



photutils

An Astropy Package for Photometry

photutils Documentation

Release 0.3.2

The Photutils Developers

Mar 31, 2017

Contents

I Photutils at a glance	3
II User Documentation	21
III Reporting Issues	369
IV Contributing	373
V Citing Photutils	377



photutils

An `Astropy` Package for Photometry

Photutils is an affiliated package of [Astropy](#) to provide tools for detecting and performing photometry of astronomical sources. It is an open source (BSD licensed) Python package. Bug reports, comments, and help with development are very welcome.

Part I

Photutils at a glance

CHAPTER 1

Installation

Requirements

Photutils has the following strict requirements:

- Python 2.7, 3.3, 3.4, 3.5 or 3.6
- Numpy 1.8 or later
- Astropy 1.0 or later

Additionally, some functionality is available only if the following optional dependencies are installed:

- Scipy 0.15 or later
- scikit-image 0.11 or later
- scikit-learn 0.18 or later
- matplotlib 1.3 or later

Warning: While Photutils will import even if these dependencies are not installed, the functionality will be severely limited. It is very strongly recommended that you install Scipy and scikit-image to use Photutils. Both are easily installed via [pip](#) or [conda](#).

Installing the latest released version

The latest released (stable) version of Photutils can be installed either with [conda](#) or [pip](#).

Using conda

Photutils can be installed with [conda](#) using the [astropy](#) Anaconda channel:

```
conda install -c astropy photutils
```

Using pip

To install using [pip](#), simply run:

```
pip install --no-deps photutils
```

Note: You will need a C compiler (e.g. gcc or clang) to be installed for the installation to succeed.

Note: The --no-deps flag is optional, but highly recommended if you already have Numpy and Astropy installed, since otherwise pip will sometimes try to “help” you by upgrading your Numpy and Astropy installations, which may not always be desired.

Note: If you get a `PermissionError` this means that you do not have the required administrative access to install new packages to your Python installation. In this case you may consider using the `--user` option to install the package into your home directory. You can read more about how to do this in the [pip documentation](#).

Do **not** install Photutils or other third-party packages using sudo unless you are fully aware of the risks.

Installing the latest development version

Prerequisites

You will need [Cython](#) (0.15 or later), a compiler suite, and the development headers for Python and Numpy in order to build Photutils from the source distribution. On Linux, using the package manager for your distribution will usually be the easiest route, while on MacOS X you will need the XCode command line tools.

The [instructions for building Numpy from source](#) are also a good resource for setting up your environment to build Python packages.

Note: If you are using MacOS X, you will need the XCode command line tools. One way to get them is to install [XCode](#). If you are using OS X 10.7 (Lion) or later, you must also explicitly install the command line tools. You can do this by opening the XCode application, going to **Preferences**, then **Downloads**, and then under **Components**, click on the Install button to the right of **Command Line Tools**. Alternatively, on 10.7 (Lion) or later, you do not need to install XCode, you can download just the command line tools from <https://developer.apple.com/downloads/index.action> (requires an Apple developer account).

Building and installing Manually

Photutils is being developed on [github](#). The latest development version of the Photutils source code can be retrieved using git:

```
git clone https://github.com/astropy/photutils.git
```

Then, to build and install Photutils (from the root of the source tree):

```
cd photutils
python setup.py install
```

Building and installing using pip

Alternatively, `pip` can be used to retrieve, build, and install the latest development version from [github](#):

```
pip install --no-deps git+https://github.com/astropy/photutils.git
```

Note: The `--no-deps` flag is optional, but highly recommended if you already have Numpy and Astropy installed, since otherwise pip will sometimes try to “help” you by upgrading your Numpy and Astropy installations, which may not always be desired.

Note: If you get a `PermissionError` this means that you do not have the required administrative access to install new packages to your Python installation. In this case you may consider using the `--user` option to install the package into your home directory. You can read more about how to do this in the [pip documentation](#).

Do **not** install Photutils or other third-party packages using `sudo` unless you are fully aware of the risks.

Testing an installed Photutils

The easiest way to test your installed version of Photutils is running correctly is to use the `photutils.test()` function:

```
>>> import photutils  
>>> photutils.test()
```

The tests should run and report any failures, which you can report to the [Photutils issue tracker](#).

Note: This way of running the tests may not work if you start Python from within the Photutils source distribution directory.

CHAPTER 2

Overview

Introduction

Photutils contains functions for:

- estimating the background and background RMS in astronomical images
- detecting sources in astronomical images
- estimating morphological parameters of those sources (e.g., centroid and shape parameters)
- performing aperture and PSF photometry

The code and the documentation are available at the following links:

- Code: <https://github.com/astropy/photutils>
- Issue Tracker: <https://github.com/astropy/photutils/issues>
- Documentation: <https://photutils.readthedocs.io/>

Coordinate Conventions

In Photutils, pixel coordinates are zero-indexed, meaning that $(x, y) = (0, 0)$ corresponds to the center of the lowest, leftmost array element. This means that the value of `data[0, 0]` is taken as the value over the range $-0.5 < x \leq 0.5, -0.5 \leq y \leq 0.5$. Note that this differs from the `SourceExtractor`, `IRAF`, `FITS`, and `ds9` conventions, in which the center of the lowest, leftmost array element is $(1, 1)$.

