CuPy Documentation

Release 13.0.0rc1

Preferred Networks, inc. and Preferred Infrastructure, inc.

May 13, 2024

CONTENTS

CHAPTER

ONE

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.

CHAPTER

TWO

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
v11.2 ~ 11.8 (x86_64 / aarch64)	`pip install cupy-cuda11x`
v12.x (x86_64 / aarch64)	`pip install cupy-cuda12x`

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 (`cudatoolkit` 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 youself, 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 `cudatoolkit` 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 `cudatoolkit` 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 `cudatoolkit` 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"
```

CHAPTER

THREE

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 hipsparse rocsparse 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/interpn`), 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)

CHAPTER

FOUR

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 log1p:

```
>>> # 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)
bool	bool	1
int8	char, signed char	1
int16	short, signed short	2
int32	int, signed int	4
int64	long long, signed long long	8
uint8	unsigned char	1
uint16	unsigned short	2
uint32	unsigned int	4
uint64	unsigned long long	8
float16	half	2
float32	float	4
float64	double	8
complex64	float2, cuFloatComplex, complex<float>	8
complex128	double2, cuDoubleComplex, complex<double>	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 perforamance 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 *`fft()`* and *`ifft()`* 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 *`rfft()`* 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 alway 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 accruacy 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 processs 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 Testing with CI/CD

When running CI/CD to test CuPy or any downstream packages that heavily rely on CuPy, depending on the use cases the developers/users may find that JIT compilation takes a non-negligible amount of time. To accelerate testing, it is advised to store the artifacts generated under the cache directory (see the above section) in a persistent location (say, a cloud storage) after the test is finished, regardless of success or failure, so that the artifacts can be re-used across runs, avoiding JIT'ing kernels at test time.

4.6.3 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.

4.6.4 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=""`.

4.6.5 Overlapping work using streams

Under construction.

4.6.6 Use JIT compiler

Under construction. For now please refer to *JIT kernel definition* for a quick introduction.

4.6.7 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 JITed 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
```

(continues on next page)

(continued from previous page)

```
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
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')
```

(continues on next page)

(continued from previous page)

```
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 framefork 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()
```

(continues on next page)

(continued from previous page)

```
# 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(
```

(continues on next page)

(continued from previous page)

```
    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",
)


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
```

(continues on next page)

(continued from previous page)

```
<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)>
>>> 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(300000, 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.

CHAPTER

FIVE

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.

cupy.ndarray(self, shape[, dtype, memptr, ...])	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.argmin() for full documentation, numpy.ndarray.argmin()

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<value.

__le__(*value*, /)
Return self<=value.

__gt__(*value*, /)
Return self>value.

__ge__(*value*, /)
Return self>=value.

__bool__()
True if self else False

Attributes

T

Shape-reversed view of the array.

If ndim < 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.ndarray*s 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.

cupy.array(obj[, dtype, copy, order, subok, ...])	Creates an array on the current device.
cupy.asarray(a[, dtype, order, blocking])	Converts an object to array.
cupy.asnumpy(a[, stream, order, out, blocking])	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.

cupy.get_array_module(*args)	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)))
```

cupyx.scipy.get_array_module(*args)	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

ufunc(name, nin, nout, _Ops ops[, preamble, ...])	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

add(x1, x2, /[, out, casting, dtype])	Adds two arrays elementwise.
subtract(x1, x2, /[, out, casting, dtype])	Subtracts arguments elementwise.
multiply(x1, x2, /[, out, casting, dtype])	Multiplies two arrays elementwise.
matmul	matmul(x1, x2, /, out=None, **kwargs)
divide	true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)
logaddexp(x1, x2, /[, out, casting, dtype])	Computes `log(exp(x1) + exp(x2))` elementwise.
logaddexp2(x1, x2, /[, out, casting, dtype])	Computes `log2(exp2(x1) + exp2(x2))` element-wise.
true_divide(x1, x2, /[, out, casting, dtype])	Elementwise true division (i.e.
floor_divide(x1, x2, /[, out, casting, dtype])	Elementwise floor division (i.e.
negative(x, /[, out, casting, dtype])	Takes numerical negative elementwise.
positive(x, /[, out, casting, dtype])	Takes numerical positive elementwise.
power(x1, x2, /[, out, casting, dtype])	Computes `x1 ** x2` elementwise.
float_power(x1, x2, /[, out, casting, dtype])	First array elements raised to powers from second array, element-wise.

continues on next page

Table 1 – continued from previous page

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

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

bitwise_and(x1, x2, /[, out, casting, dtype])	Computes the bitwise AND of two arrays elementwise.
bitwise_or(x1, x2, /[, out, casting, dtype])	Computes the bitwise OR of two arrays elementwise.
bitwise_xor(x1, x2, /[, out, casting, dtype])	Computes the bitwise XOR of two arrays elementwise.
invert(x, /[, out, casting, dtype])	Computes the bitwise NOT of an array elementwise.
left_shift(x1, x2, /[, out, casting, dtype])	Shifts the bits of each integer element to the left.
right_shift(x1, x2, /[, out, casting, dtype])	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

greater(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 > x2`.
greater_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 >= x2`.
less(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 < x2`.
less_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 <= x2`.
not_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 != x2`.
equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 == x2`.
logical_and(x1, x2, /[, out, casting, dtype])	Computes the logical AND of two arrays.
logical_or(x1, x2, /[, out, casting, dtype])	Computes the logical OR of two arrays.
logical_xor(x1, x2, /[, out, casting, dtype])	Computes the logical XOR of two arrays.
logical_not(x, /[, out, casting, dtype])	Computes the logical NOT of an array.
maximum(x1, x2, /[, out, casting, dtype])	Takes the maximum of two arrays elementwise.
minimum(x1, x2, /[, out, casting, dtype])	Takes the minimum of two arrays elementwise.
fmax(x1, x2, /[, out, casting, dtype])	Takes the maximum of two arrays elementwise.
fmin(x1, x2, /[, out, casting, dtype])	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 <= 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 != 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

isfinite(x, /[, out, casting, dtype])	Tests finiteness elementwise.
isinf(x, /[, out, casting, dtype])	Tests if each element is the positive or negative infinity.
isnan(x, /[, out, casting, dtype])	Tests if each element is a NaN.
fabs(x, /[, out, casting, dtype])	Calculates absolute values element-wise.
signbit(x, /[, out, casting, dtype])	Tests elementwise if the sign bit is set (i.e.
copysign(x1, x2, /[, out, casting, dtype])	Returns the first argument with the sign bit of the second elementwise.
nextafter(x1, x2, /[, out, casting, dtype])	Computes the nearest neighbor float values towards the second argument.
modf(x[, out1, out2], / [[, out, casting, dtype])	Extracts the fractional and integral parts of an array elementwise.
ldexp(x1, x2, /[, out, casting, dtype])	Computes `x1 * 2 ** x2` elementwise.
frexp(x[, out1, out2], / [[, out, casting, ...])	Decomposes each element to mantissa and two's exponent.
fmod(x1, x2, /[, out, casting, dtype])	Computes the remainder of C division elementwise.
floor(x, /[, out, casting, dtype])	Rounds each element of an array to its floor integer.
ceil(x, /[, out, casting, dtype])	Rounds each element of an array to its ceiling integer.
trunc(x, /[, out, casting, dtype])	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.

GeneralizedUFunc(func, signature, **kwargs)	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)->(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)->()'`, 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)->()'` or `'(m,m)->()'`. 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

empty(shape[, dtype, order])	Returns an array without initializing the elements.
empty_like(prototype[, dtype, order, subok, ...])	Returns a new array with same shape and dtype of a given array.
eye(N[, M, k, dtype, order])	Returns a 2-D array with ones on the diagonals and zeros elsewhere.
identity(n[, dtype])	Returns a 2-D identity array.
ones(shape[, dtype, order])	Returns a new array of given shape and dtype, filled with ones.
ones_like(a[, dtype, order, subok, shape])	Returns an array of ones with same shape and dtype as a given array.
zeros(shape[, dtype, order])	Returns a new array of given shape and dtype, filled with zeros.
zeros_like(a[, dtype, order, subok, shape])	Returns an array of zeros with same shape and dtype as a given array.
full(shape, fill_value[, dtype, order])	Returns a new array of given shape and dtype, filled with a given value.
full_like(a, fill_value[, dtype, order, ...])	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

array(obj[, dtype, copy, order, subok, ...])	Creates an array on the current device.
asarray(a[, dtype, order, blocking])	Converts an object to array.
asanyarray(a[, dtype, order, blocking])	Converts an object to array.
ascontiguousarray(a[, dtype])	Returns a C-contiguous array.
copy(a[, order])	Creates a copy of a given array on the current device.
frombuffer(*args, **kwargs)	Interpret a buffer as a 1-dimensional array.
fromfile(*args, **kwargs)	Reads an array from a file.
fromfunction(*args, **kwargs)	Construct an array by executing a function over each coordinate.
fromiter(*args, **kwargs)	Create a new 1-dimensional array from an iterable object.
fromstring(*args, **kwargs)	A new 1-D array initialized from text data in a string.
loadtxt(*args, **kwargs)	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

arange(start[, stop, step, dtype])	Returns an array with evenly spaced values within a given interval.
linspace(start, stop[, num, endpoint, ...])	Returns an array with evenly-spaced values within a given interval.
logspace(start, stop[, num, endpoint, base, ...])	Returns an array with evenly-spaced values on a log-scale.
meshgrid(*xi, **kwargs)	Return coordinate matrices from coordinate vectors.
mgrid	Construct a multi-dimensional "meshgrid".
ogrid	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

diag(v[, k])	Returns a diagonal or a diagonal array.
diagflat(v[, k])	Creates a diagonal array from the flattened input.
tri(N[, M, k, dtype])	Creates an array with ones at and below the given diagonal.
tril(m[, k])	Returns a lower triangle of an array.
triu(m[, k])	Returns an upper triangle of an array.
vander(x[, N, increasing])	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 = 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

copyto(dst, src[, casting, where])	Copies values from one array to another with broadcasting.
shape(a)	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

reshape(a, newshape[, order])	Returns an array with new shape and same elements.
ravel(a[, order])	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

moveaxis(a, source, destination)	Moves axes of an array to new positions.
rollaxis(a, axis[, start])	Moves the specified axis backwards to the given place.
swapaxes(a, axis1, axis2)	Swaps the two axes.
transpose(a[, axes])	Permutes 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

atleast_1d(*arys)	Converts arrays to arrays with dimensions >= 1.
atleast_2d(*arys)	Converts arrays to arrays with dimensions >= 2.
atleast_3d(*arys)	Converts arrays to arrays with dimensions >= 3.
broadcast(*arrays)	Object that performs broadcasting.
broadcast_to(array, shape)	Broadcast an array to a given shape.
broadcast_arrays(*args)	Broadcasts given arrays.
expand_dims(a, axis)	Expands given arrays.
squeeze(a[, axis])	Removes size-one axes from the shape of an array.

cupy.atleast_1d

cupy.**atleast_1d**(**arys*)

Converts arrays to arrays with dimensions >= 1.

Parameters
arys (*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**(**arys*)

Converts arrays to arrays with dimensions >= 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
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_2d()

cupy.atleast_3d

cupy.**atleast_3d**(**arys*)

Converts arrays to arrays with dimensions >= 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

asarray(a[, dtype, order, blocking])	Converts an object to array.
asanyarray(a[, dtype, order, blocking])	Converts an object to array.
asfarray(a[, dtype])	Converts array elements to float type.
asfortranarray(a[, dtype])	Return an array laid out in Fortran order in memory.
ascontiguousarray(a[, dtype])	Returns a C-contiguous array.
asarray_chkfinite(a[, dtype, order])	Converts the given input to an array, and raises an error if the input contains NaNs or Infs.
require(a[, dtype, requirements])	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

concatenate(tup[, axis, out, dtype, casting])	Joins arrays along an axis.
stack(tup[, axis, out, dtype, casting])	Stacks arrays along a new axis.
vstack(tup, *[, dtype, casting])	Stacks arrays vertically.
hstack(tup, *[, dtype, casting])	Stacks arrays horizontally.
dstack(tup)	Stacks arrays along the third axis.
column_stack(tup)	Stacks 1-D and 2-D arrays as columns into a 2-D array.
row_stack(tup, *[, dtype, casting])	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

split(ary, indices_or_sections[, axis])	Splits an array into multiple sub arrays along a given axis.
array_split(ary, indices_or_sections[, axis])	Splits an array into multiple sub arrays along a given axis.
dsplit(ary, indices_or_sections)	Splits an array into multiple sub arrays along the third axis.
hsplit(ary, indices_or_sections)	Splits an array into multiple sub arrays horizontally.
vsplit(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

tile(A, reps)	Construct an array by repeating A the number of times given by reps.
repeat(a, repeats[, axis])	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

delete(arr, indices[, axis])	Delete values from an array along the specified axis.
append(arr, values[, axis])	Append values to the end of an array.
resize(a, new_shape)	Return a new array with the specified shape.
unique(ar[, return_index, return_inverse, ...])	Find the unique elements of an array.
trim_zeros(filt[, trim])	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

flip(a[, axis])	Reverse the order of elements in an array along the given axis.
fliplr(a)	Flip array in the left/right direction.
flipud(a)	Flip array in the up/down direction.
reshape(a, newshape[, order])	Returns an array with new shape and same elements.
roll(a, shift[, axis])	Roll array elements along a given axis.
rot90(a[, k, axes])	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

bitwise_and(x1, x2, /[, out, casting, dtype])	Computes the bitwise AND of two arrays elementwise.
bitwise_or(x1, x2, /[, out, casting, dtype])	Computes the bitwise OR of two arrays elementwise.
bitwise_xor(x1, x2, /[, out, casting, dtype])	Computes the bitwise XOR of two arrays elementwise.
invert(x, /[, out, casting, dtype])	Computes the bitwise NOT of an array elementwise.
left_shift(x1, x2, /[, out, casting, dtype])	Shifts the bits of each integer element to the left.
right_shift(x1, x2, /[, out, casting, dtype])	Shifts the bits of each integer element to the right.

Bit packing

packbits(a[, axis, bitorder])	Packs the elements of a binary-valued array into bits in a uint8 array.
unpackbits(a[, axis, bitorder])	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), alghough 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

binary_repr(num[, width])	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

can_cast(from_, to[, casting])	Returns True if cast between data types can occur according to the casting rule.
min_scalar_type(a)	For scalar a, returns the data type with the smallest size and smallest scalar kind which can hold its value.
result_type(*arrays_and_dtypes)	Returns the type that results from applying the NumPy type promotion rules to the arguments.
common_type(*arrays)	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()

promote_types (alias of numpy.promote_types())
obj2sctype (alias of numpy.obj2sctype())

Creating data types

dtype (alias of numpy.dtype)
format_parser (alias of numpy.format_parser)

Data type information

finfo (alias of numpy.finfo)
iinfo (alias of numpy.iinfo)
MachAr (alias of numpy.MachAr)

Data type testing

issctype (alias of numpy.issctype())
issubdtype (alias of numpy.issubdtype())
issubsctype (alias of numpy.issubsctype())
issubclass_ (alias of numpy.issubclass_())
find_common_type (alias of numpy.find_common_type())

Miscellaneous

typename (alias of numpy.typename())
sctype2char (alias of numpy.sctype2char())
mintypecode (alias of numpy.mintypecode())

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

fft(a[, n, axis, norm])	Compute the one-dimensional FFT.
ifft(a[, n, axis, norm])	Compute the one-dimensional inverse FFT.
fft2(a[, s, axes, norm])	Compute the two-dimensional FFT.
ifft2(a[, s, axes, norm])	Compute the two-dimensional inverse FFT.
fftn(a[, s, axes, norm])	Compute the N-dimensional FFT.
ifftn(a[, s, axes, norm])	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

rfft(a[, n, axis, norm])	Compute the one-dimensional FFT for real input.
irfft(a[, n, axis, norm])	Compute the one-dimensional inverse FFT for real input.
rfft2(a[, s, axes, norm])	Compute the two-dimensional FFT for real input.
irfft2(a[, s, axes, norm])	Compute the two-dimensional inverse FFT for real input.
rfftn(a[, s, axes, norm])	Compute the N-dimensional FFT for real input.
irfftn(a[, s, axes, norm])	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

hfft(a[, n, axis, norm])	Compute the FFT of a signal that has Hermitian symmetry.
ihfft(a[, n, axis, norm])	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

fftfreq(n[, d])	Return the FFT sample frequencies.
rfftfreq(n[, d])	Return the FFT sample frequencies for real input.
fftshift(x[, axes])	Shift the zero-frequency component to the center of the spectrum.
ifftshift(x[, axes])	The inverse of *fftshift()*.

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.

config.set_cufft_callbacks(...)	A context manager for setting up load and/or store callbacks.
config.set_cufft_gpus(gpus)	Set the GPUs to be used in multi-GPU FFT.
config.get_plan_cache()	Get the per-thread, per-device plan cache, or create one if not found.
config.show_plan_cache_info()	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.

apply_along_axis(func1d, axis, arr, *args, ...)	Apply a function to 1-D slices along the given axis.
vectorize(pyfunc[, otypes, doc, excluded, ...])	Generalized function class.
piecewise(x, condlist, funclist)	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,) -> (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

Warning: 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

*c_*	
*r_*	
nonzero(a)	Return the indices of the elements that are non-zero.
where(condition[, x, y])	Return elements, either from x or y, depending on condition.
indices(dimensions[, dtype])	Returns an array representing the indices of a grid.
mask_indices(n, mask_func[, k])	Return the indices to access (n, n) arrays, given a masking function.
tril_indices(n[, k, m])	Returns the indices of the lower triangular matrix.
tril_indices_from(arr[, k])	Returns the indices for the lower-triangle of arr.
triu_indices(n[, k, m])	Returns the indices of the upper triangular matrix.
triu_indices_from(arr[, k])	Returns indices for the upper-triangle of arr.
*ix_*(*args)	Construct an open mesh from multiple sequences.
ravel_multi_index(multi_index, dims[, mode, ...])	Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.
unravel_index(indices, dims[, order])	Converts array of flat indices into a tuple of coordinate arrays.
diag_indices(n[, ndim])	Return the indices to access the main diagonal of an array.
diag_indices_from(arr)	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 dialogal. 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 range

  In '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

take(a, indices[, axis, out])	Takes elements of an array at specified indices along an axis.
take_along_axis(a, indices, axis)	Take values from the input array by matching 1d index and data slices.
choose(a, choices[, out, mode])	
compress(condition, a[, axis, out])	Returns selected slices of an array along given axis.
diag(v[, k])	Returns a diagonal or a diagonal array.
diagonal(a[, offset, axis1, axis2])	Returns specified diagonals.
select(condlist, choicelist[, default])	Return an array drawn from elements in choicelist, depending on conditions.
lib.stride_tricks.as_strided(x[, shape, strides])	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

place(arr, mask, vals)	Change elements of an array based on conditional and input values.
put(a, ind, v[, mode])	Replaces specified elements of an array with given values.
put_along_axis(arr, indices, values, axis)	Put values into the destination array by matching 1d index and data slices.
putmask(a, mask, values)	Changes elements of an array inplace, based on a conditional mask and input values.
fill_diagonal(a, val[, wrap])	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.put_along_axis

cupy.**put_along_axis**(*arr*, *indices*, *values*, *axis*)

Put values into the destination array by matching 1d index and data slices.

This iterates over matching 1d slices oriented along the specified axis in the index and data arrays, and uses the former to place values into the latter. These slices can be different lengths.

Functions returning an index along an axis, like *argsort* and *argpartition*, produce suitable indices for this function.

Parameters

- **arr** – cupy.ndarray (Ni..., M, Nk...) Destination array.
- **indices** – cupy.ndarray (Ni..., J, Nk...) Indices to change along each 1d slice of *arr*. This must match the dimension of arr, but dimensions in Ni and Nj may be 1 to broadcast against *arr*.
- **values** – array_like (Ni..., J, Nk...) values to insert at those indices. Its shape and dimension are broadcast to match that of *indices*.
- **axis** – int The axis to take 1d slices along. If axis is None, the destination array is treated as if a flattened 1d view had been created of it.

See also:

numpy.put_along_axis()

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)

load(file[, mmap_mode, allow_pickle])	Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.
save(file, arr[, allow_pickle])	Saves an array to a binary file in `.npy` format.
savez(file, *args, **kwds)	Saves one or more arrays into a file in uncompressed `.npz` format.
savez_compressed(file, *args, **kwds)	Saves one or more arrays into a file in compressed `.npz` 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 npy 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

loadtxt(*args, **kwargs)	Load data from a text file.
savetxt(fname, X, *args, **kwargs)	Save an array to a text file.
genfromtxt(*args, **kwargs)	Load data from text file, with missing values handled as specified.
fromstring(*args, **kwargs)	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

array2string(a, *args, **kwargs)	Return a string representation of an array.
array_repr(arr[, max_line_width, precision, ...])	Returns the string representation of an array.
array_str(arr[, max_line_width, precision, ...])	Returns the string representation of the content of an array.
format_float_positional(x, *args, **kwargs)	Format a floating-point scalar as a decimal string in positional notation.
format_float_scientific(x, *args, **kwargs)	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

binary_repr(num[, width])	Return the binary representation of the input number as a string.
base_repr(number[, base, padding])	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

dot(a, b[, out])	Returns a dot product of two arrays.
vdot(a, b)	Returns the dot product of two vectors.
inner(a, b)	Returns the inner product of two arrays.
outer(a, b[, out])	Returns the outer product of two vectors.
matmul	matmul(x1, x2, /, out=None, **kwargs)
tensordot(a, b[, axes])	Returns the tensor dot product of two arrays along specified axes.
einsum(subscripts, *operands[, dtype, optimize])	Evaluates the Einstein summation convention on the operands.
linalg.matrix_power(M, n)	Raise a square matrix to the (integer) power *n*.
kron(a, b)	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

linalg.cholesky(a)	Cholesky decomposition.
linalg.qr(a[, mode])	QR decomposition.
linalg.svd(a[, full_matrices, compute_uv])	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 `(..., M, M)`.

Returns
The lower-triangular matrix of shape `(..., 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 = (..., M, N)` is decomposed into `Q`, `R` with dimensions `(..., M, K)`, `(..., 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 * 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 `(..., M, N)`.
- **full_matrices** (*bool*) – If True, it returns u and v with dimensions `(..., M, M)` and `(..., N, N)`. Otherwise, the dimensions of u and v are `(..., M, K)` and `(..., 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 * 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 `cupyx.errstate()` or `cupyx.seterr()`.

Note: On CUDA, when `a.ndim > 2` and the matrix dimensions <= 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

linalg.eigh(a[, UPLO])	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.
linalg.eigvalsh(a[, UPLO])	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

linalg.norm(x[, ord, axis, keepdims])	Returns one of matrix norms specified by `ord` parameter.
linalg.det(a)	Returns the determinant of an array.
linalg.matrix_rank(M[, tol])	Return matrix rank of array using SVD method
linalg.slogdet(a)	Returns sign and logarithm of the determinant of an array.
trace(a[, offset, axis1, axis2, dtype, out])	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.shape) * 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 `(..., 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

linalg.solve(a, b)	Solves a linear matrix equation.
linalg.tensorsolve(a, b[, axes])	Solves tensor equations denoted by `ax = b`.
linalg.lstsq(a, b[, rcond])	Return the least-squares solution to a linear matrix equation.
linalg.inv(a)	Computes the inverse of a matrix.
linalg.pinv(a[, rcond])	Compute the Moore-Penrose pseudoinverse of a matrix.
linalg.tensorinv(a[, ind])	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 `x` in `ax = b`, where `a` is a square and full rank matrix.

Parameters

- **a** (cupy.ndarray) – The matrix with dimension `(..., M, M)`.
- **b** (cupy.ndarray) – The matrix with dimension `(..., M)` or `(..., M, K)`.

Returns

The matrix with dimension `(..., M)` or `(..., M, 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 $a\ x = b$ by computing a vector x that minimizes the Euclidean 2-norm $|| b - a\ x ||^2$. 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 <= 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 `(..., 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 `rcond * s` to zero. Broadcasts against the stack of matrices.

Returns
The pseudoinverse of `a` with dimension `(..., 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

all(a[, axis, out, keepdims])	Tests whether all array elements along a given axis evaluate to True.
any(a[, axis, out, keepdims])	Tests whether any array elements along a given axis evaluate to True.
union1d(arr1, arr2)	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 flattend if they are not already 1-D.
- **arr2** (cupy.ndarray) – Input arrays. They are flattend 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

isfinite(x, /[, out, casting, dtype])	Tests finiteness elementwise.
isinf(x, /[, out, casting, dtype])	Tests if each element is the positive or negative infinity.
isnan(x, /[, out, casting, dtype])	Tests if each element is a NaN.
isneginf(x[, out])	Test element-wise for negative infinity, return result as bool array.
isposinf(x[, out])	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

iscomplex(x)	Returns a bool array, where True if input element is complex.
iscomplexobj(x)	Check for a complex type or an array of complex numbers.
isfortran(a)	Returns True if the array is Fortran contiguous but *not* C contiguous.
isreal(x)	Returns a bool array, where True if input element is real.
isrealobj(x)	Return True if x is a not complex type or an array of complex numbers.
isscalar(element)	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

logical_and(x1, x2, /[, out, casting, dtype])	Computes the logical AND of two arrays.
logical_or(x1, x2, /[, out, casting, dtype])	Computes the logical OR of two arrays.
logical_not(x, /[, out, casting, dtype])	Computes the logical NOT of an array.
logical_xor(x1, x2, /[, out, casting, dtype])	Computes the logical XOR of two arrays.

Comparison

allclose(a, b[, rtol, atol, equal_nan])	Returns True if two arrays are element-wise equal within a tolerance.
isclose(a, b[, rtol, atol, equal_nan])	Returns a boolean array where two arrays are equal within a tolerance.
array_equal(a1, a2[, equal_nan])	Returns `True` if two arrays are element-wise exactly equal.
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.
greater(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 > x2`.
greater_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 >= x2`.
less(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 < x2`.
less_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 <= x2`.
equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 == x2`.
not_equal(x1, x2, /[, out, casting, dtype])	Tests elementwise if `x1 != 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 considiered 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 considiered 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

sin(x, /[, out, casting, dtype])	Elementwise sine function.
cos(x, /[, out, casting, dtype])	Elementwise cosine function.
tan(x, /[, out, casting, dtype])	Elementwise tangent function.
arcsin(x, /[, out, casting, dtype])	Elementwise inverse-sine function (a.k.a.
arccos(x, /[, out, casting, dtype])	Elementwise inverse-cosine function (a.k.a.
arctan(x, /[, out, casting, dtype])	Elementwise inverse-tangent function (a.k.a.
hypot(x1, x2, /[, out, casting, dtype])	Computes the hypoteneous of orthogonal vectors of given length.
arctan2(x1, x2, /[, out, casting, dtype])	Elementwise inverse-tangent of the ratio of two arrays.
degrees	rad2deg(x, /, out=None, *, casting='same_kind', dtype=None)
radians(x, /[, out, casting, dtype])	Converts angles from degrees to radians elementwise.
unwrap(p[, discont, axis, period])	Unwrap by taking the complement of large deltas w.r.t.
deg2rad	radians(x, /, out=None, *, casting='same_kind', dtype=None)
rad2deg(x, /[, out, casting, dtype])	Converts angles from radians to degrees elementwise.

cupy.unwrap

cupy.**unwrap**(*p*, *discont=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(discont, period/2)` to their *period*-complementary values.

For the default case where *period* is 2π and is `discont` is π, this unwraps a radian phase p such that adjacent differences are never greater than π by adding $2k\pi$ for some integer k.

Parameters

- **p** (*cupy.ndarray*) – Input array. discont (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, `discont` 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 2π.

Returns

The result array.

Return type

cupy.ndarray

See also:

`numpy.unwrap()`

Hyperbolic functions

sinh(x, /[, out, casting, dtype])	Elementwise hyperbolic sine function.
cosh(x, /[, out, casting, dtype])	Elementwise hyperbolic cosine function.
tanh(x, /[, out, casting, dtype])	Elementwise hyperbolic tangent function.
arcsinh(x, /[, out, casting, dtype])	Elementwise inverse of hyperbolic sine function.
arccosh(x, /[, out, casting, dtype])	Elementwise inverse of hyperbolic cosine function.
arctanh(x, /[, out, casting, dtype])	Elementwise inverse of hyperbolic tangent function.

Rounding

around(a[, decimals, out])	Rounds to the given number of decimals.
*round_*(a[, decimals, out])	

rint(x, /[, out, casting, dtype])	Rounds each element of an array to the nearest integer.
fix(x, /[, out, casting, dtype])	If given value x is positive, it return floor(x).
floor(x, /[, out, casting, dtype])	Rounds each element of an array to its floor integer.
ceil(x, /[, out, casting, dtype])	Rounds each element of an array to its ceiling integer.
trunc(x, /[, out, casting, dtype])	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

prod(a[, axis, dtype, out, keepdims])	Returns the product of an array along given axes.
sum(a[, axis, dtype, out, keepdims])	Returns the sum of an array along given axes.
nanprod(a[, axis, dtype, out, keepdims])	Returns the product of an array along given axes treating Not a Numbers (NaNs) as zero.
nansum(a[, axis, dtype, out, keepdims])	Returns the sum of an array along given axes treating Not a Numbers (NaNs) as zero.
cumprod(a[, axis, dtype, out])	Returns the cumulative product of an array along a given axis.
cumsum(a[, axis, dtype, out])	Returns the cumulative sum of an array along a given axis.
nancumprod(a[, axis, dtype, out])	Returns the cumulative product of an array along a given axis treating Not a Numbers (NaNs) as one.
nancumsum(a[, axis, dtype, out])	Returns the cumulative sum of an array along a given axis treating Not a Numbers (NaNs) as zero.
diff(a[, n, axis, prepend, append])	Calculate the n-th discrete difference along the given axis.
gradient(f, *varargs[, axis, edge_order])	Return the gradient of an N-dimensional array.
ediff1d(arr[, to_end, to_begin])	Calculates the difference between consecutive elements of an array.
cross(a, b[, axisa, axisb, axisc, axis])	Returns the cross product of two vectors.
trapz(y[, x, dx, axis])	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 returend 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

exp(x, /[, out, casting, dtype])	Elementwise exponential function.
expm1(x, /[, out, casting, dtype])	Computes `exp(x) - 1` elementwise.
exp2(x, /[, out, casting, dtype])	Elementwise exponentiation with base 2.
log(x, /[, out, casting, dtype])	Elementwise natural logarithm function.
log10(x, /[, out, casting, dtype])	Elementwise common logarithm function.
log2(x, /[, out, casting, dtype])	Elementwise binary logarithm function.
log1p(x, /[, out, casting, dtype])	Computes `log(1 + x)` elementwise.
logaddexp(x1, x2, /[, out, casting, dtype])	Computes `log(exp(x1) + exp(x2))` elementwise.
logaddexp2(x1, x2, /[, out, casting, dtype])	Computes `log2(exp2(x1) + exp2(x2))` elementwise.

Other special functions

i0(x, /[, out, casting, dtype])	Modified Bessel function of the first kind, order 0.
sinc(x, /[, out, casting, dtype])	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

signbit(x, /[, out, casting, dtype])	Tests elementwise if the sign bit is set (i.e.
copysign(x1, x2, /[, out, casting, dtype])	Returns the first argument with the sign bit of the second elementwise.
frexp(x[, out1, out2], / [[, out, casting, ...])	Decomposes each element to mantissa and two's exponent.
ldexp(x1, x2, /[, out, casting, dtype])	Computes `x1 * 2 ** x2` elementwise.
nextafter(x1, x2, /[, out, casting, dtype])	Computes the nearest neighbor float values towards the second argument.

Rational routines

lcm(x1, x2, /[, out, casting, dtype])	Computes lcm of `x1` and `x2` elementwise.
gcd(x1, x2, /[, out, casting, dtype])	Computes gcd of `x1` and `x2` elementwise.

Arithmetic operations

add(x1, x2, /[, out, casting, dtype])	Adds two arrays elementwise.
reciprocal(x, /[, out, casting, dtype])	Computes `1 / x` elementwise.
positive(x, /[, out, casting, dtype])	Takes numerical positive elementwise.
negative(x, /[, out, casting, dtype])	Takes numerical negative elementwise.
multiply(x1, x2, /[, out, casting, dtype])	Multiplies two arrays elementwise.
divide	true_divide(x1, x2, /, out=None, *, casting='same_kind', dtype=None)
power(x1, x2, /[, out, casting, dtype])	Computes `x1 ** x2` elementwise.
subtract(x1, x2, /[, out, casting, dtype])	Subtracts arguments elementwise.
true_divide(x1, x2, /[, out, casting, dtype])	Elementwise true division (i.e.
floor_divide(x1, x2, /[, out, casting, dtype])	Elementwise floor division (i.e.
float_power(x1, x2, /[, out, casting, dtype])	First array elements raised to powers from second array, element-wise.
fmod(x1, x2, /[, out, casting, dtype])	Computes the remainder of C division elementwise.
mod(x1, x2, /[, out, casting, dtype])	Computes the remainder of Python division elementwise.
modf(x[, out1, out2], / [[, out, casting, dtype])	Extracts the fractional and integral parts of an array elementwise.
remainder	mod(x1, x2, /, out=None, *, casting='same_kind', dtype=None)
divmod(x1, x2[, out1, out2], / [[, out, ...])	

Handling complex numbers

angle(z[, deg])	Returns the angle of the complex argument.
real(val)	Returns the real part of the elements of the array.
imag(val)	Returns the imaginary part of the elements of the array.
conj	conjugate(x, /, out=None, *, casting='same_kind', dtype=None)
conjugate(x, /[, out, casting, dtype])	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

convolve(a, v[, mode])	Returns the discrete, linear convolution of two one-dimensional sequences.
clip(a, a_min, a_max[, out])	Clips the values of an array to a given interval.
sqrt(x, /[, out, casting, dtype])	Elementwise square root function.
cbrt(x, /[, out, casting, dtype])	Elementwise cube root function.
square(x, /[, out, casting, dtype])	Elementwise square function.
absolute(x, /[, out, casting, dtype])	Elementwise absolute value function.
fabs(x, /[, out, casting, dtype])	Calculates absolute values element-wise.
sign(x, /[, out, casting, dtype])	Elementwise sign function.
maximum(x1, x2, /[, out, casting, dtype])	Takes the maximum of two arrays elementwise.
minimum(x1, x2, /[, out, casting, dtype])	Takes the minimum of two arrays elementwise.
fmax(x1, x2, /[, out, casting, dtype])	Takes the maximum of two arrays elementwise.
fmin(x1, x2, /[, out, casting, dtype])	Takes the minimum of two arrays elementwise.
nan_to_num(x[, copy, nan, posinf, neginf])	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.
heaviside(x1, x2, /[, out, casting, dtype])	Compute the Heaviside step function.
real_if_close(a[, tol])	If input is complex with all imaginary parts close to zero, return real parts.
interp(x, xp, fp[, left, right, period])	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*).

> **Warning:** 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

byte_bounds(a)	Returns pointers to the end-points of an array.
shares_memory(a, b[, max_work])	
may_share_memory(a, b[, max_work])	

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

show_config(*[, _full])	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

who([vardict])	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

polyvander(x, deg)	Computes the Vandermonde matrix of given degree.
polycompanion(c)	Computes the companion matrix of c.
polyval(x, c[, tensor])	Evaluate a polynomial at points x.
polyvalfromroots(x, r[, tensor])	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 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.shape[1:] + x.shape; that is, each polynomial is evaluated at every value of *x*. If *tensor* is `False`, the shape will be r.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 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.polyvalfroomroots`

Polyutils

Hint: NumPy API Reference: Polyutils

Functions

as_series(alist[, trim])	Returns argument as a list of 1-d arrays.
trimseq(seq)	Removes small polynomial series coefficients.
trimcoef(c[, tol])	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

poly1d(c_or_r[, r, variable])	A one-dimensional polynomial class.
cupy.poly(seq_of_zeros)	Computes the coefficients of a polynomial with the given roots sequence.
polyval(p, x)	Evaluates a polynomial at specific values.
roots(p)	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

polyfit(x, y, deg[, rcond, full, w, cov])	Returns the least squares fit of polynomial of degree deg to the data y sampled at x.

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

polyadd(a1, a2)	Computes the sum of two polynomials.
polysub(a1, a2)	Computes the difference of two polynomials.
polymul(a1, a2)	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

default_rng([seed])	Construct a new Generator with the default BitGenerator (XORWOW).
Generator(bit_generator)	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 detive the initial *BitGenerator* state. One may also pass in a *SeedSequence instance. Adiditionally, 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 α.
- **b** (*float*) – Parameter of the beta distribution β.
- **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, >= 0. Floats are also accepted, but they will be truncated to integers.
- **p** (*float or cupy.ndarray of floats*) – Parameter of the distribution, >= 0 and <= 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 α.
- **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

$$f(x) = \frac{1}{\beta} \exp(-\frac{x}{\beta}).$$

Parameters

- **scale** (*float or array_like of floats*) – The scale parameter β.
- **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

$$f(x) = \frac{1}{B(\frac{d_1}{2}, \frac{d_2}{2})} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2} x\right)^{-\frac{d_1+d_2}{2}}.$$

Parameters

- **dfnum** (*float or array_like of floats*) – 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 λ.
- **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) * 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 `low + (high-low) * 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

BitGenerator([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:

XORWOW([seed, size])	BitGenerator that uses cuRAND XORWOW device generator.
MRG32k3a([seed, size])	BitGenerator that uses cuRAND MRG32k3a device generator.
Philox4x3210([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<value.

`__le__`(*value*, /)
Return self<=value.

`__gt__`(*value*, /)
Return self>value.

`__ge__`(*value*, /)
Return self>=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.

> **Warning:** 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.

> **Warning:** 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

beta(a, b[, size, dtype])	Beta distribution.
binomial(n, p[, size, dtype])	Binomial distribution.
bytes(length)	Returns random bytes.
chisquare(df[, size, dtype])	Chi-square distribution.
choice(a[, size, replace, p])	Returns an array of random values from a given 1-D array.
dirichlet(alpha[, size, dtype])	Dirichlet distribution.
exponential(scale[, size, dtype])	Exponential distribution.
f(dfnum, dfden[, size, dtype])	F distribution.
gamma(shape[, scale, size, dtype])	Gamma distribution.
geometric(p[, size, dtype])	Geometric distribution.
gumbel([loc, scale, size, dtype])	Returns an array of samples drawn from a Gumbel distribution.
hypergeometric(ngood, nbad, nsample[, size, ...])	hypergeometric distribution.
laplace([loc, scale, size, dtype])	Laplace distribution.
logistic([loc, scale, size, dtype])	Logistic distribution.
lognormal([mean, sigma, size, dtype])	Returns an array of samples drawn from a log normal distribution.
logseries(p[, size, dtype])	Log series distribution.
multinomial(n, pvals[, size])	Returns an array from multinomial distribution.
multivariate_normal(mean, cov[, size, ...])	Multivariate normal distribution.
negative_binomial(n, p[, size, dtype])	Negative binomial distribution.
noncentral_chisquare(df, nonc[, size, dtype])	Noncentral chisquare distribution.
noncentral_f(dfnum, dfden, nonc[, size, dtype])	Noncentral F distribution.
normal([loc, scale, size, dtype])	Returns an array of normally distributed samples.
pareto(a[, size, dtype])	Pareto II or Lomax distribution.
permutation(a)	Returns a permuted range or a permutation of an array.
poisson([lam, size, dtype])	Poisson distribution.

continues on next page

Table 2 – continued from previous page

power(a[, size, dtype])	Power distribution.
rand(*size, **kwarg)	Returns an array of uniform random values over the interval `[0, 1)`.
randint(low[, high, size, dtype])	Returns a scalar or an array of integer values over `[low, high)`.
randn(*size, **kwarg)	Returns an array of standard normal random values.
random([size, dtype])	Returns an array of random values over the interval `[0, 1)`.
random_integers(low[, high, size])	Return a scalar or an array of integer values over `[low, high]`
random_sample([size, dtype])	Returns an array of random values over the interval `[0, 1)`.
ranf([size, dtype])	Returns an array of random values over the interval `[0, 1)`.
rayleigh([scale, size, dtype])	Rayleigh distribution.
sample([size, dtype])	Returns an array of random values over the interval `[0, 1)`.
seed([seed])	Resets the state of the random number generator with a seed.
shuffle(a)	Shuffles an array.
standard_cauchy([size, dtype])	Standard cauchy distribution.
standard_exponential([size, dtype])	Standard exponential distribution.
standard_gamma(shape[, size, dtype])	Standard gamma distribution.
standard_normal([size, dtype])	Returns an array of samples drawn from the standard normal distribution.
standard_t(df[, size, dtype])	Standard Student's t distribution.
triangular(left, mode, right[, size, dtype])	Triangular distribution.
uniform([low, high, size, dtype])	Returns an array of uniformly-distributed samples over an interval.
vonmises(mu, kappa[, size, dtype])	von Mises distribution.
wald(mean, scale[, size, dtype])	Wald distribution.
weibull(a[, size, dtype])	weibull distribution.
zipf(a[, size, dtype])	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 α.
- **b** (*float*) – Parameter of the beta distribution β.
- **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 α.
- **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

$$f(x) = \frac{1}{\beta} \exp(-\frac{x}{\beta}).$$

Parameters

- **scale** (*float or array_like of floats*) – The scale parameter β.
- **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 θ
- **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 μ is `loc` and η is `scale`.

Parameters

- **loc** (*float*) – The location of the mode μ.
- **scale** (*float*) – The scale parameter η.
- **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 μ.
- **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 μ.
- **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 μ.
- **cov** (*2-D array_like, of shape (N, N)*) – Covariance matrix Σ 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 λ.
- **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

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 α.
- **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 λ.

- **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 σ.
- **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

$$f(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1+\frac{x^2}{\nu}\right)^{-(\frac{\nu+1}{2})}.$$

Parameters

- **df** (*float or array_like of floats*) – Degree of freedom ν.
- **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 μ.
- **kappa** (*float*) – Parameter of the von Mises distribution κ.
- **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 μ.
- **scale** (*float*) – Parameter of the wald distribution λ.
- **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 ζ 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.

get_random_state()	Gets the state of the random number generator for the current device.
set_random_state(rs)	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

unique(ar[, return_index, return_inverse, ...])	Find the unique elements of an array.

Boolean operations

in1d(ar1, ar2[, assume_unique, invert])	Tests whether each element of a 1-D array is also present in a second array.
intersect1d(arr1, arr2[, assume_unique, ...])	Find the intersection of two arrays.
isin(element, test_elements[, ...])	Calculates element in `test_elements`, broadcasting over `element` only.
setdiff1d(ar1, ar2[, assume_unique])	Find the set difference of two arrays.
setxor1d(ar1, ar2[, assume_unique])	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 assumend 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 comparision
- **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 flattend if they are not already 1-D.
- **ar2** (cupy.ndarray) – Input arrays. They are flattend 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

sort(a[, axis, kind])	Returns a sorted copy of an array with a stable sorting algorithm.
lexsort(keys)	Perform an indirect sort using an array of keys.
argsort(a[, axis, kind])	Returns the indices that would sort an array with a stable sorting.
msort(a)	Returns a copy of an array sorted along the first axis.
sort_complex(a)	Sort a complex array using the real part first, then the imaginary part.
partition(a, kth[, axis])	Returns a partitioned copy of an array.
argpartition(a, kth[, axis])	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

argmax(a[, axis, dtype, out, keepdims])	Returns the indices of the maximum along an axis.
nanargmax(a[, axis, dtype, out, keepdims])	Return the indices of the maximum values in the specified axis ignoring NaNs.
argmin(a[, axis, dtype, out, keepdims])	Returns the indices of the minimum along an axis.
nanargmin(a[, axis, dtype, out, keepdims])	Return the indices of the minimum values in the specified axis ignoring NaNs.
argwhere(a)	Return the indices of the elements that are non-zero.
nonzero(a)	Return the indices of the elements that are non-zero.
flatnonzero(a)	Return indices that are non-zero in the flattened version of a.
where(condition[, x, y])	Return elements, either from x or y, depending on condition.
searchsorted(a, v[, side, sorter])	Finds indices where elements should be inserted to maintain order.
extract(condition, a)	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) dimantional 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

count_nonzero(a[, axis])	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

amin(a[, axis, out, keepdims])	Returns the minimum of an array or the minimum along an axis.
amax(a[, axis, out, keepdims])	Returns the maximum of an array or the maximum along an axis.
nanmin(a[, axis, out, keepdims])	Returns the minimum of an array along an axis ignoring NaN.
nanmax(a[, axis, out, keepdims])	Returns the maximum of an array along an axis ignoring NaN.
ptp(a[, axis, out, keepdims])	Returns the range of values (maximum - minimum) along an axis.
percentile(a, q[, axis, out, ...])	Computes the q-th percentile of the data along the specified axis.
quantile(a, q[, axis, out, overwrite_input, ...])	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

median(a[, axis, out, overwrite_input, keepdims])	Compute the median along the specified axis.
average(a[, axis, weights, returned, keepdims])	Returns the weighted average along an axis.
mean(a[, axis, dtype, out, keepdims])	Returns the arithmetic mean along an axis.
std(a[, axis, dtype, out, ddof, keepdims])	Returns the standard deviation along an axis.
var(a[, axis, dtype, out, ddof, keepdims])	Returns the variance along an axis.
nanmedian(a[, axis, out, overwrite_input, ...])	Compute the median along the specified axis, while ignoring NaNs.
nanmean(a[, axis, dtype, out, keepdims])	Returns the arithmetic mean along an axis ignoring NaN values.
nanstd(a[, axis, dtype, out, ddof, keepdims])	Returns the standard deviation along an axis ignoring NaN values.
nanvar(a[, axis, dtype, out, ddof, keepdims])	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

corrcoef(a[, y, rowvar, bias, ddof, dtype])	Returns the Pearson product-moment correlation coefficients of an array.
correlate(a, v[, mode])	Returns the cross-correlation of two 1-dimensional sequences.
cov(a[, y, rowvar, bias, ddof, fweights, ...])	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

histogram(x[, bins, range, weights, density])	Computes the histogram of a set of data.
histogram2d(x, y[, bins, range, weights, ...])	Compute the bi-dimensional histogram of two data samples.
histogramdd(sample[, bins, range, weights, ...])	Compute the multidimensional histogram of some data.
bincount(x[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
digitize(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, `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.

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, …)*. 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*.

assert_array_almost_equal(x, y[, decimal, ...])	Raises an AssertionError if objects are not equal up to desired precision.
assert_allclose(actual, desired[, rtol, ...])	Raises an AssertionError if objects are not equal up to desired tolerance.
assert_array_almost_equal_nulp(x, y[, nulp])	Compare two arrays relatively to their spacing.
assert_array_max_ulp(a, b[, maxulp, dtype])	Check that all items of arrays differ in at most N Units in the Last Place.
assert_array_equal(x, y[, err_msg, verbose, ...])	Raises an AssertionError if two array_like objects are not equal.
assert_array_less(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

assert_array_list_equal(xlist, ylist[, ...])	Compares lists of arrays pairwise with `assert_array_equal`.

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.

numpy_cupy_allclose([rtol, atol, err_msg, ...])	Decorator that checks NumPy results and CuPy ones are close.
numpy_cupy_array_almost_equal([decimal, ...])	Decorator that checks NumPy results and CuPy ones are almost equal.
numpy_cupy_array_almost_equal_nulp([nulp, ...])	Decorator that checks results of NumPy and CuPy are equal w.r.t.
numpy_cupy_array_max_ulp([maxulp, dtype, ...])	Decorator that checks results of NumPy and CuPy ones are equal w.r.t.
numpy_cupy_array_equal([err_msg, verbose, ...])	Decorator that checks NumPy results and CuPy ones are equal.
numpy_cupy_array_list_equal([err_msg, ...])	Decorator that checks the resulting lists of NumPy and CuPy's one are equal.
numpy_cupy_array_less([err_msg, verbose, ...])	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).

for_dtypes(dtypes[, name])	Decorator for parameterized dtype test.
for_all_dtypes([name, no_float16, no_bool, ...])	Decorator that checks the fixture with all dtypes.
for_float_dtypes([name, no_float16])	Decorator that checks the fixture with float dtypes.
for_signed_dtypes([name])	Decorator that checks the fixture with signed dtypes.
for_unsigned_dtypes([name])	Decorator that checks the fixture with unsinged dtypes.
for_int_dtypes([name, no_bool])	Decorator that checks the fixture with integer and optionally bool dtypes.
for_complex_dtypes([name])	Decorator that checks the fixture with complex dtypes.
for_dtypes_combination(types[, names, full])	Decorator that checks the fixture with a product set of dtypes.
for_all_dtypes_combination([names, ...])	Decorator that checks the fixture with a product set of all dtypes.
for_signed_dtypes_combination([names, full])	Decorator for parameterized test w.r.t.
for_unsigned_dtypes_combination([names, full])	Decorator for parameterized test w.r.t.
for_int_dtypes_combination([names, no_bool, ...])	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 unsinged 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 dtypes*) – 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.

for_orders(orders[, name])	Decorator to parameterize tests with order.
for_CF_orders([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

bartlett(M)	Returns the Bartlett window.
blackman(M)	Returns the Blackman window.
hamming(M)	Returns the Hamming window.
hanning(M)	Returns the Hanning window.
kaiser(M, beta)	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

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) + 0.08\cos\left(\frac{4\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

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

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

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

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

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

$$-\frac{M-1}{2} \le n \le \frac{M-1}{2}$$

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)

fft(x[, n, axis, norm, overwrite_x, plan])	Compute the one-dimensional FFT.
ifft(x[, n, axis, norm, overwrite_x, plan])	Compute the one-dimensional inverse FFT.
fft2(x[, s, axes, norm, overwrite_x, plan])	Compute the two-dimensional FFT.
ifft2(x[, s, axes, norm, overwrite_x, plan])	Compute the two-dimensional inverse FFT.
fftn(x[, s, axes, norm, overwrite_x, plan])	Compute the N-dimensional FFT.
ifftn(x[, s, axes, norm, overwrite_x, plan])	Compute the N-dimensional inverse FFT.
rfft(x[, n, axis, norm, overwrite_x, plan])	Compute the one-dimensional FFT for real input.
irfft(x[, n, axis, norm, overwrite_x, plan])	Compute the one-dimensional inverse FFT for real input.
rfft2(x[, s, axes, norm, overwrite_x, plan])	Compute the two-dimensional FFT for real input.
irfft2(x[, s, axes, norm, overwrite_x, plan])	Compute the two-dimensional inverse FFT for real input.
rfftn(x[, s, axes, norm, overwrite_x, plan])	Compute the N-dimensional FFT for real input.
irfftn(x[, s, axes, norm, overwrite_x, plan])	Compute the N-dimensional inverse FFT for real input.
hfft(x[, n, axis, norm, overwrite_x, plan])	Compute the FFT of a signal that has Hermitian symmetry.
ihfft(x[, n, axis, norm, overwrite_x, plan])	Compute the FFT of a signal that has Hermitian symmetry.
hfft2(x[, s, axes, norm, overwrite_x, plan])	Compute the FFT of a two-dimensional signal that has Hermitian symmetry.
ihfft2(x[, s, axes, norm, overwrite_x, plan])	Compute the Inverse FFT of a two-dimensional signal that has Hermitian symmetry.
hfftn(x[, s, axes, norm, overwrite_x, plan])	Compute the FFT of a N-dimensional signal that has Hermitian symmetry.
ihfftn(x[, s, axes, norm, overwrite_x, plan])	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.ifft

`cupyx.scipy.fft.`**`ifft`**(*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.ifft()`

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.ifft2

cupyx.scipy.fft.**ifft2**(*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)

dct(x[, type, n, axis, norm, overwrite_x])	Return the Discrete Cosine Transform of an array, x.
idct(x[, type, n, axis, norm, overwrite_x])	Return the Inverse Discrete Cosine Transform of an array, x.
dctn(x[, type, s, axes, norm, overwrite_x])	Compute a multidimensional Discrete Cosine Transform.
idctn(x[, type, s, axes, norm, overwrite_x])	Compute a multidimensional Discrete Cosine Transform.
dst(x[, type, n, axis, norm, overwrite_x])	Return the Discrete Sine Transform of an array, x.
idst(x[, type, n, axis, norm, overwrite_x])	Return the Inverse Discrete Sine Transform of an array, x.
dstn(x[, type, s, axes, norm, overwrite_x])	Compute a multidimensional Discrete Sine Transform.
idstn(x[, type, s, axes, norm, overwrite_x])	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[1]. 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".

[1] Wikipedia, "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[1]. 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 ith dimension is padded with zeros. If `s[i] < x.shape[i]`, the ith 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

[1] Wikipedia, "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 ith dimension is padded with zeros. If `s[i] < x.shape[i]`, the ith 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 ith dimension is padded with zeros. If `s[i] < x.shape[i]`, the ith 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 ith dimension is padded with zeros. If `s[i] < x.shape[i]`, the ith 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

fht(a, dln, mu[, offset, bias])	Compute the fast Hankel transform.
ifht(A, dln, mu[, offset, bias])	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[1],[2].

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.

[1] Talman J. D., 1978, J. Comp. Phys., 29, 35
[2] 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

fftshift(x[, axes])	Shift the zero-frequency component to the center of the spectrum.
ifftshift(x[, axes])	The inverse of *fftshift()*.
fftfreq(n[, d])	Return the FFT sample frequencies.
rfftfreq(n[, d])	Return the FFT sample frequencies for real input.
next_fast_len(target[, real])	Find the next fast size to `fft`.

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)

fft(x[, n, axis, overwrite_x, plan])	Compute the one-dimensional FFT.
ifft(x[, n, axis, overwrite_x, plan])	Compute the one-dimensional inverse FFT.
fft2(x[, shape, axes, overwrite_x, plan])	Compute the two-dimensional FFT.
ifft2(x[, shape, axes, overwrite_x, plan])	Compute the two-dimensional inverse FFT.
fftn(x[, shape, axes, overwrite_x, plan])	Compute the N-dimensional FFT.
ifftn(x[, shape, axes, overwrite_x, plan])	Compute the N-dimensional inverse FFT.
rfft(x[, n, axis, overwrite_x, plan])	Compute the one-dimensional FFT for real input.
irfft(x[, n, axis, overwrite_x])	Compute the one-dimensional inverse FFT for real input.
get_fft_plan(a[, shape, axes, value_type])	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 accelarate 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

BarycentricInterpolator(xi[, yi, axis])	The interpolating polynomial for a set of points.
KroghInterpolator(xi, yi[, axis])	Interpolating polynomial for a set of points.
barycentric_interpolate(xi, yi, x[, axis])	Convenience function for polynomial interpolation.
krogh_interpolate(xi, yi, x[, der, axis])	Convenience function for polynomial interpolation
pchip_interpolate(xi, yi, x[, der, axis])	Convenience function for pchip interpolation.
CubicHermiteSpline(x, y, dydx[, axis, ...])	Piecewise-cubic interpolator matching values and first derivatives.
PchipInterpolator(x, y[, axis, extrapolate])	PCHIP 1-D monotonic cubic interpolation.
Akima1DInterpolator(x, y[, axis])	Akima interpolator
PPoly(c, x[, extrapolate, axis])	Piecewise polynomial in terms of coefficients and break-points The polynomial between `x[i]` and `x[i + 1]` is written in the local power basis.
BPoly(c, x[, extrapolate, axis])	Piecewise polynomial in terms of coefficients and break-points.
CubicSpline(x, y[, axis, bc_type, extrapolate])	Cubic spline data interpolator.
interp1d(x, y[, kind, axis, copy, ...])	Interpolate a 1-D function.

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 jth 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.[1]

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:

[1] 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 k2 = 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 k2 = 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[1].

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[2].

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

[1] 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.
[2] see, e.g., C. Moler, Numerical Computing with Matlab, 2004. 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 k2 = 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 k2 = 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[1].

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

[1] 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 k2 = 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 k2 = 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) for m 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 k2 = 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 k2 = 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) for a in range(k+1)),
```

