipy.spatial.distance`), 699
- `sqrt()` (`cupyx.scipy.sparse.coo_matrix` method), 630
- `sqrt()` (`cupyx.scipy.sparse.csc_matrix` method), 640
- `sqrt()` (`cupyx.scipy.sparse.csr_matrix` method), 650
- `sqrt()` (`cupyx.scipy.sparse.dia_matrix` method), 658
- `sqrt()` (in module `cupy`), 83

- [sqrt\(\)](#) (in module `cupy.array_api`), 906  
[square\(\)](#) (in module `cupy`), 83  
[square\(\)](#) (in module `cupy.array_api`), 906  
[square\(\)](#) (in module `cupyx.scipy.signal`), 557  
[squeeze\(\)](#) (`cupy.ndarray` method), 65  
[squeeze\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 845  
[squeeze\(\)](#) (in module `cupy`), 120  
[squeeze\(\)](#) (in module `cupy.array_api`), 906  
[ss2tf\(\)](#) (in module `cupyx.scipy.signal`), 533  
[ss2zpk\(\)](#) (in module `cupyx.scipy.signal`), 533  
[stack\(\)](#) (in module `cupy`), 123  
[stack\(\)](#) (in module `cupy.array_api`), 906  
[standard\\_cauchy\(\)](#) (`cupy.random.RandomState` method), 247  
[standard\\_cauchy\(\)](#) (in module `cupy.random`), 269  
[standard\\_deviation\(\)](#) (in module `cupyx.scipy.ndimage`), 437  
[standard\\_exponential\(\)](#) (`cupy.random.Generator` method), 234  
[standard\\_exponential\(\)](#) (`cupy.random.RandomState` method), 248  
[standard\\_exponential\(\)](#) (in module `cupy.random`), 270  
[standard\\_gamma\(\)](#) (`cupy.random.Generator` method), 235  
[standard\\_gamma\(\)](#) (`cupy.random.RandomState` method), 248  
[standard\\_gamma\(\)](#) (in module `cupy.random`), 270  
[standard\\_normal\(\)](#) (`cupy.random.Generator` method), 235  
[standard\\_normal\(\)](#) (`cupy.random.RandomState` method), 248  
[standard\\_normal\(\)](#) (in module `cupy.random`), 271  
[standard\\_t\(\)](#) (`cupy.random.RandomState` method), 248  
[standard\\_t\(\)](#) (in module `cupy.random`), 271  
[state\(\)](#) (`cupy.random.MRG32k3a` method), 239  
[state\(\)](#) (`cupy.random.Philox4x3210` method), 241  
[state\(\)](#) (`cupy.random.XORWOW` method), 238  
[StateSpace](#) (class in `cupyx.scipy.signal`), 539  
[std\(\)](#) (`cupy.ndarray` method), 66  
[std\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 845  
[std\(\)](#) (in module `cupy`), 293  
[step\(\)](#) (`cupyx.scipy.signal.dlti` method), 549  
[step\(\)](#) (`cupyx.scipy.signal.lti` method), 538  
[step\(\)](#) (in module `cupyx.scipy.signal`), 546  
[stft\(\)](#) (in module `cupyx.scipy.signal`), 582  
[stop\(\)](#) (`cupyx.distributed.NCCLBackend` method), 840  
[Stream](#) (class in `cupy.cuda`), 779  
[stream\\_ref](#) (`cupy.cuda.MemoryAsync` attribute), 754  
[streamAddCallback\(\)](#) (in module `cupy.cuda.runtime`), 811  
[streamCreate\(\)](#) (in module `cupy.cuda.runtime`), 810  
[streamCreateWithFlags\(\)](#) (in module `cupy.cuda.runtime`), 810  
[streamDestroy\(\)](#) (in module `cupy.cuda.runtime`), 810  
[streamQuery\(\)](#) (in module `cupy.cuda.runtime`), 811  
[streamSynchronize\(\)](#) (in module `cupy.cuda.runtime`), 810  
[streamWaitEvent\(\)](#) (in module `cupy.cuda.runtime`), 811  
[strides](#) (`cupy.ndarray` attribute), 69  
[strides](#) (`cupyx.distributed.array.DistributedArray` attribute), 849  
[subtract\(\)](#) (in module `cupy`), 76  
[subtract\(\)](#) (in module `cupy.array_api`), 906  
[sum\(\)](#) (`cupy.ndarray` method), 66  
[sum\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 845  
[sum\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 630  
[sum\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 640  
[sum\(\)](#) (`cupyx.scipy.sparse.csr_matrix` method), 650  
[sum\(\)](#) (`cupyx.scipy.sparse.dia_matrix` method), 658  
[sum\(\)](#) (`cupyx.scipy.sparse.spmatrix` method), 664  
[sum\(\)](#) (in module `cupy`), 200  
[sum\\_duplicates\(\)](#) (`cupyx.scipy.sparse.coo_matrix` method), 630  
[sum\\_duplicates\(\)](#) (`cupyx.scipy.sparse.csc_matrix` method), 640  
[sum\\_duplicates\(\)](#) (`cupyx.scipy.sparse.csr_matrix` method), 651  
[sum\\_labels\(\)](#) (in module `cupyx.scipy.ndimage`), 437  
[SuperLU](#) (class in `cupyx.scipy.sparse.linalg`), 690  
[SurfaceObject](#) (class in `cupy.cuda.texture`), 795  
[svd\(\)](#) (in module `cupy.linalg`), 179  
[svds\(\)](#) (in module `cupyx.scipy.sparse.linalg`), 688  
[swapaxes\(\)](#) (`cupy.ndarray` method), 66  
[swapaxes\(\)](#) (`cupyx.distributed.array.DistributedArray` method), 846  
[swapaxes\(\)](#) (in module `cupy`), 116  
[symiirorder1\(\)](#) (in module `cupyx.scipy.signal`), 469  
[symiirorder2\(\)](#) (in module `cupyx.scipy.signal`), 470  
[sync](#) (in module `cupyx.jit.cg`), 833  
[synchronize\(\)](#) (`cupy.cuda.Device` method), 749  
[synchronize\(\)](#) (`cupy.cuda.Event` method), 786  
[synchronize\(\)](#) (`cupy.cuda.ExternalStream` method), 785  
[synchronize\(\)](#) (`cupy.cuda.Stream` method), 782  
[syncthreads](#) (in module `cupyx.jit`), 828  
[syncwarp](#) (in module `cupyx.jit`), 828
- ## T
- [T](#) (`cupy.array_api._array_object.Array` attribute), 910  
[T](#) (`cupy.ndarray` attribute), 68  
[T](#) (`cupyx.distributed.array.DistributedArray` attribute), 847  
[T](#) (`cupyx.scipy.sparse.coo_matrix` attribute), 633