The `x` (column) coordinate corresponds to the second (fast) array index and the `y` (row) coordinate corresponds to the first (slow) index. `data[y, x]` gives the value at coordinates (x, y) . Along with zero-indexing, this means that an array is defined over the coordinate range $-0.5 < x \leq \text{data.shape}[1] - 0.5, -0.5 \leq y \leq \text{data.shape}[0] - 0.5$.

Bundled Datasets

In this documentation, we use example datasets provided by calling functions such as `load_star_image()`. This function returns an Astropy `ImageHDU` object, and is equivalent to doing:

```
>>> from astropy.io import fits  
>>> hdu = fits.open('dataset.fits')[0]
```

where the `[0]` accesses the first HDU in the FITS file.

Contributors

For the complete list of contributors please see the [Photutils contributors page on Github](#).

CHAPTER 3

Getting Started with Photutils

The following example uses Photutils to find sources in an astronomical image and perform circular aperture photometry on them.

We start by loading an image from the bundled datasets and selecting a subset of the image. We then subtract a rough estimate of the background, calculated using the image median:

```
>>> import numpy as np
>>> from photutils import datasets
>>> hdu = datasets.load_star_image()
Downloading ...
>>> image = hdu.data[500:700, 500:700].astype(float)
>>> image -= np.median(image)
```

In the remainder of this example, we assume that the data is background-subtracted.

Photutils supports several source detection algorithms. For this example, we use `DAOStarFinder` to detect the stars in the image. We set the detection threshold at the 3-sigma noise level, estimated using the median absolute deviation (`mad_std`) of the image. The parameters of the detected sources are returned as an Astropy `Table`:

```
>>> from photutils import DAOStarFinder
>>> from astropy.stats import mad_std
>>> bkg_sigma = mad_std(image)
>>> daofind = DAOStarFinder(fwhm=4., threshold=3.*bkg_sigma)
>>> sources = daofind(image)
>>> print(sources)
   id      xcentroid     ycentroid     ...    peak      flux          mag
---  -----  -----  ...  -----  -----  ...
   1  182.838658938  0.167670190537  ...  3824.0  2.80283459469 -1.11899367311
   2  189.204308134  0.260813525338  ...  4913.0  3.87291850311 -1.47009589582
   3  5.79464911433  2.61254240807  ...  7752.0  4.1029107294 -1.53273016937
   4  36.8470627804  1.32202279582  ...  8739.0  7.43158178793 -2.17770315441
   5  3.2565602452  5.41895201748  ...  6935.0  3.81262984074 -1.45306160673
...
   148 124.313272579  188.305229159  ...  6702.0  6.63585429303 -2.05474210356
   149 24.2572074962  194.714942814  ...  8342.0  3.2671036996 -1.28540729858
```

```

150 116.449998422 195.059233325 ... 3299.0 2.87752205766 -1.1475466535
151 18.9580860645 196.342065132 ... 3854.0 2.38352961224 -0.943051379595
152 111.525751196 195.731917995 ... 8109.0 7.9278607401 -2.24789003194
Length = 152 rows

```

Using the list of source locations (`xcentroid` and `ycentroid`), we now compute the sum of the pixel values in circular apertures with a radius of 4 pixels. The `aperture_photometry()` function returns an Astropy `Table` with the results of the photometry:

```

>>> from photutils import aperture_photometry, CircularAperture
>>> positions = (sources['xcentroid'], sources['ycentroid'])
>>> apertures = CircularAperture(positions, r=4.)
>>> phot_table = aperture_photometry(image, apertures)
>>> print(phot_table)
   id      xcenter          ycenter      aperture_sum
    pix           pix
-----
 1  182.8386589381308  0.16767019053693752  18121.7594837
 2  189.20430813403388  0.26081352533766516  29836.5152158
 3  5.794649114329246   2.612542408073547  331979.819037
 4  36.84706278043582   1.3220227958153257  183705.093284
 5  3.2565602452007325  5.418952017476508  349468.978627
...
148   ...
149   ...
150 116.44999842177826 195.05923332483115 31232.9117818
151 18.958086064485013 196.3420651316401 162076.262752
152 111.52575119605933 195.73191799469373 82795.7145661
Length = 152 rows

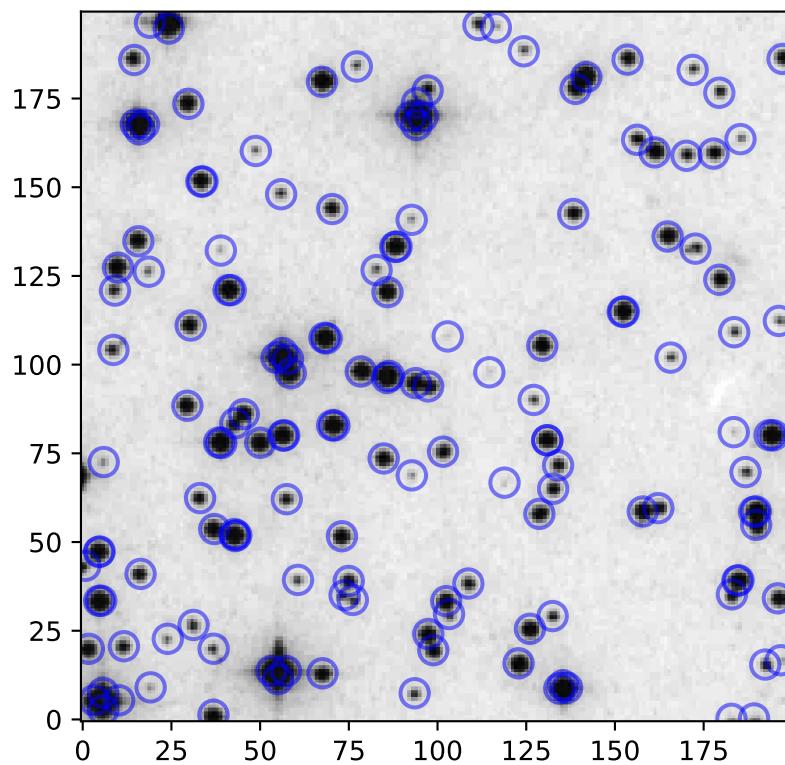
```