where `k` is the degree of the polynomial, and:

```
b(a, k; x) = 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[1][2][3].

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

$$B(x) = 1 \times b_{0,2}(x) + 2 \times b_{1,2}(x) + 3 \times b_{2,2}(x)$$
$$= 1 \times (1-x)^2 + 2 \times 2x(1-x) + 3 \times x^2$$

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.

[1] https://en.wikipedia.org/wiki/Bernstein_polynomial
[2] Kenneth I. Joy, Bernstein polynomials, http://www.idav.ucdavis.edu/education/CAGDNotes/Bernstein-Polynomials.pdf
[3] 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, ...)*) – 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** (PPoly) – 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`

cupyx.scipy.interpolate.CubicSpline

`class` cupyx.scipy.interpolate.`CubicSpline`(*x*, *y*, *axis=0*, *bc_type='not-a-knot'*, *extrapolate=None*)
Cubic spline data interpolator.

Interpolate data with a piecewise cubic polynomial which is twice continuously differentiable. The result is represented as a *PPoly* instance with breakpoints matching the given data.

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, shape (n,)*) – 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.
- **axis** (*int, optional*) – Axis along which *y* is assumed to be varying. Meaning that for `x[i]` the corresponding values are `np.take(y, i, axis=axis)`. Default is 0.
- **bc_type** (*string or 2-tuple, optional*) – Boundary condition type. Two additional equations, given by the boundary conditions, are required to determine all coefficients of polynomials on each segment.

If *bc_type* is a string, then the specified condition will be applied at both ends of a spline. Available conditions are:

- 'not-a-knot' (default): The first and second segment at a curve end are the same polynomial. It is a good default when there is no information on boundary conditions.
- 'periodic': The interpolated functions is assumed to be periodic of period `x[-1] - x[0]`. The first and last value of *y* must be identical: `y[0] == y[-1]`. This boundary condition will result in `y'[0] == y'[-1]` and `y''[0] == y''[-1]`.
- 'clamped': The first derivative at curves ends are zero. Assuming a 1D *y*, `bc_type=((1, 0.0), (1, 0.0))` is the same condition.
- 'natural': The second derivative at curve ends are zero. Assuming a 1D *y*, `bc_type=((2, 0.0), (2, 0.0))` is the same condition.

If *bc_type* is a 2-tuple, the first and the second value will be applied at the curve start and end respectively. The tuple values can be one of the previously mentioned strings (except 'periodic') or a tuple *(order, deriv_values)* allowing to specify arbitrary derivatives at curve ends:

- `order`: the derivative order, 1 or 2.
- `deriv_value`: array_like containing derivative values, shape must be the same as *y*, excluding `axis` dimension. For example, if *y* is 1-D, then *deriv_value* must be a scalar. If *y* is 3-D with the shape (n0, n1, n2) and axis=2, then *deriv_value* must be 2-D and have the shape (n0, n1).

- **`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), `extrapolate` is set to 'periodic' for `bc_type='periodic'` and to True otherwise.

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:

`scipy.interpolate.CubicSpline`

Notes

Parameters *bc_type* and `extrapolate` work independently, i.e. the former controls only construction of a spline, and the latter only evaluation.

When a boundary condition is 'not-a-knot' and n = 2, it is replaced by a condition that the first derivative is equal to the linear interpolant slope. When both boundary conditions are 'not-a-knot' and n = 3, the solution is sought as a parabola passing through given points.

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 k2 = 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 k2 = 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.interp1d

class cupyx.scipy.interpolate.**interp1d**(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=None*, *fill_value=nan*, *assume_sorted=False*)

Interpolate a 1-D function.

x and *y* are arrays of values used to approximate some function f: `y = f(x)`. This class returns a function whose call method uses interpolation to find the value of new points.

Parameters

- **x** (*(npoints,) array_like*) – A 1-D array of real values.
- **y** (*(..., npoints, ...) array_like*) – A N-D array of real values. The length of *y* along the interpolation axis must be equal to the length of *x*. Use the `axis` parameter to select correct axis. Unlike other interpolators, the default interpolation axis is the last axis of *y*.

- **kind** (*str or int, optional*) – Specifies the kind of interpolation as a string or as an integer specifying the order of the spline interpolator to use. The string has to be one of 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', or 'next'. 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order; 'previous' and 'next' simply return the previous or next value of the point; 'nearest-up' and 'nearest' differ when interpolating half-integers (e.g. 0.5, 1.5) in that 'nearest-up' rounds up and 'nearest' rounds down. Default is 'linear'.
- **axis** (*int, optional*) – Axis in the `y` array corresponding to the x-coordinate values. Unlike other interpolators, defaults to `axis=-1`.
- **copy** (*bool, optional*) – If `True`, the class makes internal copies of x and y. If `False`, references to `x` and `y` are used if possible. The default is to copy.
- **bounds_error** (*bool, optional*) – If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of x (where extrapolation is necessary). If False, out of bounds values are assigned *fill_value*. By default, an error is raised unless `fill_value="extrapolate"`.
- **fill_value** (*array-like or (array-like, array_like) or "extrapolate", optional*) –
  - if a ndarray (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN. The array-like must broadcast properly to the dimensions of the non-interpolation axes.
  - If a two-element tuple, then the first element is used as a fill value for `x_new < x[0]` and the second element is used for `x_new > x[-1]`. Anything that is not a 2-element tuple (e.g., list or ndarray, regardless of shape) is taken to be a single array-like argument meant to be used for both bounds as `below, above = fill_value, fill_value`. Using a two-element tuple or ndarray requires `bounds_error=False`.
  - If "extrapolate", then points outside the data range will be extrapolated.
- **assume_sorted** (*bool, optional*) – If False, values of *x* can be in any order and they are sorted first. If True, *x* has to be an array of monotonically increasing values.

See also:

`scipy.interpolate.interp1d`

Notes

Calling *interp1d* with NaNs present in input values results in undefined behaviour.

Input values *x* and *y* must be convertible to *float* values like *int* or *float*.

If the values in *x* are not unique, the resulting behavior is undefined and specific to the choice of *kind*, i.e., changing *kind* will change the behavior for duplicates.

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*.

__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

fill_value

The fill value.

1-D Splines

BSpline(t, c, k[, extrapolate, axis])	Univariate spline in the B-spline basis.
make_interp_spline(x, y[, k, t, bc_type, ...])	Compute the (coefficients of) interpolating B-spline.
make_lsq_spline(x, y, t[, k, w, axis, ...])	Construct a BSpline via an LSQ (Least SQuared) fit.
splder(tck[, n])	Compute the spline representation of the derivative of a given spline
splantider(tck[, n])	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 (>=n, ...)*) – spline coefficients
- **k** (*int*) – B-spline degree
- **extrapolate** (*bool or 'periodic', optional*) – whether to extrapolate beyond the base interval, `t[k] .. 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

$$B_{i,0}(x) = 1, \text{if } t_i \le x < t_{i+1}, \text{otherwise } 0,$$
$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)$$

Implementation details

- At least `k+1` coefficients are required for a spline of degree *k*, so that `n >= 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] <= x <= t[n]`.
- Based on[1] and[2]

See also:

scipy.interpolate.BSpline

[1] Tom Lyche and Knut Morken, Spline methods, http://www.uio.no/studier/emner/matnat/ifi/INF-MAT5340/v05/undervisningsmateriale/
[2] 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 | t[0], ..., 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] .. 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 `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*, *extrapolate=True*, *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*, *extrapolate=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] .. 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 `len(deriv_l) + len(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.make_lsq_spline

cupyx.scipy.interpolate.**make_lsq_spline**(*x*, *y*, *t*, *k=3*, *w=None*, *axis=0*, *check_finite=True*, ***, *method='qr'*)

Construct a BSpline via an LSQ (Least SQuared) fit.

The result is a linear combination

$$S(x) = \sum_j c_j B_j(x; t)$$

of the B-spline basis elements, $B_j(x; t)$, which minimizes

$$\sum_j \left(w_j \times (S(x_j) - y_j)\right)^2$$

Parameters

- **x** (*array_like, shape (m,)*) – Abscissas.
- **y** (*array_like, shape (m, ...)*) – Ordinates.

- **t** (*array_like, shape (n + k + 1,).*) – Knots. Knots and data points must satisfy Schoenberg-Whitney conditions.
- **k** (*int, optional*) – B-spline degree. Default is cubic, `k = 3`.
- **w** (*array_like, shape (m,), optional*) – Weights for spline fitting. Must be positive. If `None`, then weights are all equal. Default is `None`.
- **axis** (*int, optional*) – Interpolation axis. Default is zero.
- **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` with knots `t`.

See also:

`scipy.interpolate.make_lsq_spline`

BSpline
base class representing the B-spline objects

make_interp_spline
a similar factory function for interpolating splines

Notes

The number of data points must be larger than the spline degree `k`. Knots `t` must satisfy the Schoenberg-Whitney conditions, i.e., there must be a subset of data points `x[j]` such that `t[j] < x[j] < t[j+k+1]`, for `j=0, 1, ...,n-k-2`.

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 k2=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 k2=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`

Smoothing Splines

UnivariateSpline(x, y[, w, bbox, k, s, ext])	1-D smoothing spline fit to a given set of data points.
InterpolatedUnivariateSpline(x, y[, w, ...])	1-D interpolating spline for a given set of data points.
LSQUnivariateSpline(x, y, t[, w, bbox, k, ext])	1-D spline with explicit internal knots.

cupyx.scipy.interpolate.UnivariateSpline

class cupyx.scipy.interpolate.**UnivariateSpline**(*x*, *y*, *w=None*, *bbox=[None, None]*, *k=3*, *s=None*, *ext=0*)

1-D smoothing spline fit to a given set of data points.

Fits a spline y = spl(x) of degree *k* to the provided *x*, *y* data. *s* specifies the number of knots by specifying a smoothing condition.

Parameters

- **x** (*(N,) array_like*) – 1-D array of independent input data. Must be increasing; must be strictly increasing if *s* is 0.
- **y** (*(N,) array_like*) – 1-D array of dependent input data, of the same length as *x*.
- **w** (*(N,) array_like, optional*) – Weights for spline fitting. Must be positive. If *w* is None, weights are all 1. Default is None.
- **bbox** (*(2,) array_like, optional*) – 2-sequence specifying the boundary of the approximation interval. If *bbox* is None, `bbox=[x[0], x[-1]]`. Default is None.
- **k** (*int, optional*) – Degree of the smoothing spline. `k = 3` is a cubic spline. Default is 3.
- **s** (*float or None, optional*) – Positive smoothing factor used to choose the number of knots. Number of knots will be increased until the smoothing condition is satisfied:

```
sum((w[i] * (y[i]-spl(x[i])))**2, axis=0) <= s
```

However, because of numerical issues, the actual condition is:

```
abs(sum((w[i] * (y[i]-spl(x[i])))**2, axis=0) - s) < 0.001 * s
```

If *s* is None, *s* will be set as *len(w)* for a smoothing spline that uses all data points. If 0, spline will interpolate through all data points. This is equivalent to *InterpolatedUnivariateSpline*. Default is None. The user can use the *s* to control the tradeoff between closeness and smoothness of fit. Larger *s* means more smoothing while smaller values of *s* indicate less smoothing. Recommended values of *s* depend on the weights, *w*. If the weights represent the inverse of the standard-deviation of *y*, then a good *s* value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m is the number of datapoints in *x*, *y*, and *w*. This means `s = len(w)` should be a good value if `1/w[i]` is an estimate of the standard deviation of `y[i]`.

- **ext** (*int or str, optional*) – Controls the extrapolation mode for elements not in the interval defined by the knot sequence.
  - if ext=0 or ‘extrapolate’, return the extrapolated value.
  - if ext=1 or ‘zeros’, return 0
  - if ext=2 or ‘raise’, raise a ValueError
  - if ext=3 or ‘const’, return the boundary value.

  Default is 0.

See also:

scipy.interpolate.UnivariateSpline

Methods

__call__(*x*, *nu=0*, *ext=None*)

Evaluate spline (or its nu-th derivative) at positions x.

Parameters

- **x** (*ndarray*) – A 1-D array of points at which to return the value of the smoothed spline or its derivatives. Note: *x* can be unordered but the evaluation is more efficient if *x* is (partially) ordered.
- **nu** (*int*) – The order of derivative of the spline to compute.
- **ext** (*int*) – Controls the value returned for elements of *x* not in the interval defined by the knot sequence.
  - if ext=0 or 'extrapolate', return the extrapolated value.
  - if ext=1 or 'zeros', return 0
  - if ext=2 or 'raise', raise a ValueError
  - if ext=3 or 'const', return the boundary value.

  The default value is 0, passed from the initialization of UnivariateSpline.

antiderivative(*n=1*)

Construct a new spline representing the antiderivative of this spline.

Parameters
n (*int, optional*) – Order of antiderivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k+n representing the antiderivative of this spline.

Return type
UnivariateSpline

derivative(*n=1*)

Construct a new spline representing the derivative of this spline.

Parameters
n (*int, optional*) – Order of derivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k-n representing the derivative of this spline.

Return type
UnivariateSpline

derivatives(*x*)

Return all derivatives of the spline at the point x.

Parameters
x (*float*) – The point to evaluate the derivatives at.

Returns
der – Derivatives of the orders 0 to k.

Return type
ndarray, shape(k+1,)

get_coeffs()

Return spline coefficients.

get_knots()

Return positions of interior knots of the spline.

Internally, the knot vector contains `2*k` additional boundary knots.

get_residual()

Return weighted sum of squared residuals of the spline approx.

This is equivalent to:

```
sum((w[i] * (y[i]-spl(x[i])))**2, axis=0
```

integral(*a*, *b*)

Return definite integral of the spline between two given points.

Parameters

- **a** (*float*) – Lower limit of integration.
- **b** (*float*) – Upper limit of integration.

Returns

integral – The value of the definite integral of the spline between limits.

Return type

float

set_smoothing_factor(*s*, *t=None*)

Continue spline computation with the given smoothing factor s and with the knots found at the last call.

This routine modifies the spline in place.

__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.InterpolatedUnivariateSpline

class cupyx.scipy.interpolate.**InterpolatedUnivariateSpline**(*x*, *y*, *w=None*, *bbox=[None, None]*, *k=3*, *ext=0*)

1-D interpolating spline for a given set of data points.

Fits a spline y = spl(x) of degree *k* to the provided *x*, *y* data. Spline function passes through all provided points. Equivalent to *UnivariateSpline* with *s* = 0.

Parameters

- **x** (*(N,) array_like*) – Input dimension of data points – must be strictly increasing
- **y** (*(N,) array_like*) – input dimension of data points
- **w** (*(N,) array_like, optional*) – Weights for spline fitting. Must be positive. If None (default), weights are all 1.
- **bbox** (*(2,) array_like, optional*) – 2-sequence specifying the boundary of the approximation interval. If None (default), `bbox=[x[0], x[-1]]`.
- **k** (*int, optional*) – Degree of the smoothing spline. Default is `k = 3`, a cubic spline.
- **ext** (*int or str, optional*) – Controls the extrapolation mode for elements not in the interval defined by the knot sequence.
  - if ext=0 or 'extrapolate', return the extrapolated value.
  - if ext=1 or 'zeros', return 0
  - if ext=2 or 'raise', raise a ValueError
  - if ext=3 of 'const', return the boundary value.

  The default value is 0.

See also:

scipy.interpolate.InterpolatedUnivariateSpline

Methods

__call__(*x*, *nu=0*, *ext=None*)

Evaluate spline (or its nu-th derivative) at positions x.

Parameters

- **x** (ndarray) – A 1-D array of points at which to return the value of the smoothed spline or its derivatives. Note: *x* can be unordered but the evaluation is more efficient if *x* is (partially) ordered.
- **nu** (*int*) – The order of derivative of the spline to compute.
- **ext** (*int*) – Controls the value returned for elements of *x* not in the interval defined by the knot sequence.
  - if ext=0 or 'extrapolate', return the extrapolated value.
  - if ext=1 or 'zeros', return 0
  - if ext=2 or 'raise', raise a ValueError
  - if ext=3 or 'const', return the boundary value.

The default value is 0, passed from the initialization of UnivariateSpline.

antiderivative(*n=1*)

Construct a new spline representing the antiderivative of this spline.

Parameters
n (*int, optional*) – Order of antiderivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k+n representing the antiderivative of this spline.

Return type
UnivariateSpline

derivative(*n=1*)

Construct a new spline representing the derivative of this spline.

Parameters
n (*int, optional*) – Order of derivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k-n representing the derivative of this spline.

Return type
UnivariateSpline

derivatives(*x*)

Return all derivatives of the spline at the point x.

Parameters
x (*float*) – The point to evaluate the derivatives at.

Returns
der – Derivatives of the orders 0 to k.

Return type
ndarray, shape(k+1,)

get_coeffs()

Return spline coefficients.

get_knots()

Return positions of interior knots of the spline.

Internally, the knot vector contains `2*k` additional boundary knots.

get_residual()

Return weighted sum of squared residuals of the spline approx.

This is equivalent to:

```
sum((w[i] * (y[i]-spl(x[i])))**2, axis=0
```

integral(*a*, *b*)

Return definite integral of the spline between two given points.

Parameters

- **a** (*float*) – Lower limit of integration.
- **b** (*float*) – Upper limit of integration.

Returns
integral – The value of the definite integral of the spline between limits.

Return type
float

`set_smoothing_factor`(*s*, *t=None*)

Continue spline computation with the given smoothing factor s and with the knots found at the last call.

This routine modifies the spline in place.

`__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.LSQUnivariateSpline

`class cupyx.scipy.interpolate.LSQUnivariateSpline`(*x*, *y*, *t*, *w=None*, *bbox=[None, None]*, *k=3*, *ext=0*)

1-D spline with explicit internal knots.

Fits a spline y = spl(x) of degree *k* to the provided *x*, *y* data. *t* specifies the internal knots of the spline

Parameters

- **x** (*(N,) array_like*) – Input dimension of data points – must be increasing
- **y** (*(N,) array_like*) – Input dimension of data points
- **t** (*(M,) array_like*) – interior knots of the spline. Must be in ascending order and:

```
bbox[0] < t[0] < ... < t[-1] < bbox[-1]
```

- **w** (*(N,) array_like, optional*) – weights for spline fitting. Must be positive. If None (default), weights are all 1.
- **bbox** (*(2,) array_like, optional*) – 2-sequence specifying the boundary of the approximation interval. If None (default), `bbox = [x[0], x[-1]]`.
- **k** (*int, optional*) – Degree of the smoothing spline. Default is *k* = 3, a cubic spline.
- **ext** (*int or str, optional*) – Controls the extrapolation mode for elements not in the interval defined by the knot sequence.
  - if ext=0 or ‘extrapolate’, return the extrapolated value.
  - if ext=1 or ‘zeros’, return 0

- if ext=2 or 'raise', raise a ValueError
- if ext=3 of 'const', return the boundary value.

The default value is 0.

Raises
ValueError – If the interior knots do not satisfy the Schoenberg-Whitney conditions

See also:

scipy.interpolate.LSQUnivariateSpline

Methods

__call__(*x*, *nu=0*, *ext=None*)

Evaluate spline (or its nu-th derivative) at positions x.

Parameters

- **x** (ndarray) – A 1-D array of points at which to return the value of the smoothed spline or its derivatives. Note: *x* can be unordered but the evaluation is more efficient if *x* is (partially) ordered.
- **nu** (*int*) – The order of derivative of the spline to compute.
- **ext** (*int*) – Controls the value returned for elements of *x* not in the interval defined by the knot sequence.
  - if ext=0 or 'extrapolate', return the extrapolated value.
  - if ext=1 or 'zeros', return 0
  - if ext=2 or 'raise', raise a ValueError
  - if ext=3 or 'const', return the boundary value.

  The default value is 0, passed from the initialization of UnivariateSpline.

antiderivative(*n=1*)

Construct a new spline representing the antiderivative of this spline.

Parameters
n (*int, optional*) – Order of antiderivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k+n representing the antiderivative of this spline.

Return type
UnivariateSpline

derivative(*n=1*)

Construct a new spline representing the derivative of this spline.

Parameters
n (*int, optional*) – Order of derivative to evaluate. Default: 1

Returns
spline – Spline of order k2=k-n representing the derivative of this spline.

Return type
UnivariateSpline

derivatives(*x*)

Return all derivatives of the spline at the point x.

Parameters
x (*float*) – The point to evaluate the derivatives at.

Returns
der – Derivatives of the orders 0 to k.

Return type
ndarray, shape(k+1,)

get_coeffs()

Return spline coefficients.

get_knots()

Return positions of interior knots of the spline.

Internally, the knot vector contains `2*k` additional boundary knots.

get_residual()

Return weighted sum of squared residuals of the spline approx.

This is equivalent to:

```
sum((w[i] * (y[i]-spl(x[i])))**2, axis=0
```

integral(*a*, *b*)

Return definite integral of the spline between two given points.

Parameters

- **a** (*float*) – Lower limit of integration.
- **b** (*float*) – Upper limit of integration.

Returns
integral – The value of the definite integral of the spline between limits.

Return type
float

set_smoothing_factor(*s*, *t=None*)

Continue spline computation with the given smoothing factor s and with the knots found at the last call.

This routine modifies the spline in place.

__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.

Multivariate interpolation

Unstructured data:

LinearNDInterpolator(points, values[, ...])	Piecewise linear interpolant in N > 1 dimensions.
CloughTocher2DInterpolator(points, values[, ...])	CloughTocher2DInterpolator(points, values, tol=1e-6).
RBFInterpolator(y, d[, neighbors, ...])	Radial basis function (RBF) interpolation in N dimensions.

cupyx.scipy.interpolate.LinearNDInterpolator

class cupyx.scipy.interpolate.**LinearNDInterpolator**(*points*, *values*, *fill_value=cupy.nan*, *rescale=False*)

Piecewise linear interpolant in N > 1 dimensions.

Parameters

- **points** (ndarray of floats, shape (npoints, ndims); or Delaunay) – 2-D array of data point coordinates, or a precomputed Delaunay triangulation.
- **values** (*ndarray of float or complex, shape (npoints, ...), optional*) – N-D array of data values at *points*. The length of *values* along the first axis must be equal to the length of *points*. Unlike some interpolators, the interpolation axis cannot be changed.
- **fill_value** (*float, optional*) – Value used to fill in for requested points outside of the convex hull of the input points. If not provided, then the default is nan.
- **rescale** (*bool, optional*) – Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

Notes

The interpolant is constructed by triangulating the input data with GDel2D[1], and on each triangle performing linear barycentric interpolation.

Note: For data on a regular grid use *interpn* instead.

[1] A GPU accelerated algorithm for 3D Delaunay triangulation (2014). Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, Tiow-Seng Tan. Proc. 18th ACM SIGGRAPH Symp. Interactive 3D Graphics and Games, 47-55.

Examples

We can interpolate values on a 2D plane:

```
>>> from scipy.interpolate import LinearNDInterpolator
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> rng = np.random.default_rng()
>>> x = rng.random(10) - 0.5
>>> y = rng.random(10) - 0.5
>>> z = np.hypot(x, y)
>>> X = np.linspace(min(x), max(x))
>>> Y = np.linspace(min(y), max(y))
>>> X, Y = np.meshgrid(X, Y)  # 2D grid for interpolation
>>> interp = LinearNDInterpolator(list(zip(x, y)), z)
>>> Z = interp(X, Y)
>>> plt.pcolormesh(X, Y, Z, shading='auto')
>>> plt.plot(x, y, "ok", label="input point")
>>> plt.legend()
>>> plt.colorbar()
>>> plt.axis("equal")
>>> plt.show()
```

See also:

griddata
Interpolate unstructured D-D data.

NearestNDInterpolator
Nearest-neighbor interpolation in N dimensions.

CloughTocher2DInterpolator
Piecewise cubic, C1 smooth, curvature-minimizing interpolant in 2D.

interpn
Interpolation on a regular grid or rectilinear grid.

RegularGridInterpolator
Interpolation on a regular or rectilinear grid in arbitrary dimensions (*interpn* wraps this class).

References

Methods

__call__(**args*)
interpolator(xi)

Evaluate interpolator at given points.

Parameters

- **x1** (*array-like of float*) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`

- **x2** (*array-like of float*) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`
- **xn** (`...`) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`

__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.CloughTocher2DInterpolator

class cupyx.scipy.interpolate.**CloughTocher2DInterpolator**(*points*, *values*, *fill_value=nan*, *tol=1e-06*, *maxiter=400*, *rescale=False*)

CloughTocher2DInterpolator(points, values, tol=1e-6).

Piecewise cubic, C1 smooth, curvature-minimizing interpolator in 2D.

Parameters

- **points** (ndarray *of floats, shape (npoints, ndims); or* Delaunay) – 2-D array of data point coordinates, or a precomputed Delaunay triangulation.
- **values** (ndarray *of float or complex, shape (npoints, ...)*) – N-D array of data values at *points*. The length of *values* along the first axis must be equal to the length of *points*. Unlike some interpolators, the interpolation axis cannot be changed.
- **fill_value** (*float, optional*) – Value used to fill in for requested points outside of the convex hull of the input points. If not provided, then the default is `nan`.
- **tol** (*float, optional*) – Absolute/relative tolerance for gradient estimation.
- **maxiter** (*int, optional*) – Maximum number of iterations in gradient estimation.
- **rescale** (*bool, optional*) – Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

Notes

The interpolant is constructed by triangulating the input data with GDel2D[1], and constructing a piecewise cubic interpolating Bezier polynomial on each triangle, using a Clough-Tocher scheme [CT]. The interpolant is guaranteed to be continuously differentiable.

The gradients of the interpolant are chosen so that the curvature of the interpolating surface is approximatively minimized. The gradients necessary for this are estimated using the global algorithm described in [Nielson83] and [Renka84].

Note: For data on a regular grid use *interpn* instead.

Examples

We can interpolate values on a 2D plane:

```
>>> from scipy.interpolate import CloughTocher2DInterpolator
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> rng = np.random.default_rng()
>>> x = rng.random(10) - 0.5
>>> y = rng.random(10) - 0.5
>>> z = np.hypot(x, y)
>>> X = np.linspace(min(x), max(x))
>>> Y = np.linspace(min(y), max(y))
>>> X, Y = np.meshgrid(X, Y)  # 2D grid for interpolation
>>> interp = CloughTocher2DInterpolator(list(zip(x, y)), z)
>>> Z = interp(X, Y)
>>> plt.pcolormesh(X, Y, Z, shading='auto')
>>> plt.plot(x, y, "ok", label="input point")
>>> plt.legend()
>>> plt.colorbar()
>>> plt.axis("equal")
>>> plt.show()
```

See also:

griddata
: Interpolate unstructured D-D data.

LinearNDInterpolator
: Piecewise linear interpolator in N > 1 dimensions.

NearestNDInterpolator
: Nearest-neighbor interpolator in N > 1 dimensions.

interpn
: Interpolation on a regular grid or rectilinear grid.

RegularGridInterpolator
: Interpolator on a regular or rectilinear grid in arbitrary dimensions (*interpn* wraps this class).

[1] A GPU accelerated algorithm for 3D Delaunay triangulation (2014). Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, Tiow-Seng Tan. Proc. 18th ACM SIGGRAPH Symp. Interactive 3D Graphics and Games, 47-55.

References

Methods

__call__(**args*)

interpolator(xi)

Evaluate interpolator at given points.

Parameters

- **x1** (*array-like of float*) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`
- **x2** (*array-like of float*) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`
- **xn** (`...`) – Points where to interpolate data at. x1, x2, … xn can be array-like of float with broadcastable shape. or x1 can be array-like of float with shape `(..., ndim)`

__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.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 λ 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:

interpn(points, values, xi[, method, ...])	Multidimensional interpolation on regular or rectilinear grids.
RegularGridInterpolator(points, values[, ...])	Interpolator 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 (*interpn* 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*)

Interpolator 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, nearest-neighbor, spline 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.
- **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.

The 'slinear'(k=1), 'cubic'(k=3), and 'quintic'(k=5) methods are tensor-product spline interpolators, where k is the spline degree, If any dimension has fewer points than $k + 1$, an error will be raised.

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.

** Choosing a spline method **

Spline methods, "slinear", "cubic" and "quintic" involve solving a large sparse linear system at instantiation time. Alternatively, you may instead use the legacy methods, "slinear_legacy", "cubic_legacy" and "quintic_legacy". These methods allow faster construction but evaluations will be much slower.

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:

`scipy.interpolate.RegularGridInterpolator`

interpn
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

[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*, *, *nu=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", "nearest", "slinear", "cubic", "quintic" and "pchip". Default is the method chosen when the interpolator was created.
- **nu** (*sequence of ints, length ndim, optional*) – If not None, the orders of the derivatives to evaluate. Each entry must be non-negative. Only allowed for methods "slinear", "cubic" and "quintic".

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
NdBSpline(t, c, k, *[, extrapolate])	Tensor product spline object.

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 k2 = 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] - nu[0], . . . , k[n] - 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]), ..., (a[ndim-1], b[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]] x . . . x [a[ndim-1], b[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:

$$p(y, z, ...) = \int_a^b dx\, p(x, y, z, ...)$$

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.

cupyx.scipy.interpolate.NdBSpline

class cupyx.scipy.interpolate.**NdBSpline**(*t*, *c*, *k*, *, *extrapolate=None*)

Tensor product spline object.

The value at point `xp = (x1, x2, ..., xN)` is evaluated as a linear combination of products of one-dimensional b-splines in each of the `N` dimensions:

```
c[i1, i2, ..., iN] * B(x1; i1, t1) * B(x2; i2, t2) * ... * B(xN; iN, tN)
```

Here `B(x; i, t)` is the `i`-th b-spline defined by the knot vector `t` evaluated at `x`.

Parameters

- **t** (*tuple of 1D ndarrays*) – knot vectors in directions 1, 2, … N, `len(t[i]) == n[i] + k + 1`
- **c** (*ndarray, shape (n1, n2, ..., nN, ...)*) – b-spline coefficients
- **k** (*int or length-d tuple of integers*) – spline degrees. A single integer is interpreted as having this degree for all dimensions.
- **extrapolate** (*bool, optional*) – Whether to extrapolate out-of-bounds inputs, or return *nan*. Default is to extrapolate.

Variables

- **t** (*tuple of ndarrays*) – Knots vectors.
- **c** (*ndarray*) – Coefficients of the tensor-produce spline.
- **k** (*tuple of integers*) – Degrees for each dimension.
- **extrapolate** (*bool, optional*) – Whether to extrapolate or return nans for out-of-bounds inputs. Defaults to true.

See also:

BSpline
a one-dimensional B-spline object

NdPPoly
an N-dimensional piecewise tensor product polynomial

Methods

__call__(*xi*, ***, *nu=None*, *extrapolate=None*)

Evaluate the tensor product b-spline at `xi`.

Parameters

- **xi** (*array_like, shape(..., ndim)*) – The coordinates to evaluate the interpolator at. This can be a list or tuple of ndim-dimensional points or an array with the shape (num_points, ndim).
- **nu** (*array_like, optional, shape (ndim,)*) – Orders of derivatives to evaluate. Each must be non-negative. Defaults to the zeroth derivivative.
- **extrapolate** (*bool, optional*) – Whether to exrapolate based on first and last intervals in each dimension, or return *nan*. Default is to `self.extrapolate`.

Returns
values – Interpolated values at `xi`

Return type
ndarray, shape `xi.shape[:-1] + self.c.shape[ndim:]`

classmethod design_matrix(*xvals*, *t*, *k*, *extrapolate=True*)

Construct the design matrix as a CSR format sparse array.

Parameters

- **xvals** (*ndarray, shape(npts, ndim)*) – Data points. `xvals[j, :]` gives the `j`-th data point as an `ndim`-dimensional array.
- **t** (*tuple of 1D ndarrays, length-ndim*) – Knot vectors in directions 1, 2, … ndim,

- **k** (*int*) – B-spline degree.
- **extrapolate** (*bool, optional*) – Whether to extrapolate out-of-bounds values of raise a *ValueError*

Returns
design_matrix – Each row of the design matrix corresponds to a value in *xvals* and contains values of b-spline basis elements which are non-zero at this value.

Return type
a CSR matrix

__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

solve_triangular(a, b[, trans, lower, ...])	Solve the equation a x = b for x, assuming a is a triangular matrix.
tril(m[, k])	Make a copy of a matrix with elements above the k-th diagonal zeroed.
triu(m[, k])	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 kth 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[1]: a [13 / 13] Pade approximant with scaling and squaring.

Simplifications:

- we always use a [13/13] approximate
- no matrix balancing

[1] N. Higham, SIAM J. MATRIX ANAL. APPL. Vol. 26(4), p. 1179 (2005) https://doi.org/10.1137/04061101X

References

Decompositions

lu(a[, permute_l, overwrite_a, check_finite])	LU decomposition.
lu_factor(a[, overwrite_a, check_finite])	LU decomposition.
lu_solve(lu_and_piv, b[, trans, ...])	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, nontermination) 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, nontermination) 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 <= 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, nontermination) 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

block_diag(*arrs)	Create a block diagonal matrix from provided arrays.
circulant(c)	Construct a circulant matrix.
companion(a)	Create a companion matrix.
convolution_matrix(a, n[, mode])	Construct a convolution matrix.
dft(n[, scale])	Discrete Fourier transform matrix.
fiedler(a)	Returns a symmetric Fiedler matrix
fiedler_companion(a)	Returns a Fiedler companion matrix
hadamard(n[, dtype])	Construct an Hadamard matrix.
hankel(c[, r])	Construct a Hankel matrix.
helmert(n[, full])	Create an Helmert matrix of order `n`.
hilbert(n)	Create a Hilbert matrix of order `n`.
kron(a, b)	Kronecker product.
leslie(f, s)	Create a Leslie matrix.
toeplitz(c[, r])	Construct a Toeplitz matrix.
tri(N[, M, k, dtype])	Construct (`N`, `M`) matrix filled with ones at and below the `k`-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] = 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, *optionnal*) – 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 … a[0,-1]*b a[1,0]*b a[1,1]*b … a[1,-1]*b … a[-1,0]*b a[-1,1]*b … 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

convolve(input, weights[, output, mode, ...])	Multi-dimensional convolution.
convolve1d(input, weights[, axis, output, ...])	One-dimensional convolution.
correlate(input, weights[, output, mode, ...])	Multi-dimensional correlate.
correlate1d(input, weights[, axis, output, ...])	One-dimensional correlate.
gaussian_filter(input, sigma[, order, ...])	Multi-dimensional Gaussian filter.
gaussian_filter1d(input, sigma[, axis, ...])	One-dimensional Gaussian filter along the given axis.
gaussian_gradient_magnitude(input, sigma[, ...])	Multi-dimensional gradient magnitude using Gaussian derivatives.
gaussian_laplace(input, sigma[, output, ...])	Multi-dimensional Laplace filter using Gaussian second derivatives.
generic_filter(input, function[, size, ...])	Compute a multi-dimensional filter using the provided raw kernel or reduction kernel.
generic_filter1d(input, function, filter_size)	Compute a 1D filter along the given axis using the provided raw kernel.
generic_gradient_magnitude(input, derivative)	Multi-dimensional gradient magnitude filter using a provided derivative function.
generic_laplace(input, derivative2[, ...])	Multi-dimensional Laplace filter using a provided second derivative function.
laplace(input[, output, mode, cval])	Multi-dimensional Laplace filter based on approximate second derivatives.
maximum_filter(input[, size, footprint, ...])	Multi-dimensional maximum filter.
maximum_filter1d(input, size[, axis, ...])	Compute the maximum filter along a single axis.
median_filter(input[, size, footprint, ...])	Multi-dimensional median filter.
minimum_filter(input[, size, footprint, ...])	Multi-dimensional minimum filter.
minimum_filter1d(input, size[, axis, ...])	Compute the minimum filter along a single axis.
percentile_filter(input, percentile[, size, ...])	Multi-dimensional percentile filter.
prewitt(input[, axis, output, mode, cval])	Compute a Prewitt filter along the given axis.
rank_filter(input, rank[, size, footprint, ...])	Multi-dimensional rank filter.
sobel(input[, axis, output, mode, cval])	Compute a Sobel filter along the given axis.
uniform_filter(input[, size, output, mode, ...])	Multi-dimensional uniform filter.
uniform_filter1d(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 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 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 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 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 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 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 `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 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 `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 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 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 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 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 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

fourier_ellipsoid(input, size[, n, axis, output])	Multidimensional ellipsoid Fourier filter.
fourier_gaussian(input, sigma[, n, axis, output])	Multidimensional Gaussian shift filter.
fourier_shift(input, shift[, n, axis, output])	Multidimensional Fourier shift filter.
fourier_uniform(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

affine_transform(input, matrix[, offset, ...])	Apply an affine transformation.
map_coordinates(input, coordinates[, ...])	Map the input array to new coordinates by interpolation.
rotate(input, angle[, axes, reshape, ...])	Rotate an array.
shift(input, shift[, output, order, mode, ...])	Shift an array.
spline_filter(input[, order, output, mode])	Multidimensional spline filter.
spline_filter1d(input[, order, axis, ...])	Calculate a 1-D spline filter along the given axis.
zoom(input, zoom[, output, order, mode, ...])	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 `'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 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 >= 2 and <= 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

center_of_mass(input[, labels, index])	Calculate the center of mass of the values of an array at labels.
extrema(input[, labels, index])	Calculate the minimums and maximums of the values of an array at labels, along with their positions.
histogram(input, min, max, bins[, labels, index])	Calculate the histogram of the values of an array, optionally at labels.
label(input[, structure, output])	Labels features in an array.
labeled_comprehension(input, labels, index, ...)	Array resulting from applying `func` to each labeled region.
maximum(input[, labels, index])	Calculate the maximum of the values of an array over labeled regions.
maximum_position(input[, labels, index])	Find the positions of the maximums of the values of an array at labels.
mean(input[, labels, index])	Calculates the mean of the values of an n-D image array, optionally
median(input[, labels, index])	Calculate the median of the values of an array over labeled regions.
minimum(input[, labels, index])	Calculate the minimum of the values of an array over labeled regions.
minimum_position(input[, labels, index])	Find the positions of the minimums of the values of an array at labels.
standard_deviation(input[, labels, index])	Calculates the standard deviation of the values of an n-D image array, optionally at specified sub-regions.
sum_labels(input[, labels, index])	Calculates the sum of the values of an n-D image array, optionally
value_indices(arr, *[, ignore_value, ...])	Find indices of each distinct value in given array.
variance(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 enerated 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 determaxed 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 determaxed 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

binary_closing(input[, structure, ...])	Multidimensional binary closing with the given structuring element.
binary_dilation(input[, structure, ...])	Multidimensional binary dilation with the given structuring element.
binary_erosion(input[, structure, ...])	Multidimensional binary erosion with a given structuring element.
binary_fill_holes(input[, structure, ...])	Fill the holes in binary objects.
binary_hit_or_miss(input[, structure1, ...])	Multidimensional binary hit-or-miss transform.
binary_opening(input[, structure, ...])	Multidimensional binary opening with the given structuring element.
binary_propagation(input[, structure, mask, ...])	Multidimensional binary propagation with the given structuring element.
black_tophat(input[, size, footprint, ...])	Multidimensional black tophat filter.
distance_transform_edt(image[, sampling, ...])	Exact Euclidean distance transform.
generate_binary_structure(rank, connectivity)	Generate a binary structure for binary morphological operations.
grey_closing(input[, size, footprint, ...])	Calculates a multi-dimensional greyscale closing.
grey_dilation(input[, size, footprint, ...])	Calculates a greyscale dilation.
grey_erosion(input[, size, footprint, ...])	Calculates a greyscale erosion.
grey_opening(input[, size, footprint, ...])	Calculates a multi-dimensional greyscale opening.
iterate_structure(structure, iterations[, ...])	Iterate a structure by dilating it with itself.
morphological_gradient(input[, size, ...])	Multidimensional morphological gradient.
morphological_laplace(input[, size, ...])	Multidimensional morphological laplace.
white_tophat(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 or tuple or int, 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). If a tuple of integers is provided, a structuring element of the specified shape is used (all elements True). If an integer is provided, the structuring element will have the same size along all axes.

- **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 *or* *tuple* *or* *int*, *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). If a tuple of integers is provided, a structuring element of the specified shape is used (all elements True). If an integer is provided, the structuring element will have the same size along all axes.

- **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.ndarry

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_indicies* 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[2]. 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.

[2] https://www.comp.nus.edu.sg/~tants/pba.html

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.

The parallel banding algorithm implemented here was originally described in[1]. The kernels used here correspond to the revised PBA+ implementation that is described on the author's website[Page 471, 2]. The source code of the author's PBA+ implementation is available at[3].

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],
```

(continues on next page)

[1] 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

[3] https://github.com/orzzzjq/Parallel-Banding-Algorithm-plus

(continued from previous page)

```
        [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]]])
```

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
```

(continues on next page)

(continued from previous page)

```
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

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

convolve(in1, in2[, mode, method])	Convolve two N-dimensional arrays.
correlate(in1, in2[, mode, method])	Cross-correlate two N-dimensional arrays.
fftconvolve(in1, in2[, mode, axes])	Convolve two N-dimensional arrays using FFT.
oaconvolve(in1, in2[, mode, axes])	Convolve two N-dimensional arrays using the overlap-add method.
convolve2d(in1, in2[, mode, boundary, fillvalue])	Convolve two 2-dimensional arrays.
correlate2d(in1, in2[, mode, boundary, ...])	Cross-correlate two 2-dimensional arrays.
sepfir2d(input, hrow, hcol)	Convolve with a 2-D separable FIR filter.
choose_conv_method(in1, in2[, mode])	Find the fastest convolution/correlation method.
correlation_lags(in1_len, in2_len[, mode])	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 of 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 of 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 off 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

gauss_spline(x, n)	Gaussian approximation to B-spline basis function of order n.
cspline1d(signal[, lamb])	Compute cubic spline coefficients for rank-1 array.
qspline1d(signal[, lamb])	Compute quadratic spline coefficients for rank-1 array.
cspline2d(signal[, lamb, precision])	Coefficients for 2-D cubic (3rd order) B-spline.
qspline2d(signal[, lamb, precision])	Coefficients for 2-D quadratic (2nd order) B-spline.
cspline1d_eval(cj, newx[, dx, x0])	Evaluate a cubic spline at the new set of points.
qspline1d_eval(cj, newx[, dx, x0])	Evaluate a quadratic spline at the new set of points.
spline_filter(Iin[, lmbda])	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 >= 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 :

$$\frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{x^2}{2\sigma})$$

See[1],[2] 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 .

[1] 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

[2] http://folk.uio.no/inf3330/scripting/doc/python/SciPy/tutorial/old/node24.html

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.

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 j=0...N-1, with N=len(cj)

Edges are handled using mirror-symmetric boundary conditions.

Parameters

- **cj** (ndarray) – cublic 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:

```
oldx = x0 + j*dx  j=0...N-1, with N=len(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 smooghing fall-off value, default is *5.0*.

Returns
res – filterd input data

Return type
ndarray

Filtering

order_filter(a, domain, rank)	Perform an order filter on an N-D array.
medfilt(volume[, kernel_size])	Perform a median filter on an N-dimensional array.
medfilt2d(input[, kernel_size])	Median filter a 2-dimensional array.
wiener(im[, mysize, noise])	Perform a Wiener filter on an N-dimensional array.
symiirorder1(input, c0, z1[, precision])	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::.
symiirorder2(input, r, omega[, precision])	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::.
lfilter(b, a, x[, axis, zi])	Filter data along one-dimension with an IIR or FIR filter.
lfiltic(b, a, y[, x])	Construct initial conditions for lfilter given input and output vectors.
lfilter_zi(b, a)	Construct initial conditions for lfilter for step response steady-state.
filtfilt(b, a, x[, axis, padtype, padlen, ...])	Apply a digital filter forward and backward to a signal.
savgol_filter(x, window_length, polyorder[, ...])	Apply a Savitzky-Golay filter to an array.
deconvolve(signal, divisor)	Deconvolves `divisor` out of `signal` using inverse filtering.
sosfilt(sos, x[, axis, zi])	Filter data along one dimension using cascaded second-order sections.
sosfilt_zi(sos)	Construct initial conditions for sosfilt for step response steady-state.
sosfiltfilt(sos, x[, axis, padtype, padlen])	A forward-backward digital filter using cascaded second-order sections.
hilbert(x[, N, axis])	Compute the analytic signal, using the Hilbert transform.
hilbert2(x[, N])	Compute the '2-D' analytic signal of *x*
decimate(x, q[, n, ftype, axis, zero_phase])	Downsample the signal after applying an anti-aliasing filter.
detrend(data[, axis, type, bp, overwrite_data])	Remove linear trend along axis from data.
resample(x, num[, t, axis, window, domain])	Resample *x* to *num* samples using Fourier method along the given axis.
resample_poly(x, up, down[, axis, window, ...])	Resample *x* along the given axis using polyphase filtering.
upfirdn(h, x[, up, down, axis, mode, cval])	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:

```
                c0
H(z) = ----------------------
        (1-z1/z) (1 - z1 z)
```

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:

```
                        cs^2
H(z) = ---------------------------------------
       (1 - a2/z - a3/z^2) (1 - a2 z - a3 z^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:

```
a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M]
                      - a[1]*y[n-1] - ... - a[N]*y[n-N]
```

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,
      r(p_i, zi)             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, ..., N)-fibonacci sequence. For more information see[1].

The rational transfer function describing this filter in the z-transform domain is:

```
                     -1              -M
        b[0] + b[1]z  + ... + b[M] z
Y(z) = --------------------------------- X(z)
                     -1              -N
        a[0] + a[1]z  + ... + a[N] z
```

References

cupyx.scipy.signal.lfiltic

cupyx.scipy.signal.**lfiltic**(*b*, *a*, *y*, *x=None*)

Construct initial conditions for lfilter 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 zi which is used by *lfilter* 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 = len(a) - 1`, then `y = {y[-1], y[-2], ..., y[-N]}`. If y is too short, it is padded with zeros.
- **x** (*array_like, optional*) – Initial conditions. If `M = len(b) - 1`, then `x = {x[-1], x[-2], ..., x[-M]}`. If x is not given, its initial conditions are assumed zero. If x is too short, it is padded with zeros.

[1] Sepideh Maleki and Martin Burtscher. 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 `zi = {z_0[-1], z_1[-1], ..., 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[1] 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*.

[1] F. Gustaffson, "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 *lfilter_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.[1]

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.

[1] Wikipedia, "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 `len(x) / num * (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. fftfreq(x.shape[axis])).

If *window* is an array of the same length as *x.shape[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 * 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(up, down) // gcd(up, 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 *up//gcd(up, 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 `dx * down / float(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.-',                    cupy.
↪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[1] (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

[1] P. P. Vaidyanathan, Multirate Systems and Filter Banks, Prentice Hall, 1993.

Filter design

bilinear(b, a[, fs])	Return a digital IIR filter from an analog one using a bilinear transform.
bilinear_zpk(z, p, k, fs)	Return a digital IIR filter from an analog one using a bilinear transform.
findfreqs(num, den, N[, kind])	Find array of frequencies for computing the response of an analog filter.
freqs(b, a[, worN, plot])	Compute frequency response of analog filter.
freqs_zpk(z, p, k[, worN])	Compute frequency response of analog filter.
freqz(b[, a, worN, whole, plot, fs, ...])	Compute the frequency response of a digital filter.
freqz_zpk(z, p, k[, worN, whole, fs])	Compute the frequency response of a digital filter in ZPK form.
sosfreqz(sos[, worN, whole, fs])	Compute the frequency response of a digital filter in SOS format.
firwin(numtaps, cutoff[, width, window, ...])	FIR filter design using the window method.
firwin2(numtaps, freq, gain[, nfreqs, ...])	FIR filter design using the window method.
firls(numtaps, bands, desired[, weight, fs])	FIR filter design using least-squares error minimization.
minimum_phase(h[, method, n_fft])	Convert a linear-phase FIR filter to minimum phase
savgol_coeffs(window_length, polyorder[, ...])	Compute the coefficients for a 1-D Savitzky-Golay FIR filter.
gammatone(freq, ftype[, order, numtaps, fs])	Gammatone filter design.
group_delay(system[, w, whole, fs])	Compute the group delay of a digital filter.
iirdesign(wp, ws, gpass, gstop[, analog, ...])	Complete IIR digital and analog filter design.
iirfilter(N, Wn[, rp, rs, btype, analog, ...])	IIR digital and analog filter design given order and critical points.
kaiser_atten(numtaps, width)	Compute the attenuation of a Kaiser FIR filter.
kaiser_beta(a)	Compute the Kaiser parameter *beta*, given the attenuation *a*.
kaiserord(ripple, width)	Determine the filter window parameters for the Kaiser window method.
unique_roots(p[, tol, rtype])	Determine unique roots and their multiplicities from a list of roots.
residue(b, a[, tol, rtype])	Compute partial-fraction expansion of b(s) / a(s).
residuez(b, a[, tol, rtype])	Compute partial-fraction expansion of b(z) / a(z).
invres(r, p, k[, tol, rtype])	Compute b(s) and a(s) from partial fraction expansion.
invresz(r, p, k[, tol, rtype])	Compute b(z) and a(z) from partial fraction expansion.
BadCoefficients	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:

```
         b[0]*(jw)**M + b[1]*(jw)**(M-1) + ... + b[M]
H(w) = ----------------------------------------------
         a[0]*(jw)**N + a[1]*(jw)**(N-1) + ... + 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:

```
           (jw-z[0]) * (jw-z[1]) * ... * (jw-z[-1])
H(w) = k * ----------------------------------------
           (jw-p[0]) * (jw-p[1]) * ... * (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:

```
            jw                 -jw              -jwM
   jw    B(e  )    b[0] + b[1]e    + ... + b[M]e
H(e  ) = ------ = ----------------------------------
            jw                 -jw              -jwN
         A(e  )    a[0] + a[1]e    + ... + a[N]e
```

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:

```
       B0(z)   B1(z)         B{n-1}(z)
H(z) = ----- * ----- * ... * ---------
       A0(z)   A1(z)         A{n-1}(z)
```

for z = exp(omega*1j), 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 w 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='nuttall')
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 nx2 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[4][5] 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[1] 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 `(length(h) + 1) // 2`.

Return type
array

See also:

`scipy.signal.minimum_phase`

[4] J. S. Lim, Advanced Topics in Signal Processing. Englewood Cliffs, N.J.: Prentice Hall, 1988.
[5] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, "Discrete-Time Signal Processing," 2nd edition. Upper Saddle River, N.J.: Prentice Hall, 1999.
[1] 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[Page 519, 1] or homomorphic[Page 519, 4Page 519, 5] 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:

```
epsilon = 2. * n_stop / 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:

```
n_fft = 2 ** int(np.ceil(np.log2(2 * (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[2] and spectral factorization[3Page 519, 4Page 519, 5]. 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.