- `T` (*cupyx.scipy.sparse.csc\_matrix* attribute), 643
- `T` (*cupyx.scipy.sparse.csr\_matrix* attribute), 653
- `T` (*cupyx.scipy.sparse.dia\_matrix* attribute), 660
- `T` (*cupyx.scipy.sparse.linalg.LinearOperator* attribute), 678
- `T` (*cupyx.scipy.sparse.spmatrix* attribute), 665
- `take()` (*cupy.ndarray* method), 66
- `take()` (*cupyx.distributed.array.DistributedArray* method), 846
- `take()` (in module *cupy*), 162
- `take()` (in module *cupy.array\_api*), 906
- `take_along_axis()` (in module *cupy*), 162
- `tan()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tan()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `tan()` (*cupyx.scipy.sparse.csr\_matrix* method), 651
- `tan()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tan()` (in module *cupy*), 85
- `tan()` (in module *cupy.array\_api*), 907
- `tandg()` (in module *cupyx.scipy.special*), 730
- `tanh()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tanh()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `tanh()` (*cupyx.scipy.sparse.csr\_matrix* method), 651
- `tanh()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tanh()` (in module *cupy*), 86
- `tanh()` (in module *cupy.array\_api*), 907
- `taylor()` (in module *cupyx.scipy.signal.windows*), 621
- `tck` (*cupyx.scipy.interpolate.BSpline* attribute), 379
- `tensor_dot()` (in module *cupy*), 175
- `tensorinv()` (in module *cupy.linalg*), 186
- `tensorsolve()` (in module *cupy.linalg*), 184
- `TexDesc` (*cupy.cuda.texture.TextureObject* attribute), 795
- `TextureDescriptor` (class in *cupy.cuda.texture*), 793
- `TextureObject` (class in *cupy.cuda.texture*), 794
- `tf2sos()` (in module *cupyx.scipy.signal*), 532
- `tf2ss()` (in module *cupyx.scipy.signal*), 532
- `tf2zpk()` (in module *cupyx.scipy.signal*), 531
- `this_grid` (in module *cupyx.jit.cg*), 833
- `this_thread_block` (in module *cupyx.jit.cg*), 833
- `threadIdx` (in module *cupyx.jit*), 825
- `tile()` (in module *cupy*), 128
- `time_range` (class in *cupyx.profiler*), 745
- `to_device()` (*cupy.array\_api.\_array\_object.Array* method), 909
- `to_discrete()` (*cupyx.scipy.signal.lti* method), 538
- `to_ss()` (*cupyx.scipy.signal.StateSpace* method), 540
- `to_ss()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `to_ss()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `to_tf()` (*cupyx.scipy.signal.StateSpace* method), 540
- `to_tf()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `to_tf()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `to_zpk()` (*cupyx.scipy.signal.StateSpace* method), 540
- `to_zpk()` (*cupyx.scipy.signal.TransferFunction* method), 542
- `to_zpk()` (*cupyx.scipy.signal.ZerosPolesGain* method), 544
- `toarray()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `toarray()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `toarray()` (*cupyx.scipy.sparse.csr\_matrix* method), 651
- `toarray()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `toarray()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `tobsr()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tobsr()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `tobsr()` (*cupyx.scipy.sparse.csr\_matrix* method), 651
- `tobsr()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tobsr()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `tobytes()` (*cupy.ndarray* method), 67
- `tobytes()` (*cupyx.distributed.array.DistributedArray* method), 846
- `tocoo()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tocoo()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `tocoo()` (*cupyx.scipy.sparse.csr\_matrix* method), 651
- `tocoo()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tocoo()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `tocsc()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tocsc()` (*cupyx.scipy.sparse.csc\_matrix* method), 641
- `tocsc()` (*cupyx.scipy.sparse.csr\_matrix* method), 652
- `tocsc()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tocsc()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `tocsr()` (*cupyx.scipy.sparse.coo\_matrix* method), 631
- `tocsr()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `tocsr()` (*cupyx.scipy.sparse.csr\_matrix* method), 652
- `tocsr()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `tocsr()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `todense()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `todense()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `todense()` (*cupyx.scipy.sparse.csr\_matrix* method), 652
- `todense()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `todense()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `todia()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `todia()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `todia()` (*cupyx.scipy.sparse.csr\_matrix* method), 652
- `todia()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `todia()` (*cupyx.scipy.sparse.spmatrix* method), 664
- `toDlpack()` (*cupy.ndarray* method), 66
- `toDlpack()` (*cupyx.distributed.array.DistributedArray* method), 846
- `todok()` (*cupyx.scipy.sparse.coo\_matrix* method), 632
- `todok()` (*cupyx.scipy.sparse.csc\_matrix* method), 642
- `todok()` (*cupyx.scipy.sparse.csr\_matrix* method), 652
- `todok()` (*cupyx.scipy.sparse.dia\_matrix* method), 659
- `todok()` (*cupyx.scipy.sparse.spmatrix* method), 665
- `toeplitz()` (in module *cupyx.scipy.linalg*), 404
- `tofile()` (*cupy.ndarray* method), 67