The sum of the pixel values within the apertures are given in the column `aperture_sum`. We now plot the image and the defined apertures:

```

>>> import matplotlib.pyplot as plt
>>> plt.imshow(image, cmap='gray_r', origin='lower')
>>> apertures.plot(color='blue', lw=1.5, alpha=0.5)

```



CHAPTER 4

Changelog

0.3.2 (2017-03-31)

General

- Fixed file permissions in the released source distribution.

0.3.1 (2017-03-02)

General

- Dropped numpy 1.7 support. Minimal required version is now numpy 1.8. [#327]
- photutils.datasets
 - The load_* functions that use remote data now retrieve the data from `data.astropy.org` (the astropy data repository). [#472]

Bug Fixes

- photutils.background
 - Fixed issue with `Background2D` with `edge_method='pad'` that occurred when unequal padding needed to be applied to each axis. [#498]
 - Fixed issue with `Background2D` that occurred when zero padding needed to apply along only one axis. [#500]
- photutils.geometry
 - Fixed a bug in `circular_overlap_grid` affecting 32-bit machines that could cause errors circular aperture photometry. [#475]

- `photutils.psf`
 - Fixed a bug in how `FittableImageModel` represents its center. [#460]
 - Fix bug which modified user's input table when doing forced photometry. [#485]

0.3 (2016-11-06)

General

New Features

- `photutils.aperture`
 - Added new `origin` keyword to aperture plot methods. [#395]
 - Added new `id` column to `aperture_photometry` output table. [#446]
 - Added `__len__` method for aperture classes. [#446]
 - Added new `to_mask` method to `PixelAperture` classes. [#453]
 - Added new `ApertureMask` class to generate masks from apertures. [#453]
 - Added new `mask_area()` method to `PixelAperture` classes. [#453]
 - The `aperture_photometry()` function now accepts a list of aperture objects. [#454]
- `photutils.background`
 - Added new `MeanBackground`, `MedianBackground`, `MMMBackground`, `SExtractorBackground`, `BiweightLocationBackground`, `StdBackgroundRMS`, `MADStdBackgroundRMS`, and `BiweightMidvarianceBackgroundRMS` classes. [#370]
 - Added `axis` keyword to new background classes. [#392]
 - Added new `removed_masked`, `meshpix_threshold`, and `edge_method` keywords for the 2D background classes. [#355]
 - Added new `std_blocksum` function. [#355]
 - Added new `SigmaClip` class. [#423]
 - Added new `BkgZoomInterpolator` and `BkgIDWInterpolator` classes. [#437]
- `photutils.datasets`
 - Added `load_irac_psf` function. [#403]
- `photutils.detection`
 - Added new `make_source_mask` convenience function. [#355]
 - Added `filter_data` function. [#398]
 - Added `DAOStarFinder` and `IRAFStarFinder` as oop interfaces for `daofind` and `irafstarfinder`, respectively, which are now deprecated. [#379]
- `photutils.psf`
 - Added `BasicPSFPhotometry`, `IterativelySubtractedPSFPhotometry`, and `DAOPhotPSFPhotometry` classes to perform PSF photometry in crowded fields. [#427]
 - Added `DAOGroup` and `DBSCANGroup` classes for grouping overlapping sources. [#369]

- photutils.psf_match
 - Added `create_matching_kernel` and `resize_psf` functions. Also added `CosineBellWindow`, `HanningWindow`, `SplitCosineBellWindow`, `TopHatWindow`, and `TukeyWindow` classes. [#403]
- photutils.segmentation
 - Created new `photutils.segmentation` subpackage. [#442]
 - Added `copy` and `area` methods and an `areas` property to `SegmentationImage`. [#331]

API changes

- photutils.aperture
 - Removed the `effective_gain` keyword from `aperture_photometry`. Users must now input the total error, which can be calculated using the `calc_total_error` function. [#368]
 - `aperture_photometry` now outputs a `QTable`. [#446]
 - Renamed `source_id` keyword to `indices` in the `aperture plot()` method. [#453]
 - Added `mask` and `unit` keywords to `aperture do_photometry()` methods. [#453]
- photutils.background
 - For the `background` classes, the `filter_shape` keyword was renamed to `filter_size`. The `background_low_res` and `background_rms_low_res` class attributes were renamed to `background_mesh` and `background_rms_mesh`, respectively. [#355, #437]
 - The `Background2D` method and `backfunc` keywords have been removed. In its place one can input callable objects via the `sigma_clip`, `bkg_estimator`, and `bkgrms_estimator` keywords. [#437]
 - The interpolator to be used by the `Background2D` class can be input as a callable object via the new `interpolator` keyword. [#437]
- photutils.centroids
 - Created `photutils.centroids` subpackage, which contains the `centroid_com`, `centroid_1dg`, and `centroid_2dg` functions. These functions now return a two-element numpy ndarray. [#428]
- photutils.detection
 - Changed finding algorithm implementations (`daofind` and `starfind`) from functional to object-oriented style. Deprecated old style. [#379]
- photutils.morphology
 - Created `photutils.morphology` subpackage. [#428]
 - Removed `marginalize_data2d` function. [#428]
 - Moved `cutout_footprint` from `photutils.morphology` to `photutils.utils`. [#428]
 - Added a function to calculate the Gini coefficient (`gini`). [#343]
- photutils.psf
 - Removed the `effective_gain` keyword from `psf_photometry`. Users must now input the total error, which can be calculated using the `calc_total_error` function. [#368]
- photutils.segmentation
 - Removed the `effective_gain` keyword from `SourceProperties` and `source_properties`. Users must now input the total error, which can be calculated using the `calc_total_error` function. [#368]