[2] 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.

[3] 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[1].

Parameters

- **freq** (*float*) – Center frequency of the filter (expressed in the same units as *fs*).

[1] 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        jw
D(w) = - -- arg H(e)
         dw
```

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[1].

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

[1] Richard G. Lyons, "Understanding Digital Signal Processing, 3rd edition", p. 830.

- 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.

- **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, Schafer, "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:

```
abs(A(w) - D(w)) < 10**(-ripple/20)
```

  for 0 <= w <= 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*:

```
        b(s)     b[0] s**(M) + b[1] s**(M-1) + ... + b[M]
H(s) = ------ = ----------------------------------------
        a(s)     a[0] s**(N) + a[1] s**(N-1) + ... + a[N]
```

then the partial-fraction expansion H(s) is defined as:

```
    r[0]       r[1]             r[-1]
= -------- + -------- + ... + --------- + k(s)
  (s-p[0])   (s-p[1])         (s-p[-1])
```

If there are any repeated roots (closer together than *tol*), then H(s) has terms like:

```
  r[i]      r[i+1]              r[i+n-1]
-------- + ----------- + ... + -----------
(s-p[i])  (s-p[i])**2           (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[1].

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.

[1] 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*:

```
        b(z)     b[0] + b[1] z**(-1) + ... + b[M] z**(-M)
H(z) = ------ = ----------------------------------------
        a(z)     a[0] + a[1] z**(-1) + ... + a[N] z**(-N)
```

then the partial-fraction expansion H(z) is defined as:

```
        r[0]                   r[-1]
= ---------------- + ... + ----------------- + k[0] + k[1]z**(-1) ...
  (1-p[0]z**(-1))          (1-p[-1]z**(-1))
```

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

```
     r[i]              r[i+1]                    r[i+n-1]
-------------- + ------------------ + ... + ------------------
(1-p[i]z**(-1))  (1-p[i]z**(-1))**2          (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*:

```
        b(s)     b[0] s**(M) + b[1] s**(M-1) + ... + b[M]
H(s) = ------ = ----------------------------------------
        a(s)     a[0] s**(N) + a[1] s**(N-1) + ... + a[N]
```

then the partial-fraction expansion H(s) is defined as:

```
    r[0]       r[1]             r[-1]
= -------- + -------- + ... + --------- + k(s)
  (s-p[0])   (s-p[1])         (s-p[-1])
```

If there are any repeated roots (closer together than *tol*), then H(s) has terms like:

```
  r[i]      r[i+1]              r[i+n-1]
-------- + ----------- + ... + -----------
(s-p[i])  (s-p[i])**2          (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*:

```
        b(z)     b[0] + b[1] z**(-1) + ... + b[M] z**(-M)
H(z) = ------ = ----------------------------------------
        a(z)     a[0] + a[1] z**(-1) + ... + a[N] z**(-N)
```

then the partial-fraction expansion H(z) is defined as:

```
        r[0]                   r[-1]
= --------------- + ... + ---------------- + k[0] + k[1]z**(-1) ...
  (1-p[0]z**(-1))         (1-p[-1]z**(-1))
```

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

```
     r[i]              r[i+1]                    r[i+n-1]
-------------- + ------------------ + ... + ------------------
(1-p[i]z**(-1))  (1-p[i]z**(-1))**2          (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

butter(N, Wn[, btype, analog, output, fs])	Butterworth digital and analog filter design.
buttord(wp, ws, gpass, gstop[, analog, fs])	Butterworth filter order selection.
ellip(N, rp, rs, Wn[, btype, analog, output, fs])	Elliptic (Cauer) digital and analog filter design.
ellipord(wp, ws, gpass, gstop[, analog, fs])	Elliptic (Cauer) filter order selection.
cheby1(N, rp, Wn[, btype, analog, output, fs])	Chebyshev type I digital and analog filter design.
cheb1ord(wp, ws, gpass, gstop[, analog, fs])	Chebyshev type I filter order selection.
cheby2(N, rs, Wn[, btype, analog, output, fs])	Chebyshev type II digital and analog filter design.
cheb2ord(wp, ws, gpass, gstop[, analog, fs])	Chebyshev type II filter order selection.
iircomb(w0, Q[, ftype, fs, pass_zero])	Design IIR notching or peaking digital comb filter.
iirnotch(w0, Q[, fs])	Design second-order IIR notch digital filter.
iirpeak(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 >= 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: 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.

  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:

cheb1ord, *cheb1ap*, *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, ...], and the peaks are centered on the midpoints [w0/2, 3*w0/2, 5*w0/2, ...]. If True, the peaks are centered on [0, w0, 2*w0, ...] (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:

`scpy.signal.iirpeak`

References

Sophocles J. Orfanidis, "Introduction To Signal Processing",
Prentice-Hall, 1996

Low-level filter design functions

abcd_normalize([A, B, C, D])	Check state-space matrices and ensure they are 2-D.
band_stop_obj(wp, ind, passb, stopb, gpass, ...)	Band Stop Objective Function for order minimization.
buttap(N)	Return (z,p,k) for analog prototype of Nth-order Butterworth filter.
cheb1ap(N, rp)	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
cheb2ap(N, rs)	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
ellipap(N, rp, rs)	Return (z,p,k) of Nth-order elliptic analog lowpass filter.
lp2bp(b, a[, wo, bw])	Transform a lowpass filter prototype to a bandpass filter.
lp2bp_zpk(z, p, k[, wo, bw])	Transform a lowpass filter prototype to a bandpass filter.
lp2bs(b, a[, wo, bw])	Transform a lowpass filter prototype to a bandstop filter.
lp2bs_zpk(z, p, k[, wo, bw])	Transform a lowpass filter prototype to a bandstop filter.
lp2hp(b, a[, wo])	Transform a lowpass filter prototype to a highpass filter.
lp2hp_zpk(z, p, k[, wo])	Transform a lowpass filter prototype to a highpass filter.
lp2lp(b, a[, wo])	Transform a lowpass filter prototype to a different frequency.
lp2lp_zpk(z, p, k[, wo])	Transform a lowpass filter prototype to a different frequency.
normalize(b, a)	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

$$s \rightarrow \frac{s^2 + {\omega_0}^2}{s \cdot \mathrm{BW}}$$

This is the "wideband" transformation, producing a passband with geometric (log frequency) symmetry about *wo*.

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 *wo* 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

$$s \rightarrow \frac{s^2 + {\omega_0}^2}{s \cdot \mathrm{BW}}$$

This is the "wideband" transformation, producing a passband with geometric (log frequency) symmetry about *wo*.

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

$$s \rightarrow \frac{s \cdot \mathrm{BW}}{s^2 + {\omega_0}^2}$$

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

$$s \rightarrow \frac{s \cdot \mathrm{BW}}{s^2 + {\omega_0}^2}$$

This is the "wideband" transformation, producing a stopband with geometric (log frequency) symmetry about *wo*.

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 *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 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

$$s \to \frac{\omega_0}{s}$$

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

$$s \to \frac{s}{\omega_0}$$

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

zpk2tf(z, p, k)	Return polynomial transfer function representation from zeros and poles
zpk2sos(z, p, k[, pairing, analog])	Return second-order sections from zeros, poles, and gain of a system
zpk2ss(z, p, k)	Zero-pole-gain representation to state-space representation
tf2zpk(b, a)	Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.
tf2sos(b, a[, pairing, analog])	Return second-order sections from transfer function representation
tf2ss(num, den)	Transfer function to state-space representation.
ss2tf(A, B, C, D[, input])	State-space to transfer function.
ss2zpk(A, B, C, D[, input])	State-space representation to zero-pole-gain representation.
sos2tf(sos)	Return a single transfer function from a series of second-order sections
sos2zpk(sos)	Return zeros, poles, and gain of a series of second-order sections
cont2discrete(system, dt[, method, alpha])	Transform a continuous to a discrete state-space system.
place_poles(A, B, poles[, method, rtol, maxiter])	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 $b = [b_0, b_1, ..., b_M]$ and $a = [a_0, a_1, ..., a_N]$ can represent an analog filter of the form:

$$H(s) = \frac{b_0 s^M + b_1 s^{(M-1)} + \cdots + b_M}{a_0 s^N + a_1 s^{(N-1)} + \cdots + a_N}$$

or a discrete-time filter of the form:

$$H(z) = \frac{b_0 z^M + b_1 z^{(M-1)} + \cdots + b_M}{a_0 z^N + a_1 z^{(N-1)} + \cdots + a_N}$$

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:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}}$$

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 0

  See 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 congugates 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 * diag(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 `diag(poles) = (A - B*K)`, or 0 when the optimization algorithms can't do anything i.e when `B.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 `diag(poles) = (A - B*K)`, or 0 when the optimization algorithms can't do anything i.e when `B.shape[1] == 1`.

Return type
Bunch object

Notes

The Tits and Yang (YT),[2] paper is an update of the original Kautsky et al. (KNV) paper[1]. KNV relies on rank-1 updates to find the transfer matrix X such that `X * diag(poles) = (A - B*K)*X`, whereas YT uses rank-2 updates. This yields on average more robust solutions (see[Page 561, 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.

[2] 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

lti(*system)	Continuous-time linear time invariant system base class.
StateSpace(*system, **kwargs)	Linear Time Invariant system in state-space form.
TransferFunction(*system, **kwargs)	Linear Time Invariant system class in transfer function form.
ZerosPolesGain(*system, **kwargs)	Linear Time Invariant system class in zeros, poles, gain form.
lsim(system, U, T[, X0, interp])	Simulate output of a continuous-time linear system.
impulse(system[, X0, T, N])	Impulse response of continuous-time system.
step(system[, X0, T, N])	Step response of continuous-time system.
freqresp(system[, w, n])	Calculate the frequency response of a continuous-time system.
bode(system[, w, n])	Calculate Bode magnitude and phase data of a continuous-time system.

[2] 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.

[1] 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 == len(b) - 1`, `M == 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

dlti(*system, **kwargs)	Discrete-time linear time invariant system base class.
StateSpace(*system, **kwargs)	Linear Time Invariant system in state-space form.
TransferFunction(*system, **kwargs)	Linear Time Invariant system class in transfer function form.
ZerosPolesGain(*system, **kwargs)	Linear Time Invariant system class in zeros, poles, gain form.
dlsim(system, u[, t, x0])	Simulate output of a discrete-time linear system.
dimpulse(system[, x0, t, n])	Impulse response of discrete-time system.
dstep(system[, x0, t, n])	Step response of discrete-time system.
dfreqresp(system[, w, n, whole])	Calculate the frequency response of a discrete-time system.
dbode(system[, w, n])	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

chirp(t, f0, t1, f1[, method, phi, vertex_zero])	Frequency-swept cosine generator.
gausspulse(t[, fc, bw, bwr, tpr, retquad, ...])	Return a Gaussian modulated sinusoid:
max_len_seq(nbits[, state, length, taps])	Maximum length sequence (MLS) generator.
sawtooth(t[, width])	Return a periodic sawtooth or triangle waveform.
square(t[, duty])	Return a periodic square-wave waveform.
unit_impulse(shape[, idx, dtype])	Unit impulse signal (discrete delta function) or unit basis vector.
sweep_poly(t, poly[, phi])	Frequency-swept cosine generator, with a time-dependent frequency.

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(phase + (pi/180)*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

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

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 `width*2*pi`, then drops from 1 to -1 on the interval `width*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*duty` and -1 from `2*pi*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.]])
```

cupyx.scipy.signal.sweep_poly

cupyx.scipy.signal.**sweep_poly**(*t*, *poly*, *phi=0*)

Frequency-swept cosine generator, with a time-dependent frequency.

This function generates a sinusoidal function whose instantaneous frequency varies with time. The frequency at time *t* is given by the *poly* array.

Parameters

- **t** (ndarray) – Times at which to evaluate the waveform.
- **poly** (*1-D array_like or instance of numpy.poly1d*) – The desired frequency expressed as a polynomial. If *poly* is a list or ndarray of length n, then the elements of *poly* are the coefficients of the polynomial, and the instantaneous frequency is

  ```
  f(t) = poly[0]*t**(n-1) + poly[1]*t**(n-2) + ... + poly[n-1]
  ```

  If *poly* is an instance of cupy.poly1d, then the instantaneous frequency is

  ```
  f(t) = poly(t)
  ```

- **phi** (*float, optional*) – Phase offset, in degrees, Default: 0.

Returns

sweep_poly – A numpy array containing the signal evaluated at *t* with the requested time-varying frequency. More precisely, the function returns `cos(phase + (pi/180)*phi)`, where *phase* is the integral (from 0 to t) of `2 * pi * f(t)`; `f(t)` is defined above.

Return type

ndarray

See also:

scipy.signal.sweep_poly, *chirp*

Notes

If *poly* is an ndarray of length *n*, then the elements of *poly* are the coefficients of the polynomial, and the instantaneous frequency is:

```
f(t) = poly[0]*t**(n-1) + poly[1]*t**(n-2) + ... + poly[n-1]
```

If *poly* is an instance of *numpy.poly1d*, then the instantaneous frequency is:

```
f(t) = poly(t)
```

Finally, the output *s* is:

```
cos(phase + (pi/180)*phi)
```

where *phase* is the integral from 0 to *t* of `2 * pi * f(t)`, `f(t)` as defined above.

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:

get_window(window, Nx[, fftbins])	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 *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])
```

Wavelets

morlet(M[, w, s, complete])	Complex Morlet wavelet.
qmf(hk)	Return high-pass qmf filter from low-pass
ricker(points, a)	Return a Ricker wavelet, also known as the "Mexican hat wavelet".
morlet2(M, s[, w])	Complex Morlet wavelet, designed to work with *cwt*. Returns the complete version of morlet wavelet, normalised according to *s*::.
cwt(data, wavelet, widths)	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*w*x/s) * exp(-0.5*(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*fs / (2*s*np.pi)
```

where `fs` 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

argrelmin(data[, axis, order, mode])	Calculate the relative minima of *data*.
argrelmax(data[, axis, order, mode])	Calculate the relative maxima of *data*.
argrelextrema(data, comparator[, axis, ...])	Calculate the relative extrema of *data*.
find_peaks(x[, height, threshold, distance, ...])	Find peaks inside a signal based on peak properties.
peak_prominences(x, peaks[, wlen])	Calculate the prominence of each peak in a signal.
peak_widths(x, peaks[, rel_height, ...])	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:

argrelextrema, *argrelmax*, *find_peaks*

Notes

This function uses *argrelextrema* 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:

argrelextrema, *argrelmin*, *find_peaks*

Notes

This function uses *argrelextrema* 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.argrelextrema

cupyx.scipy.signal.**argrelextrema**(*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 (>= 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 constrains 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* where 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 <= left_base <= peak <= right_base < x.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

periodogram(x[, fs, window, nfft, detrend, ...])	Estimate power spectral density using a periodogram.
welch(x[, fs, window, nperseg, noverlap, ...])	Estimate power spectral density using Welch's method.
csd(x, y[, fs, window, nperseg, noverlap, ...])	Estimate the cross power spectral density, Pxy, using Welch's method.
coherence(x, y[, fs, window, nperseg, ...])	Estimate the magnitude squared coherence estimate, Cxy, of discrete-time signals X and Y using Welch's method.
spectrogram(x[, fs, window, nperseg, ...])	Compute a spectrogram with consecutive Fourier transforms.
lombscargle(x, y, freqs)	Computes the Lomb-Scargle periodogram.
vectorstrength(events, period)	Determine the vector strength of the events corresponding to the given period.
stft(x[, fs, window, nperseg, noverlap, ...])	Compute the Short Time Fourier Transform (STFT).
istft(Zxx[, fs, window, nperseg, noverlap, ...])	Perform the inverse Short Time Fourier transform (iSTFT).
check_COLA(window, nperseg, noverlap[, tol])	Check whether the Constant OverLap Add (COLA) constraint is met.
check_NOLA(window, nperseg, noverlap[, tol])	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 *Pxx* has units of V**2/Hz and computing the power spectrum ('spectrum') where *Pxx* 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[1] 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.
- **scaling** (*{ 'density', 'spectrum' }, optional*) – Selects between computing the power spectral density ('density') where *Pxx* has units of V**2/Hz and computing the power spectrum ('spectrum') where *Pxx* 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:

[1] 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.

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[2].

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, Pxy, 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.
- **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.

[2] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika, vol. 37, pp. 1-16, 1950.

- **scaling** (*{ 'density', 'spectrum' }, optional*) – Selects between computing the cross spectral density ('density') where *Pxy* has units of V**2/Hz and computing the cross spectrum ('spectrum') where *Pxy* 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, Pxy 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, Cxy, of discrete-time signals X and Y using Welch's method.

`Cxy = abs(Pxy)**2/(Pxx*Pyy)`, where *Pxx* and *Pyy* are power spectral density estimates of X and Y, and *Pxy* 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[1] and[2] for more information.

[1] 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.
[2] Stoica, Petre, and Randolph Moses, "Spectral Analysis of Signals" Prentice Hall, 2005

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.

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[1] 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()
```

[1] Oppenheim, Alan V., Ronald W. Schafer, 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[1] and further extended by Scargle[2] 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[3] 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 * 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

[1] N.R. Lomb "Least-squares frequency analysis of unequally spaced data", Astrophysics and Space Science, vol 39, pp. 447-462, 1976

[2] 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

[3] 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[1],[2] and[3] 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

[1] van Hemmen, JL, Longtin, A, and Vollmayr, AN. Testing resonating vector strength: Auditory system, electric fish, and noise. Chaos 21, 047508 (2011).
[2] van Hemmen, JL. Vector strength after Goldberg, Brown, and von Mises: biological and mathematical perspectives. Biol Cybern. 2013 Aug;107(4):385-96.
[3] 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 `abs(Zxx)**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 denomimator 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[1],[2] 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
```

[1] Oppenheim, Alan V., Ronald W. Schafer, John R. Buck "Discrete-Time Signal Processing", Prentice Hall, 1999.
[2] 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 realizible signal. This function implements the iSTFT via the least-squares estimation algorithm detailed in[2], which produces a signal that minimizes the mean squared error between the STFT of the returned signal and the modified STFT.

See[1, Page 612, 2] for more information.

[2] Daniel W. Griffin, Jae S. Lim "Signal Estimation from Modified Short-Time Fourier Transform", IEEE 1984, 10.1109/TASSP.1984.1164317
[1] Oppenheim, Alan V., Ronald W. Schafer, 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[2], wherein the COLA condition is satisfied when the "Amplitude Flatness" is unity. See[1] 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

[2] 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

[1] 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[1],[2] for more information.

References

Chirp Z-transform and Zoom FFT

czt(x[, m, w, a, axis])	Compute the frequency response around a spiral in the Z plane.
zoom_fft(x, fn[, m, fs, endpoint, axis])	Compute the DFT of *x* only for frequencies in range *fn*.
CZT(n[, m, w, a])	Create a callable chirp z-transform function.
ZoomFFT(n, fn[, m, fs, endpoint])	Create a callable zoom FFT transform function.
czt_points(m[, w, a])	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

[1] Julius O. Smith III, "Spectral Audio Signal Processing", W3K Publishing, 2011,ISBN 978-0-9745607-3-1.

[2] 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.[1]

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+2df, …, f2, where df=(f2-f1)/m.

Return type

ndarray

See also:

[1] 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 > 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[1], 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'.)

(The name "chirp z-transform" comes from the use of a chirp in the Bluestein algorithm[2]. 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.

[1] 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).

[2] 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.[1]

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`

[1] 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 > len(x)`, that `f(x, 2, m)` is equivalent to `fft.fft(x, m)`.

Sampling frequency is 1/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*).[2]

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.

[2] 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)

get_window(window, Nx[, fftbins])	Return a window of a given length and type.
barthann(M[, sym])	Return a modified Bartlett-Hann window.
bartlett(M[, sym])	Return a Bartlett window.
blackman(M[, sym])	Return a Blackman window.
blackmanharris(M[, sym])	Return a minimum 4-term Blackman-Harris window.
bohman(M[, sym])	Return a Bohman window.
boxcar(M[, sym])	Return a boxcar or rectangular window.
chebwin(M, at[, sym])	Return a Dolph-Chebyshev window.
cosine(M[, sym])	Return a window with a simple cosine shape.
exponential(M[, center, tau, sym])	Return an exponential (or Poisson) window.
flattop(M[, sym])	Return a flat top window.
gaussian(M, std[, sym])	Return a Gaussian window.
general_cosine(M, a[, sym])	Generic weighted sum of cosine terms window
general_gaussian(M, p, sig[, sym])	Return a window with a generalized Gaussian shape.
general_hamming(M, alpha[, sym])	Return a generalized Hamming window.
hamming(M[, sym])	Return a Hamming window.
hann(M[, sym])	Return a Hann window.
kaiser(M, beta[, sym])	Return a Kaiser window.
kaiser_bessel_derived(M, beta[, sym])	Return a Kaiser-Bessel derived window.
nuttall(M[, sym])	Return a minimum 4-term Blackman-Harris window according to Nuttall.
parzen(M[, sym])	Return a Parzen window.
taylor(M[, nbar, sll, norm, sym])	Return a Taylor window.
triang(M[, sym])	Return a triangular window.
lanczos(M[, sym])	Return a Lanczos window also known as a sinc window.
tukey(M[, alpha, sym])	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.[2]

For more information, see[1], Page 626, [2], [3], [4] and[5]

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")
```

[2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.
[1] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika 37, 1-16, 1950.
[3] A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal Processing", Prentice-Hall, 1999, pp. 468-471.
[4] Wikipedia, "Window function", https://en.wikipedia.org/wiki/Window_function
[5] 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.[3]

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[1],[2], and[Page 627, 3]

[3] 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

[1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[2] Oppenheim, A.V., and R.W. Schafer. 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 <= abs(k) <= 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[1],[2] and[3]

[1] 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

[2] Peter Lynch, "The Dolph-Chebyshev Window: A Simple Optimal Filter", American Meteorological Society (April 1997) http://mathsci.ucd.ie/~plynch/Publications/Dolph.pdf

[3] 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.[1]

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]")
```

[1] 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[1] and[2]

References

Examples

Heinzel describes a flat-top window named "HFT90D" with formula:[Page 637, 2]

$$w_j = 1 - 1.942604\cos(z) + 1.340318\cos(2z) - 0.440811\cos(3z) + 0.043097\cos(4z)$$

where

$$z = \frac{2\pi j}{N}, j = 0...N-1$$

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")
```

[1] 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

[2] 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

```
>>> 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[1].

Parameters

- **M** (*int*) – Number of points in the output window. If zero or less, an empty array is returned.
- **alpha** (*float*) – The window coefficient, α
- **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

[1] DSPRelated, "Generalized Hamming Window Family", https://www.dsprelated.com/freebooks/sasp/Generalized_Hamming_Window_Family.html

Notes

The generalized Hamming window is defined as

$$w(n) = \alpha - (1 - \alpha) \cos\left(\frac{2\pi n}{M - 1}\right) \qquad 0 \leq n \leq M - 1$$

Both the common Hamming window and Hann window are special cases of the generalized Hamming window with α = 0.54 and α = 0.5, respectively[2].

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[3]. The facility uses various values for the α parameter based on operating mode of the SAR instrument. Some common α values include 0.75, 0.7 and 0.52[4]. 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")
```

[2] Wikipedia, "Window function", https://en.wikipedia.org/wiki/Window_function

[3] Riccardo Piantanida ESA, "Sentinel-1 Level 1 Detailed Algorithm Definition", https://sentinel.esa.int/documents/247904/1877131/Sentinel-1-Level-1-Detailed-Algorithm-Definition

[4] 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

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

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[1],[2],[3] and[4]

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
```

[1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.
[2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.
[3] Wikipedia, "Window function", https://en.wikipedia.org/wiki/Window_function
[4] 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

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \le n \le M-1$$

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[1],[2],[3], and[4]

[1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.
[2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 106-108.
[3] Wikipedia, "Window function", https://en.wikipedia.org/wiki/Window_function
[4] 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.)[4]

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[1], [2], [3], and[Page 644, 4]

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
```

[4] 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

[1] 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).

[2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 177-178.

[3] Wikipedia, "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.kaiser_bessel_derived

cupyx.scipy.signal.windows.**kaiser_bessel_derived**(*M*, *beta*, *sym=True*)

Return a Kaiser-Bessel derived 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. Note that this window is only defined for an even number of points.
- **beta** (*float*) – Kaiser window shape parameter.
- **sym** (*bool, optional*) – This parameter only exists to comply with the interface offered by the other window functions and to be callable by *get_window*. When True (default), generates a symmetric window, for use in filter design.

Returns

w – The window, normalized to fulfil the Princen-Bradley condition.

Return type

ndarray

See also:

kaiser, scipy.signal.windows.kaiser_bessel_derived

Notes

It is designed to be suitable for use with the modified discrete cosine transform (MDCT) and is mainly used in audio signal processing and audio coding.[1]

[1] Bosi, Marina, and Richard E. Goldberg. Introduction to Digital Audio Coding and Standards. Dordrecht: Kluwer, 2003.

References

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.[2]

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[1] and[Page 646, 2]

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])
```

(continues on next page)

[2] 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

[1] 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

(continued from previous page)

```
>>> 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[1].

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)
```

(continues on next page)

[1] E. Parzen, “Mathematical Considerations in the Estimation of Spectra”, Technometrics, Vol. 3, No. 2 (May, 1961), pp. 167-190

(continued from previous page)

```
>>> 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[2]. The SAR (synthetic aperature radar) community commonly uses Taylor weighting for image formation processing because it provides strong, selectable sidelobe suppression with minimum broadening of the mainlobe[1].

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.
- **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

[2] 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

[1] W. Carrara, R. Goodman, and R. Majewski, "Spotlight Synthetic Aperture Radar: Signal Processing Algorithms" Pages 512-513, July 1995.

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))
```

(continues on next page)

(continued from previous page)

```
>>> 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.lanczos

cupyx.scipy.signal.windows.**lanczos**(*M*, *sym=True*)

Return a Lanczos window also known as a sinc 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

The Lanczos window is defined as

$$w(n) = sinc\left(\frac{2n}{M-1} - 1\right)$$

where

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

The Lanczos window has reduced Gibbs oscillations and is widely used for filtering climate timeseries with good properties in the physical and spectral domains.

See also:

scipy.signal.windows.lanczos

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[1] and[2].

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])
```

(continues on next page)

[1] 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

[2] Wikipedia, "Window function", https://en.wikipedia.org/wiki/Window_function#Tukey_window

(continued from previous page)

```
>>> 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.

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

`coo_matrix`(arg1[, shape, dtype, copy])	COOrdinate format sparse matrix.
`csc_matrix`(arg1[, shape, dtype, copy])	Compressed Sparse Column matrix.
`csr_matrix`(arg1[, shape, dtype, copy])	Compressed Sparse Row matrix.
`dia_matrix`(arg1[, shape, dtype, copy])	Sparse matrix with DIAgonal storage.
`spmatrix`([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 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.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 comuted. 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 COOdinate 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-dimenaional *cupy.ndarray*.

csc_matrix((data, indices, indptr))
All `data`, `indices` and `indptr` 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 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.narray 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.narray 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 comuted. 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 COOdinate 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-dimenaional *cupy.ndarray*.

csr_matrix((data, indices, indptr))

All `data`, `indices` and `indptr` 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 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.narray 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.narray 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 comuted. 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.csr_matrix.toarray()`

tobsr(*blocksize=None*, *copy=False*)

Convert this matrix to Block Sparse Row format.

tocoo(*copy=False*)

Converts the matrix to COOdinate 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<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 = '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 comuted. 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 comuted. 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:

eye(m[, n, k, dtype, format])	Creates a sparse matrix with ones on diagonal.
identity(n[, dtype, format])	Creates an identity matrix in sparse format.
kron(A, B[, format])	Kronecker product of sparse matrices A and B.
kronsum(A, B[, format])	Kronecker sum of sparse matrices A and B.
diags(diagonals[, offsets, shape, format, dtype])	Construct a sparse matrix from diagonals.
spdiags(data, diags, m, n[, format])	Creates a sparse matrix from diagonals.
tril(A[, k, format])	Returns the lower triangular portion of a matrix in sparse format
triu(A[, k, format])	Returns the upper triangular portion of a matrix in sparse format
bmat(blocks[, format, dtype])	Builds a sparse matrix from sparse sub-blocks
hstack(blocks[, format, dtype])	Stacks sparse matrices horizontally (column wise)
vstack(blocks[, format, dtype])	Stacks sparse matrices vertically (row wise)
rand(m, n[, density, format, dtype, ...])	Generates a random sparse matrix.
random(m, n[, density, format, dtype, ...])	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 `kron(I_n, A) + 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 `density` from `(m, n)` elements. So the number of non-zero elements `k` is `k = m * n * 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:

find(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:

issparse(x)	Checks if a given matrix is a sparse matrix.
isspmatrix(x)	Checks if a given matrix is a sparse matrix.
isspmatrix_csc(x)	Checks if a given matrix is of CSC format.
isspmatrix_csr(x)	Checks if a given matrix is of CSR format.
isspmatrix_coo(x)	Checks if a given matrix is of COO format.
isspmatrix_dia(x)	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

LinearOperator(shape, matvec[, rmatvec, ...])	Common interface for performing matrix vector products
aslinearoperator(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 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

norm(x[, ord, axis])	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:

spsolve(A, b)	Solves a sparse linear system `A x = b`
spsolve_triangular(A, b[, lower, ...])	Solves a sparse triangular system `A x = b`.
factorized(A)	Return a function for solving a sparse linear system, with A pre-factorized.

cupyx.scipy.sparse.linalg.spsolve

cupyx.scipy.sparse.linalg.**spsolve**(*A*, *b*)

Solves a sparse linear system `A x = 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 x = 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 x = 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 trinagular 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 x = 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:

cg(A, b[, x0, tol, maxiter, M, callback, atol])	Uses Conjugate Gradient iteration to solve `Ax = b`.
gmres(A, b[, x0, tol, restart, maxiter, M, ...])	Uses Generalized Minimal RESidual iteration to solve `Ax = b`.
cgs(A, b[, x0, tol, maxiter, M, callback, atol])	Use Conjugate Gradient Squared iteration to solve `Ax = b`.
minres(A, b[, x0, shift, tol, maxiter, M, ...])	Uses MINimum RESidual iteration to solve `Ax = 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 `Ax = 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 arugment.
- **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 shift != 0 then the method solves `(A - shift*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:

lsqr(A, b)	Solves linear system with QR decomposition.
lsmr(A, b[, x0, damp, atol, btol, conlim, ...])	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 ||(b) - (  A   )x||
    ||(0)   (damp*I) ||_2
```

  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 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): norm(b-Ax)
- *normar* (float): norm(A^T (b - Ax))
- *norma* (float): norm(A)

- *conda* (float): Condition number of A.
- *normx* (float): `norm(x)`

Return type
tuple

See also:

`scipy.sparse.linalg.lsmr()`

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:

eigsh(a[, k, which, v0, ncv, maxiter, tol, ...])	Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex Hermitian matrix A.
lobpcg(A, X[, B, M, Y, tol, maxiter, ...])	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 *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 <= 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 `shape=(n,k)`.
- **B** (*array-like*) – The right hand side operator in a generalized eigenproblem. By default, `B = 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 = 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 <= k < min(m, n)`.
- **ncv** (*int*) – The number of Lanczos vectors generated. Must be `k + 1 < 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:

splu(A[, permc_spec, diag_pivot_thresh, ...])	Computes the LU decomposition of a sparse square matrix.
spilu(A[, drop_tol, fill_factor, drop_rule, ...])	Computes the incomplete LU decomposition of a sparse square matrix.
SuperLU(obj)	

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

connected_components(csgraph[, directed, ...])	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.

Nearest-neighbor queries

KDTree(data[, leafsize, compact_nodes, ...])	KDTree(data, leafsize=16, compact_nodes=True, copy_data=False,

cupyx.scipy.spatial.KDTree

class cupyx.scipy.spatial.**KDTree**(*data*, *leafsize=10*, *compact_nodes=True*, *copy_data=False*, *balanced_tree=True*, *boxsize=None*)

KDTree(data, leafsize=16, compact_nodes=True, copy_data=False,
balanced_tree=True, boxsize=None)

kd-tree for quick nearest-neighbor lookup

This class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors of any point.

Parameters

- **data** (*array_like, shape (n,m)*) – The n data points of dimension m to be indexed. This array is not copied unless this is necessary to produce a contiguous array of doubles, and so modifying this data will result in bogus results. The data are also copied if the kd-tree is built with copy_data=True.
- **leafsize** (*positive int, optional*) – The number of points at which the algorithm switches over to brute-force. Default: 16.
- **compact_nodes** (*bool, optional*) – If True, the kd-tree is built to shrink the hyperrectangles to the actual data range. This usually gives a more compact tree that is robust against degenerated input data and gives faster queries at the expense of longer build time. Default: True.
- **copy_data** (*bool, optional*) – If True the data is always copied to protect the kd-tree against data corruption. Default: False.
- **balanced_tree** (*bool, optional*) – If True, the median is used to split the hyperrectangles instead of the midpoint. This usually gives a more compact tree and faster queries at the expense of longer build time. Default: True.
- **boxsize** (*array_like or scalar, optional*) – Apply a m-d toroidal topology to the KDTree.. The topology is generated by $x_i + n_i L_i$ where n_i are integers and L_i is the boxsize along i-th dimension. The input data shall be wrapped into $[0, L_i)$. A ValueError is raised if any of the data is outside of this bound.

Notes

The algorithm used is described in Wald, I. 2022[1]. The general idea is that the kd-tree is a binary tree, each of whose nodes represents an axis-aligned hyperrectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value.

The tree can be queried for the r closest neighbors of any given point (optionally returning only those within some maximum distance of the point). It can also be queried, with a substantial gain in efficiency, for the r approximate closest neighbors. See[2] for more information regarding the implementation.

[1] Wald, I., GPU-friendly, Parallel, and (Almost-)In-Place Construction of Left-Balanced k-d Trees, 2022. doi:10.48550/arXiv.2211.00120.
[2] Wald, I., A Stack-Free Traversal Algorithm for Left-Balanced k-d Trees, 2022. doi:10.48550/arXiv.2210.12859.

For large dimensions (20 is already large) do not expect this to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.

Variables

- **data** (*ndarray, shape (n,m)*) – The n data points of dimension m to be indexed. This array is not copied unless this is necessary to produce a contiguous array of doubles. The data are also copied if the kd-tree is built with *copy_data=True*.
- **leafsize** (*positive int*) – The number of points at which the algorithm switches over to brute-force.
- **m** (*int*) – The dimension of a single data-point.
- **n** (*int*) – The number of data points.
- **maxes** (*ndarray, shape (m,)*) – The maximum value in each dimension of the n data points.
- **mins** (*ndarray, shape (m,)*) – The minimum value in each dimension of the n data points.
- **tree** (*ndarray*) – This attribute exposes the array representation of the tree.
- **size** (*int*) – The number of nodes in the tree.

References

Methods

count_neighbors(*other*, *r*, *p=2.0*, *weights=None*, *cumulative=True*)

Count how many nearby pairs can be formed.

Count the number of pairs `(x1,x2)` can be formed, with `x1` drawn from `self` and `x2` drawn from `other`, and where `distance(x1, x2, p) <= r`.

Data points on `self` and `other` are optionally weighted by the `weights` argument. (See below)

This is adapted from the "two-point correlation" algorithm described by Gray and Moore[Page 720, 1]. See notes for further discussion.

