dask-image Documentation

Release 0.2.0+12.g8ff0f16.dirty

John Kirkham

Mar 21, 2019

Contents

CHAPTER 1

Features

- Support focuses on Dask Arrays.
- Provides support for loading image files.
- Implements commonly used N-D filters.
- Includes a few N-D Fourier filters.
- Provides some functions for working with N-D label images.
- Supports a few N-D morphological operators.

CHAPTER 2

Contents

2.1 Installation

2.1.1 Stable release

To install dask-image, run this command in your terminal:

```
$ conda install -c conda-forge dask-image
```

This is the preferred method to install dask-image, as it will always install the most recent stable release.

If you don't have conda installed, you can download and install it with the Anaconda distribution here.

Alternatively, you can install dask-image with pip:

```
$ pip install dask-image
```

If you don't have pip installed, this *Python installation guide* can guide you through the process. http://docs.python-guide.org/en/latest/starting/installation/

2.1.2 From sources

The sources for dask-image can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/dask/dask-image
```

Or download the tarball:

```
$ curl  -OL https://github.com/dask/dask-image/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

2.2 Quickstart

2.2.1 Importing dask-image

Import dask image is with an underscore, like this example:

```
import dask_image.imread
import dask_image.ndfilters
```

2.2.2 Dask Examples

We highly recommend checking out the dask-image-quickstart.ipynb notebook (and any other dask-image example notebooks) at the dask-examples repository. You can find the dask-image quickstart notebook in the `applications` folder of this repository:

https://github.com/dask/dask-examples

All the example notebooks are available to launch with mybinder and test out interactively.

2.2.3 An Even Quicker Start

You can read files stored on disk into a dask array by passing the filename, or regex matching multiple filenames into `imread()`.

```
filename_pattern = 'path/to/image-*.png'
images = dask_image.imread.imread(filename_pattern)
```

If your images are parts of a much larger image, dask can stack, concatenate or block chunks together: http://docs.dask.org/en/latest/array-stack.html

Calling dask-image functions is also easy.

```
import dask_image.ndfilters
blurred_image = dask_image.ndfilters.gaussian_filter(images, sigma=10)
```

Many other functions can be applied to dask arrays. See the dask_array_documentation for more detail on general array operations.

```
result = function_name(images)
```

2.2.4 Further Reading

Good places to start include:

- The dask-image API documentation: http://image.dask.org/en/latest/api.html
- The documentation on working with dask arrays: http://docs.dask.org/en/latest/array.html

2.3 API

2.3.1 dask_image package

Subpackages

dask_image.imread package

dask_image.imread.**imread**(*fname*, *nframes=1*)

Read image data into a Dask Array.

Provides a simple, fast mechanism to ingest image data into a Dask Array.

Parameters

- **fname** (*str*) – A glob like string that may match one or multiple filenames.
- **nframes** (*int, optional*) – Number of the frames to include in each chunk (default: 1).

Returns **array** – A Dask Array representing the contents of all image files.

Return type dask.array.Array

dask_image.ndfilters package

dask_image.ndfilters.**convolve**(*input*, *weights*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.convolve"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multidimensional convolution.

The array is convolved with the given kernel.

Parameters

- **input** (*array_like*) – The input array.
- **weights** (*array_like*) – Array of weights, same number of dimensions as input
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **result** – The result of convolution of *input* with *weights*.

Return type ndarray

See also:

correlate() Correlate an image with a kernel.

Notes

Each value in result is $C_i = \sum_j I_{i+k-j} W_j$, where W is the *weights* kernel, j is the n-D spatial index over W, I is the *input* and k is the coordinate of the center of W, specified by *origin* in the input parameters.

Examples

Perhaps the simplest case to understand is `mode='constant', cval=0.0`, because in this case borders (i.e. where the *weights* kernel, centered on any one value, extends beyond an edge of *input*) are treated as zeros.