- photutils.utils
 - Renamed calculate_total_error to calc_total_error. [#368]

Bug Fixes

- photutils.aperture
 - Fixed a bug in aperture_photometry so that single-row output tables do not return a multidimensional column. [#446]
- photutils.centroids
 - Fixed a bug in centroid_1dg and centroid_2dg that occurred when the input data contained invalid (NaN or inf) values. [#428]
- photutils.segmentation
 - Fixed a bug in SourceProperties where error and background units were sometimes dropped. [#441]

0.2.2 (2016-07-06)

General

- Dropped numpy 1.6 support. Minimal required version is now numpy 1.7. [#327]
- Fixed configparser for Python 3.5. [#366, #384]

Bug Fixes

- photutils.detection
 - Fixed an issue to update segmentation image slices after deblending. [#340]
 - Fixed source deblending to pass the pixel connectivity to the watershed algorithm. [#347]
 - SegmentationImage properties are now cached instead of recalculated, which significantly improves performance. [#361]
- photutils.utils
 - Fixed a bug in pixel_to_icrs_coords where the incorrect pixel origin was being passed. [#348]

0.2.1 (2016-01-15)

Bug Fixes

- photutils.background
 - Added more robust version checking of Astropy. [#318]
- photutils.detection
 - Added more robust version checking of Astropy. [#318]
- photutils.segmentation

- Fixed issue where SegmentationImage slices were not being updated. [#317]
- Added more robust version checking of scikit-image. [#318]

0.2 (2015-12-31)

General

- Photutils has the following requirements:
 - Python 2.7 or 3.3 or later
 - Numpy 1.6 or later
 - Astropy v1.0 or later

New Features

- photutils.detection
 - find_peaks now returns an Astropy Table containing the (x, y) positions and peak values. [#240]
 - find_peaks has new mask, error, wcs and subpixel precision options. [#244]
 - detect_sources will now issue a warning if the filter kernel is not normalized to 1. [#298]
 - Added new deblend_sources function, an experimental source deblender. [#314]
- photutils.morphology
 - Added new GaussianConst2D (2D Gaussian plus a constant) model. [#244]
 - Added new marginalize_data2d function. [#244]
 - Added new cutout_footprint function. [#244]
- photutils.segmentation
 - Added new SegmentationImage class. [#306]
 - Added new check_label, keep_labels, and outline_segments methods for modifying SegmentationImage. [#306]
- photutils.utils
 - Added new random_cmap function to generate a colormap comprised of random colors. [#299]
 - Added new ShepardIDWInterpolator class to perform Inverse Distance Weighted (IDW) interpolation. [#307]
 - The interpolate_masked_data function can now interpolate higher-dimensional data. [#310]

API changes

- photutils.segmentation
 - The relabel_sequential, relabel_segments, remove_segments, remove_border_segments, and remove_masked_segments functions are now SegmentationImage methods (with slightly different names). [#306]

- The SegmentProperties class has been renamed to SourceProperties. Likewise the segment_properties function has been renamed to source_properties. [#306]
- The segment_sum and segment_sum_err attributes have been renamed to source_sum and source_sum_err, respectively. [#306]
- The background_atcentroid attribute has been renamed to background_at_centroid. [#306]

Bug Fixes

- photutils.aperture_photometry
 - Fixed an issue where np.nan or np.inf were not properly masked. [#267]
- photutils.geometry
 - overlap_area_triangle_unit_circle handles correctly a corner case in some i386 systems where the area of the aperture was not computed correctly. [#242]
 - rectangular_overlap_grid and elliptical_overlap_grid fixes to normalization of subsampled pixels. [#265]
 - overlap_area_triangle_unit_circle handles correctly the case where a line segment intersects at a triangle vertex. [#277]

Other Changes and Additions

- Updated astropy-helpers to v1.1. [#302]

0.1 (2014-12-22)

Photutils 0.1 was released on December 22, 2014. It requires Astropy version 0.4 or later.

Part II

User Documentation

CHAPTER 5

Background Estimation (`photutils.background`)

Introduction

To accurately measure the photometry and morphological properties of astronomical sources, one requires an accurate estimate of the background, which can be from both the sky and the detector. Similarly, having an accurate estimate of the background noise is important for determining the significance of source detections and for estimating photometric errors.

Unfortunately, accurate background and background noise estimation is a difficult task. Further, because astronomical images can cover a wide variety of scenes, there is not a single background estimation method that will always be applicable. Photutils provides tools for estimating the background and background noise in your data, but they will likely require some tweaking to optimize the background estimate for your data.