Parameters

- **other** (*KDTree instance*) – The other tree to draw points from, can be the same tree as self.
- **r** (*float or one-dimensional array of floats*) – The radius to produce a count for. Multiple radii are searched with a single tree traversal. If the count is non-cumulative(`cumulative=False`), `r` defines the edges of the bins, and must be non-decreasing.
- **p** (*float, optional*) – 1<=p<=infinity. Which Minkowski p-norm to use. Default 2.0. A finite large p may cause a ValueError if overflow can occur.
- **weights** (*tuple, array_like, or None, optional*) – If None, the pair-counting is unweighted. If given as a tuple, weights[0] is the weights of points in `self`, and weights[1] is the weights of points in `other`; either can be None to indicate the points are unweighted. If given as an array_like, weights is the weights of points in `self` and `other`. For this to make sense, `self` and `other` must be the same tree. If `self` and `other` are two different trees, a `ValueError` is raised. Default: None

- **cumulative** (*bool, optional*) – Whether the returned counts are cumulative. When cumulative is set to `False` the algorithm is optimized to work with a large number of bins (>10) specified by `r`. When `cumulative` is set to True, the algorithm is optimized to work with a small number of `r`. Default: True

Returns

result – The number of pairs. For unweighted counts, the result is integer. For weighted counts, the result is float. If cumulative is False, `result[i]` contains the counts with `(-inf if i == 0 else r[i-1]) < R <= r[i]`

Return type

scalar or 1-D array

query(*x*, *k=1*, *eps=0.0*, *p=2.0*, *distance_upper_bound=inf*)

Query the kd-tree for nearest neighbors.

Parameters

- **x** (*array_like, last dimension self.m*) – An array of points to query.
- **k** (*list of integer or integer*) – The list of k-th nearest neighbors to return. If k is an integer it is treated as a list of [1, … k] (range(1, k+1)). Note that the counting starts from 1.
- **eps** (*non-negative float*) – Return approximate nearest neighbors; the k-th returned value is guaranteed to be no further than (1+eps) times the distance to the real k-th nearest neighbor.
- **p** (*float, 1<=p<=infinity*) – Which Minkowski p-norm to use. 1 is the sum-of-absolute-values "Manhattan" distance 2 is the usual Euclidean distance infinity is the maximum-coordinate-difference distance A finite large p may cause a ValueError if overflow can occur.
- **distance_upper_bound** (*nonnegative float*) – Return only neighbors within this distance. This is used to prune tree searches, so if you are doing a series of nearest-neighbor queries, it may help to supply the distance to the nearest neighbor of the most recent point.

Returns

- **d** (*array of floats*) – The distances to the nearest neighbors. If `x` has shape `tuple+(self.m,)`, then `d` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with infinite distances.
- **i** (*ndarray of ints*) – The index of each neighbor in `self.data`. If `x` has shape `tuple+(self.m,)`, then `i` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with `self.n`.

Notes

If the KD-Tree is periodic, the position `x` is wrapped into the box.

When the input k is a list, a query for arange(max(k)) is performed, but only columns that store the requested values of k are preserved. This is implemented in a manner that reduces memory usage.

Examples

```
>>> import cupy as cp
>>> from cupyx.scipy.spatial import KDTree
>>> x, y = cp.mgrid[0:5, 2:8]
>>> tree = KDTree(cp.c_[x.ravel(), y.ravel()])
```

To query the nearest neighbours and return squeezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=1)
>>> print(dd, ii, sep='\n')
[2.         0.2236068]
[ 0 13]
```

To query the nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[1])
>>> print(dd, ii, sep='\n')
[[2.        ]
 [0.2236068]]
[[ 0]
 [13]]
```

To query the second nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[2])
>>> print(dd, ii, sep='\n')
[[2.23606798]
 [0.80622577]]
[[ 6]
 [19]]
```

To query the first and second nearest neighbours, use

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=2)
>>> print(dd, ii, sep='\n')
[[2.         2.23606798]
 [0.2236068  0.80622577]]
[[ 0  6]
 [13 19]]
```

or, be more specific

```
>>> dd, ii = tree.query([[0, 0], [2.2, 2.9]], k=[1, 2])
>>> print(dd, ii, sep='\n')
[[2.         2.23606798]
 [0.2236068  0.80622577]]
[[ 0  6]
 [13 19]]
```

query_ball_point(*x*, *r*, *p=2.0*, *eps=0*, *return_sorted=None*, *return_length=False*)

Find all points within distance r of point(s) x.

Parameters

- **x** (*array_like, shape tuple + (self.m,)*) – The point or points to search for neighbors of.
- **r** (*array_like, float*) – The radius of points to return, shall broadcast to the length of x.
- **p** (*float, optional*) – Which Minkowski p-norm to use. Should be in the range [1, inf]. A finite large p may cause a ValueError if overflow can occur.
- **eps** (*nonnegative float, optional*) – Approximate search. Branches of the tree are not explored if their nearest points are further than `r / (1 + eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1 + eps)`.
- **return_sorted** (*bool, optional*) – Sorts returned indices if True and does not sort them if False. If None, does not sort single point queries, but does sort multi-point queries which was the behavior before this option was added in SciPy.
- **return_length** (*bool, optional*) – Return the number of points inside the radius instead of a list of the indices.

Returns

results – If *x* is a single point, returns a list of the indices of the neighbors of *x*. If *x* is an array of points, returns an object array of shape tuple containing lists of neighbors.

Return type

list or array of lists

Notes

If you have many points whose neighbors you want to find, you may save substantial amounts of time by putting them in a KDTree and using query_ball_tree.

Examples

```
>>> import cupy as cp
>>> from cupyx.scipy import spatial
>>> x, y = cp.mgrid[0:4, 0:4]
>>> points = cp.c_[x.ravel(), y.ravel()]
>>> tree = spatial.KDTree(points)
>>> tree.query_ball_point([2, 0], 1)
[4, 8, 9, 12]
```

query_ball_tree(*other*, *r*, *p=2.0*, *eps=0.0*)

Find all pairs of points between *self* and *other* whose distance is at most r.

Parameters

- **other** (*KDTree instance*) – The tree containing points to search against.
- **r** (*float*) – The maximum distance, has to be positive.
- **p** (*float, optional*) – Which Minkowski norm to use. *p* has to meet the condition `1 <= p <= infinity`. A finite large p may cause a ValueError if overflow can occur.
- **eps** (*float, optional*) – Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

Returns

results – For each element `self.data[i]` of this tree, `results[i]` is a list of the indices of its neighbors in `other.data`.

Return type

list of ndarrays

Examples

You can search all pairs of points between two kd-trees within a distance:

```
>>> import matplotlib.pyplot as plt
>>> import cupy as cp
>>> from cupyx.scipy.spatial import KDTree
>>> points1 = cp.random.rand((15, 2))
>>> points2 = cp.random.rand((15, 2))
>>> plt.figure(figsize=(6, 6))
>>> plt.plot(points1[:, 0], points1[:, 1], "xk", markersize=14)
>>> plt.plot(points2[:, 0], points2[:, 1], "og", markersize=14)
>>> kd_tree1 = KDTree(points1)
>>> kd_tree2 = KDTree(points2)
>>> indexes = kd_tree1.query_ball_tree(kd_tree2, r=0.2)
>>> for i in range(len(indexes)):
...     for j in indexes[i]:
...         plt.plot([points1[i, 0], points2[j, 0]],
...             [points1[i, 1], points2[j, 1]], "-r")
>>> plt.show()
```

query_pairs(*r*, *p=2.0*, *eps=0*, *output_type='ndarray'*)

Find all pairs of points in *self* whose distance is at most r.

Parameters

- **r** (*positive float*) – The maximum distance.
- **p** (*float, optional*) – Which Minkowski norm to use. `p` has to meet the condition `1 <= p <= infinity`. A finite large p may cause a ValueError if overflow can occur.
- **eps** (*float, optional*) – Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.
- **output_type** (*string, optional*) – Choose the output container, 'set' or 'ndarray'. Default: 'ndarray' Note: 'set' output is not supported.

Returns

results – An ndarray of size `(total_pairs, 2)`, containing each pair `(i,j)`, with `i < j`, for which the corresponding positions are close.

Return type

ndarray

Notes

This method does not support the *set* output type.

Examples

You can search all pairs of points in a kd-tree within a distance:

```
>>> import matplotlib.pyplot as plt
>>> import cupy as cp
>>> from cupyx.scipy.spatial import KDTree
>>> points = cp.random.rand((20, 2))
>>> plt.figure(figsize=(6, 6))
>>> plt.plot(points[:, 0], points[:, 1], "xk", markersize=14)
>>> kd_tree = KDTree(points)
>>> pairs = kd_tree.query_pairs(r=0.2)
>>> for (i, j) in pairs:
...     plt.plot([points[i, 0], points[j, 0]],
...              [points[i, 1], points[j, 1]], "-r")
>>> plt.show()
```

sparse_distance_matrix(*other*, *max_distance*, *p=2.0*, *output_type='coo_matrix'*)

Compute a sparse distance matrix

Computes a distance matrix between two KDTrees, leaving as zero any distance greater than max_distance.

Parameters

- **other** (*KDTree*) –
- **max_distance** (*positive float*) –
- **p** (*float, 1<=p<=infinity*) – Which Minkowski p-norm to use. A finite large p may cause a ValueError if overflow can occur.
- **output_type** (*string, optional*) – Which container to use for output data. Options: 'coo_matrix' or 'ndarray'. Default: 'coo_matrix'.

Returns

result – Sparse matrix representing the results in "dictionary of keys" format. If output_type is 'ndarray' an NxM distance matrix will be returned.

Return type

coo_matrix or *ndarray*

Examples

You can compute a sparse distance matrix between two kd-trees:

```
>>> import cupy
>>> from cupyx.scipy.spatial import KDTree
>>> points1 = cupy.random.rand((5, 2))
>>> points2 = cupy.random.rand((5, 2))
>>> kd_tree1 = KDTree(points1)
>>> kd_tree2 = KDTree(points2)
>>> sdm = kd_tree1.sparse_distance_matrix(kd_tree2, 0.3)
```

(continues on next page)

(continued from previous page)

```
>>> sdm.toarray()
array([[0.        , 0.        , 0.12295571, 0.        , 0.        ],
   [0.        , 0.        , 0.        , 0.        , 0.        ],
   [0.28942611, 0.        , 0.        , 0.2333084 , 0.        ],
   [0.        , 0.        , 0.        , 0.        , 0.        ],
   [0.24617575, 0.29571802, 0.26836782, 0.        , 0.        ]])
```

You can check distances above the *max_distance* are zeros:

```
>>> from cupyx.scipy.spatial import distance_matrix
>>> distance_matrix(points1, points2)
array([[0.56906522, 0.39923701, 0.12295571, 0.8658745 , 0.79428925],
   [0.37327919, 0.7225693 , 0.87665969, 0.32580855, 0.75679479],
   [0.28942611, 0.30088013, 0.6395831 , 0.2333084 , 0.33630734],
   [0.31994999, 0.72658602, 0.71124834, 0.55396483, 0.90785663],
   [0.24617575, 0.29571802, 0.26836782, 0.57714465, 0.6473269 ]])
```

__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.

Delaunay triangulation

Delaunay(points[, furthest_site, incremental])	Delaunay tessellation in 2 dimensions.

cupyx.scipy.spatial.Delaunay

class cupyx.scipy.spatial.**Delaunay**(*points*, *furthest_site=False*, *incremental=False*)

Delaunay tessellation in 2 dimensions.

Parameters

- **points** (*ndarray of floats, shape (npoints, ndim)*) – Coordinates of points to triangulate
- **furthest_site** (*bool, optional*) – Whether to compute a furthest-site Delaunay triangulation. This option will be ignored, since it is not supported by CuPy Default: False

- **incremental** (*bool, optional*) – Allow adding new points incrementally. This takes up some additional resources. This option will be ignored, since it is not supported by CuPy. Default: False

Variables

- **points** (ndarray *of double, shape (npoints, ndim)*) – Coordinates of input points.
- **simplices** (ndarray *of ints, shape (nsimplex, ndim+1)*) – Indices of the points forming the simplices in the triangulation. For 2-D, the points are oriented counterclockwise.
- **neighbors** (ndarray *of ints, shape (nsimplex, ndim+1)*) – Indices of neighbor simplices for each simplex. The kth neighbor is opposite to the kth vertex. For simplices at the boundary, -1 denotes no neighbor.0
- **vertex_neighbor_vertices** (*tuple of two ndarrays of int; (indptr, indices)*) – Neighboring vertices of vertices. The indices of neighboring vertices of vertex *k* are `indices[indptr[k]:indptr[k+1]]`.

Notes

This implementation makes use of GDel2D to perform the triangulation in 2D. See[1] for more information.

References

Methods

find_simplex(*xi*, *bruteforce=False*, *tol=None*)

Find the simplices containing the given points.

Parameters

- **xi** (ndarray *of double, shape (..., ndim)*) – Points to locate
- **bruteforce** (*bool, optional*) – Whether to only perform a brute-force search. Not used by CuPy
- **tol** (*float, optional*) – Tolerance allowed in the inside-triangle check. Default is `100*eps`.

Returns

i – Indices of simplices containing each point. Points outside the triangulation get the value -1.

Return type

ndarray of int, same shape as *xi*

vertex_neighbor_vertices()

Neighboring vertices of vertices.

Tuple of two ndarrays of int: (indptr, indices). The indices of neighboring vertices of vertex *k* are `indices[indptr[k]:indptr[k+1]]`.

__eq__(*value*, */*)

Return self==value.

[1] A GPU accelerated algorithm for 3D Delaunay triangulation (2014). Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, Tiow-Seng Tan. Proc. 18th ACM SIGGRAPH Symp. Interactive 3D Graphics and Games, 47-55.

__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

distance_matrix(x, y[, p])	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 <= p <= 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.

pdist(X[, metric, out])	Compute distance between observations in n-dimensional space.
cdist(XA, XB[, metric, out])	Compute distance between each pair of the two collections of inputs.
distance_matrix(x, y[, p])	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 `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 <= p <= 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*)

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.

minkowski(u, v, p)	Compute the Minkowski distance between two 1-D arrays.
canberra(u, v)	Compute the Canberra distance between two 1-D arrays.
chebyshev(u, v)	Compute the Chebyshev distance between two 1-D arrays.
cityblock(u, v)	Compute the City Block (Manhattan) distance between two 1-D arrays.
correlation(u, v)	Compute the correlation distance between two 1-D arrays.
cosine(u, v)	Compute the Cosine distance between two 1-D arrays.
hamming(u, v)	Compute the Hamming distance between two 1-D arrays.
euclidean(u, v)	Compute the Euclidean distance between two 1-D arrays.
jensenshannon(u, v)	Compute the Jensen-Shannon distance between two 1-D arrays.
russellrao(u, v)	Compute the Russell-Rao distance between two 1-D arrays.
sqeuclidean(u, v)	Compute the squared Euclidean distance between two 1-D arrays.
hellinger(u, v)	Compute the Hellinger distance between two 1-D arrays.
kl_divergence(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

$$d(u, v) = \left(\sum_i (u_i - v_i)^2 \right)^{12}$$

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

j0	Bessel function of the first kind of order 0.
j1	Bessel function of the first kind of order 1.
k0	Modified Bessel function of the second kind of order 0.
k0e	Exponentially scaled modified Bessel function K of order 0
k1	Modified Bessel function of the second kind of order 1.
k1e	Exponentially scaled modified Bessel function K of order 1
y0	Bessel function of the second kind of order 0.
y1	Bessel function of the second kind of order 1.
yn	Bessel function of the second kind of order n.
i0	Modified Bessel function of order 0.
i0e	Exponentially scaled modified Bessel function of order 0.
i1	Modified Bessel function of order 1.
i1e	Exponentially scaled modified Bessel function of order 1.
spherical_yn(n, z[, derivative])	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

bdtr	Binomial distribution cumulative distribution function.
bdtrc	Binomial distribution survival function.
bdtri	Inverse function to *bdtr* with respect to *p*.
btdtr(a, b, x[, out])	
btdtri(a, b, p[, out])	
fdtr	F cumulative distribution function.
fdtrc	F survival function.
fdtri	The p-th quantile of the F-distribution.
gdtr	Gamma distribution cumulative distribution function.
gdtrc	Gamma distribution survival function.
nbdtr	Negative binomial distribution cumulative distribution function.
nbdtrc	Negative binomial distribution survival function.
nbdtri	Inverse function to *nbdtr* with respect to *p*.
pdtr	Poisson cumulative distribution function.
pdtrc	Binomial distribution survival function.
pdtri	Inverse function to *pdtr* with respect to *m*.
chdtr	Chi-square cumulative distribution function.
chdtrc	Chi square survival function.
chdtri	Inverse to *chdtrc* with respect to *x*.
ndtr	Cumulative distribution function of normal distribution.
log_ndtr	Logarithm of Gaussian cumulative distribution function.
ndtri	Inverse of the cumulative distribution function of the standard
logit	Logit function.
expit	Logistic sigmoid function (expit).
log_expit	Logarithm of the logistic sigmoid function.
boxcox	Compute the Box-Cox transformation.
boxcox1p	Compute the Box-Cox transformation op 1 + *x*.
inv_boxcox	Compute the Box-Cox transformation.
inv_boxcox1p	Compute the Box-Cox transformation op 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`**(*a*, *b*, *x*, *out=None*)

cupyx.scipy.special.btdtri

`cupyx.scipy.special.`**`btdtri`**(*a*, *b*, *p*, *out=None*)

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.fdtr()`

cupyx.scipy.special.fdtrc

`cupyx.scipy.special.`**`fdtrc`**()

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.fdtrc()`

cupyx.scipy.special.fdtri

`cupyx.scipy.special.fdtri()`

The p-th quantile of the F-distribution.

This function is the inverse of the F-distribution CDF, *fdtr*, returning the *x* such that *fdtr(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.fdtri()`

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(expit(x))

Return type
cupy.ndarray

See also:

scipy.special.log_expit

Note: The function is mathematically equivalent to `log(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 boxcox(x)

Return type
cupy.ndarray

See also:

scipy.special.boxcox

cupyx.scipy.special.boxcox1p

cupyx.scipy.special.**boxcox1p**()

Compute the Box-Cox transformation op 1 + *x*.

Parameters
x (cupy.ndarray) – input data (must be real)

Returns
values of 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 op 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

entr	Elementwise function for computing entropy.
rel_entr	Elementwise function for computing relative entropy.
kl_div	Elementwise function for computing Kullback-Leibler divergence.
huber	Elementwise function for computing the Huber loss.
pseudo_huber	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

gamma	Gamma function.
gammaln	Logarithm of the absolute value of the Gamma function.
loggamma	Principal branch of the logarithm of the gamma function.
gammainc	Elementwise function for scipy.special.gammainc
gammaincinv	Elementwise function for scipy.special.gammaincinv
gammaincc	Elementwise function for scipy.special.gammaincc
gammainccinv	Elementwise function for scipy.special.gammainccinv
beta	Beta function.
betaln	Natural logarithm of absolute value of beta function.
betainc	Incomplete beta function.
betaincinv	Inverse of the incomplete beta function.
psi	The digamma function.
rgamma	Reciprocal gamma function.
polygamma(n, x)	Polygamma function n.
multigammaln(a, d)	Returns the log of multivariate gamma, also sometimes called the generalized gamma.
digamma	The digamma function.
poch	Elementwise function for scipy.special.poch (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 plain 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(abs(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 <= x <= 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

erf	Error function.
erfc	Complementary error function.
erfcx	Scaled complementary error function.
erfinv	Inverse function of error function.
erfcinv	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

lpmv	Associated Legendre function of integer order and real degree.
sph_harm	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

exp1	Exponential integral E1.
expi	Exponential integral Ei.
expn	Generalized exponential integral En.
exprel	Computes `(exp(x) - 1) / x`.
softmax(x[, axis])	Softmax function.
lambertw(z[, k, tol])	Lambert W function.
log_softmax(x[, axis])	Compute logarithm of softmax function
zeta	Hurwitz zeta function.
zetac	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.lambertw

cupyx.scipy.special.**lambertw**(*z*, *k=0*, *tol=1e-08*)

Lambert W function.

See also:

`scipy.special.lambertw()`

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

cbrt	Cube root.
exp10	Computes `10**x`.
exp2	Computes `2**x`.
radian	Degrees, minutes, seconds to radians:
cosdg	Cosine of x with x in degrees.
sindg	Sine of x with x in degrees.
tandg	Tangent of x with x in degrees.
cotdg	Cotangent of x with x in degrees.
log1p	Elementwise function for scipy.special.log1p
expm1	Computes `exp(x) - 1`.
cosm1	Computes `cos(x) - 1`.
round(a[, decimals, out])	

xlogy	Compute `x*log(y)` so that the result is 0 if `x = 0`.
xlog1py	Compute `x*log1p(y)` so that the result is 0 if `x = 0`.
logsumexp(a[, axis, b, keepdims, return_sign])	Compute the log of the sum of exponentials of input elements.
sinc(x, /[, out, casting, dtype])	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*log(y)` so that the result is 0 if `x = 0`.

Parameters
x (cupy.ndarray) – input data

Returns
values of `x * log(y)`

Return type
cupy.ndarray

See also:

scipy.special.xlogy

cupyx.scipy.special.xlog1py

cupyx.scipy.special.**xlog1py**()

Compute `x*log1p(y)` so that the result is 0 if `x = 0`.

Parameters
x (cupy.ndarray) – input data

Returns
values of `x * log1p(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, `cp.log(cp.sum(cp.exp(a)))` calculated in a numerically more stable way. If *b* is given then `cp.log(cp.sum(b*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

trim_mean(a, proportiontocut[, axis])	Return mean of array after trimming distribution from both tails.
entropy(pk[, qk, base, axis])	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

boxcox_llf(lmb, data)	The boxcox log-likelihood function.
zmap(scores, compare[, axis, ddof, nan_policy])	Calculate the relative z-scores.
zscore(a[, axis, ddof, nan_policy])	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 arrsy *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.

cupyx.rsqrt	Returns the reciprocal square root.
cupyx.scatter_add(a, slices, value)	Adds given values to specified elements of an array.
cupyx.scatter_max(a, slices, value)	Stores a maximum value of elements specified by indices to an array.
cupyx.scatter_min(a, slices, value)	Stores a minimum value of elements specified by indices to an array.
cupyx.empty_pinned(shape[, dtype, order])	Returns a new, uninitialized NumPy array with the given shape and dtype.
cupyx.empty_like_pinned(a[, dtype, order, ...])	Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.
cupyx.zeros_pinned(shape[, dtype, order])	Returns a new, zero-initialized NumPy array with the given shape and dtype.
cupyx.zeros_like_pinned(a[, dtype, order, ...])	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.

cupyx.signal.channelize_poly(x, h, n_chans)	Polyphase channelize signal into n channels
cupyx.signal.convolve1d3o(in1, in2[, mode, ...])	Convolve a 1-dimensional array with a 3rd order filter.
cupyx.signal.pulse_compression(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)
cupyx.signal.pulse_doppler(x[, window, nfft])	Pulse doppler processing yields a range/doppler data matrix that represents moving target data that's separated from clutter.
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.
cupyx.signal.ca_cfar(array, guard_cells, ...)	Computes the cell-averaged constant false alarm rate (CA CFAR) detector threshold and returns for a given array.
cupyx.signal.freq_shift(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 referenc 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

cupyx.profiler.benchmark(func[, args, ...])	Timing utility for measuring time spent by both CPU and GPU.
cupyx.profiler.time_range([message, ...])	Mark function calls with ranges using NVTX/rocTX.
cupyx.profiler.profile()	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 argumens 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):
...     # 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*.

cupy.from_dlpack(array)	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`)

cupyx.optimizing.optimize(*[, key, path, ...])	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

cupy.get_default_memory_pool()	Returns CuPy default memory pool for GPU memory.
cupy.get_default_pinned_memory_pool()	Returns CuPy default memory pool for pinned memory.
cupy.cuda.Memory(size_t size)	Memory allocation on a CUDA device.
cupy.cuda.MemoryAsync(size_t size, stream)	Asynchronous memory allocation on a CUDA device.
cupy.cuda.ManagedMemory(size_t size)	Managed memory (Unified memory) allocation on a CUDA device.
cupy.cuda.UnownedMemory(intptr_t ptr, ...)	CUDA memory that is not owned by CuPy.
cupy.cuda.PinnedMemory(size[, flags])	Pinned memory allocation on host.
cupy.cuda.MemoryPointer(BaseMemory mem, ...)	Pointer to a point on a device memory.
cupy.cuda.PinnedMemoryPointer(mem, ...)	Pointer of a pinned memory.
cupy.cuda.malloc_managed(size_t size)	Allocate managed memory (unified memory).
cupy.cuda.malloc_async(size_t size)	(Experimental) Allocate memory from Stream Ordered Memory Allocator.
cupy.cuda.alloc(size)	Calls the current allocator.
cupy.cuda.alloc_pinned_memory(size_t size)	Calls the current allocator.
cupy.cuda.get_allocator()	Returns the current allocator for GPU memory.
cupy.cuda.set_allocator([allocator])	Sets the current allocator for GPU memory.
cupy.cuda.using_allocator([allocator])	Sets a thread-local allocator for GPU memory inside
cupy.cuda.set_pinned_memory_allocator([...])	Sets the current allocator for the pinned memory.
cupy.cuda.MemoryPool([allocator])	Memory pool for all GPU devices on the host.
cupy.cuda.MemoryAsyncPool([pool_handles])	(Experimental) CUDA memory pool for all GPU devices on the host.
cupy.cuda.PinnedMemoryPool([allocator])	Memory pool for pinned memory on the host.
cupy.cuda.PythonFunctionAllocator(...)	Allocator with python functions to perform memory allocation.
cupy.cuda.CFunctionAllocator(intptr_t param, ...)	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 preferrable.

Read more at: 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 >= 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

cupy.cuda.MemoryHook()	Base class of hooks for Memory allocations.
cupy.cuda.memory_hooks.DebugPrintHook([...])	Memory hook that prints debug information.
cupy.cuda.memory_hooks.LineProfileHook([...])	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 descirbe 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 descirbe 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

cupy.cuda.Stream([null, non_blocking, ptds, ...])	CUDA stream.
cupy.cuda.ExternalStream(ptr[, device_id])	CUDA stream not managed by CuPy.
cupy.cuda.get_current_stream(int device_id=-1)	Gets the current CUDA stream for the specified CUDA device.
cupy.cuda.Event([block, disable_timing, ...])	CUDA event, a synchronization point of CUDA streams.
cupy.cuda.get_elapsed_time(start_event, ...)	Gets the elapsed time between two events.

cupy.cuda.Stream

class cupy.cuda.**Stream**(*null=False*, *non_blocking=False*, *ptds=False*, *priority=None*)
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.
- **priority** (*int*) – Priority of the stream. Lower numbers represent higher priorities.

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.

is_non_blocking

True if the stream is non_blocking. False indicates the default stream creation flag.

null = <Stream 0 (device -1)>

priority

Query the priority of a stream.

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.

is_non_blocking

True if the stream is non_blocking. False indicates the default stream creation flag.

priority

Query the priority of a stream.

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

cupy.cuda.texture.ChannelFormatDescriptor(...)	A class that holds the channel format description.
cupy.cuda.texture.CUDAarray(...)	Allocate a CUDA array (*cudaArray_t*) that can be used as texture memory.
cupy.cuda.texture.ResourceDescriptor(...)	A class that holds the resource description.
cupy.cuda.texture.TextureDescriptor([...])	A class that holds the texture description.
cupy.cuda.texture.TextureObject(...)	A class that holds a texture object.
cupy.cuda.texture.SurfaceObject(...)	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 intance 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

cupy.cuda.nvtx.Mark(message, int id_color=-1)	Marks an instantaneous event (marker) in the application.
cupy.cuda.nvtx.MarkC(message, uint32_t color=0)	Marks an instantaneous event (marker) in the application.
cupy.cuda.nvtx.RangePush(message, ...)	Starts a nested range.
cupy.cuda.nvtx.RangePushC(message, ...)	Starts a nested range.
cupy.cuda.nvtx.RangePop()	Ends a nested range started by a `RangePush*()` 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

cupy.cuda.nccl.NcclCommunicator(int ndev, ...)	Initialize an NCCL communicator for one device controlled by one process.
cupy.cuda.nccl.get_build_version()	
cupy.cuda.nccl.get_version()	Returns the runtime version of NCCL.
cupy.cuda.nccl.get_unique_id()	
cupy.cuda.nccl.groupStart()	Start a group of NCCL calls.
cupy.cuda.nccl.groupEnd()	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 recvcount*, *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

cupy.cuda.runtime.getDevice()	
cupy.cuda.runtime.getDeviceProperties(int device)	
cupy.cuda.runtime.deviceGetAttribute(...)	
cupy.cuda.runtime.deviceGetByPCIBusId(...)	
cupy.cuda.runtime.deviceGetPCIBusId(int device)	
cupy.cuda.runtime.deviceGetDefaultMemPool(...)	Get the default mempool on the current device.
cupy.cuda.runtime.deviceGetMemPool(int device)	Get the current mempool on the current device.
cupy.cuda.runtime.deviceSetMemPool(...)	Set the current mempool on the current device to pool.
cupy.cuda.runtime.memPoolCreate(...)	
cupy.cuda.runtime.memPoolDestroy(intptr_t pool)	
cupy.cuda.runtime.memPoolTrimTo(...)	
cupy.cuda.runtime.getDeviceCount()	
cupy.cuda.runtime.setDevice(int device)	
cupy.cuda.runtime.deviceSynchronize()	
cupy.cuda.runtime.deviceCanAccessPeer(...)	
cupy.cuda.runtime.deviceEnablePeerAccess(...)	
cupy.cuda.runtime.deviceGetLimit(int limit)	
cupy.cuda.runtime.deviceSetLimit(int limit, ...)	
cupy.cuda.runtime.malloc(size_t size)	
cupy.cuda.runtime.mallocManaged(size_t size, ...)	
cupy.cuda.runtime.malloc3DArray(...)	
cupy.cuda.runtime.mallocArray(...)	
cupy.cuda.runtime.mallocAsync(size_t size, ...)	
cupy.cuda.runtime.mallocFromPoolAsync(...)	
cupy.cuda.runtime.hostAlloc(size_t size, ...)	
cupy.cuda.runtime.hostRegister(intptr_t ptr, ...)	

continues on next page

Table 3 – continued from previous page

cupy.cuda.runtime.hostUnregister(intptr_t ptr)
cupy.cuda.runtime.free(intptr_t ptr)
cupy.cuda.runtime.freeHost(intptr_t ptr)
cupy.cuda.runtime.freeArray(intptr_t ptr)
cupy.cuda.runtime.freeAsync(intptr_t ptr, ...)
cupy.cuda.runtime.memGetInfo()
cupy.cuda.runtime.memcpy(intptr_t dst, ...)
cupy.cuda.runtime.memcpyAsync(intptr_t dst, ...)
cupy.cuda.runtime.memcpyPeer(intptr_t dst, ...)
cupy.cuda.runtime.memcpyPeerAsync(...)
cupy.cuda.runtime.memcpy2D(intptr_t dst, ...)
cupy.cuda.runtime.memcpy2DAsync(...)
cupy.cuda.runtime.memcpy2DFromArray(...)
cupy.cuda.runtime.memcpy2DFromArrayAsync(...)
cupy.cuda.runtime.memcpy2DToArray(...)
cupy.cuda.runtime.memcpy2DToArrayAsync(...)
cupy.cuda.runtime.memcpy3D(...)
cupy.cuda.runtime.memcpy3DAsync(...)
cupy.cuda.runtime.memset(intptr_t ptr, ...)
cupy.cuda.runtime.memsetAsync(intptr_t ptr, ...)
cupy.cuda.runtime.memPrefetchAsync(...)
cupy.cuda.runtime.memAdvise(intptr_t devPtr, ...)
cupy.cuda.runtime.pointerGetAttributes(...)
cupy.cuda.runtime.streamCreate()
cupy.cuda.runtime.streamCreateWithFlags(...)

continues on next page

Table 3 – continued from previous page

cupy.cuda.runtime.streamCreateWithPriority(...)	
cupy.cuda.runtime.streamDestroy(intptr_t stream)	
cupy.cuda.runtime.streamSynchronize(...)	
cupy.cuda.runtime.streamAddCallback(...)	
cupy.cuda.runtime.streamQuery(intptr_t stream)	
cupy.cuda.runtime.streamWaitEvent(...)	
cupy.cuda.runtime.launchHostFunc(...)	
cupy.cuda.runtime.eventCreate()	
cupy.cuda.runtime.eventCreateWithFlags(...)	
cupy.cuda.runtime.eventDestroy(intptr_t event)	
cupy.cuda.runtime.eventElapsedTime(...)	
cupy.cuda.runtime.eventQuery(intptr_t event)	
cupy.cuda.runtime.eventRecord(...)	
cupy.cuda.runtime.eventSynchronize(...)	
cupy.cuda.runtime.ipcGetMemHandle(...)	
cupy.cuda.runtime.ipcOpenMemHandle(...)	
cupy.cuda.runtime.ipcCloseMemHandle(...)	
cupy.cuda.runtime.ipcGetEventHandle(...)	
cupy.cuda.runtime.ipcOpenEventHandle(...)	
cupy.cuda.runtime.profilerStart()	Enable profiling.
cupy.cuda.runtime.profilerStop()	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.streamCreateWithPriority

cupy.cuda.runtime.**streamCreateWithPriority**(*unsigned int flags*, *int priority*) → 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

cupy.ElementwiseKernel(in_params, ...[, ...])	User-defined elementwise kernel.
cupy.ReductionKernel(unicode in_params, ...)	User-defined reduction kernel.
cupy.RawKernel(unicode code, unicode name, ...)	User-defined custom kernel.
cupy.RawModule(unicode code=None, *, ...[, ...])	User-defined custom module.
cupy.fuse(*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.
- **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 proprety 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 proprety 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 proprety 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 proprety 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 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 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

cupyx.jit.threadIdx	dim3 threadIdx
cupyx.jit.blockDim	dim3 blockDim
cupyx.jit.blockIdx	dim3 blockIdx
cupyx.jit.gridDim	dim3 gridDim
cupyx.jit.grid(ndim)	Compute the thread index in the grid.
cupyx.jit.gridsize(ndim)	Compute the grid size.
cupyx.jit.laneid()	Returns the lane ID of the calling thread, ranging in `[0, jit.warpsize)`.
cupyx.jit.warpsize	Returns the number of threads in a warp.
cupyx.jit.range(*args[, unroll])	Range with loop unrolling support.
cupyx.jit.syncthreads()	Calls `__syncthreads()`.
cupyx.jit.syncwarp(*[, mask])	Calls `__syncwarp()`.
cupyx.jit.shfl_sync(mask, var, val_id, *[, ...])	Calls the `__shfl_sync` function.
cupyx.jit.shfl_up_sync(mask, var, val_id, *)	Calls the `__shfl_up_sync` function.
cupyx.jit.shfl_down_sync(mask, var, val_id, *)	Calls the `__shfl_down_sync` function.
cupyx.jit.shfl_xor_sync(mask, var, val_id, *)	Calls the `__shfl_xor_sync` function.
cupyx.jit.shared_memory(dtype, size[, alignment])	Allocates shared memory and returns it as a 1-D array.
cupyx.jit.atomic_add(array, index, value[, ...])	Calls the `atomicAdd` function to operate atomically on `array[index]`.
cupyx.jit.atomic_sub(array, index, value[, ...])	Calls the `atomicSub` function to operate atomically on `array[index]`.
cupyx.jit.atomic_exch(array, index, value[, ...])	Calls the `atomicExch` function to operate atomically on `array[index]`.
cupyx.jit.atomic_min(array, index, value[, ...])	Calls the `atomicMin` function to operate atomically on `array[index]`.
cupyx.jit.atomic_max(array, index, value[, ...])	Calls the `atomicMax` function to operate atomically on `array[index]`.
cupyx.jit.atomic_inc(array, index, value[, ...])	Calls the `atomicInc` function to operate atomically on `array[index]`.
cupyx.jit.atomic_dec(array, index, value[, ...])	Calls the `atomicDec` function to operate atomically on `array[index]`.
cupyx.jit.atomic_cas(array, index, value[, ...])	Calls the `atomicCAS` function to operate atomically on `array[index]`.
cupyx.jit.atomic_and(array, index, value[, ...])	Calls the `atomicAnd` function to operate atomically on `array[index]`.
cupyx.jit.atomic_or(array, index, value[, ...])	Calls the `atomicOr` function to operate atomically on `array[index]`.
cupyx.jit.atomic_xor(array, index, value[, ...])	Calls the `atomicXor` function to operate atomically on `array[index]`.
cupyx.jit.cg.this_grid()	Returns the current grid group (`_GridGroup`).
cupyx.jit.cg.this_thread_block()	Returns the current thread block group (`_ThreadBlockGroup`).
cupyx.jit.cg.sync(group)	Calls `cg::sync()`.
cupyx.jit.cg.memcpy_async(group, dst, ...[, ...])	Calls `cg::memcpy_sync()`.
cupyx.jit.cg.wait(group)	Calls `cg::wait()`.
cupyx.jit.cg.wait_prior(group)	Calls `cg::wait_prior<N>()`.
cupyx.jit._interface._JitRawKernel(func, ...)	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 inteded 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 proprety 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 proprety method is for debugging purpose. The return value is not guaranteed to keep backward compatibility.