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> k = np.array([[1,1,1],[1,1,0],[1,0,0]])
>>> from scipy import ndimage
>>> ndimage.convolve(a, k, mode='constant', cval=0.0)
array([[11, 10,  7,  4],
       [10,  3, 11, 11],
       [15, 12, 14,  7],
       [12,  3,  7,  0]])
```

Setting `cval=1.0` is equivalent to padding the outer edge of *input* with 1.0's (and then extracting only the original region of the result).

```
>>> ndimage.convolve(a, k, mode='constant', cval=1.0)
array([[13, 11,  8,  7],
       [11,  3, 11, 14],
       [16, 12, 14, 10],
       [15,  6, 10,  5]])
```

With `mode='reflect'` (the default), outer values are reflected at the edge of *input* to fill in missing values.

```
>>> b = np.array([[2, 0, 0],
...               [1, 0, 0],
...               [0, 0, 0]])
```

(continues on next page)

(continued from previous page)

```
>>> k = np.array([[0,1,0], [0,1,0], [0,1,0]])
>>> ndimage.convolve(b, k, mode='reflect')
array([[5, 0, 0],
       [3, 0, 0],
       [1, 0, 0]])
```

This includes diagonally at the corners.

```
>>> k = np.array([[1,0,0],[0,1,0],[0,0,1]])
>>> ndimage.convolve(b, k)
array([[4, 2, 0],
       [3, 2, 0],
       [1, 1, 0]])
```

With `mode='nearest'`, the single nearest value in to an edge in *input* is repeated as many times as needed to match the overlapping *weights*.

```
>>> c = np.array([[2, 0, 1],
...               [1, 0, 0],
...               [0, 0, 0]])
>>> k = np.array([[0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0]])
>>> ndimage.convolve(c, k, mode='nearest')
array([[7, 0, 3],
       [5, 0, 2],
       [3, 0, 1]])
```

dask_image.ndfilters.**correlate**(*input*, *weights*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.correlate"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional correlation.

The array is correlated with the given kernel.

Parameters

- **input** (*array_like*) – The input array.
- **weights** (*ndarray*) – array of weights, same number of dimensions as input
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

See also:

convolve() Convolve an image with a kernel.

dask_image.ndfilters.**gaussian_filter**(*input*, *sigma*, *order=0*, *mode='reflect'*, *cval=0.0*, *truncate=4.0*)

Wrapped copy of "scipy.ndimage.filters.gaussian_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multidimensional Gaussian filter.

Parameters

- **input** (*array_like*) – The input array.
- **sigma** (*scalar or sequence of scalars*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **order** (*int or sequence of ints, optional*) – The order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – 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 **gaussian_filter** – Returned array of same shape as *input*.

Return type ndarray

Notes

The multidimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Examples

```
>>> from scipy.ndimage import gaussian_filter
>>> a = np.arange(50, step=2).reshape((5,5))
>>> a
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]])
>>> gaussian_filter(a, sigma=1)
array([[ 4,  6,  8,  9, 11],
       [10, 12, 14, 15, 17],
       [20, 22, 24, 25, 27],
       [29, 31, 33, 34, 36],
       [35, 37, 39, 40, 42]])
```

```
>>> from scipy import misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = gaussian_filter(ascent, sigma=5)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**gaussian_gradient_magnitude**(*input*, *sigma*, *mode='reflect'*, *cval=0.0*, *truncate=4.0*, ***kwargs*)

Wrapped copy of "scipy.ndimage.filters.gaussian_gradient_magnitude"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multidimensional gradient magnitude using Gaussian derivatives.

Parameters

- **input** (*array_like*) – The input array.

- **sigma** (*scalar or sequence of scalars*) – The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes..
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **keyword arguments will be passed to gaussian_filter()** (*Extra*) –

Returns **gaussian_gradient_magnitude** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.gaussian_gradient_magnitude(ascent, sigma=5)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**gaussian_laplace**(*input*, *sigma*, *mode='reflect'*, *cval=0.0*, *truncate=4.0*, ***kwargs*)

Wrapped copy of "scipy.ndimage.filters.gaussian_laplace"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multidimensional Laplace filter using gaussian second derivatives.

Parameters

- **input** (*array_like*) – The input array.