Scalar Background and Noise Estimation

Simple Statistics

If the background level and noise are relatively constant across an image, the simplest way to estimate these values is to derive scalar quantities using simple approximations. Of course, when computing the image statistics one must take into account the astronomical sources present in the images, which add a positive tail to the distribution of pixel intensities. For example, one may consider using the image median as the background level and the image standard deviation as the 1-sigma background noise, but the resulting values are obviously biased by the presence of real sources.

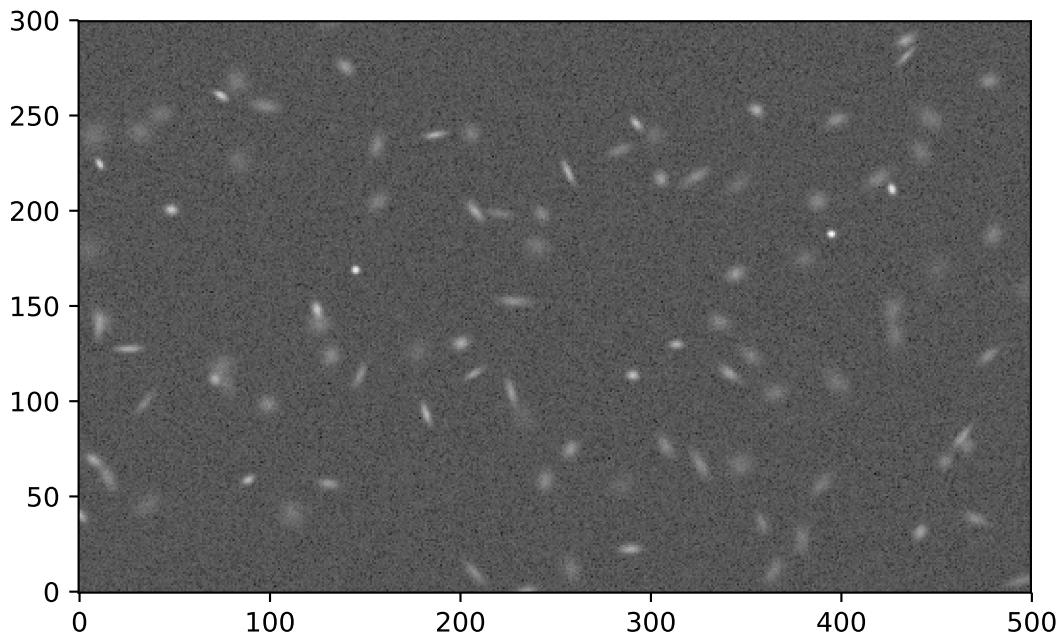
A slightly better method involves using statistics that are robust against the presence of outliers, such as the biweight location for the background level and biweight midvariance or [median absolute deviation \(MAD\)](#) for the background noise estimation. However, for most astronomical scenes these methods will also be biased by the presence of astronomical sources in the image.

As an example, we load a synthetic image comprised of 100 sources with a Gaussian-distributed background whose mean is 5 and standard deviation is 2:

```
>>> from photutils.datasets import make_100gaussians_image
>>> data = make_100gaussians_image()
```

Let's plot the image:

```
>>> import matplotlib.pyplot as plt
>>> from astropy.visualization import SqrtStretch
>>> from astropy.visualization.mpl_normalize import ImageNormalize
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(data, norm=norm, origin='lower', cmap='Greys_r')
```



The image median and biweight location are both larger than the true background level of 5:

```
>>> import numpy as np
>>> from astropy.stats import biweight_location
>>> print(np.median(data))
5.2255295184
>>> print(biweight_location(data))
5.1867597555
```

Similarly, using the biweight midvariance and median absolute deviation to estimate the background noise level give values that are larger than the true value of 2:

```
>>> from astropy.stats import biweight_midvariance, mad_std
>>> print(biweight_midvariance(data))
2.22011175104
```

```
>>> print(mad_std(data))
2.1443728009
```

Sigma Clipping Sources

The most widely used technique to remove the sources from the image statistics is called sigma clipping. Briefly, pixels that are above or below a specified sigma level from the median are discarded and the statistics are recalculated. The procedure is typically repeated over a number of iterations or until convergence is reached. This method provides a better estimate of the background and background noise levels:

```
>>> from astropy.stats import sigma_clipped_stats
>>> mean, median, std = sigma_clipped_stats(data, sigma=3.0, iters=5)
>>> print((mean, median, std))
(5.1991386516217908, 5.1555874333582912, 2.0942752121329691)
```

Masking Sources

An even better procedure is to exclude the sources in the image by masking them. Of course, this technique requires one to identify the sources in the data, which in turn depends on the background and background noise. Therefore, this method for estimating the background and background RMS requires an iterative procedure.

Photutils provides a convenience function, `make_source_mask()`, for creating source masks. It uses sigma-clipped statistics as the first estimate of the background and noise levels for the source detection. Sources are then identified using image segmentation. Finally, the source masks are dilated to mask more extended regions around the detected sources.

Here we use an aggressive 2-sigma detection threshold to maximize the source detections and dilate using a 11x11 box:

```
>>> from photutils import make_source_mask
>>> mask = make_source_mask(data, snr=2, npixels=5, dilate_size=11)
>>> mean, median, std = sigma_clipped_stats(data, sigma=3.0, mask=mask)
>>> print((mean, median, std))
(5.0010134754755695, 5.0005849056043763, 1.970887100626572)
```

Of course, the source detection and masking procedure can be iterated further. Even with one iteration we are within 0.02% of the true background and 1.5% of the true background RMS.