5.7.7 Kernel binary memoization

cupy.memoize(bool for_each_device=False)	Makes a function memoizing the result for each argument and device.
cupy.clear_memo()	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

init_process_group(n_devices, rank, *[, ...])	Start *cupyx.distributed* and obtain a communicator.
NCCLBackend(n_devices, rank[, host, port, ...])	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 appropiated 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* <= *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* <= *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 with 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 with 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 with 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 with 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

distributed_array(array, index_map[, mode])	Creates a distributed array from the given data.
DistributedArray(self, shape, dtype, chunks_map)	Multi-dimensional array distributed across multiple CUDA devices.
make_2d_index_map(i_partitions, ...)	Create an `index_map` for a 2D matrix with a specified blocking.
matmul(a, b[, out])	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 ignoed 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
numpy.DataSource	cupy.DataSource (*alias of* numpy.DataSource)
numpy.ScalarType	-
numpy.abs	cupy.abs
numpy.absolute	*cupy.absolute*
numpy.add	*cupy.add*
numpy.all	*cupy.all*
numpy.allclose	*cupy.allclose*
numpy.alltrue	cupy.alltrue
numpy.amax	*cupy.amax*
numpy.amin	*cupy.amin*
numpy.angle	*cupy.angle*
numpy.any	*cupy.any*
numpy.append	*cupy.append*
numpy.apply_along_axis	*cupy.apply_along_axis*
numpy.apply_over_axes	-
numpy.arange	*cupy.arange*
numpy.arccos	*cupy.arccos*
numpy.arccosh	*cupy.arccosh*
numpy.arcsin	*cupy.arcsin*
numpy.arcsinh	*cupy.arcsinh*
numpy.arctan	*cupy.arctan*
numpy.arctan2	*cupy.arctan2*
numpy.arctanh	*cupy.arctanh*
numpy.argmax	*cupy.argmax*
numpy.argmin	*cupy.argmin*
numpy.argpartition	*cupy.argpartition*
numpy.argsort	*cupy.argsort*
numpy.argwhere	*cupy.argwhere*
numpy.around	*cupy.around*
numpy.array	*cupy.array*
numpy.array2string	*cupy.array2string*
numpy.array_equal	*cupy.array_equal*
numpy.array_equiv	*cupy.array_equiv*
numpy.array_repr	*cupy.array_repr*
numpy.array_split	*cupy.array_split*
numpy.array_str	*cupy.array_str*
numpy.asanyarray	*cupy.asanyarray*
numpy.asarray	*cupy.asarray*
numpy.asarray_chkfinite	*cupy.asarray_chkfinite*

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.ascontiguousarray	*cupy.ascontiguousarray*
numpy.asfarray	*cupy.asfarray*
numpy.asfortranarray	*cupy.asfortranarray*
numpy.asmatrix	-[1]
numpy.atleast_1d	*cupy.atleast_1d*
numpy.atleast_2d	*cupy.atleast_2d*
numpy.atleast_3d	*cupy.atleast_3d*
numpy.average	*cupy.average*
numpy.bartlett	*cupy.bartlett*
numpy.base_repr	*cupy.base_repr*
numpy.binary_repr	*cupy.binary_repr*
numpy.bincount	*cupy.bincount*
numpy.bitwise_and	*cupy.bitwise_and*
numpy.bitwise_not	cupy.bitwise_not
numpy.bitwise_or	*cupy.bitwise_or*
numpy.bitwise_xor	*cupy.bitwise_xor*
numpy.blackman	*cupy.blackman*
numpy.block	-
numpy.bmat	-[Page 901, 1]
numpy.bool_	cupy.bool_ (*alias of* numpy.bool_)
numpy.broadcast	*cupy.broadcast*
numpy.broadcast_arrays	*cupy.broadcast_arrays*
numpy.broadcast_shapes	cupy.broadcast_shapes (*alias of* numpy.broadcast_shapes)
numpy.broadcast_to	*cupy.broadcast_to*
numpy.busday_count	-[2]
numpy.busday_offset	-[Page 901, 2]
numpy.busdaycalendar	-[Page 901, 2]
numpy.byte	cupy.byte (*alias of* numpy.byte)
numpy.byte_bounds	*cupy.byte_bounds*
numpy.bytes_	-[3]
numpy.c_	*cupy.c_*
numpy.can_cast	*cupy.can_cast*
numpy.cast	-
numpy.cbrt	*cupy.cbrt*
numpy.cdouble	cupy.cdouble (*alias of* numpy.cdouble)
numpy.ceil	*cupy.ceil*
numpy.cfloat	cupy.cfloat (*alias of* numpy.cfloat)
numpy.character	-[Page 901, 3]
numpy.chararray	-[Page 901, 3]
numpy.choose	*cupy.choose*
numpy.clip	*cupy.clip*
numpy.clongdouble	-
numpy.clongfloat	-
numpy.column_stack	*cupy.column_stack*
numpy.common_type	*cupy.common_type*
numpy.compare_chararrays	-[Page 901, 3]
numpy.complex128	cupy.complex128 (*alias of* numpy.complex128)
numpy.complex256	-

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.complex64	cupy.complex64 (*alias of* numpy.complex64)
numpy.complex_	cupy.complex_ (*alias of* numpy.complex_)
numpy.complexfloating	cupy.complexfloating (*alias of* numpy.complexfloating)
numpy.compress	*cupy.compress*
numpy.concatenate	*cupy.concatenate*
numpy.conj	*cupy.conj*
numpy.conjugate	*cupy.conjugate*
numpy.convolve	*cupy.convolve*
numpy.copy	*cupy.copy*
numpy.copysign	*cupy.copysign*
numpy.copyto	*cupy.copyto*
numpy.corrcoef	*cupy.corrcoef*
numpy.correlate	*cupy.correlate*
numpy.cos	*cupy.cos*
numpy.cosh	*cupy.cosh*
numpy.count_nonzero	*cupy.count_nonzero*
numpy.cov	*cupy.cov*
numpy.cross	*cupy.cross*
numpy.csingle	cupy.csingle (*alias of* numpy.csingle)
numpy.cumprod	*cupy.cumprod*
numpy.cumproduct	cupy.cumproduct
numpy.cumsum	*cupy.cumsum*
numpy.datetime64	_[Page 901, 2]
numpy.datetime_as_string	_[Page 901, 2]
numpy.datetime_data	_[Page 901, 2]
numpy.deg2rad	*cupy.deg2rad*
numpy.degrees	*cupy.degrees*
numpy.delete	*cupy.delete*
numpy.deprecate	-
numpy.deprecate_with_doc	-
numpy.diag	*cupy.diag*
numpy.diag_indices	*cupy.diag_indices*
numpy.diag_indices_from	*cupy.diag_indices_from*
numpy.diagflat	*cupy.diagflat*
numpy.diagonal	*cupy.diagonal*
numpy.diff	*cupy.diff*
numpy.digitize	*cupy.digitize*
numpy.disp	cupy.disp (*alias of* numpy.disp)
numpy.divide	*cupy.divide*
numpy.divmod	*cupy.divmod*
numpy.dot	*cupy.dot*
numpy.double	cupy.double (*alias of* numpy.double)
numpy.dsplit	*cupy.dsplit*
numpy.dstack	*cupy.dstack*
numpy.dtype	cupy.dtype (*alias of* numpy.dtype)
numpy.ediff1d	*cupy.ediff1d*
numpy.einsum	*cupy.einsum*
numpy.einsum_path	-

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.empty	*cupy.empty*
numpy.empty_like	*cupy.empty_like*
numpy.equal	*cupy.equal*
numpy.errstate	-[4]
numpy.exp	*cupy.exp*
numpy.exp2	*cupy.exp2*
numpy.expand_dims	*cupy.expand_dims*
numpy.expm1	*cupy.expm1*
numpy.extract	*cupy.extract*
numpy.eye	*cupy.eye*
numpy.fabs	*cupy.fabs*
numpy.fill_diagonal	*cupy.fill_diagonal*
numpy.find_common_type	cupy.find_common_type (*alias of* numpy.find_common_type)
numpy.finfo	cupy.finfo (*alias of* numpy.finfo)
numpy.fix	*cupy.fix*
numpy.flatiter	*cupy.flatiter*
numpy.flatnonzero	*cupy.flatnonzero*
numpy.flexible	-[Page 901, 3]
numpy.flip	*cupy.flip*
numpy.fliplr	*cupy.fliplr*
numpy.flipud	*cupy.flipud*
numpy.float128	-
numpy.float16	cupy.float16 (*alias of* numpy.float16)
numpy.float32	cupy.float32 (*alias of* numpy.float32)
numpy.float64	cupy.float64 (*alias of* numpy.float64)
numpy.float_	cupy.float_ (*alias of* numpy.float_)
numpy.float_power	*cupy.float_power*
numpy.floating	cupy.floating (*alias of* numpy.floating)
numpy.floor	*cupy.floor*
numpy.floor_divide	*cupy.floor_divide*
numpy.fmax	*cupy.fmax*
numpy.fmin	*cupy.fmin*
numpy.fmod	*cupy.fmod*
numpy.format_float_positional	*cupy.format_float_positional*
numpy.format_float_scientific	*cupy.format_float_scientific*
numpy.format_parser	cupy.format_parser (*alias of* numpy.format_parser)
numpy.frexp	*cupy.frexp*
numpy.from_dlpack	*cupy.from_dlpack*
numpy.frombuffer	*cupy.frombuffer*
numpy.fromfile	*cupy.fromfile*
numpy.fromfunction	*cupy.fromfunction*
numpy.fromiter	*cupy.fromiter*
numpy.frompyfunc	-
numpy.fromregex	-[5]
numpy.fromstring	*cupy.fromstring*
numpy.full	*cupy.full*
numpy.full_like	*cupy.full_like*

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.gcd	*cupy.gcd*
numpy.generic	cupy.generic (*alias of* numpy.generic)
numpy.genfromtxt	*cupy.genfromtxt*
numpy.geomspace	-
numpy.get_array_wrap	cupy.get_array_wrap (*alias of* numpy.get_array_wrap)
numpy.get_include	-
numpy.get_printoptions	cupy.get_printoptions (*alias of* numpy.get_printoptions)
numpy.getbufsize	-
numpy.geterr	_Page 901, 4
numpy.geterrcall	_Page 901, 4
numpy.geterrobj	_Page 901, 4
numpy.gradient	*cupy.gradient*
numpy.greater	*cupy.greater*
numpy.greater_equal	*cupy.greater_equal*
numpy.half	cupy.half (*alias of* numpy.half)
numpy.hamming	*cupy.hamming*
numpy.hanning	*cupy.hanning*
numpy.heaviside	*cupy.heaviside*
numpy.histogram	*cupy.histogram*
numpy.histogram2d	*cupy.histogram2d*
numpy.histogram_bin_edges	-
numpy.histogramdd	*cupy.histogramdd*
numpy.hsplit	*cupy.hsplit*
numpy.hstack	*cupy.hstack*
numpy.hypot	*cupy.hypot*
numpy.i0	*cupy.i0*
numpy.identity	*cupy.identity*
numpy.iinfo	cupy.iinfo (*alias of* numpy.iinfo)
numpy.imag	*cupy.imag*
numpy.in1d	*cupy.in1d*
numpy.index_exp	cupy.index_exp (*alias of* numpy.index_exp)
numpy.indices	*cupy.indices*
numpy.inexact	cupy.inexact (*alias of* numpy.inexact)
numpy.info	-
numpy.inner	*cupy.inner*
numpy.insert	-
numpy.int16	cupy.int16 (*alias of* numpy.int16)
numpy.int32	cupy.int32 (*alias of* numpy.int32)
numpy.int64	cupy.int64 (*alias of* numpy.int64)
numpy.int8	cupy.int8 (*alias of* numpy.int8)
numpy.int_	cupy.int_ (*alias of* numpy.int_)
numpy.intc	cupy.intc (*alias of* numpy.intc)
numpy.integer	cupy.integer (*alias of* numpy.integer)
numpy.interp	*cupy.interp*
numpy.intersect1d	*cupy.intersect1d*
numpy.intp	cupy.intp (*alias of* numpy.intp)
numpy.invert	*cupy.invert*

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.is_busday	_Page 901, 2
numpy.isclose	*cupy.isclose*
numpy.iscomplex	*cupy.iscomplex*
numpy.iscomplexobj	*cupy.iscomplexobj*
numpy.isfinite	*cupy.isfinite*
numpy.isfortran	*cupy.isfortran*
numpy.isin	*cupy.isin*
numpy.isinf	*cupy.isinf*
numpy.isnan	*cupy.isnan*
numpy.isnat	_Page 901, 2
numpy.isneginf	*cupy.isneginf*
numpy.isposinf	*cupy.isposinf*
numpy.isreal	*cupy.isreal*
numpy.isrealobj	*cupy.isrealobj*
numpy.isscalar	*cupy.isscalar*
numpy.issctype	cupy.issctype (*alias of* numpy.issctype)
numpy.issubclass_	cupy.issubclass_ (*alias of* numpy.issubclass_)
numpy.issubdtype	cupy.issubdtype (*alias of* numpy.issubdtype)
numpy.issubsctype	cupy.issubsctype (*alias of* numpy.issubsctype)
numpy.iterable	cupy.iterable (*alias of* numpy.iterable)
numpy.ix_	*cupy.ix_*
numpy.kaiser	*cupy.kaiser*
numpy.kron	*cupy.kron*
numpy.lcm	*cupy.lcm*
numpy.ldexp	*cupy.ldexp*
numpy.left_shift	*cupy.left_shift*
numpy.less	*cupy.less*
numpy.less_equal	*cupy.less_equal*
numpy.lexsort	*cupy.lexsort*
numpy.linspace	*cupy.linspace*
numpy.load	*cupy.load*
numpy.loadtxt	*cupy.loadtxt*
numpy.log	*cupy.log*
numpy.log10	*cupy.log10*
numpy.log1p	*cupy.log1p*
numpy.log2	*cupy.log2*
numpy.logaddexp	*cupy.logaddexp*
numpy.logaddexp2	*cupy.logaddexp2*
numpy.logical_and	*cupy.logical_and*
numpy.logical_not	*cupy.logical_not*
numpy.logical_or	*cupy.logical_or*
numpy.logical_xor	*cupy.logical_xor*
numpy.logspace	*cupy.logspace*
numpy.longcomplex	-
numpy.longdouble	-
numpy.longfloat	cupy.longfloat (*alias of* numpy.longfloat)
numpy.longlong	cupy.longlong (*alias of* numpy.longlong)
numpy.lookfor	-
numpy.mask_indices	*cupy.mask_indices*

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.mat	_Page 901, 1
numpy.matmul	*cupy.matmul*
numpy.matrix	_Page 901, 1
numpy.max	cupy.max
numpy.maximum	*cupy.maximum*
numpy.maximum_sctype	-
numpy.may_share_memory	*cupy.may_share_memory*
numpy.mean	*cupy.mean*
numpy.median	*cupy.median*
numpy.memmap	-
numpy.meshgrid	*cupy.meshgrid*
numpy.mgrid	*cupy.mgrid*
numpy.min	cupy.min
numpy.min_scalar_type	*cupy.min_scalar_type*
numpy.minimum	*cupy.minimum*
numpy.mintypecode	cupy.mintypecode (*alias of* numpy.mintypecode)
numpy.mod	*cupy.mod*
numpy.modf	*cupy.modf*
numpy.moveaxis	*cupy.moveaxis*
numpy.msort	*cupy.msort*
numpy.multiply	*cupy.multiply*
numpy.nan_to_num	*cupy.nan_to_num*
numpy.nanargmax	*cupy.nanargmax*
numpy.nanargmin	*cupy.nanargmin*
numpy.nancumprod	*cupy.nancumprod*
numpy.nancumsum	*cupy.nancumsum*
numpy.nanmax	*cupy.nanmax*
numpy.nanmean	*cupy.nanmean*
numpy.nanmedian	*cupy.nanmedian*
numpy.nanmin	*cupy.nanmin*
numpy.nanpercentile	-
numpy.nanprod	*cupy.nanprod*
numpy.nanquantile	-
numpy.nanstd	*cupy.nanstd*
numpy.nansum	*cupy.nansum*
numpy.nanvar	*cupy.nanvar*
numpy.nbytes	-
numpy.ndarray	*cupy.ndarray*
numpy.ndenumerate	-
numpy.ndim	cupy.ndim
numpy.ndindex	cupy.ndindex (*alias of* numpy.ndindex)
numpy.nditer	-
numpy.negative	*cupy.negative*
numpy.nested_iters	-
numpy.newaxis	cupy.newaxis (*alias of* numpy.newaxis)
numpy.nextafter	*cupy.nextafter*
numpy.nonzero	*cupy.nonzero*
numpy.not_equal	*cupy.not_equal*
numpy.number	cupy.number (*alias of* numpy.number)

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.obj2sctype	cupy.obj2sctype (*alias of* numpy.obj2sctype)
numpy.object_	_Page 901, 3
numpy.ogrid	*cupy.ogrid*
numpy.ones	*cupy.ones*
numpy.ones_like	*cupy.ones_like*
numpy.outer	*cupy.outer*
numpy.packbits	*cupy.packbits*
numpy.pad	*cupy.pad*
numpy.partition	*cupy.partition*
numpy.percentile	*cupy.percentile*
numpy.piecewise	*cupy.piecewise*
numpy.place	*cupy.place*
numpy.poly	*cupy.poly*[6]
numpy.poly1d	*cupy.poly1d*
numpy.polyadd	*cupy.polyadd*
numpy.polyder	_Page 901, 6
numpy.polydiv	_Page 901, 6
numpy.polyfit	*cupy.polyfit*
numpy.polyint	_Page 901, 6
numpy.polymul	*cupy.polymul*
numpy.polysub	*cupy.polysub*
numpy.polyval	*cupy.polyval*
numpy.positive	*cupy.positive*
numpy.power	*cupy.power*
numpy.printoptions	cupy.printoptions (*alias of* numpy.printoptions)
numpy.prod	*cupy.prod*
numpy.product	cupy.product
numpy.promote_types	cupy.promote_types (*alias of* numpy.promote_types)
numpy.ptp	*cupy.ptp*
numpy.put	*cupy.put*
numpy.put_along_axis	*cupy.put_along_axis*
numpy.putmask	*cupy.putmask*
numpy.quantile	*cupy.quantile*
numpy.r_	*cupy.r_*
numpy.rad2deg	*cupy.rad2deg*
numpy.radians	*cupy.radians*
numpy.ravel	*cupy.ravel*
numpy.ravel_multi_index	*cupy.ravel_multi_index*
numpy.real	*cupy.real*
numpy.real_if_close	*cupy.real_if_close*
numpy.recarray	_Page 901, 5
numpy.recfromcsv	_Page 901, 5
numpy.recfromtxt	_Page 901, 5
numpy.reciprocal	*cupy.reciprocal*
numpy.record	_Page 901, 5
numpy.remainder	*cupy.remainder*
numpy.repeat	*cupy.repeat*

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.require	*cupy.require*
numpy.reshape	*cupy.reshape*
numpy.resize	*cupy.resize*
numpy.result_type	*cupy.result_type*
numpy.right_shift	*cupy.right_shift*
numpy.rint	*cupy.rint*
numpy.roll	*cupy.roll*
numpy.rollaxis	*cupy.rollaxis*
numpy.roots	*cupy.roots*
numpy.rot90	*cupy.rot90*
numpy.round	cupy.round
numpy.round_	*cupy.round_*
numpy.row_stack	*cupy.row_stack*
numpy.s_	cupy.s_ (*alias of* numpy.s_)
numpy.safe_eval	cupy.safe_eval (*alias of* numpy.safe_eval)
numpy.save	*cupy.save*
numpy.savetxt	*cupy.savetxt*
numpy.savez	*cupy.savez*
numpy.savez_compressed	*cupy.savez_compressed*
numpy.sctype2char	cupy.sctype2char (*alias of* numpy.sctype2char)
numpy.sctypeDict	-
numpy.sctypes	-
numpy.searchsorted	*cupy.searchsorted*
numpy.select	*cupy.select*
numpy.set_numeric_ops	-[7]
numpy.set_printoptions	cupy.set_printoptions (*alias of* numpy.set_printoptions)
numpy.set_string_function	cupy.set_string_function (*alias of* numpy.set_string_function)
numpy.setbufsize	-
numpy.setdiff1d	*cupy.setdiff1d*
numpy.seterr	_[Page 901, 4]
numpy.seterrcall	_[Page 901, 4]
numpy.seterrobj	_[Page 901, 4]
numpy.setxor1d	*cupy.setxor1d*
numpy.shape	*cupy.shape*
numpy.shares_memory	*cupy.shares_memory*
numpy.short	cupy.short (*alias of* numpy.short)
numpy.show_config	*cupy.show_config*
numpy.show_runtime	-
numpy.sign	*cupy.sign*
numpy.signbit	*cupy.signbit*
numpy.signedinteger	cupy.signedinteger (*alias of* numpy.signedinteger)
numpy.sin	*cupy.sin*
numpy.sinc	*cupy.sinc*
numpy.single	cupy.single (*alias of* numpy.single)
numpy.singlecomplex	cupy.singlecomplex (*alias of* numpy.singlecomplex)

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.sinh	*cupy.sinh*
numpy.size	cupy.size
numpy.sometrue	cupy.sometrue
numpy.sort	*cupy.sort*
numpy.sort_complex	*cupy.sort_complex*
numpy.source	-
numpy.spacing	-
numpy.split	*cupy.split*
numpy.sqrt	*cupy.sqrt*
numpy.square	*cupy.square*
numpy.squeeze	*cupy.squeeze*
numpy.stack	*cupy.stack*
numpy.std	*cupy.std*
numpy.str_	_Page 901, 3
numpy.string_	_Page 901, 3
numpy.subtract	*cupy.subtract*
numpy.sum	*cupy.sum*
numpy.swapaxes	*cupy.swapaxes*
numpy.take	*cupy.take*
numpy.take_along_axis	*cupy.take_along_axis*
numpy.tan	*cupy.tan*
numpy.tanh	*cupy.tanh*
numpy.tensordot	*cupy.tensordot*
numpy.tile	*cupy.tile*
numpy.timedelta64	_Page 901, 2
numpy.trace	*cupy.trace*
numpy.transpose	*cupy.transpose*
numpy.trapz	*cupy.trapz*
numpy.tri	*cupy.tri*
numpy.tril	*cupy.tril*
numpy.tril_indices	*cupy.tril_indices*
numpy.tril_indices_from	*cupy.tril_indices_from*
numpy.trim_zeros	*cupy.trim_zeros*
numpy.triu	*cupy.triu*
numpy.triu_indices	*cupy.triu_indices*
numpy.triu_indices_from	*cupy.triu_indices_from*
numpy.true_divide	*cupy.true_divide*
numpy.trunc	*cupy.trunc*
numpy.typecodes	-
numpy.typename	cupy.typename (*alias of* numpy.typename)
numpy.ubyte	cupy.ubyte (*alias of* numpy.ubyte)
numpy.ufunc	*cupy.ufunc*
numpy.uint	cupy.uint (*alias of* numpy.uint)
numpy.uint16	cupy.uint16 (*alias of* numpy.uint16)
numpy.uint32	cupy.uint32 (*alias of* numpy.uint32)
numpy.uint64	cupy.uint64 (*alias of* numpy.uint64)
numpy.uint8	cupy.uint8 (*alias of* numpy.uint8)
numpy.uintc	cupy.uintc (*alias of* numpy.uintc)
numpy.uintp	cupy.uintp (*alias of* numpy.uintp)

continues on next page

Table 5 – continued from previous page

NumPy	CuPy
numpy.ulonglong	cupy.ulonglong (*alias of* numpy.ulonglong)
numpy.unicode_	_[Page 901, 3]
numpy.union1d	*cupy.union1d*
numpy.unique	*cupy.unique*
numpy.unpackbits	*cupy.unpackbits*
numpy.unravel_index	*cupy.unravel_index*
numpy.unsignedinteger	cupy.unsignedinteger (*alias of* numpy.unsignedinteger)
numpy.unwrap	*cupy.unwrap*
numpy.ushort	cupy.ushort (*alias of* numpy.ushort)
numpy.vander	*cupy.vander*
numpy.var	*cupy.var*
numpy.vdot	*cupy.vdot*
numpy.vectorize	*cupy.vectorize*
numpy.void	_[3]
numpy.vsplit	*cupy.vsplit*
numpy.vstack	*cupy.vstack*
numpy.where	*cupy.where*
numpy.who	*cupy.who*
numpy.zeros	*cupy.zeros*
numpy.zeros_like	*cupy.zeros_like*

Multi-Dimensional Array

NumPy	CuPy
numpy.ndarray.T	*cupy.ndarray.T*
numpy.ndarray.all	*cupy.ndarray.all*
numpy.ndarray.any	*cupy.ndarray.any*
numpy.ndarray.argmax	*cupy.ndarray.argmax*
numpy.ndarray.argmin	*cupy.ndarray.argmin*
numpy.ndarray.argpartition	*cupy.ndarray.argpartition*
numpy.ndarray.argsort	*cupy.ndarray.argsort*
numpy.ndarray.astype	*cupy.ndarray.astype*
numpy.ndarray.base	*cupy.ndarray.base*
numpy.ndarray.byteswap	_[8]
numpy.ndarray.choose	*cupy.ndarray.choose*
numpy.ndarray.clip	*cupy.ndarray.clip*
numpy.ndarray.compress	*cupy.ndarray.compress*
numpy.ndarray.conj	*cupy.ndarray.conj*
numpy.ndarray.conjugate	*cupy.ndarray.conjugate*
numpy.ndarray.copy	*cupy.ndarray.copy*

continues on next page

[1] Use of numpy.matrix is discouraged in NumPy and thus we have no plan to add it to CuPy.
[2] *datetime64* and *timedelta64* dtypes are currently unsupported.
[3] *object* and string dtypes are not supported in GPU and thus left unimplemented in CuPy.
[4] Floating point error handling depends on CPU-specific features which is not available in GPU.
[5] Structured arrays and record arrays are currently unsupported.
[6] Use of numpy.poly1d is discouraged in NumPy and thus we have stopped adding functions with the interface.
[7] Not supported as it has been deprecated in NumPy.

Table 6 – continued from previous page

NumPy	CuPy
numpy.ndarray.ctypes	-
numpy.ndarray.cumprod	*cupy.ndarray.cumprod*
numpy.ndarray.cumsum	*cupy.ndarray.cumsum*
numpy.ndarray.data	*cupy.ndarray.data*
numpy.ndarray.diagonal	*cupy.ndarray.diagonal*
numpy.ndarray.dot	*cupy.ndarray.dot*
numpy.ndarray.dtype	*cupy.ndarray.dtype*
numpy.ndarray.dump	*cupy.ndarray.dump*
numpy.ndarray.dumps	*cupy.ndarray.dumps*
numpy.ndarray.fill	*cupy.ndarray.fill*
numpy.ndarray.flags	*cupy.ndarray.flags*
numpy.ndarray.flat	*cupy.ndarray.flat*
numpy.ndarray.flatten	*cupy.ndarray.flatten*
numpy.ndarray.getfield	-
numpy.ndarray.imag	*cupy.ndarray.imag*
numpy.ndarray.item	*cupy.ndarray.item*
numpy.ndarray.itemset	-
numpy.ndarray.itemsize	*cupy.ndarray.itemsize*
numpy.ndarray.max	*cupy.ndarray.max*
numpy.ndarray.mean	*cupy.ndarray.mean*
numpy.ndarray.min	*cupy.ndarray.min*
numpy.ndarray.nbytes	*cupy.ndarray.nbytes*
numpy.ndarray.ndim	*cupy.ndarray.ndim*
numpy.ndarray.newbyteorder	_Page 903, 8
numpy.ndarray.nonzero	*cupy.ndarray.nonzero*
numpy.ndarray.partition	*cupy.ndarray.partition*
numpy.ndarray.prod	*cupy.ndarray.prod*
numpy.ndarray.ptp	*cupy.ndarray.ptp*
numpy.ndarray.put	*cupy.ndarray.put*
numpy.ndarray.ravel	*cupy.ndarray.ravel*
numpy.ndarray.real	*cupy.ndarray.real*
numpy.ndarray.repeat	*cupy.ndarray.repeat*
numpy.ndarray.reshape	*cupy.ndarray.reshape*
numpy.ndarray.resize	-
numpy.ndarray.round	*cupy.ndarray.round*
numpy.ndarray.searchsorted	*cupy.ndarray.searchsorted*
numpy.ndarray.setfield	-
numpy.ndarray.setflags	-
numpy.ndarray.shape	*cupy.ndarray.shape*
numpy.ndarray.size	*cupy.ndarray.size*
numpy.ndarray.sort	*cupy.ndarray.sort*
numpy.ndarray.squeeze	*cupy.ndarray.squeeze*
numpy.ndarray.std	*cupy.ndarray.std*
numpy.ndarray.strides	*cupy.ndarray.strides*
numpy.ndarray.sum	*cupy.ndarray.sum*
numpy.ndarray.swapaxes	*cupy.ndarray.swapaxes*
numpy.ndarray.take	*cupy.ndarray.take*
numpy.ndarray.tobytes	*cupy.ndarray.tobytes*
numpy.ndarray.tofile	*cupy.ndarray.tofile*

continues on next page

Table 6 – continued from previous page

NumPy	CuPy
numpy.ndarray.tolist	*cupy.ndarray.tolist*
numpy.ndarray.tostring	_Page 901, 7
numpy.ndarray.trace	*cupy.ndarray.trace*
numpy.ndarray.transpose	*cupy.ndarray.transpose*
numpy.ndarray.var	*cupy.ndarray.var*
numpy.ndarray.view	*cupy.ndarray.view*

Linear Algebra

NumPy	CuPy
numpy.linalg.cholesky	*cupy.linalg.cholesky*
numpy.linalg.cond	-
numpy.linalg.det	*cupy.linalg.det*
numpy.linalg.eig	-
numpy.linalg.eigh	*cupy.linalg.eigh*
numpy.linalg.eigvals	-
numpy.linalg.eigvalsh	*cupy.linalg.eigvalsh*
numpy.linalg.inv	*cupy.linalg.inv*
numpy.linalg.lstsq	*cupy.linalg.lstsq*
numpy.linalg.matrix_power	*cupy.linalg.matrix_power*
numpy.linalg.matrix_rank	*cupy.linalg.matrix_rank*
numpy.linalg.multi_dot	-
numpy.linalg.norm	*cupy.linalg.norm*
numpy.linalg.pinv	*cupy.linalg.pinv*
numpy.linalg.qr	*cupy.linalg.qr*
numpy.linalg.slogdet	*cupy.linalg.slogdet*
numpy.linalg.solve	*cupy.linalg.solve*
numpy.linalg.svd	*cupy.linalg.svd*
numpy.linalg.tensorinv	*cupy.linalg.tensorinv*
numpy.linalg.tensorsolve	*cupy.linalg.tensorsolve*

[8] Not supported as GPUs only support little-endian byte-encoding.