tofile() (*cupyx.distributed.array.DistributedArray* method), 846  
 tolil() (*cupyx.scipy.sparse.coo\_matrix* method), 632  
 tolil() (*cupyx.scipy.sparse.csc\_matrix* method), 642  
 tolil() (*cupyx.scipy.sparse.csr\_matrix* method), 652  
 tolil() (*cupyx.scipy.sparse.dia\_matrix* method), 659  
 tolil() (*cupyx.scipy.sparse.spmatrix* method), 665  
 tolist() (*cupy.ndarray* method), 67  
 tolist() (*cupyx.distributed.array.DistributedArray* method), 846  
 tomaxint() (*cupy.random.RandomState* method), 248  
 total\_bytes() (*cupy.cuda.MemoryAsyncPool* method), 768  
 total\_bytes() (*cupy.cuda.MemoryPool* method), 765  
 trace() (*cupy.ndarray* method), 67  
 trace() (*cupyx.distributed.array.DistributedArray* method), 846  
 trace() (*in module cupy*), 183  
 TransferFunction (*class in cupyx.scipy.signal*), 541  
 transpose() (*cupy.ndarray* method), 67  
 transpose() (*cupyx.distributed.array.DistributedArray* method), 846  
 transpose() (*cupyx.scipy.sparse.coo\_matrix* method), 632  
 transpose() (*cupyx.scipy.sparse.csc\_matrix* method), 642  
 transpose() (*cupyx.scipy.sparse.csr\_matrix* method), 652  
 transpose() (*cupyx.scipy.sparse.dia\_matrix* method), 660  
 transpose() (*cupyx.scipy.sparse.linalg.LinearOperator* method), 677  
 transpose() (*cupyx.scipy.sparse.spmatrix* method), 665  
 transpose() (*in module cupy*), 116  
 trapz() (*in module cupy*), 205  
 tri() (*in module cupy*), 111  
 tri() (*in module cupyx.scipy.linalg*), 404  
 triang() (*in module cupyx.scipy.signal.windows*), 622  
 triangular() (*cupy.random.RandomState* method), 248  
 triangular() (*in module cupy.random*), 272  
 tril() (*in module cupy*), 111  
 tril() (*in module cupy.array\_api*), 907  
 tril() (*in module cupyx.scipy.linalg*), 393  
 tril() (*in module cupyx.scipy.sparse*), 669  
 tril\_indices() (*in module cupy*), 156  
 tril\_indices\_from() (*in module cupy*), 156  
 trim\_mean() (*in module cupyx.scipy.stats*), 733  
 trim\_zeros() (*in module cupy*), 131  
 trimcoef() (*in module cupy.polynomial.polyutils*), 221  
 trimseq() (*in module cupy.polynomial.polyutils*), 220  
 triu() (*in module cupy*), 112  
 triu() (*in module cupy.array\_api*), 907  
 triu() (*in module cupyx.scipy.linalg*), 394  
 triu() (*in module cupyx.scipy.sparse*), 670