2D Background and Noise Estimation

If the background or the background noise varies across the image, then you will generally want to generate a 2D image of the background and background RMS (or compute these values locally). This can be accomplished by applying the above techniques to subregions of the image. A common procedure is to use sigma-clipped statistics in each mesh of a grid that covers the input data to create a low-resolution background image. The final background or background RMS image can then be generated by interpolating the low-resolution image.

Photutils provides the `Background2D` class to estimate the 2D background and background noise in an astronomical image. `Background2D` requires the size of the box (`box_size`) in which to estimate the background. Selecting the box size requires some care by the user. The box size should generally be larger than the typical size of sources in the image, but small enough to encapsulate any background variations. For best results, the box size should also be chosen so that the data are covered by an integer number of boxes in both dimensions. If that is not the case, the `edge_method`

keyword determines whether to pad or crop the image such that there is an integer multiple of the `box_size` in both dimensions.

The background level in each of the meshes is calculated using the function or callable object (e.g. class instance) input via `bkg_estimator` keyword. Photutils provides a several background classes that can be used:

- `MeanBackground`
- `MedianBackground`
- `ModeEstimatorBackground`
- `MMMBgground`
- `SExtractorBackground`
- `BiweightLocationBackground`

The default is a `SExtractorBackground` instance. For this method, the background in each mesh is calculated as $(2.5 * \text{median}) - (1.5 * \text{mean})$. However, if $(\text{mean} - \text{median}) / \text{std} > 0.3$ then the median is used instead (despite what the `SExtractor` User's Manual says, this is the method it always uses).

Likewise, the background RMS level in each mesh is calculated using the function or callable object input via the `bkgrms_estimator` keyword. Photutils provides the following classes for this purpose:

- `StdBackgroundRMS`
- `MADStdBackgroundRMS`
- `BiweightMidvarianceBackgroundRMS`

For even more flexibility, users may input a custom function or callable object to the `bkg_estimator` and/or `bkgrms_estimator` keywords.

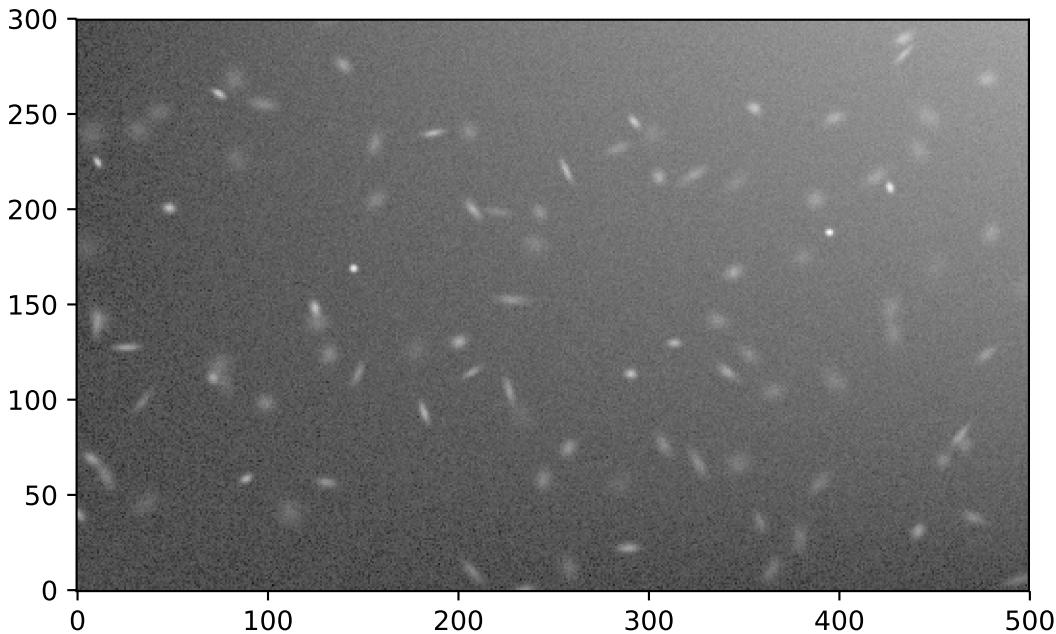
By default the `bkg_estimator` and `bkgrms_estimator` are applied to sigma clipped data. Sigma clipping is defined by inputting a `SigmaClip` object to the `sigma_clip` keyword. The default is to perform sigma clipping with `sigma=3` and `iters=10`. Sigma clipping can be turned off by setting `sigma_clip=None`.

After the background level has been determined in each of the boxes, the low-resolution background image can be median filtered, with a window of size of `filter_size`, to suppress local under- or overestimations (e.g., due to bright galaxies in a particular box). Likewise, the median filter can be applied only to those boxes where the background level is above a specified threshold (`filter_threshold`).

The low-resolution background and background RMS images are resized to the original data size using the function or callable object input via the `interpolator` keyword. Photutils provides two interpolator classes: `BkgZoomInterpolator` (default), which performs spline interpolation, and `BkgIDWInterpolator`, which uses inverse-distance weighted (IDW) interpolation.