- **sigma** (*scalar or sequence of scalars*) – The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.
- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **keyword arguments will be passed to gaussian_filter()** (*Extra*) –

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> ascent = misc.ascent()
```

```
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
```

```
>>> result = ndimage.gaussian_laplace(ascent, sigma=1)
>>> ax1.imshow(result)
```

```
>>> result = ndimage.gaussian_laplace(ascent, sigma=3)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**generic_filter**(*input*, *function*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*, *extra_arguments=()*, *extra_keywords={}*)

Wrapped copy of "scipy.ndimage.filters.generic_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multi-dimensional filter using the given function.

At each element the provided function is called. The input values within the filter footprint at that element are passed to the function as a 1D array of double values.

Parameters

- **input** (*array_like*) – The input array.
- **function** (*{callable, scipy.LowLevelCallable}*) – Function to apply at each element.
- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.
- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.
- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.
- **extra_arguments** (*sequence, optional*) – Sequence of extra positional arguments to pass to passed function.
- **extra_keywords** (*dict, optional*) – dict of extra keyword arguments to pass to passed function.

Notes

This function also accepts low-level callback functions with one of the following signatures and wrapped in *scipy.LowLevelCallable*:

```
int callback(double *buffer, npy_intp filter_size,
             double *return_value, void *user_data)
int callback(double *buffer, intptr_t filter_size,
             double *return_value, void *user_data)
```

The calling function iterates over the elements of the input and output arrays, calling the callback function at each element. The elements within the footprint of the filter at the current element are passed through the `buffer` parameter, and the number of elements within the footprint through `filter_size`. The calculated value is returned in `return_value`. `user_data` is the data pointer provided to *scipy.LowLevelCallable* as-is.

The callback function must return an integer error status that is zero if something went wrong and one otherwise. If an error occurs, you should normally set the python error status with an informative message before returning, otherwise a default error message is set by the calling function.

In addition, some other low-level function pointer specifications are accepted, but these are for backward compatibility only and should not be used in new code.

`dask_image.ndfilters.`**`laplace`**(*input*, *mode='reflect'*, *cval=0.0*)

Wrapped copy of "scipy.ndimage.filters.laplace"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

N-dimensional Laplace filter based on approximate second derivatives.

Parameters

- **`input`** (*`array_like`*) – The input array.
- **`mode`** (*`str or sequence, optional`*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **`cval`** (*`scalar, optional`*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
```

(continues on next page)

(continued from previous page)

```
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.laplace(ascent)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**maximum_filter**(*input*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.maximum_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multi-dimensional maximum filter.

Parameters

- **input** (*array_like*) – The input array.
- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.
- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **maximum_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.maximum_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**median_filter**(*input*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.median_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multidimensional median filter.

Parameters

- **input** (*array_like*) – The input array.
- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.
- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **median_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.median_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**minimum_filter**(*input*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.minimum_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multi-dimensional minimum filter.

Parameters

- **input** (*array_like*) – The input array.
- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.
- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **minimum_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.minimum_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**percentile_filter**(*input*, *percentile*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.percentile_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multi-dimensional percentile filter.

Parameters

- **input** (*array_like*) – The input array.
- **percentile** (*scalar*) – The percentile parameter may be less then zero, i.e., percentile = -20 equals percentile = 80
- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.

- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **percentile_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.percentile_filter(ascent, percentile=20, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**prewitt**(*input*, *axis=-1*, *mode='reflect'*, *cval=0.0*)

Wrapped copy of "scipy.ndimage.filters.prewitt"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a Prewitt filter.

Parameters

- **input** (*array_like*) – The input array.
- **axis** (*int, optional*) – The axis of *input* along which to calculate. Default is -1.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.prewitt(ascent)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**rank_filter**(*input*, *rank*, *size=None*, *footprint=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.rank_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a multi-dimensional rank filter.

Parameters

- **input** (*array_like*) – The input array.
- **rank** (*int*) – The rank parameter may be less then zero, i.e., rank = -1 indicates the largest element.