triu\_indices() (*in module cupy*), 156  
 triu\_indices\_from() (*in module cupy*), 157  
 true\_divide() (*in module cupy*), 78  
 trunc() (*cupyx.scipy.sparse.coo\_matrix* method), 632  
 trunc() (*cupyx.scipy.sparse.csc\_matrix* method), 642  
 trunc() (*cupyx.scipy.sparse.csr\_matrix* method), 653  
 trunc() (*cupyx.scipy.sparse.dia\_matrix* method), 660  
 trunc() (*in module cupy*), 96  
 trunc() (*in module cupy.array\_api*), 907  
 tukey() (*in module cupyx.scipy.signal.windows*), 623  
 types (*cupy.ufunc* attribute), 75

## U

ufunc (*class in cupy*), 73  
 uniform() (*cupy.random.Generator* method), 235  
 uniform() (*cupy.random.RandomState* method), 249  
 uniform() (*in module cupy.random*), 272  
 uniform\_filter() (*in module cupyx.scipy.ndimage*), 421  
 uniform\_filter1d() (*in module cupyx.scipy.ndimage*), 421  
 union1d() (*in module cupy*), 188  
 unique() (*in module cupy*), 130  
 unique\_all() (*in module cupy.array\_api*), 907  
 unique\_inverse() (*in module cupy.array\_api*), 907  
 unique\_roots() (*in module cupyx.scipy.signal*), 504  
 unique\_values() (*in module cupy.array\_api*), 907  
 unit\_impulse() (*in module cupyx.scipy.signal*), 558  
 UnownedMemory (*class in cupy.cuda*), 755  
 unpackbits() (*in module cupy*), 135  
 unravel\_index() (*in module cupy*), 159  
 unwrap() (*in module cupy*), 197  
 upfirdn() (*in module cupyx.scipy.signal*), 484  
 upload() (*cupy.cuda.Graph* method), 788  
 use() (*cupy.cuda.Device* method), 749  
 use() (*cupy.cuda.ExternalStream* method), 785  
 use() (*cupy.cuda.Stream* method), 782  
 used\_bytes() (*cupy.cuda.MemoryAsyncPool* method), 768  
 used\_bytes() (*cupy.cuda.MemoryPool* method), 766  
 using\_allocator() (*in module cupy.cuda*), 763