Discrete Fourier Transform

NumPy	CuPy
numpy.fft.fft	*cupy.fft.fft*
numpy.fft.fft2	*cupy.fft.fft2*
numpy.fft.fftfreq	*cupy.fft.fftfreq*
numpy.fft.fftn	*cupy.fft.fftn*
numpy.fft.fftshift	*cupy.fft.fftshift*
numpy.fft.hfft	*cupy.fft.hfft*
numpy.fft.ifft	*cupy.fft.ifft*
numpy.fft.ifft2	*cupy.fft.ifft2*
numpy.fft.ifftn	*cupy.fft.ifftn*
numpy.fft.ifftshift	*cupy.fft.ifftshift*
numpy.fft.ihfft	*cupy.fft.ihfft*
numpy.fft.irfft	*cupy.fft.irfft*
numpy.fft.irfft2	*cupy.fft.irfft2*
numpy.fft.irfftn	*cupy.fft.irfftn*
numpy.fft.rfft	*cupy.fft.rfft*
numpy.fft.rfft2	*cupy.fft.rfft2*
numpy.fft.rfftfreq	*cupy.fft.rfftfreq*
numpy.fft.rfftn	*cupy.fft.rfftn*

Random Sampling

NumPy	CuPy
numpy.random.BitGenerator	*cupy.random.BitGenerator*
numpy.random.Generator	-
numpy.random.MT19937	-
numpy.random.PCG64	-
numpy.random.PCG64DXSM	-
numpy.random.Philox	-
numpy.random.RandomState	*cupy.random.RandomState*
numpy.random.SFC64	-
numpy.random.SeedSequence	-
numpy.random.beta	*cupy.random.beta*
numpy.random.binomial	*cupy.random.binomial*
numpy.random.bytes	*cupy.random.bytes*
numpy.random.chisquare	*cupy.random.chisquare*
numpy.random.choice	*cupy.random.choice*
numpy.random.default_rng	*cupy.random.default_rng*
numpy.random.dirichlet	*cupy.random.dirichlet*
numpy.random.exponential	*cupy.random.exponential*
numpy.random.f	*cupy.random.f*
numpy.random.gamma	*cupy.random.gamma*
numpy.random.geometric	*cupy.random.geometric*
numpy.random.get_bit_generator	-
numpy.random.get_state	-
numpy.random.gumbel	*cupy.random.gumbel*
numpy.random.hypergeometric	*cupy.random.hypergeometric*

continues on next page

Table 7 – continued from previous page

NumPy	CuPy
numpy.random.laplace	*cupy.random.laplace*
numpy.random.logistic	*cupy.random.logistic*
numpy.random.lognormal	*cupy.random.lognormal*
numpy.random.logseries	*cupy.random.logseries*
numpy.random.multinomial	*cupy.random.multinomial*
numpy.random.multivariate_normal	*cupy.random.multivariate_normal*
numpy.random.negative_binomial	*cupy.random.negative_binomial*
numpy.random.noncentral_chisquare	*cupy.random.noncentral_chisquare*
numpy.random.noncentral_f	*cupy.random.noncentral_f*
numpy.random.normal	*cupy.random.normal*
numpy.random.pareto	*cupy.random.pareto*
numpy.random.permutation	*cupy.random.permutation*
numpy.random.poisson	*cupy.random.poisson*
numpy.random.power	*cupy.random.power*
numpy.random.rand	*cupy.random.rand*
numpy.random.randint	*cupy.random.randint*
numpy.random.randn	*cupy.random.randn*
numpy.random.random	*cupy.random.random*
numpy.random.random_integers	*cupy.random.random_integers*
numpy.random.random_sample	*cupy.random.random_sample*
numpy.random.ranf	*cupy.random.ranf*
numpy.random.rayleigh	*cupy.random.rayleigh*
numpy.random.sample	*cupy.random.sample*
numpy.random.seed	*cupy.random.seed*
numpy.random.set_bit_generator	-
numpy.random.set_state	-
numpy.random.shuffle	*cupy.random.shuffle*
numpy.random.standard_cauchy	*cupy.random.standard_cauchy*
numpy.random.standard_exponential	*cupy.random.standard_exponential*
numpy.random.standard_gamma	*cupy.random.standard_gamma*
numpy.random.standard_normal	*cupy.random.standard_normal*
numpy.random.standard_t	*cupy.random.standard_t*
numpy.random.triangular	*cupy.random.triangular*
numpy.random.uniform	*cupy.random.uniform*
numpy.random.vonmises	*cupy.random.vonmises*
numpy.random.wald	*cupy.random.wald*
numpy.random.weibull	*cupy.random.weibull*
numpy.random.zipf	*cupy.random.zipf*

Polynomials

NumPy	CuPy
numpy.polynomial.Chebyshev	-
numpy.polynomial.Hermite	-
numpy.polynomial.HermiteE	-
numpy.polynomial.Laguerre	-
numpy.polynomial.Legendre	-
numpy.polynomial.Polynomial	-
numpy.polynomial.set_default_printstyle	-

Power Series

NumPy	CuPy
numpy.polynomial.polynomial.ABCPolyBase	-
numpy.polynomial.polynomial.Polynomial	-
numpy.polynomial.polynomial.normalize_axis_index	-
numpy.polynomial.polynomial.polyadd	-
numpy.polynomial.polynomial.polycompanion	*cupy.polynomial.polynomial.polycompanion*
numpy.polynomial.polynomial.polyder	-
numpy.polynomial.polynomial.polydiv	-
numpy.polynomial.polynomial.polydomain	-
numpy.polynomial.polynomial.polyfit	-
numpy.polynomial.polynomial.polyfromroots	-
numpy.polynomial.polynomial.polygrid2d	-
numpy.polynomial.polynomial.polygrid3d	-
numpy.polynomial.polynomial.polyint	-
numpy.polynomial.polynomial.polyline	-
numpy.polynomial.polynomial.polymul	-
numpy.polynomial.polynomial.polymulx	-
numpy.polynomial.polynomial.polyone	-
numpy.polynomial.polynomial.polypow	-
numpy.polynomial.polynomial.polyroots	-
numpy.polynomial.polynomial.polysub	-
numpy.polynomial.polynomial.polytrim	-
numpy.polynomial.polynomial.polyval	*cupy.polynomial.polynomial.polyval*
numpy.polynomial.polynomial.polyval2d	-
numpy.polynomial.polynomial.polyval3d	-
numpy.polynomial.polynomial.polyvalfromroots	*cupy.polynomial.polynomial.polyvalfromroots*
numpy.polynomial.polynomial.polyvander	*cupy.polynomial.polynomial.polyvander*
numpy.polynomial.polynomial.polyvander2d	-
numpy.polynomial.polynomial.polyvander3d	-
numpy.polynomial.polynomial.polyx	-
numpy.polynomial.polynomial.polyzero	-

Polyutils

NumPy	CuPy
numpy.polynomial.polyutils.absolute	-
numpy.polynomial.polyutils.as_series	*cupy.polynomial.polyutils.as_series*
numpy.polynomial.polyutils.dragon4_positional	-
numpy.polynomial.polyutils.dragon4_scientific	-
numpy.polynomial.polyutils.format_float	-
numpy.polynomial.polyutils.getdomain	-
numpy.polynomial.polyutils.mapdomain	-
numpy.polynomial.polyutils.mapparms	-
numpy.polynomial.polyutils.trimcoef	*cupy.polynomial.polyutils.trimcoef*
numpy.polynomial.polyutils.trimseq	*cupy.polynomial.polyutils.trimseq*

5.10.2 SciPy / CuPy APIs

Discrete Fourier Transform

SciPy	CuPy
scipy.fft.dct	*cupyx.scipy.fft.dct*
scipy.fft.dctn	*cupyx.scipy.fft.dctn*
scipy.fft.dst	*cupyx.scipy.fft.dst*
scipy.fft.dstn	*cupyx.scipy.fft.dstn*
scipy.fft.fft	*cupyx.scipy.fft.fft*
scipy.fft.fft2	*cupyx.scipy.fft.fft2*
scipy.fft.fftfreq	*cupyx.scipy.fft.fftfreq*
scipy.fft.fftn	*cupyx.scipy.fft.fftn*
scipy.fft.fftshift	*cupyx.scipy.fft.fftshift*
scipy.fft.fht	*cupyx.scipy.fft.fht*
scipy.fft.fhtoffset	-
scipy.fft.get_workers	-
scipy.fft.hfft	*cupyx.scipy.fft.hfft*
scipy.fft.hfft2	*cupyx.scipy.fft.hfft2*
scipy.fft.hfftn	*cupyx.scipy.fft.hfftn*
scipy.fft.idct	*cupyx.scipy.fft.idct*
scipy.fft.idctn	*cupyx.scipy.fft.idctn*
scipy.fft.idst	*cupyx.scipy.fft.idst*
scipy.fft.idstn	*cupyx.scipy.fft.idstn*
scipy.fft.ifft	*cupyx.scipy.fft.ifft*
scipy.fft.ifft2	*cupyx.scipy.fft.ifft2*
scipy.fft.ifftn	*cupyx.scipy.fft.ifftn*
scipy.fft.ifftshift	*cupyx.scipy.fft.ifftshift*
scipy.fft.ifht	*cupyx.scipy.fft.ifht*
scipy.fft.ihfft	*cupyx.scipy.fft.ihfft*
scipy.fft.ihfft2	*cupyx.scipy.fft.ihfft2*
scipy.fft.ihfftn	*cupyx.scipy.fft.ihfftn*
scipy.fft.irfft	*cupyx.scipy.fft.irfft*
scipy.fft.irfft2	*cupyx.scipy.fft.irfft2*
scipy.fft.irfftn	*cupyx.scipy.fft.irfftn*
scipy.fft.next_fast_len	*cupyx.scipy.fft.next_fast_len*
scipy.fft.register_backend	-
scipy.fft.rfft	*cupyx.scipy.fft.rfft*
scipy.fft.rfft2	*cupyx.scipy.fft.rfft2*
scipy.fft.rfftfreq	*cupyx.scipy.fft.rfftfreq*
scipy.fft.rfftn	*cupyx.scipy.fft.rfftn*
scipy.fft.set_backend	-
scipy.fft.set_global_backend	-
scipy.fft.set_workers	-
scipy.fft.skip_backend	-

Legacy Discrete Fourier Transform

SciPy	CuPy
scipy.fftpack.cc_diff	-
scipy.fftpack.cs_diff	-
scipy.fftpack.dct	-
scipy.fftpack.dctn	-
scipy.fftpack.diff	-
scipy.fftpack.dst	-
scipy.fftpack.dstn	-
scipy.fftpack.fft	*cupyx.scipy.fftpack.fft*
scipy.fftpack.fft2	*cupyx.scipy.fftpack.fft2*
scipy.fftpack.fftfreq	-
scipy.fftpack.fftn	*cupyx.scipy.fftpack.fftn*
scipy.fftpack.fftshift	-
scipy.fftpack.hilbert	-
scipy.fftpack.idct	-
scipy.fftpack.idctn	-
scipy.fftpack.idst	-
scipy.fftpack.idstn	-
scipy.fftpack.ifft	*cupyx.scipy.fftpack.ifft*
scipy.fftpack.ifft2	*cupyx.scipy.fftpack.ifft2*
scipy.fftpack.ifftn	*cupyx.scipy.fftpack.ifftn*
scipy.fftpack.ifftshift	-
scipy.fftpack.ihilbert	-
scipy.fftpack.irfft	*cupyx.scipy.fftpack.irfft*
scipy.fftpack.itilbert	-
scipy.fftpack.next_fast_len	-
scipy.fftpack.rfft	*cupyx.scipy.fftpack.rfft*
scipy.fftpack.rfftfreq	-
scipy.fftpack.sc_diff	-
scipy.fftpack.shift	-
scipy.fftpack.ss_diff	-
scipy.fftpack.tilbert	-

Interpolation

SciPy	CuPy
scipy.interpolate.Akima1DInterpolator	*cupyx.scipy.interpolate.Akima1DInterpolator*
scipy.interpolate.BPoly	*cupyx.scipy.interpolate.BPoly*
scipy.interpolate.BSpline	*cupyx.scipy.interpolate.BSpline*
scipy.interpolate.BarycentricInterpolator	*cupyx.scipy.interpolate.BarycentricInterpolator*
scipy.interpolate.BivariateSpline	-
scipy.interpolate.CloughTocher2DInterpolator	*cupyx.scipy.interpolate.CloughTocher2DInterpolator*
scipy.interpolate.CubicHermiteSpline	*cupyx.scipy.interpolate.CubicHermiteSpline*
scipy.interpolate.CubicSpline	*cupyx.scipy.interpolate.CubicSpline*
scipy.interpolate.InterpolatedUnivariateSpline	*cupyx.scipy.interpolate.InterpolatedUnivariateSpline*
scipy.interpolate.KroghInterpolator	*cupyx.scipy.interpolate.KroghInterpolator*
scipy.interpolate.LSQBivariateSpline	-

continues on next page

Table 11 – continued from previous page

SciPy	CuPy
scipy.interpolate.LSQSphereBivariateSpline	-
scipy.interpolate.LSQUnivariateSpline	*cupyx.scipy.interpolate.LSQUnivariateSpline*
scipy.interpolate.LinearNDInterpolator	*cupyx.scipy.interpolate.LinearNDInterpolator*
scipy.interpolate.NdPPoly	*cupyx.scipy.interpolate.NdPPoly*
scipy.interpolate.NearestNDInterpolator	-
scipy.interpolate.PPoly	*cupyx.scipy.interpolate.PPoly*
scipy.interpolate.PchipInterpolator	*cupyx.scipy.interpolate.PchipInterpolator*
scipy.interpolate.RBFInterpolator	*cupyx.scipy.interpolate.RBFInterpolator*
scipy.interpolate.Rbf	-
scipy.interpolate.RectBivariateSpline	-
scipy.interpolate.RectSphereBivariateSpline	-
scipy.interpolate.RegularGridInterpolator	*cupyx.scipy.interpolate.RegularGridInterpolator*
scipy.interpolate.SmoothBivariateSpline	-
scipy.interpolate.SmoothSphereBivariateSpline	-
scipy.interpolate.UnivariateSpline	*cupyx.scipy.interpolate.UnivariateSpline*
scipy.interpolate.approximate_taylor_polynomial	-
scipy.interpolate.barycentric_interpolate	*cupyx.scipy.interpolate.barycentric_interpolate*
scipy.interpolate.bisplev	-
scipy.interpolate.bisplrep	-
scipy.interpolate.griddata	-
scipy.interpolate.insert	-
scipy.interpolate.interp1d	*cupyx.scipy.interpolate.interp1d*
scipy.interpolate.interp2d	-
scipy.interpolate.interpn	*cupyx.scipy.interpolate.interpn*
scipy.interpolate.krogh_interpolate	*cupyx.scipy.interpolate.krogh_interpolate*
scipy.interpolate.lagrange	-
scipy.interpolate.make_interp_spline	*cupyx.scipy.interpolate.make_interp_spline*
scipy.interpolate.make_lsq_spline	*cupyx.scipy.interpolate.make_lsq_spline*
scipy.interpolate.make_smoothing_spline	-
scipy.interpolate.pade	-
scipy.interpolate.pchip	cupyx.scipy.interpolate.pchip
scipy.interpolate.pchip_interpolate	*cupyx.scipy.interpolate.pchip_interpolate*
scipy.interpolate.spalde	-
scipy.interpolate.splantider	*cupyx.scipy.interpolate.splantider*
scipy.interpolate.splder	*cupyx.scipy.interpolate.splder*
scipy.interpolate.splev	-
scipy.interpolate.splint	-
scipy.interpolate.splprep	-
scipy.interpolate.splrep	-
scipy.interpolate.sproot	-

Advanced Linear Algebra

SciPy	CuPy
scipy.linalg.bandwidth	cupyx.scipy.linalg.bandwidth
scipy.linalg.block_diag	*cupyx.scipy.linalg.block_diag*
scipy.linalg.cdf2rdf	-
scipy.linalg.cho_factor	-
scipy.linalg.cho_solve	-
scipy.linalg.cho_solve_banded	-
scipy.linalg.cholesky_banded	-
scipy.linalg.circulant	*cupyx.scipy.linalg.circulant*
scipy.linalg.clarkson_woodruff_transform	-
scipy.linalg.companion	*cupyx.scipy.linalg.companion*
scipy.linalg.convolution_matrix	*cupyx.scipy.linalg.convolution_matrix*
scipy.linalg.coshm	-
scipy.linalg.cosm	-
scipy.linalg.cossin	-
scipy.linalg.dft	*cupyx.scipy.linalg.dft*
scipy.linalg.diagsvd	-
scipy.linalg.eig_banded	-
scipy.linalg.eigh_tridiagonal	-
scipy.linalg.eigvals_banded	-
scipy.linalg.eigvalsh_tridiagonal	-
scipy.linalg.expm	*cupyx.scipy.linalg.expm*
scipy.linalg.expm_cond	-
scipy.linalg.expm_frechet	-
scipy.linalg.fiedler	*cupyx.scipy.linalg.fiedler*
scipy.linalg.fiedler_companion	*cupyx.scipy.linalg.fiedler_companion*
scipy.linalg.find_best_blas_type	-
scipy.linalg.fractional_matrix_power	-
scipy.linalg.funm	-
scipy.linalg.get_blas_funcs	-
scipy.linalg.get_lapack_funcs	-
scipy.linalg.hadamard	*cupyx.scipy.linalg.hadamard*
scipy.linalg.hankel	*cupyx.scipy.linalg.hankel*
scipy.linalg.helmert	*cupyx.scipy.linalg.helmert*
scipy.linalg.hessenberg	-
scipy.linalg.hilbert	*cupyx.scipy.linalg.hilbert*
scipy.linalg.invhilbert	-
scipy.linalg.invpascal	-
scipy.linalg.ishermitian	-
scipy.linalg.issymmetric	-
scipy.linalg.khatri_rao	cupyx.scipy.linalg.khatri_rao
scipy.linalg.kron	*cupyx.scipy.linalg.kron*
scipy.linalg.ldl	-
scipy.linalg.leslie	*cupyx.scipy.linalg.leslie*
scipy.linalg.logm	-
scipy.linalg.lu	*cupyx.scipy.linalg.lu*
scipy.linalg.lu_factor	*cupyx.scipy.linalg.lu_factor*
scipy.linalg.lu_solve	*cupyx.scipy.linalg.lu_solve*

continues on next page

Table 12 – continued from previous page

SciPy	CuPy
scipy.linalg.matmul_toeplitz	-
scipy.linalg.matrix_balance	-
scipy.linalg.null_space	-
scipy.linalg.ordqz	-
scipy.linalg.orth	-
scipy.linalg.orthogonal_procrustes	-
scipy.linalg.pascal	-
scipy.linalg.pinvh	-
scipy.linalg.polar	-
scipy.linalg.qr_delete	-
scipy.linalg.qr_insert	-
scipy.linalg.qr_multiply	-
scipy.linalg.qr_update	-
scipy.linalg.qz	-
scipy.linalg.rq	-
scipy.linalg.rsf2csf	-
scipy.linalg.schur	-
scipy.linalg.signm	-
scipy.linalg.sinhm	-
scipy.linalg.sinm	-
scipy.linalg.solve_banded	-
scipy.linalg.solve_circulant	-
scipy.linalg.solve_continuous_are	-
scipy.linalg.solve_continuous_lyapunov	-
scipy.linalg.solve_discrete_are	-
scipy.linalg.solve_discrete_lyapunov	-
scipy.linalg.solve_lyapunov	-
scipy.linalg.solve_sylvester	-
scipy.linalg.solve_toeplitz	-
scipy.linalg.solve_triangular	*cupyx.scipy.linalg.solve_triangular*
scipy.linalg.solveh_banded	-
scipy.linalg.sqrtm	-
scipy.linalg.subspace_angles	-
scipy.linalg.svdvals	-
scipy.linalg.tanhm	-
scipy.linalg.tanm	-
scipy.linalg.toeplitz	*cupyx.scipy.linalg.toeplitz*
scipy.linalg.tri	*cupyx.scipy.linalg.tri*
scipy.linalg.tril	*cupyx.scipy.linalg.tril*
scipy.linalg.triu	*cupyx.scipy.linalg.triu*

Multidimensional Image Processing

SciPy	CuPy
scipy.ndimage.affine_transform	*cupyx.scipy.ndimage.affine_transform*
scipy.ndimage.binary_closing	*cupyx.scipy.ndimage.binary_closing*
scipy.ndimage.binary_dilation	*cupyx.scipy.ndimage.binary_dilation*
scipy.ndimage.binary_erosion	*cupyx.scipy.ndimage.binary_erosion*
scipy.ndimage.binary_fill_holes	*cupyx.scipy.ndimage.binary_fill_holes*
scipy.ndimage.binary_hit_or_miss	*cupyx.scipy.ndimage.binary_hit_or_miss*
scipy.ndimage.binary_opening	*cupyx.scipy.ndimage.binary_opening*
scipy.ndimage.binary_propagation	*cupyx.scipy.ndimage.binary_propagation*
scipy.ndimage.black_tophat	*cupyx.scipy.ndimage.black_tophat*
scipy.ndimage.center_of_mass	*cupyx.scipy.ndimage.center_of_mass*
scipy.ndimage.convolve	*cupyx.scipy.ndimage.convolve*
scipy.ndimage.convolve1d	*cupyx.scipy.ndimage.convolve1d*
scipy.ndimage.correlate	*cupyx.scipy.ndimage.correlate*
scipy.ndimage.correlate1d	*cupyx.scipy.ndimage.correlate1d*
scipy.ndimage.distance_transform_bf	-
scipy.ndimage.distance_transform_cdt	-
scipy.ndimage.distance_transform_edt	*cupyx.scipy.ndimage.distance_transform_edt*
scipy.ndimage.extrema	*cupyx.scipy.ndimage.extrema*
scipy.ndimage.find_objects	-
scipy.ndimage.fourier_ellipsoid	*cupyx.scipy.ndimage.fourier_ellipsoid*
scipy.ndimage.fourier_gaussian	*cupyx.scipy.ndimage.fourier_gaussian*
scipy.ndimage.fourier_shift	*cupyx.scipy.ndimage.fourier_shift*
scipy.ndimage.fourier_uniform	*cupyx.scipy.ndimage.fourier_uniform*
scipy.ndimage.gaussian_filter	*cupyx.scipy.ndimage.gaussian_filter*
scipy.ndimage.gaussian_filter1d	*cupyx.scipy.ndimage.gaussian_filter1d*
scipy.ndimage.gaussian_gradient_magnitude	*cupyx.scipy.ndimage.gaussian_gradient_magnitude*
scipy.ndimage.gaussian_laplace	*cupyx.scipy.ndimage.gaussian_laplace*
scipy.ndimage.generate_binary_structure	*cupyx.scipy.ndimage.generate_binary_structure*
scipy.ndimage.generic_filter	*cupyx.scipy.ndimage.generic_filter*
scipy.ndimage.generic_filter1d	*cupyx.scipy.ndimage.generic_filter1d*
scipy.ndimage.generic_gradient_magnitude	*cupyx.scipy.ndimage.generic_gradient_magnitude*
scipy.ndimage.generic_laplace	*cupyx.scipy.ndimage.generic_laplace*
scipy.ndimage.geometric_transform	-
scipy.ndimage.grey_closing	*cupyx.scipy.ndimage.grey_closing*
scipy.ndimage.grey_dilation	*cupyx.scipy.ndimage.grey_dilation*
scipy.ndimage.grey_erosion	*cupyx.scipy.ndimage.grey_erosion*
scipy.ndimage.grey_opening	*cupyx.scipy.ndimage.grey_opening*
scipy.ndimage.histogram	*cupyx.scipy.ndimage.histogram*
scipy.ndimage.iterate_structure	*cupyx.scipy.ndimage.iterate_structure*
scipy.ndimage.label	*cupyx.scipy.ndimage.label*
scipy.ndimage.labeled_comprehension	*cupyx.scipy.ndimage.labeled_comprehension*
scipy.ndimage.laplace	*cupyx.scipy.ndimage.laplace*
scipy.ndimage.map_coordinates	*cupyx.scipy.ndimage.map_coordinates*
scipy.ndimage.maximum	*cupyx.scipy.ndimage.maximum*
scipy.ndimage.maximum_filter	*cupyx.scipy.ndimage.maximum_filter*
scipy.ndimage.maximum_filter1d	*cupyx.scipy.ndimage.maximum_filter1d*
scipy.ndimage.maximum_position	*cupyx.scipy.ndimage.maximum_position*

continues on next page

Table 13 – continued from previous page

SciPy	CuPy
scipy.ndimage.mean	*cupyx.scipy.ndimage.mean*
scipy.ndimage.median	*cupyx.scipy.ndimage.median*
scipy.ndimage.median_filter	*cupyx.scipy.ndimage.median_filter*
scipy.ndimage.minimum	*cupyx.scipy.ndimage.minimum*
scipy.ndimage.minimum_filter	*cupyx.scipy.ndimage.minimum_filter*
scipy.ndimage.minimum_filter1d	*cupyx.scipy.ndimage.minimum_filter1d*
scipy.ndimage.minimum_position	*cupyx.scipy.ndimage.minimum_position*
scipy.ndimage.morphological_gradient	*cupyx.scipy.ndimage.morphological_gradient*
scipy.ndimage.morphological_laplace	*cupyx.scipy.ndimage.morphological_laplace*
scipy.ndimage.percentile_filter	*cupyx.scipy.ndimage.percentile_filter*
scipy.ndimage.prewitt	*cupyx.scipy.ndimage.prewitt*
scipy.ndimage.rank_filter	*cupyx.scipy.ndimage.rank_filter*
scipy.ndimage.rotate	*cupyx.scipy.ndimage.rotate*
scipy.ndimage.shift	*cupyx.scipy.ndimage.shift*
scipy.ndimage.sobel	*cupyx.scipy.ndimage.sobel*
scipy.ndimage.spline_filter	*cupyx.scipy.ndimage.spline_filter*
scipy.ndimage.spline_filter1d	*cupyx.scipy.ndimage.spline_filter1d*
scipy.ndimage.standard_deviation	*cupyx.scipy.ndimage.standard_deviation*
scipy.ndimage.sum	cupyx.scipy.ndimage.sum
scipy.ndimage.sum_labels	*cupyx.scipy.ndimage.sum_labels*
scipy.ndimage.uniform_filter	*cupyx.scipy.ndimage.uniform_filter*
scipy.ndimage.uniform_filter1d	*cupyx.scipy.ndimage.uniform_filter1d*
scipy.ndimage.value_indices	*cupyx.scipy.ndimage.value_indices*
scipy.ndimage.variance	*cupyx.scipy.ndimage.variance*
scipy.ndimage.watershed_ift	-
scipy.ndimage.white_tophat	*cupyx.scipy.ndimage.white_tophat*
scipy.ndimage.zoom	*cupyx.scipy.ndimage.zoom*

Signal processing

SciPy	CuPy
scipy.signal.CZT	*cupyx.scipy.signal.CZT*
scipy.signal.StateSpace	*cupyx.scipy.signal.StateSpace*
scipy.signal.TransferFunction	*cupyx.scipy.signal.TransferFunction*
scipy.signal.ZerosPolesGain	*cupyx.scipy.signal.ZerosPolesGain*
scipy.signal.ZoomFFT	*cupyx.scipy.signal.ZoomFFT*
scipy.signal.abcd_normalize	*cupyx.scipy.signal.abcd_normalize*
scipy.signal.argrelextrema	*cupyx.scipy.signal.argrelextrema*
scipy.signal.argrelmax	*cupyx.scipy.signal.argrelmax*
scipy.signal.argrelmin	*cupyx.scipy.signal.argrelmin*
scipy.signal.band_stop_obj	*cupyx.scipy.signal.band_stop_obj*
scipy.signal.barthann	-
scipy.signal.bartlett	-
scipy.signal.bessel	-
scipy.signal.besselap	-
scipy.signal.bilinear	*cupyx.scipy.signal.bilinear*
scipy.signal.bilinear_zpk	*cupyx.scipy.signal.bilinear_zpk*
scipy.signal.blackman	-

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
scipy.signal.blackmanharris	-
scipy.signal.bode	*cupyx.scipy.signal.bode*
scipy.signal.bohman	-
scipy.signal.boxcar	-
scipy.signal.bspline	-
scipy.signal.buttap	*cupyx.scipy.signal.buttap*
scipy.signal.butter	*cupyx.scipy.signal.butter*
scipy.signal.buttord	*cupyx.scipy.signal.buttord*
scipy.signal.cascade	-
scipy.signal.cheb1ap	*cupyx.scipy.signal.cheb1ap*
scipy.signal.cheb1ord	*cupyx.scipy.signal.cheb1ord*
scipy.signal.cheb2ap	*cupyx.scipy.signal.cheb2ap*
scipy.signal.cheb2ord	*cupyx.scipy.signal.cheb2ord*
scipy.signal.chebwin	-
scipy.signal.cheby1	*cupyx.scipy.signal.cheby1*
scipy.signal.cheby2	*cupyx.scipy.signal.cheby2*
scipy.signal.check_COLA	*cupyx.scipy.signal.check_COLA*
scipy.signal.check_NOLA	*cupyx.scipy.signal.check_NOLA*
scipy.signal.chirp	*cupyx.scipy.signal.chirp*
scipy.signal.choose_conv_method	*cupyx.scipy.signal.choose_conv_method*
scipy.signal.cmplx_sort	-
scipy.signal.coherence	*cupyx.scipy.signal.coherence*
scipy.signal.cont2discrete	*cupyx.scipy.signal.cont2discrete*
scipy.signal.convolve	*cupyx.scipy.signal.convolve*
scipy.signal.convolve2d	*cupyx.scipy.signal.convolve2d*
scipy.signal.correlate	*cupyx.scipy.signal.correlate*
scipy.signal.correlate2d	*cupyx.scipy.signal.correlate2d*
scipy.signal.correlation_lags	*cupyx.scipy.signal.correlation_lags*
scipy.signal.cosine	-
scipy.signal.csd	*cupyx.scipy.signal.csd*
scipy.signal.cspline1d	*cupyx.scipy.signal.cspline1d*
scipy.signal.cspline1d_eval	*cupyx.scipy.signal.cspline1d_eval*
scipy.signal.cspline2d	*cupyx.scipy.signal.cspline2d*
scipy.signal.cubic	-
scipy.signal.cwt	*cupyx.scipy.signal.cwt*
scipy.signal.czt	*cupyx.scipy.signal.czt*
scipy.signal.czt_points	*cupyx.scipy.signal.czt_points*
scipy.signal.daub	-
scipy.signal.dbode	*cupyx.scipy.signal.dbode*
scipy.signal.decimate	*cupyx.scipy.signal.decimate*
scipy.signal.deconvolve	*cupyx.scipy.signal.deconvolve*
scipy.signal.detrend	*cupyx.scipy.signal.detrend*
scipy.signal.dfreqresp	*cupyx.scipy.signal.dfreqresp*
scipy.signal.dimpulse	*cupyx.scipy.signal.dimpulse*
scipy.signal.dlsim	*cupyx.scipy.signal.dlsim*
scipy.signal.dlti	*cupyx.scipy.signal.dlti*
scipy.signal.dstep	*cupyx.scipy.signal.dstep*
scipy.signal.ellip	*cupyx.scipy.signal.ellip*
scipy.signal.ellipap	*cupyx.scipy.signal.ellipap*

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
scipy.signal.ellipord	*cupyx.scipy.signal.ellipord*
scipy.signal.exponential	-
scipy.signal.fftconvolve	*cupyx.scipy.signal.fftconvolve*
scipy.signal.filtfilt	*cupyx.scipy.signal.filtfilt*
scipy.signal.find_peaks	*cupyx.scipy.signal.find_peaks*
scipy.signal.find_peaks_cwt	-
scipy.signal.findfreqs	*cupyx.scipy.signal.findfreqs*
scipy.signal.firls	*cupyx.scipy.signal.firls*
scipy.signal.firwin	*cupyx.scipy.signal.firwin*
scipy.signal.firwin2	*cupyx.scipy.signal.firwin2*
scipy.signal.flattop	-
scipy.signal.freqresp	*cupyx.scipy.signal.freqresp*
scipy.signal.freqs	*cupyx.scipy.signal.freqs*
scipy.signal.freqs_zpk	*cupyx.scipy.signal.freqs_zpk*
scipy.signal.freqz	*cupyx.scipy.signal.freqz*
scipy.signal.freqz_zpk	*cupyx.scipy.signal.freqz_zpk*
scipy.signal.gammatone	*cupyx.scipy.signal.gammatone*
scipy.signal.gauss_spline	*cupyx.scipy.signal.gauss_spline*
scipy.signal.gaussian	-
scipy.signal.gausspulse	*cupyx.scipy.signal.gausspulse*
scipy.signal.general_gaussian	-
scipy.signal.get_window	*cupyx.scipy.signal.get_window*
scipy.signal.group_delay	*cupyx.scipy.signal.group_delay*
scipy.signal.hamming	-
scipy.signal.hann	-
scipy.signal.hilbert	*cupyx.scipy.signal.hilbert*
scipy.signal.hilbert2	*cupyx.scipy.signal.hilbert2*
scipy.signal.iircomb	*cupyx.scipy.signal.iircomb*
scipy.signal.iirdesign	*cupyx.scipy.signal.iirdesign*
scipy.signal.iirfilter	*cupyx.scipy.signal.iirfilter*
scipy.signal.iirnotch	*cupyx.scipy.signal.iirnotch*
scipy.signal.iirpeak	*cupyx.scipy.signal.iirpeak*
scipy.signal.impulse	*cupyx.scipy.signal.impulse*
scipy.signal.impulse2	-
scipy.signal.invres	*cupyx.scipy.signal.invres*
scipy.signal.invresz	*cupyx.scipy.signal.invresz*
scipy.signal.istft	*cupyx.scipy.signal.istft*
scipy.signal.kaiser	-
scipy.signal.kaiser_atten	*cupyx.scipy.signal.kaiser_atten*
scipy.signal.kaiser_beta	*cupyx.scipy.signal.kaiser_beta*
scipy.signal.kaiserord	*cupyx.scipy.signal.kaiserord*
scipy.signal.lfilter	*cupyx.scipy.signal.lfilter*
scipy.signal.lfilter_zi	*cupyx.scipy.signal.lfilter_zi*
scipy.signal.lfiltic	*cupyx.scipy.signal.lfiltic*
scipy.signal.lombscargle	*cupyx.scipy.signal.lombscargle*
scipy.signal.lp2bp	*cupyx.scipy.signal.lp2bp*
scipy.signal.lp2bp_zpk	*cupyx.scipy.signal.lp2bp_zpk*
scipy.signal.lp2bs	*cupyx.scipy.signal.lp2bs*
scipy.signal.lp2bs_zpk	*cupyx.scipy.signal.lp2bs_zpk*

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
scipy.signal.lp2hp	*cupyx.scipy.signal.lp2hp*
scipy.signal.lp2hp_zpk	*cupyx.scipy.signal.lp2hp_zpk*
scipy.signal.lp2lp	*cupyx.scipy.signal.lp2lp*
scipy.signal.lp2lp_zpk	*cupyx.scipy.signal.lp2lp_zpk*
scipy.signal.lsim	*cupyx.scipy.signal.lsim*
scipy.signal.lsim2	-
scipy.signal.lti	*cupyx.scipy.signal.lti*
scipy.signal.max_len_seq	*cupyx.scipy.signal.max_len_seq*
scipy.signal.medfilt	*cupyx.scipy.signal.medfilt*
scipy.signal.medfilt2d	*cupyx.scipy.signal.medfilt2d*
scipy.signal.minimum_phase	*cupyx.scipy.signal.minimum_phase*
scipy.signal.morlet	*cupyx.scipy.signal.morlet*
scipy.signal.morlet2	*cupyx.scipy.signal.morlet2*
scipy.signal.normalize	*cupyx.scipy.signal.normalize*
scipy.signal.nuttall	-
scipy.signal.oaconvolve	*cupyx.scipy.signal.oaconvolve*
scipy.signal.order_filter	*cupyx.scipy.signal.order_filter*
scipy.signal.parzen	-
scipy.signal.peak_prominences	*cupyx.scipy.signal.peak_prominences*
scipy.signal.peak_widths	*cupyx.scipy.signal.peak_widths*
scipy.signal.periodogram	*cupyx.scipy.signal.periodogram*
scipy.signal.place_poles	*cupyx.scipy.signal.place_poles*
scipy.signal.qmf	*cupyx.scipy.signal.qmf*
scipy.signal.qspline1d	*cupyx.scipy.signal.qspline1d*
scipy.signal.qspline1d_eval	*cupyx.scipy.signal.qspline1d_eval*
scipy.signal.qspline2d	*cupyx.scipy.signal.qspline2d*
scipy.signal.quadratic	-
scipy.signal.remez	-
scipy.signal.resample	*cupyx.scipy.signal.resample*
scipy.signal.resample_poly	*cupyx.scipy.signal.resample_poly*
scipy.signal.residue	*cupyx.scipy.signal.residue*
scipy.signal.residuez	*cupyx.scipy.signal.residuez*
scipy.signal.ricker	*cupyx.scipy.signal.ricker*
scipy.signal.savgol_coeffs	*cupyx.scipy.signal.savgol_coeffs*
scipy.signal.savgol_filter	*cupyx.scipy.signal.savgol_filter*
scipy.signal.sawtooth	*cupyx.scipy.signal.sawtooth*
scipy.signal.sepfir2d	*cupyx.scipy.signal.sepfir2d*
scipy.signal.sos2tf	*cupyx.scipy.signal.sos2tf*
scipy.signal.sos2zpk	*cupyx.scipy.signal.sos2zpk*
scipy.signal.sosfilt	*cupyx.scipy.signal.sosfilt*
scipy.signal.sosfilt_zi	*cupyx.scipy.signal.sosfilt_zi*
scipy.signal.sosfiltfilt	*cupyx.scipy.signal.sosfiltfilt*
scipy.signal.sosfreqz	*cupyx.scipy.signal.sosfreqz*
scipy.signal.spectrogram	*cupyx.scipy.signal.spectrogram*
scipy.signal.spline_filter	*cupyx.scipy.signal.spline_filter*
scipy.signal.square	*cupyx.scipy.signal.square*
scipy.signal.ss2tf	*cupyx.scipy.signal.ss2tf*
scipy.signal.ss2zpk	*cupyx.scipy.signal.ss2zpk*
scipy.signal.step	*cupyx.scipy.signal.step*

continues on next page

Table 14 – continued from previous page

SciPy	CuPy
scipy.signal.step2	-
scipy.signal.stft	*cupyx.scipy.signal.stft*
scipy.signal.sweep_poly	*cupyx.scipy.signal.sweep_poly*
scipy.signal.symiirorder1	*cupyx.scipy.signal.symiirorder1*
scipy.signal.symiirorder2	*cupyx.scipy.signal.symiirorder2*
scipy.signal.tf2sos	*cupyx.scipy.signal.tf2sos*
scipy.signal.tf2ss	*cupyx.scipy.signal.tf2ss*
scipy.signal.tf2zpk	*cupyx.scipy.signal.tf2zpk*
scipy.signal.triang	-
scipy.signal.tukey	-
scipy.signal.unique_roots	*cupyx.scipy.signal.unique_roots*
scipy.signal.unit_impulse	*cupyx.scipy.signal.unit_impulse*
scipy.signal.upfirdn	*cupyx.scipy.signal.upfirdn*
scipy.signal.vectorstrength	*cupyx.scipy.signal.vectorstrength*
scipy.signal.welch	*cupyx.scipy.signal.welch*
scipy.signal.wiener	*cupyx.scipy.signal.wiener*
scipy.signal.zoom_fft	*cupyx.scipy.signal.zoom_fft*
scipy.signal.zpk2sos	*cupyx.scipy.signal.zpk2sos*
scipy.signal.zpk2ss	*cupyx.scipy.signal.zpk2ss*
scipy.signal.zpk2tf	*cupyx.scipy.signal.zpk2tf*