- **size** (*scalar or tuple, optional*) – See footprint, below. Ignored if footprint is given.
- **footprint** (*array, optional*) – Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2). When *footprint* is given, *size* is ignored.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **rank_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.rank_filter(ascent, rank=42, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**sobel**(*input*, *axis=-1*, *mode='reflect'*, *cval=0.0*)

Wrapped copy of "scipy.ndimage.filters.sobel"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Calculate a Sobel filter.

Parameters

- **input** (*array_like*) – The input array.
- **axis** (*int, optional*) – The axis of *input* along which to calculate. Default is -1.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.sobel(ascent)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfilters.**uniform_filter**(*input*, *size=3*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Wrapped copy of "scipy.ndimage.filters.uniform_filter"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional uniform filter.

Parameters

- **input** (*array_like*) – The input array.
- **size** (*int or sequence of ints, optional*) – The sizes of the uniform filter are given for each axis as a sequence, or as a single number, in which case the size is equal for all axes.
- **mode** (*str or sequence, optional*) – The *mode* parameter determines how the input array is extended when the filter overlaps a border. By passing a sequence of modes with length equal to the number of dimensions of the input array, different modes can be specified along each axis. Default value is 'reflect'. The valid values and their behavior is as follows:

  'reflect' (*d c b a | a b c d | d c b a*) The input is extended by reflecting about the edge of the last pixel.

  'constant' (*k k k k | a b c d | k k k k*) The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  'nearest' (*a a a a | a b c d | d d d d*) The input is extended by replicating the last pixel.

  'mirror' (*d c b | a b c d | c b a*) The input is extended by reflecting about the center of the last pixel.

  'wrap' (*a b c d | a b c d | a b c d*) The input is extended by wrapping around to the opposite edge.

- **cval** (*scalar, optional*) – Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
- **origin** (*int or sequence, optional*) – Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right. By passing a sequence of origins with length equal to the number of dimensions of the input array, different shifts can be specified along each axis.

Returns **uniform_filter** – Filtered array. Has the same shape as *input*.

Return type ndarray

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.uniform_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```

dask_image.ndfourier package

dask_image.ndfourier.**fourier_gaussian**(*input*, *sigma*, *n=-1*, *axis=-1*)

Multi-dimensional Gaussian fourier filter.

The array is multiplied with the fourier transform of a Gaussian kernel.

Parameters

- **input** (*array_like*) – The input array.
- **sigma** (*float or sequence*) – 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.

Returns **fourier_gaussian**

Return type Dask Array

Examples

```
>>> from scipy import ndimage, misc
>>> import numpy.fft
>>> import matplotlib.pyplot as plt
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_gaussian(input_, sigma=4)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
```

dask_image.ndfourier.**fourier_shift**(*input*, *shift*, *n=-1*, *axis=-1*)

Multi-dimensional fourier shift filter.

The array is multiplied with the fourier transform of a shift operation.

Parameters

- **input** (*array_like*) – The input array.
- **shift** (*float or sequence*) – The size of the box used for filtering. 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.

Returns **fourier_shift**

Return type Dask Array

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> import numpy.fft
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_shift(input_, shift=200)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```

dask_image.ndfourier.**fourier_uniform**(*input*, *size*, *n=-1*, *axis=-1*)

Multi-dimensional uniform fourier filter.

The array is multiplied with the fourier transform of a box of given size.

Parameters

- **input** (*array_like*) – The input array.
- **size** (*float or sequence*) – 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.

Returns **fourier_uniform** – The filtered input. If *output* is given as a parameter, None is returned.