For this example, we will create a test image by adding a strong background gradient to the image defined above:

```
>>> ny, nx = data.shape
>>> y, x = np.mgrid[:ny, :nx]
>>> gradient = x * y / 5000.
>>> data2 = data + gradient
>>> plt.imshow(data2, norm=norm, origin='lower', cmap='Greys_r')
```



We start by creating a `Background2D` object using a box size of 50x50 and a 3x3 median filter. We will estimate the background level in each mesh as the sigma-clipped median using an instance of `MedianBackground`.

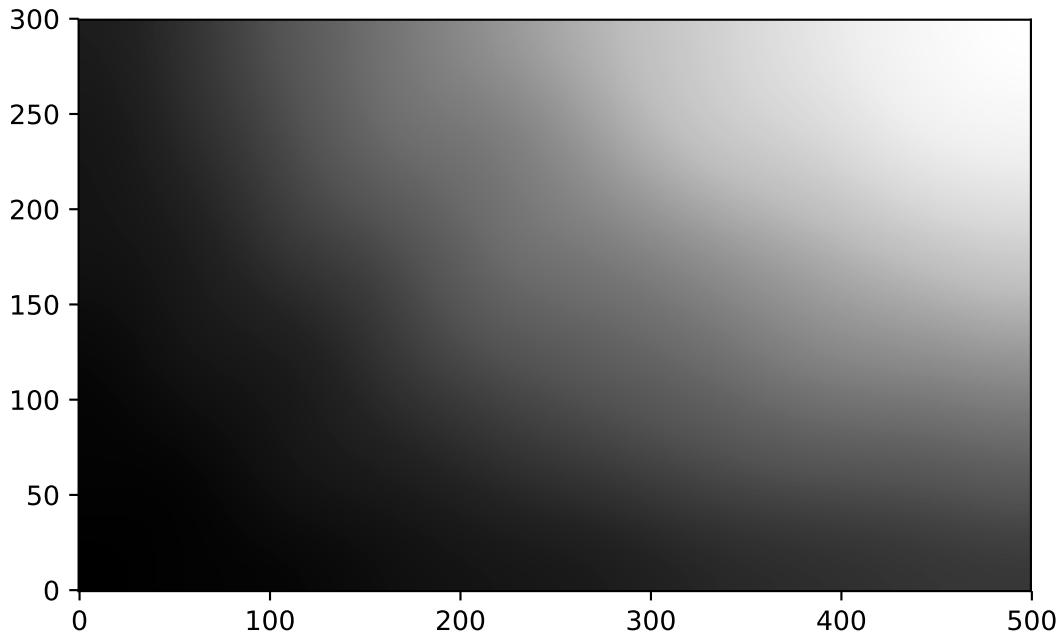
```
>>> from photutils import Background2D, SigmaClip, MedianBackground
>>> sigma_clip = SigmaClip(sigma=3., iters=10)
>>> bkg_estimator = MedianBackground()
>>> bkg = Background2D(data2, (50, 50), filter_size=(3, 3),
...                     sigma_clip=sigma_clip, bkg_estimator=bkg_estimator)
```

The 2D background and background RMS images are retrieved using the `background` and `background_rms` attributes, respectively, on the returned object. The low-resolution versions of these images are stored in the `background_mesh` and `background_rms_mesh` attributes, respectively. The global median value of the low-resolution background and background RMS image can be accessed with the `background_median` and `background_rms_median` attributes, respectively:

```
>>> print(bkg.background_median)
10.8219978626
>>> print(bkg.background_rms_median)
2.29882053968
```

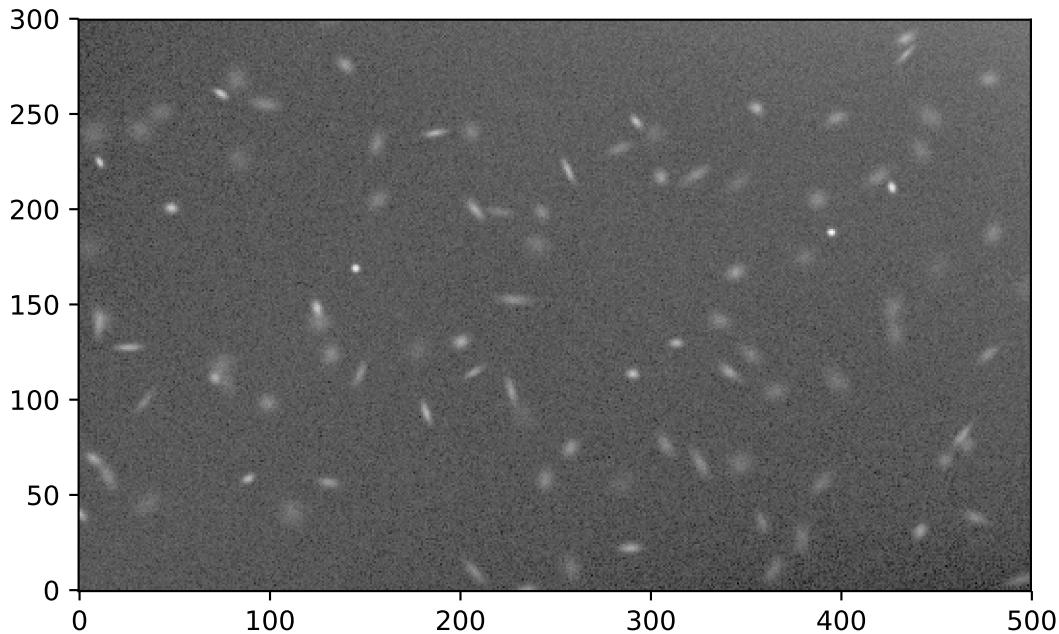
Let's plot the background image:

```
>>> plt.imshow(bkg.background, origin='lower', cmap='Greys_r')
```



and the background-subtracted image:

```
>>> plt.imshow(data2 - bkg.background, norm=norm, origin='lower',
...             cmap='Greys_r')
```