Sparse Matrices

SciPy	CuPy
scipy.sparse.block_diag	-
scipy.sparse.bmat	*cupyx.scipy.sparse.bmat*
scipy.sparse.bsr_array	-
scipy.sparse.bsr_matrix	-
scipy.sparse.coo_array	-
scipy.sparse.coo_matrix	*cupyx.scipy.sparse.coo_matrix*
scipy.sparse.csc_array	-
scipy.sparse.csc_matrix	*cupyx.scipy.sparse.csc_matrix*
scipy.sparse.csr_array	-
scipy.sparse.csr_matrix	*cupyx.scipy.sparse.csr_matrix*
scipy.sparse.dia_array	-
scipy.sparse.dia_matrix	*cupyx.scipy.sparse.dia_matrix*
scipy.sparse.diags	*cupyx.scipy.sparse.diags*
scipy.sparse.dok_array	-
scipy.sparse.dok_matrix	-
scipy.sparse.eye	*cupyx.scipy.sparse.eye*
scipy.sparse.find	*cupyx.scipy.sparse.find*
scipy.sparse.hstack	*cupyx.scipy.sparse.hstack*
scipy.sparse.identity	*cupyx.scipy.sparse.identity*
scipy.sparse.issparse	*cupyx.scipy.sparse.issparse*
scipy.sparse.isspmatrix	*cupyx.scipy.sparse.isspmatrix*
scipy.sparse.isspmatrix_bsr	-
scipy.sparse.isspmatrix_coo	*cupyx.scipy.sparse.isspmatrix_coo*
scipy.sparse.isspmatrix_csc	*cupyx.scipy.sparse.isspmatrix_csc*

continues on next page

Table 15 – continued from previous page

SciPy	CuPy
scipy.sparse.isspmatrix_csr	*cupyx.scipy.sparse.isspmatrix_csr*
scipy.sparse.isspmatrix_dia	*cupyx.scipy.sparse.isspmatrix_dia*
scipy.sparse.isspmatrix_dok	-
scipy.sparse.isspmatrix_lil	-
scipy.sparse.kron	*cupyx.scipy.sparse.kron*
scipy.sparse.kronsum	*cupyx.scipy.sparse.kronsum*
scipy.sparse.lil_array	-
scipy.sparse.lil_matrix	-
scipy.sparse.load_npz	-
scipy.sparse.rand	*cupyx.scipy.sparse.rand*
scipy.sparse.random	*cupyx.scipy.sparse.random*
scipy.sparse.save_npz	-
scipy.sparse.sparray	-
scipy.sparse.spdiags	*cupyx.scipy.sparse.spdiags*
scipy.sparse.spmatrix	*cupyx.scipy.sparse.spmatrix*
scipy.sparse.tril	*cupyx.scipy.sparse.tril*
scipy.sparse.triu	*cupyx.scipy.sparse.triu*
scipy.sparse.vstack	*cupyx.scipy.sparse.vstack*

Sparse Linear Algebra

SciPy	CuPy
scipy.sparse.linalg.LinearOperator	*cupyx.scipy.sparse.linalg.LinearOperator*
scipy.sparse.linalg.SuperLU	*cupyx.scipy.sparse.linalg.SuperLU*
scipy.sparse.linalg.aslinearoperator	*cupyx.scipy.sparse.linalg.aslinearoperator*
scipy.sparse.linalg.bicg	-
scipy.sparse.linalg.bicgstab	-
scipy.sparse.linalg.cg	*cupyx.scipy.sparse.linalg.cg*
scipy.sparse.linalg.cgs	*cupyx.scipy.sparse.linalg.cgs*
scipy.sparse.linalg.eigs	-
scipy.sparse.linalg.eigsh	*cupyx.scipy.sparse.linalg.eigsh*
scipy.sparse.linalg.expm	-
scipy.sparse.linalg.expm_multiply	-
scipy.sparse.linalg.factorized	*cupyx.scipy.sparse.linalg.factorized*
scipy.sparse.linalg.gcrotmk	-
scipy.sparse.linalg.gmres	*cupyx.scipy.sparse.linalg.gmres*
scipy.sparse.linalg.inv	-
scipy.sparse.linalg.lgmres	-
scipy.sparse.linalg.lobpcg	*cupyx.scipy.sparse.linalg.lobpcg*
scipy.sparse.linalg.lsmr	*cupyx.scipy.sparse.linalg.lsmr*
scipy.sparse.linalg.lsqr	*cupyx.scipy.sparse.linalg.lsqr*
scipy.sparse.linalg.minres	*cupyx.scipy.sparse.linalg.minres*
scipy.sparse.linalg.norm	*cupyx.scipy.sparse.linalg.norm*
scipy.sparse.linalg.onenormest	-
scipy.sparse.linalg.qmr	-
scipy.sparse.linalg.spilu	*cupyx.scipy.sparse.linalg.spilu*
scipy.sparse.linalg.splu	*cupyx.scipy.sparse.linalg.splu*
scipy.sparse.linalg.spsolve	*cupyx.scipy.sparse.linalg.spsolve*

continues on next page

Table 16 – continued from previous page

SciPy	CuPy
scipy.sparse.linalg.spsolve_triangular	*cupyx.scipy.sparse.linalg.spsolve_triangular*
scipy.sparse.linalg.svds	*cupyx.scipy.sparse.linalg.svds*
scipy.sparse.linalg.tfqmr	-
scipy.sparse.linalg.use_solver	-

Compressed sparse graph routines

SciPy	CuPy
scipy.sparse.csgraph.bellman_ford	-
scipy.sparse.csgraph.breadth_first_order	-
scipy.sparse.csgraph.breadth_first_tree	-
scipy.sparse.csgraph.connected_components	*cupyx.scipy.sparse.csgraph.connected_components*
scipy.sparse.csgraph.construct_dist_matrix	-
scipy.sparse.csgraph.csgraph_from_dense	-
scipy.sparse.csgraph.csgraph_from_masked	-
scipy.sparse.csgraph.csgraph_masked_from_dense	-
scipy.sparse.csgraph.csgraph_to_dense	-
scipy.sparse.csgraph.csgraph_to_masked	-
scipy.sparse.csgraph.depth_first_order	-
scipy.sparse.csgraph.depth_first_tree	-
scipy.sparse.csgraph.dijkstra	-
scipy.sparse.csgraph.floyd_warshall	-
scipy.sparse.csgraph.johnson	-
scipy.sparse.csgraph.laplacian	-
scipy.sparse.csgraph.maximum_bipartite_matching	-
scipy.sparse.csgraph.maximum_flow	-
scipy.sparse.csgraph.min_weight_full_bipartite_matching	-
scipy.sparse.csgraph.minimum_spanning_tree	-
scipy.sparse.csgraph.reconstruct_path	-
scipy.sparse.csgraph.reverse_cuthill_mckee	-
scipy.sparse.csgraph.shortest_path	-
scipy.sparse.csgraph.structural_rank	-

Special Functions

SciPy	CuPy
scipy.special.agm	-
scipy.special.ai_zeros	-
scipy.special.airy	-
scipy.special.airye	-
scipy.special.assoc_laguerre	-
scipy.special.bdtr	*cupyx.scipy.special.bdtr*

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.bdtrc	*cupyx.scipy.special.bdtrc*
scipy.special.bdtri	*cupyx.scipy.special.bdtri*
scipy.special.bdtrik	-
scipy.special.bdtrin	-
scipy.special.bei	-
scipy.special.bei_zeros	-
scipy.special.beip	-
scipy.special.beip_zeros	-
scipy.special.ber	-
scipy.special.ber_zeros	-
scipy.special.bernoulli	-
scipy.special.berp	-
scipy.special.berp_zeros	-
scipy.special.besselpoly	-
scipy.special.beta	*cupyx.scipy.special.beta*
scipy.special.betainc	*cupyx.scipy.special.betainc*
scipy.special.betaincinv	*cupyx.scipy.special.betaincinv*
scipy.special.betaln	*cupyx.scipy.special.betaln*
scipy.special.bi_zeros	-
scipy.special.binom	cupyx.scipy.special.binom
scipy.special.boxcox	*cupyx.scipy.special.boxcox*
scipy.special.boxcox1p	*cupyx.scipy.special.boxcox1p*
scipy.special.btdtr	*cupyx.scipy.special.btdtr*
scipy.special.btdtri	*cupyx.scipy.special.btdtri*
scipy.special.btdtria	-
scipy.special.btdtrib	-
scipy.special.c_roots	-
scipy.special.cbrt	*cupyx.scipy.special.cbrt*
scipy.special.cg_roots	-
scipy.special.chdtr	*cupyx.scipy.special.chdtr*
scipy.special.chdtrc	*cupyx.scipy.special.chdtrc*
scipy.special.chdtri	*cupyx.scipy.special.chdtri*
scipy.special.chdtriv	-
scipy.special.chebyc	-
scipy.special.chebys	-
scipy.special.chebyt	-
scipy.special.chebyu	-
scipy.special.chndtr	-
scipy.special.chndtridf	-
scipy.special.chndtrinc	-
scipy.special.chndtrix	-
scipy.special.clpmn	-
scipy.special.comb	-
scipy.special.cosdg	*cupyx.scipy.special.cosdg*
scipy.special.cosm1	*cupyx.scipy.special.cosm1*
scipy.special.cotdg	*cupyx.scipy.special.cotdg*
scipy.special.dawsn	-
scipy.special.digamma	*cupyx.scipy.special.digamma*
scipy.special.diric	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.ellip_harm	-
scipy.special.ellip_harm_2	-
scipy.special.ellip_normal	-
scipy.special.ellipe	-
scipy.special.ellipeinc	-
scipy.special.ellipj	cupyx.scipy.special.ellipj
scipy.special.ellipk	cupyx.scipy.special.ellipk
scipy.special.ellipkinc	-
scipy.special.ellipkm1	cupyx.scipy.special.ellipkm1
scipy.special.elliprc	-
scipy.special.elliprd	-
scipy.special.elliprf	-
scipy.special.elliprg	-
scipy.special.elliprj	-
scipy.special.entr	*cupyx.scipy.special.entr*
scipy.special.erf	*cupyx.scipy.special.erf*
scipy.special.erf_zeros	-
scipy.special.erfc	*cupyx.scipy.special.erfc*
scipy.special.erfcinv	*cupyx.scipy.special.erfcinv*
scipy.special.erfcx	*cupyx.scipy.special.erfcx*
scipy.special.erfi	-
scipy.special.erfinv	*cupyx.scipy.special.erfinv*
scipy.special.errstate	-
scipy.special.euler	-
scipy.special.eval_chebyc	-
scipy.special.eval_chebys	-
scipy.special.eval_chebyt	-
scipy.special.eval_chebyu	-
scipy.special.eval_gegenbauer	-
scipy.special.eval_genlaguerre	-
scipy.special.eval_hermite	-
scipy.special.eval_hermitenorm	-
scipy.special.eval_jacobi	-
scipy.special.eval_laguerre	-
scipy.special.eval_legendre	-
scipy.special.eval_sh_chebyt	-
scipy.special.eval_sh_chebyu	-
scipy.special.eval_sh_jacobi	-
scipy.special.eval_sh_legendre	-
scipy.special.exp1	*cupyx.scipy.special.exp1*
scipy.special.exp10	*cupyx.scipy.special.exp10*
scipy.special.exp2	*cupyx.scipy.special.exp2*
scipy.special.expi	*cupyx.scipy.special.expi*
scipy.special.expit	*cupyx.scipy.special.expit*
scipy.special.expm1	*cupyx.scipy.special.expm1*
scipy.special.expn	*cupyx.scipy.special.expn*
scipy.special.exprel	*cupyx.scipy.special.exprel*
scipy.special.factorial	-
scipy.special.factorial2	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.factorialk	-
scipy.special.fdtr	*cupyx.scipy.special.fdtr*
scipy.special.fdtrc	*cupyx.scipy.special.fdtrc*
scipy.special.fdtri	*cupyx.scipy.special.fdtri*
scipy.special.fdtridfd	-
scipy.special.fresnel	-
scipy.special.fresnel_zeros	-
scipy.special.fresnelc_zeros	-
scipy.special.fresnels_zeros	-
scipy.special.gamma	*cupyx.scipy.special.gamma*
scipy.special.gammainc	*cupyx.scipy.special.gammainc*
scipy.special.gammaincc	*cupyx.scipy.special.gammaincc*
scipy.special.gammainccinv	*cupyx.scipy.special.gammainccinv*
scipy.special.gammaincinv	*cupyx.scipy.special.gammaincinv*
scipy.special.gammaln	*cupyx.scipy.special.gammaln*
scipy.special.gammasgn	cupyx.scipy.special.gammasgn
scipy.special.gdtr	*cupyx.scipy.special.gdtr*
scipy.special.gdtrc	*cupyx.scipy.special.gdtrc*
scipy.special.gdtria	-
scipy.special.gdtrib	-
scipy.special.gdtrix	-
scipy.special.gegenbauer	-
scipy.special.genlaguerre	-
scipy.special.geterr	-
scipy.special.h1vp	-
scipy.special.h2vp	-
scipy.special.h_roots	-
scipy.special.hankel1	-
scipy.special.hankel1e	-
scipy.special.hankel2	-
scipy.special.hankel2e	-
scipy.special.he_roots	-
scipy.special.hermite	-
scipy.special.hermitenorm	-
scipy.special.huber	*cupyx.scipy.special.huber*
scipy.special.hyp0f1	-
scipy.special.hyp1f1	-
scipy.special.hyp2f1	-
scipy.special.hyperu	-
scipy.special.i0	*cupyx.scipy.special.i0*
scipy.special.i0e	*cupyx.scipy.special.i0e*
scipy.special.i1	*cupyx.scipy.special.i1*
scipy.special.i1e	*cupyx.scipy.special.i1e*
scipy.special.inv_boxcox	*cupyx.scipy.special.inv_boxcox*
scipy.special.inv_boxcox1p	*cupyx.scipy.special.inv_boxcox1p*
scipy.special.it2i0k0	-
scipy.special.it2j0y0	-
scipy.special.it2struve0	-
scipy.special.itairy	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.iti0k0	-
scipy.special.itj0y0	-
scipy.special.itmodstruve0	-
scipy.special.itstruve0	-
scipy.special.iv	-
scipy.special.ive	-
scipy.special.ivp	-
scipy.special.j0	*cupyx.scipy.special.j0*
scipy.special.j1	*cupyx.scipy.special.j1*
scipy.special.j_roots	-
scipy.special.jacobi	-
scipy.special.jn	-
scipy.special.jn_zeros	-
scipy.special.jnjnp_zeros	-
scipy.special.jnp_zeros	-
scipy.special.jnyn_zeros	-
scipy.special.js_roots	-
scipy.special.jv	-
scipy.special.jve	-
scipy.special.jvp	-
scipy.special.k0	*cupyx.scipy.special.k0*
scipy.special.k0e	*cupyx.scipy.special.k0e*
scipy.special.k1	*cupyx.scipy.special.k1*
scipy.special.k1e	*cupyx.scipy.special.k1e*
scipy.special.kei	-
scipy.special.kei_zeros	-
scipy.special.keip	-
scipy.special.keip_zeros	-
scipy.special.kelvin	-
scipy.special.kelvin_zeros	-
scipy.special.ker	-
scipy.special.ker_zeros	-
scipy.special.kerp	-
scipy.special.kerp_zeros	-
scipy.special.kl_div	*cupyx.scipy.special.kl_div*
scipy.special.kn	-
scipy.special.kolmogi	-
scipy.special.kolmogorov	-
scipy.special.kv	-
scipy.special.kve	-
scipy.special.kvp	-
scipy.special.l_roots	-
scipy.special.la_roots	-
scipy.special.laguerre	-
scipy.special.lambertw	*cupyx.scipy.special.lambertw*
scipy.special.legendre	-
scipy.special.lmbda	-
scipy.special.log1p	*cupyx.scipy.special.log1p*
scipy.special.log_expit	*cupyx.scipy.special.log_expit*

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.log_ndtr	*cupyx.scipy.special.log_ndtr*
scipy.special.log_softmax	*cupyx.scipy.special.log_softmax*
scipy.special.loggamma	*cupyx.scipy.special.loggamma*
scipy.special.logit	*cupyx.scipy.special.logit*
scipy.special.logsumexp	*cupyx.scipy.special.logsumexp*
scipy.special.lpmn	-
scipy.special.lpmv	*cupyx.scipy.special.lpmv*
scipy.special.lpn	-
scipy.special.lqmn	-
scipy.special.lqn	-
scipy.special.mathieu_a	-
scipy.special.mathieu_b	-
scipy.special.mathieu_cem	-
scipy.special.mathieu_even_coef	-
scipy.special.mathieu_modcem1	-
scipy.special.mathieu_modcem2	-
scipy.special.mathieu_modsem1	-
scipy.special.mathieu_modsem2	-
scipy.special.mathieu_odd_coef	-
scipy.special.mathieu_sem	-
scipy.special.modfresnelm	-
scipy.special.modfresnelp	-
scipy.special.modstruve	-
scipy.special.multigammaln	*cupyx.scipy.special.multigammaln*
scipy.special.nbdtr	*cupyx.scipy.special.nbdtr*
scipy.special.nbdtrc	*cupyx.scipy.special.nbdtrc*
scipy.special.nbdtri	*cupyx.scipy.special.nbdtri*
scipy.special.nbdtrik	-
scipy.special.nbdtrin	-
scipy.special.ncfdtr	-
scipy.special.ncfdtri	-
scipy.special.ncfdtridfd	-
scipy.special.ncfdtridfn	-
scipy.special.ncfdtrinc	-
scipy.special.nctdtr	-
scipy.special.nctdtridf	-
scipy.special.nctdtrinc	-
scipy.special.nctdtrit	-
scipy.special.ndtr	*cupyx.scipy.special.ndtr*
scipy.special.ndtri	*cupyx.scipy.special.ndtri*
scipy.special.ndtri_exp	-
scipy.special.nrdtrimn	-
scipy.special.nrdtrisd	-
scipy.special.obl_ang1	-
scipy.special.obl_ang1_cv	-
scipy.special.obl_cv	-
scipy.special.obl_cv_seq	-
scipy.special.obl_rad1	-
scipy.special.obl_rad1_cv	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.obl_rad2	-
scipy.special.obl_rad2_cv	-
scipy.special.owens_t	-
scipy.special.p_roots	-
scipy.special.pbdn_seq	-
scipy.special.pbdv	-
scipy.special.pbdv_seq	-
scipy.special.pbvv	-
scipy.special.pbvv_seq	-
scipy.special.pbwa	-
scipy.special.pdtr	*cupyx.scipy.special.pdtr*
scipy.special.pdtrc	*cupyx.scipy.special.pdtrc*
scipy.special.pdtri	*cupyx.scipy.special.pdtri*
scipy.special.pdtrik	-
scipy.special.perm	-
scipy.special.poch	*cupyx.scipy.special.poch*
scipy.special.polygamma	*cupyx.scipy.special.polygamma*
scipy.special.powm1	-
scipy.special.pro_ang1	-
scipy.special.pro_ang1_cv	-
scipy.special.pro_cv	-
scipy.special.pro_cv_seq	-
scipy.special.pro_rad1	-
scipy.special.pro_rad1_cv	-
scipy.special.pro_rad2	-
scipy.special.pro_rad2_cv	-
scipy.special.ps_roots	-
scipy.special.pseudo_huber	*cupyx.scipy.special.pseudo_huber*
scipy.special.psi	*cupyx.scipy.special.psi*
scipy.special.radian	*cupyx.scipy.special.radian*
scipy.special.rel_entr	*cupyx.scipy.special.rel_entr*
scipy.special.rgamma	*cupyx.scipy.special.rgamma*
scipy.special.riccati_jn	-
scipy.special.riccati_yn	-
scipy.special.roots_chebyc	-
scipy.special.roots_chebys	-
scipy.special.roots_chebyt	-
scipy.special.roots_chebyu	-
scipy.special.roots_gegenbauer	-
scipy.special.roots_genlaguerre	-
scipy.special.roots_hermite	-
scipy.special.roots_hermitenorm	-
scipy.special.roots_jacobi	-
scipy.special.roots_laguerre	-
scipy.special.roots_legendre	-
scipy.special.roots_sh_chebyt	-
scipy.special.roots_sh_chebyu	-
scipy.special.roots_sh_jacobi	-
scipy.special.roots_sh_legendre	-

continues on next page

Table 17 – continued from previous page

SciPy	CuPy
scipy.special.round	*cupyx.scipy.special.round*
scipy.special.s_roots	-
scipy.special.seterr	-
scipy.special.sh_chebyt	-
scipy.special.sh_chebyu	-
scipy.special.sh_jacobi	-
scipy.special.sh_legendre	-
scipy.special.shichi	-
scipy.special.sici	-
scipy.special.sinc	*cupyx.scipy.special.sinc*
scipy.special.sindg	*cupyx.scipy.special.sindg*
scipy.special.smirnov	-
scipy.special.smirnovi	-
scipy.special.softmax	*cupyx.scipy.special.softmax*
scipy.special.spence	-
scipy.special.sph_harm	*cupyx.scipy.special.sph_harm*
scipy.special.spherical_in	-
scipy.special.spherical_jn	-
scipy.special.spherical_kn	-
scipy.special.spherical_yn	*cupyx.scipy.special.spherical_yn*
scipy.special.stdtr	-
scipy.special.stdtridf	-
scipy.special.stdtrit	-
scipy.special.struve	-
scipy.special.t_roots	-
scipy.special.tandg	*cupyx.scipy.special.tandg*
scipy.special.tklmbda	-
scipy.special.ts_roots	-
scipy.special.u_roots	-
scipy.special.us_roots	-
scipy.special.voigt_profile	-
scipy.special.wofz	-
scipy.special.wright_bessel	-
scipy.special.wrightomega	-
scipy.special.xlog1py	*cupyx.scipy.special.xlog1py*
scipy.special.xlogy	*cupyx.scipy.special.xlogy*
scipy.special.y0	*cupyx.scipy.special.y0*
scipy.special.y0_zeros	-
scipy.special.y1	*cupyx.scipy.special.y1*
scipy.special.y1_zeros	-
scipy.special.y1p_zeros	-
scipy.special.yn	*cupyx.scipy.special.yn*
scipy.special.yn_zeros	-
scipy.special.ynp_zeros	-
scipy.special.yv	-
scipy.special.yve	-
scipy.special.yvp	-
scipy.special.zeta	*cupyx.scipy.special.zeta*
scipy.special.zetac	*cupyx.scipy.special.zetac*

Statistical Functions

SciPy	CuPy
scipy.stats.BootstrapMethod	-
scipy.stats.CensoredData	-
scipy.stats.Covariance	-
scipy.stats.MonteCarloMethod	-
scipy.stats.PermutationMethod	-
scipy.stats.alexandergovern	-
scipy.stats.alpha	-
scipy.stats.anderson	-
scipy.stats.anderson_ksamp	-
scipy.stats.anglit	-
scipy.stats.ansari	-
scipy.stats.arcsine	-
scipy.stats.argus	-
scipy.stats.barnard_exact	-
scipy.stats.bartlett	-
scipy.stats.bayes_mvs	-
scipy.stats.bernoulli	-
scipy.stats.beta	-
scipy.stats.betabinom	-
scipy.stats.betaprime	-
scipy.stats.binned_statistic	-
scipy.stats.binned_statistic_2d	-
scipy.stats.binned_statistic_dd	-
scipy.stats.binom	-
scipy.stats.binom_test	-
scipy.stats.binomtest	-
scipy.stats.boltzmann	-
scipy.stats.bootstrap	-
scipy.stats.boschloo_exact	-
scipy.stats.boxcox	-
scipy.stats.boxcox_llf	*cupyx.scipy.stats.boxcox_llf*
scipy.stats.boxcox_normmax	-
scipy.stats.boxcox_normplot	-
scipy.stats.bradford	-
scipy.stats.brunnermunzel	-
scipy.stats.burr	-
scipy.stats.burr12	-
scipy.stats.cauchy	-
scipy.stats.chi	-
scipy.stats.chi2	-
scipy.stats.chi2_contingency	-
scipy.stats.chisquare	-
scipy.stats.circmean	-
scipy.stats.circstd	-
scipy.stats.circvar	-
scipy.stats.combine_pvalues	-
scipy.stats.cosine	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
scipy.stats.cramervonmises	-
scipy.stats.cramervonmises_2samp	-
scipy.stats.crystalball	-
scipy.stats.cumfreq	-
scipy.stats.describe	-
scipy.stats.dgamma	-
scipy.stats.differential_entropy	-
scipy.stats.directional_stats	-
scipy.stats.dirichlet	-
scipy.stats.dirichlet_multinomial	-
scipy.stats.dlaplace	-
scipy.stats.dunnett	-
scipy.stats.dweibull	-
scipy.stats.ecdf	-
scipy.stats.energy_distance	-
scipy.stats.entropy	*cupyx.scipy.stats.entropy*
scipy.stats.epps_singleton_2samp	-
scipy.stats.erlang	-
scipy.stats.expectile	-
scipy.stats.expon	-
scipy.stats.exponnorm	-
scipy.stats.exponpow	-
scipy.stats.exponweib	-
scipy.stats.f	-
scipy.stats.f_oneway	-
scipy.stats.false_discovery_control	-
scipy.stats.fatiguelife	-
scipy.stats.find_repeats	-
scipy.stats.fisher_exact	-
scipy.stats.fisk	-
scipy.stats.fit	-
scipy.stats.fligner	-
scipy.stats.foldcauchy	-
scipy.stats.foldnorm	-
scipy.stats.friedmanchisquare	-
scipy.stats.gamma	-
scipy.stats.gausshyper	-
scipy.stats.gaussian_kde	-
scipy.stats.genexpon	-
scipy.stats.genextreme	-
scipy.stats.gengamma	-
scipy.stats.genhalflogistic	-
scipy.stats.genhyperbolic	-
scipy.stats.geninvgauss	-
scipy.stats.genlogistic	-
scipy.stats.gennorm	-
scipy.stats.genpareto	-
scipy.stats.geom	-
scipy.stats.gibrat	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
scipy.stats.gmean	-
scipy.stats.gompertz	-
scipy.stats.goodness_of_fit	-
scipy.stats.gstd	-
scipy.stats.gumbel_l	-
scipy.stats.gumbel_r	-
scipy.stats.gzscore	-
scipy.stats.halfcauchy	-
scipy.stats.halfgennorm	-
scipy.stats.halflogistic	-
scipy.stats.halfnorm	-
scipy.stats.hmean	-
scipy.stats.hypergeom	-
scipy.stats.hypsecant	-
scipy.stats.invgamma	-
scipy.stats.invgauss	-
scipy.stats.invweibull	-
scipy.stats.invwishart	-
scipy.stats.iqr	-
scipy.stats.jarque_bera	-
scipy.stats.johnsonsb	-
scipy.stats.johnsonsu	-
scipy.stats.kappa3	-
scipy.stats.kappa4	-
scipy.stats.kendalltau	-
scipy.stats.kruskal	-
scipy.stats.ks_1samp	-
scipy.stats.ks_2samp	-
scipy.stats.ksone	-
scipy.stats.kstat	-
scipy.stats.kstatvar	-
scipy.stats.kstest	-
scipy.stats.kstwo	-
scipy.stats.kstwobign	-
scipy.stats.kurtosis	-
scipy.stats.kurtosistest	-
scipy.stats.laplace	-
scipy.stats.laplace_asymmetric	-
scipy.stats.levene	-
scipy.stats.levy	-
scipy.stats.levy_l	-
scipy.stats.levy_stable	-
scipy.stats.linregress	-
scipy.stats.loggamma	-
scipy.stats.logistic	-
scipy.stats.loglaplace	-
scipy.stats.lognorm	-
scipy.stats.logrank	-
scipy.stats.logser	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
scipy.stats.loguniform	-
scipy.stats.lomax	-
scipy.stats.mannwhitneyu	-
scipy.stats.matrix_normal	-
scipy.stats.maxwell	-
scipy.stats.median_abs_deviation	-
scipy.stats.median_test	-
scipy.stats.mielke	-
scipy.stats.mode	-
scipy.stats.moment	-
scipy.stats.monte_carlo_test	-
scipy.stats.mood	-
scipy.stats.moyal	-
scipy.stats.multinomial	-
scipy.stats.multiscale_graphcorr	-
scipy.stats.multivariate_hypergeom	-
scipy.stats.multivariate_normal	-
scipy.stats.multivariate_t	-
scipy.stats.mvsdist	-
scipy.stats.nakagami	-
scipy.stats.nbinom	-
scipy.stats.ncf	-
scipy.stats.nchypergeom_fisher	-
scipy.stats.nchypergeom_wallenius	-
scipy.stats.nct	-
scipy.stats.ncx2	-
scipy.stats.nhypergeom	-
scipy.stats.norm	-
scipy.stats.normaltest	-
scipy.stats.norminvgauss	-
scipy.stats.obrientransform	-
scipy.stats.ortho_group	-
scipy.stats.page_trend_test	-
scipy.stats.pareto	-
scipy.stats.pearson3	-
scipy.stats.pearsonr	-
scipy.stats.percentileofscore	-
scipy.stats.permutation_test	-
scipy.stats.planck	-
scipy.stats.pmean	-
scipy.stats.pointbiserialr	-
scipy.stats.poisson	-
scipy.stats.poisson_means_test	-
scipy.stats.power_divergence	-
scipy.stats.powerlaw	-
scipy.stats.powerlognorm	-
scipy.stats.powernorm	-
scipy.stats.ppcc_max	-
scipy.stats.ppcc_plot	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
scipy.stats.probplot	-
scipy.stats.randint	-
scipy.stats.random_correlation	-
scipy.stats.random_table	-
scipy.stats.rankdata	-
scipy.stats.ranksums	-
scipy.stats.rayleigh	-
scipy.stats.rdist	-
scipy.stats.recipinvgauss	-
scipy.stats.reciprocal	-
scipy.stats.rel_breitwigner	-
scipy.stats.relfreq	-
scipy.stats.rice	-
scipy.stats.rv_continuous	-
scipy.stats.rv_discrete	-
scipy.stats.rv_histogram	-
scipy.stats.rvs_ratio_uniforms	-
scipy.stats.scoreatpercentile	-
scipy.stats.sem	-
scipy.stats.semicircular	-
scipy.stats.shapiro	-
scipy.stats.siegelslopes	-
scipy.stats.sigmaclip	-
scipy.stats.skellam	-
scipy.stats.skew	-
scipy.stats.skewcauchy	-
scipy.stats.skewnorm	-
scipy.stats.skewtest	-
scipy.stats.sobol_indices	-
scipy.stats.somersd	-
scipy.stats.spearmanr	-
scipy.stats.special_ortho_group	-
scipy.stats.studentized_range	-
scipy.stats.t	-
scipy.stats.theilslopes	-
scipy.stats.tiecorrect	-
scipy.stats.tmax	-
scipy.stats.tmean	-
scipy.stats.tmin	-
scipy.stats.trapezoid	-
scipy.stats.trapz	-
scipy.stats.triang	-
scipy.stats.trim1	-
scipy.stats.trim_mean	*cupyx.scipy.stats.trim_mean*
scipy.stats.trimboth	-
scipy.stats.truncexpon	-
scipy.stats.truncnorm	-
scipy.stats.truncpareto	-
scipy.stats.truncweibull_min	-

continues on next page

Table 18 – continued from previous page

SciPy	CuPy
scipy.stats.tsem	-
scipy.stats.tstd	-
scipy.stats.ttest_1samp	-
scipy.stats.ttest_ind	-
scipy.stats.ttest_ind_from_stats	-
scipy.stats.ttest_rel	-
scipy.stats.tukey_hsd	-
scipy.stats.tukeylambda	-
scipy.stats.tvar	-
scipy.stats.uniform	-
scipy.stats.uniform_direction	-
scipy.stats.unitary_group	-
scipy.stats.variation	-
scipy.stats.vonmises	-
scipy.stats.vonmises_fisher	-
scipy.stats.vonmises_line	-
scipy.stats.wald	-
scipy.stats.wasserstein_distance	-
scipy.stats.weibull_max	-
scipy.stats.weibull_min	-
scipy.stats.weightedtau	-
scipy.stats.wilcoxon	-
scipy.stats.wishart	-
scipy.stats.wrapcauchy	-
scipy.stats.yeojohnson	-
scipy.stats.yeojohnson_llf	-
scipy.stats.yeojohnson_normmax	-
scipy.stats.yeojohnson_normplot	-
scipy.stats.yulesimon	-
scipy.stats.zipf	-
scipy.stats.zipfian	-
scipy.stats.zmap	*cupyx.scipy.stats.zmap*
scipy.stats.zscore	*cupyx.scipy.stats.zscore*

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()*.

Array(*args, **kwargs)	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.

CHAPTER

SIX

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 prelease 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.

CHAPTER

SEVEN

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

Previoulys, 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

Page 968, 1

CuPy	CC	CUDA	ROCm	cuTENSOR	NCCL	cuDNN	Python	NumPy	SciPy	Baseline API Spec.	Docs
v14	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
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	stable
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	1.7~1.11	NumPy 1.24 & SciPy 1.9	v12.3.0
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	1.6~1.9	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	1.4~1.8	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	7.6~8.2	3.6~3.9	1.17~1.21	1.4~1.7	NumPy 1.20 & SciPy 1.6	v9.6.0
v8	3.0~8.x	9.0 & 9.2~11.2	3.x[2]	1.2	2.0~2.8	7.0~8.1	3.5~3.9	1.16~1.20	1.3~1.6	NumPy 1.19 & SciPy 1.5	v8.6.0
v7	3.0~8.x	8.0~11.0	2.x[2]	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

[1] CUDA Compute Capability
[2] Highly experimental support with limited features.

CHAPTER

EIGHT

LICENSE

8.1 NumPy

The CuPy is designed based on NumPy's API. CuPy's source code and documents contain the original NumPy ones.

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.

BIBLIOGRAPHY

[CT] See, for example, P. Alfeld, ''A trivariate Clough-Tocher scheme for tetrahedral data". Computer Aided Geometric Design, 1, 169 (1984); G. Farin, ''Triangular Bernstein-Bezier patches". Computer Aided Geometric Design, 3, 83 (1986).

[Nielson83] G. Nielson, ''A method for interpolating scattered data based upon a minimum norm network". Math. Comp., 40, 253 (1983).

[Renka84] R. J. Renka and A. K. Cline. ''A Triangle-based C1 interpolation method.", Rocky Mountain J. Math., 14, 223 (1984).

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z