Return type Dask Array

Examples

```
>>> from scipy import ndimage, misc
>>> import numpy.fft
>>> import matplotlib.pyplot as plt
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_uniform(input_, size=20)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```

dask_image.ndmeasure package

dask_image.ndmeasure.**center_of_mass**(*input*, *labels=None*, *index=None*)

Find the center of mass over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **center_of_mass** – Coordinates of centers-of-mass of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**extrema**(*input*, *labels=None*, *index=None*)

Find the min and max with positions over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **minimums, maximums, min_positions, max_positions** – Values and coordinates of minimums and maximums in each feature.

Return type tuple of ndarrays

dask_image.ndmeasure.**histogram**(*input*, *min*, *max*, *bins*, *labels=None*, *index=None*)

Find the histogram over an image at specified subregions.

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** (*ndarray*) – N-D image data
- **min** (*int*) – Minimum value of range of histogram bins.
- **max** (*int*) – Maximum value of range of histogram bins.
- **bins** (*int*) – Number of bins.
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **histogram** – Histogram of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**label**(*input*, *structure=None*)

Label features in an array.

Parameters

- **input** (*ndarray*) – An array-like object to be labeled. Any non-zero values in `input` are counted as features and zero values are considered the background.
- **structure** (*ndarray, optional*) – A structuring element that defines feature connections. `structure` must be symmetric. If no structuring element is provided, one is automatically generated with a squared connectivity equal to one. That is, for a 2-D `input` array, the default structuring element is:

```
[[0,1,0],
 [1,1,1],
 [0,1,0]]
```

Returns

- **label** (*ndarray or int*) – An integer ndarray where each unique feature in `input` has a unique label in the returned array.
- **num_features** (*int*) – How many objects were found.

dask_image.ndmeasure.**labeled_comprehension**(*input*, *labels*, *index*, *func*, *out_dtype*, *default*, *pass_positions=False*)

Compute a function over an image at specified subregions.

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** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.
- **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. Default is False.

Returns result – Result of applying `func` on `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**maximum**(*input*, *labels=None*, *index=None*)

Find the maxima over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data

- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **maxima** – Maxima of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**maximum_position**(*input*, *labels=None*, *index=None*)

Find the positions of maxima over an image at specified subregions.

For each region specified by `labels`, the position of the maximum value of `input` within the region is returned.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **maxima_positions** – Maxima positions of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**mean**(*input*, *labels=None*, *index=None*)

Find the mean over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **means** – Mean of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmeasure.**median**(*input*, *labels=None*, *index=None*)

Find the median over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

The `index` argument only works when `labels` is specified.

Returns **medians** – Median of `input` over the `index` selected regions from `labels`.

Return type ndarray

`dask_image.ndmeasure.`**`minimum`**(*input*, *labels=None*, *index=None*)

Find the minima over an image at specified subregions.

Parameters

- **`input`** (*`ndarray`*) – N-D image data
- **`labels`** (*`ndarray, optional`*) – Image features noted by integers. If None (default), all values.
- **`index`** (*`int or sequence of ints, optional`*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **minima** – Minima of `input` over the `index` selected regions from `labels`.

Return type ndarray

`dask_image.ndmeasure.`**`minimum_position`**(*input*, *labels=None*, *index=None*)

Find the positions of minima over an image at specified subregions.

Parameters

- **`input`** (*`ndarray`*) – N-D image data
- **`labels`** (*`ndarray, optional`*) – Image features noted by integers. If None (default), all values.
- **`index`** (*`int or sequence of ints, optional`*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **minima_positions** – Maxima positions of `input` over the `index` selected regions from `labels`.

Return type ndarray

`dask_image.ndmeasure.`**`standard_deviation`**(*input*, *labels=None*, *index=None*)

Find the standard deviation over an image at specified subregions.

Parameters

- **`input`** (*`ndarray`*) – N-D image data
- **`labels`** (*`ndarray, optional`*) – Image features noted by integers. If None (default), all values.
- **`index`** (*`int or sequence of ints, optional`*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **standard_deviation** – Standard deviation of `input` over the `index` selected regions from `labels`.

Return type ndarray

`dask_image.ndmeasure.`**`sum`**(*input*, *labels=None*, *index=None*)

Find the sum over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **sum** – Sum of `input` over the `index` selected regions from `labels`.

Return type ndarray

`dask_image.ndmeasure.`**variance**(*input*, *labels=None*, *index=None*)

Find the variance over an image at specified subregions.

Parameters

- **input** (*ndarray*) – N-D image data
- **labels** (*ndarray, optional*) – Image features noted by integers. If None (default), all values.
- **index** (*int or sequence of ints, optional*) – Labels to include in output. If None (default), all values where non-zero `labels` are used.

  The `index` argument only works when `labels` is specified.

Returns **variance** – Variance of `input` over the `index` selected regions from `labels`.

Return type ndarray

dask_image.ndmorph package

`dask_image.ndmorph.`**binary_closing**(*input*, *structure=None*, *iterations=1*, *origin=0*)

Wrapped copy of "scipy.ndimage.morphology.binary_closing"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional 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** (*array_like*) – Binary array_like to be closed. Non-zero (True) elements form the subset to be closed.
- **structure** (*array_like, optional*) – Structuring element used for the closing. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one (i.e., only nearest neighbors are connected to the center, diagonally-connected elements are not considered neighbors).
- **iterations** (*{int, float}, optional*) – The dilation step of the closing, then the erosion step are each repeated *iterations* times (one, by default). If iterations is less than 1, each operations is repeated until the result does not change anymore.

- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.
- **mask** (*array_like, optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.

  New in version 1.1.0.
- **border_value** (*int (cast to 0 or 1), optional*) – Value at the border in the output array.

  New in version 1.1.0.
- **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 in the current iteration; if true al pixels are considered as candidates for update, regardless of what happened in the previous iteration. False by default.

  New in version 1.1.0.

Returns **binary_closing** – Closing of the input by the structuring element.

Return type ndarray of bools

See also:

`grey_closing()`, *binary_opening()*, *binary_dilation()*, *binary_erosion()*, `generate_binary_structure()`

Notes

Closing [1]_ is a mathematical morphology operation [2]_ that consists in the succession of a dilation and an erosion of the input with the same structuring element. Closing therefore fills holes smaller than the structuring element.

Together with *opening* (*binary_opening*), closing can be used for noise removal.

References

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:-1, 1:-1] = 1; a[2,2] = 0
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Closing removes small holes
>>> ndimage.binary_closing(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Closing is the erosion of the dilation of the input
```