## V

value\_indices() (*in module cupyx.scipy.ndimage*), 438  
 values (*cupy.broadcast* attribute), 119  
 vander() (*in module cupy*), 112  
 var() (*cupy.ndarray* method), 67  
 var() (*cupyx.distributed.array.DistributedArray* method), 846  
 var() (*in module cupy*), 294  
 variable (*cupy.poly1d* attribute), 223  
 variance() (*in module cupyx.scipy.ndimage*), 439  
 vdot() (*in module cupy*), 174  
 vectorize (*class in cupy*), 151



`vectorstrength()` (in module `cupyx.scipy.signal`), 582  
`view()` (`cupy.ndarray` method), 67  
`view()` (`cupyx.distributed.array.DistributedArray` method), 846  
`vonmises()` (`cupy.random.RandomState` method), 249  
`vonmises()` (in module `cupy.random`), 273  
`vsplit()` (in module `cupy`), 127  
`vstack()` (in module `cupy`), 124  
`vstack()` (in module `cupyx.scipy.sparse`), 672

## W

`wait` (in module `cupyx.jit.cg`), 834  
`wait_event()` (`cupy.cuda.ExternalStream` method), 785  
`wait_event()` (`cupy.cuda.Stream` method), 782  
`wait_prior` (in module `cupyx.jit.cg`), 834  
`wald()` (`cupy.random.RandomState` method), 249  
`wald()` (in module `cupy.random`), 273  
`warpsize` (in module `cupyx.jit`), 827  
`weibull()` (`cupy.random.RandomState` method), 249  
`weibull()` (in module `cupy.random`), 274  
`welch()` (in module `cupyx.scipy.signal`), 574  
`where()` (in module `cupy`), 154  
`where()` (in module `cupy.array_api`), 908  
`white_tophat()` (in module `cupyx.scipy.ndimage`), 454  
`who()` (in module `cupy`), 213  
`width` (`cupy.cuda.texture.CUDAArray` attribute), 791  
`wiener()` (in module `cupyx.scipy.signal`), 469

## X

`x` (`cupyx.scipy.interpolate.Akima1DInterpolator` attribute), 366  
`x` (`cupyx.scipy.interpolate.BPoly` attribute), 375  
`x` (`cupyx.scipy.interpolate.CubicHermiteSpline` attribute), 357  
`x` (`cupyx.scipy.interpolate.PchipInterpolator` attribute), 362  
`x` (`cupyx.scipy.interpolate.PPoly` attribute), 371  
`xlog1py()` (in module `cupyx.scipy.special`), 732  
`xlogy()` (in module `cupyx.scipy.special`), 731  
`XORWOW` (class in `cupy.random`), 238

## Y

`y0()` (in module `cupyx.scipy.special`), 703  
`y1()` (in module `cupyx.scipy.special`), 703  
`yn()` (in module `cupyx.scipy.special`), 703

## Z

`zeros` (`cupyx.scipy.signal.dlti` attribute), 550  
`zeros` (`cupyx.scipy.signal.lti` attribute), 539  
`zeros` (`cupyx.scipy.signal.StateSpace` attribute), 541  
`zeros` (`cupyx.scipy.signal.TransferFunction` attribute), 543  
`zeros` (`cupyx.scipy.signal.ZerosPolesGain` attribute), 544

`zeros()` (in module `cupy`), 102  
`zeros()` (in module `cupy.array_api`), 908  
`zeros_like()` (in module `cupy`), 102  
`zeros_like()` (in module `cupy.array_api`), 908  
`zeros_like_pinned()` (in module `cupyx`), 740  
`zeros_pinned()` (in module `cupyx`), 740  
`ZerosPolesGain` (class in `cupyx.scipy.signal`), 543  
`zeta()` (in module `cupyx.scipy.special`), 728  
`zetac()` (in module `cupyx.scipy.special`), 728  
`zipf()` (`cupy.random.RandomState` method), 249  
`zipf()` (in module `cupy.random`), 274  
`zmap()` (in module `cupyx.scipy.stats`), 735  
`zoom()` (in module `cupyx.scipy.ndimage`), 429  
`zoom_fft()` (in module `cupyx.scipy.signal`), 591  
`ZoomFFT` (class in `cupyx.scipy.signal`), 594  
`zpk2sos()` (in module `cupyx.scipy.signal`), 530  
`zpk2ss()` (in module `cupyx.scipy.signal`), 530  
`zpk2tf()` (in module `cupyx.scipy.signal`), 529  
`zscore()` (in module `cupyx.scipy.stats`), 736