(continues on next page)

(continued from previous page)

```
>>> ndimage.binary_dilation(a).astype(int)
array([[0, 1, 1, 1, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 1, 1, 1, 0]])
>>> ndimage.binary_erosion(ndimage.binary_dilation(a)).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
```

```
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 2:5] = 1; a[1:3,3] = 0
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # In addition to removing holes, closing can also
>>> # coarsen boundaries with fine hollows.
>>> ndimage.binary_closing(a).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_closing(a, structure=np.ones((2,2))).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

dask_image.ndmorph.**binary_dilation**(*input*, *structure=None*, *iterations=1*, *mask=None*, *border_value=0*, *origin=0*, *brute_force=False*)

Wrapped copy of "scipy.ndimage.morphology.binary_dilation"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional binary dilation with the given structuring element.

Parameters

- **input** (*array_like*) – Binary array_like to be dilated. Non-zero (True) elements form the subset to be dilated.
- **structure** (*array_like, optional*) – 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.

- **iterations** (*{int, float}, 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.
- **mask** (*array_like, optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.
- **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. False by default.

Returns **binary_dilation** – Dilation of the input by the structuring element.

Return type ndarray of bools

See also:

`grey_dilation()`, *`binary_erosion()`*, *`binary_closing()`*, *`binary_opening()`*, `generate_binary_structure()`

Notes

Dilation [1]_ is a mathematical morphology operation [2]_ that uses a structuring element for expanding the shapes in an image. The binary dilation of an image by a structuring element is the locus of the points covered by the structuring element, when its center lies within the non-zero points of the image.

References

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a)
array([[False, False, False, False, False],
       [False, False,  True, False, False],
       [False,  True,  True,  True, False],
       [False, False,  True, False, False],
       [False, False, False, False, False]], dtype=bool)
>>> ndimage.binary_dilation(a).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

(continues on next page)

(continued from previous page)

```
>>> # 3x3 structuring element with connectivity 1, used by default
>>> struct1 = ndimage.generate_binary_structure(2, 1)
>>> struct1
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> # 3x3 structuring element with connectivity 2
>>> struct2 = ndimage.generate_binary_structure(2, 2)
>>> struct2
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
>>> ndimage.binary_dilation(a, structure=struct1).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a, structure=struct2).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a, structure=struct1,\
... iterations=2).astype(a.dtype)
array([[ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.]])
```

dask_image.ndmorph.**binary_erosion**(*input*, *structure=None*, *iterations=1*, *mask=None*, *border_value=0*, *origin=0*, *brute_force=False*)

Wrapped copy of "scipy.ndimage.morphology.binary_erosion"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional binary erosion with a given structuring element.

Binary erosion is a mathematical morphology operation used for image processing.

Parameters

- **input** (*array_like*) – Binary image to be eroded. Non-zero (True) elements form the subset to be eroded.
- **structure** (*array_like, optional*) – 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.
- **iterations** (*{int, float}, 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.
- **mask** (*array_like, optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.

- **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. False by default.

Returns **binary_erosion** – Erosion of the input by the structuring element.

Return type ndarray of bools

See also:

`grey_erosion()`, *`binary_dilation()`*, *`binary_closing()`*, *`binary_opening()`*, `generate_binary_structure()`

Notes

Erosion [1]_ is a mathematical morphology operation [2]_ that uses a structuring element for shrinking the shapes in an image. The binary erosion of an image by a structuring element is the locus of the points where a superimposition of the structuring element centered on the point is entirely contained in the set of non-zero elements of the image.

References

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> #Erosion removes objects smaller than the structure
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

dask_image.ndmorph.**binary_opening**(*input*, *structure=None*, *iterations=1*, *origin=0*)

Wrapped copy of "scipy.ndimage.morphology.binary_opening"

Excludes the output parameter as it would not work with Dask arrays.

Original docstring:

Multi-dimensional 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** (*array_like*) – Binary array_like to be opened. Non-zero (True) elements form the subset to be opened.
- **structure** (*array_like, optional*) – Structuring element used for the opening. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one (i.e., only nearest neighbors are connected to the center, diagonally-connected elements are not considered neighbors).
- **iterations** (*{int, float}, optional*) – The erosion step of the opening, then the dilation step are each repeated *iterations* times (one, by default). If *iterations* is less than 1, each operation is repeated until the result does not change anymore.
- **origin** (*int or tuple of ints, optional*) – Placement of the filter, by default 0.
- **mask** (*array_like, optional*) – If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.

  New in version 1.1.0.
- **border_value** (*int (cast to 0 or 1), optional*) – Value at the border in the output array.

  New in version 1.1.0.
- **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 in the current iteration; if true all pixels are considered as candidates for update, regardless of what happened in the previous iteration. False by default.

  New in version 1.1.0.

Returns **binary_opening** – Opening of the input by the structuring element.

Return type ndarray of bools

See also:

grey_opening(), *binary_closing()*, *binary_erosion()*, *binary_dilation()*, generate_binary_structure()

Notes

Opening [1]_ is a mathematical morphology operation [2]_ that consists in the succession of an erosion and a dilation of the input with the same structuring element. Opening therefore removes objects smaller than the structuring element.

Together with *closing* (*binary_closing*), opening can be used for noise removal.

References

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:4, 1:4] = 1; a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> ndimage.binary_opening(a, structure=np.ones((3,3))).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening is the dilation of the erosion of the input
>>> ndimage.binary_erosion(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> ndimage.binary_dilation(ndimage.binary_erosion(a)).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

2.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dask/dask-image/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

dask-image could always use more documentation, whether as part of the official dask-image docs, in docstrings, or even on the web in blog posts, articles, and such.

To build the documentation locally and preview your changes, first set up the conda environment for building the dask-image documentation:

```
$ conda env create -f environment_doc.yml
$ conda activate dask_image_doc_env
```

This conda environment contains dask-image and its dependencies, sphinx, and the dask-sphinx-theme.

Next, build the documentation with sphinx:

```
$ cd dask-image/docs
$ make html
```

Now you can preview the html documentation in your browser by opening the file: dask-image/docs/_build/html/index.html

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dask/dask-image/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.4.2 Get Started!

Ready to contribute? Here's how to set up *dask-image* for local development.

1. Fork the *dask-image* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/dask-image.git
```

3. Install your local copy into an environment. Assuming you have conda installed, this is how you set up your fork for local development (on Windows drop *source*). Replace *"<some version>"* with the Python version used for testing.:

```
$ conda create -n dask-image-env python="<some version>"
$ source activate dask-image-env
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions:

```
$ flake8 dask_image tests
$ python setup.py test or py.test
```

   To get flake8, just conda install it into your environment.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for all supported Python versions. Check CIs and make sure that the tests pass for all supported Python versions and platforms.

2.4.4 Tips

To run a subset of tests:

```
$ py.test tests/test_dask_image.py
```

2.5 Credits

2.5.1 Development Lead

- John Kirkham <kirkhamj@janelia.hhmi.org>

2.5.2 Contributors

None yet. Why not be the first?

2.6 History

2.6.1 0.2.0 (2018-10-10)

- Construct separate label masks in *labeled_comprehension* (#82)
- Use *full* to construct 1-D NumPy array (#83)
- Use NumPy's *ndindex* in *labeled_comprehension* (#81)
- Cleanup *test_labeled_comprehension_struct* (#80)
- Use 1-D structured array fields for position-based kernels in *ndmeasure* (#79)
- Rewrite *center_of_mass* using *labeled_comprehension* (#78)
- Adjust *extrema*'s internal structured type handling (#77)
- Test labeled_comprehension with object type (#76)
- Rewrite *histogram* to use *labeled_comprehension* (#75)
- Use labeled_comprehension directly in more function in ndmeasure (#74)
- Update mean's variables to match other functions (#73)
- Consolidate summation in *_ravel_shape_indices* (#72)
- Update HISTORY for 0.1.2 release (#71)
- Bump dask-sphinx-theme to 1.1.0 (#70)

2.6.2 0.1.2 (2018-09-17)

- Ensure *labeled_comprehension*'s *default* is 1D. (#69)
- Bump dask-sphinx-theme to 1.0.5. (#68)
- Use nout=2 in ndmeasure's label. (#67)
- Use custom kernel for extrema. (#61)
- Handle structured dtype in labeled_comprehension. (#66)
- Fixes for *_unravel_index*. (#65)
- Bump dask-sphinx-theme to 1.0.4. (#64)
- Unwrap some lines. (#63)

- Use dask-sphinx-theme. (#62)
- Refactor out *_unravel_index* function. (#60)
- Divide *sigma* by *-2*. (#59)
- Use Python 3's definition of division in Python 2. (#58)
- Force dtype of *prod* in *_ravel_shape_indices*. (#57)
- Drop vendored compatibility code. (#54)
- Drop vendored copy of indices and uses thereof. (#56)
- Drop duplicate utility tests from *ndmorph*. (#55)
- Refactor utility module for imread. (#53)
- Reuse *ndfilter* utility function in *ndmorph*. (#52)
- Cleanup freq_grid_i construction in _get_freq_grid. (#51)
- Use shared Python 2/3 compatibility module. (#50)
- Consolidate Python 2/3 compatibility code. (#49)
- Refactor Python 2/3 compatibility from imread. (#48)
- Perform *2 * pi* first in *_get_ang_freq_grid*. (#47)
- Ensure *J* is negated first in *fourier_shift*. (#46)
- Breakout common changes in fourier_gaussian. (#45)
- Use conda-forge badge. (#44)

2.6.3 0.1.1 (2018-08-31)

- Fix a bug in an ndmeasure test of an internal function.

2.6.4 0.1.0 (2018-08-31)

- First release on PyPI.
- Pulls in content from dask-image org.
- Supports reading of image files into Dask.
- Provides basic N-D filters with options to extend.
- Provides a few N-D Fourier filters.
- Provides a few N-D morphological filters.
- Provides a few N-D measurement functions for label images.
- Has 100% line coverage in test suite.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

R

S

U

V