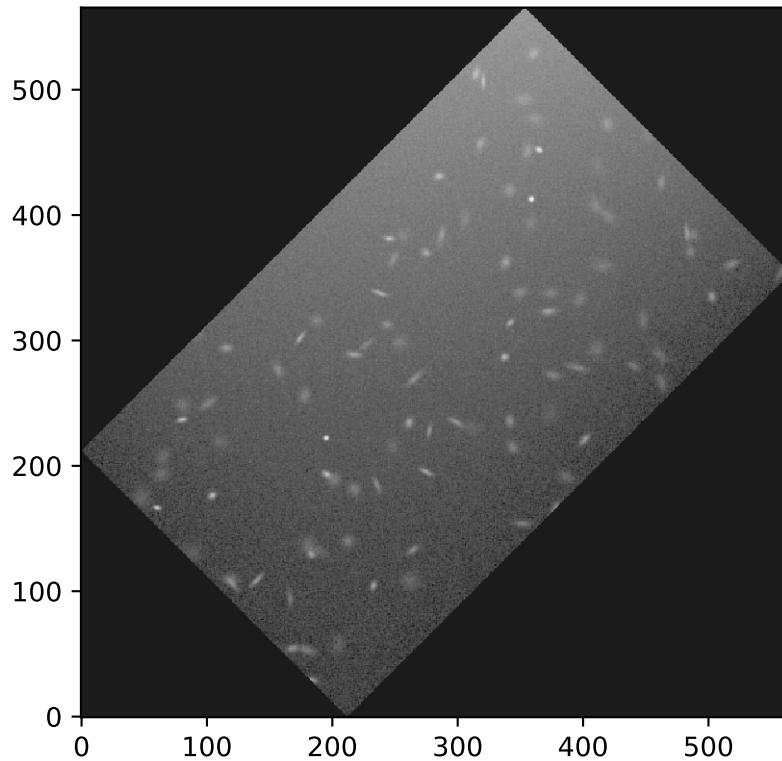
Masking

Masks can also be input into `Background2D`. As described above, this can be employed to mask sources in the image prior to estimating the background levels.

Additionally, input masks are often necessary if your data array includes regions without data coverage (e.g., from a rotated image or an image from a mosaic). Otherwise the data values in the regions without coverage (usually zeros or NaNs) will adversely contribute to the background statistics.

Let's create such an image and plot it (NOTE: this example requires `scipy`):

```
>>> from scipy.ndimage import rotate
>>> data3 = rotate(data2, -45.)
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(data3, origin='lower', cmap='Greys_r', norm=norm)
```

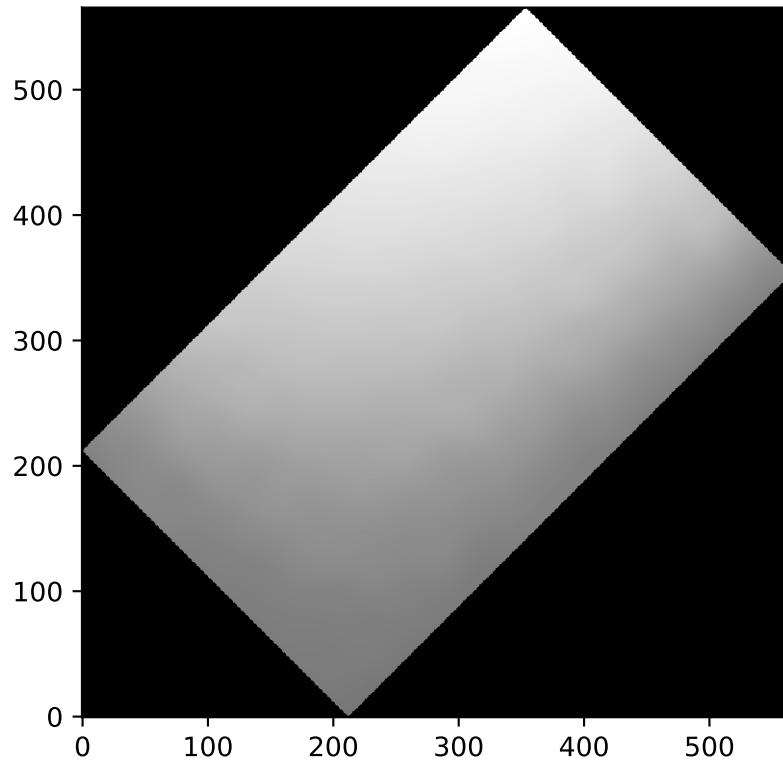


Now we create a coverage mask and input it into `Background2D` to exclude the regions where we have no data. For real data, one can usually create a coverage mask from a weight or noise image. In this example we also use a smaller box size to help capture the strong gradient in the background:

```
>>> mask = (data3 == 0)
>>> bkg3 = Background2D(data3, (25, 25), filter_size=(3, 3), mask=mask)
```

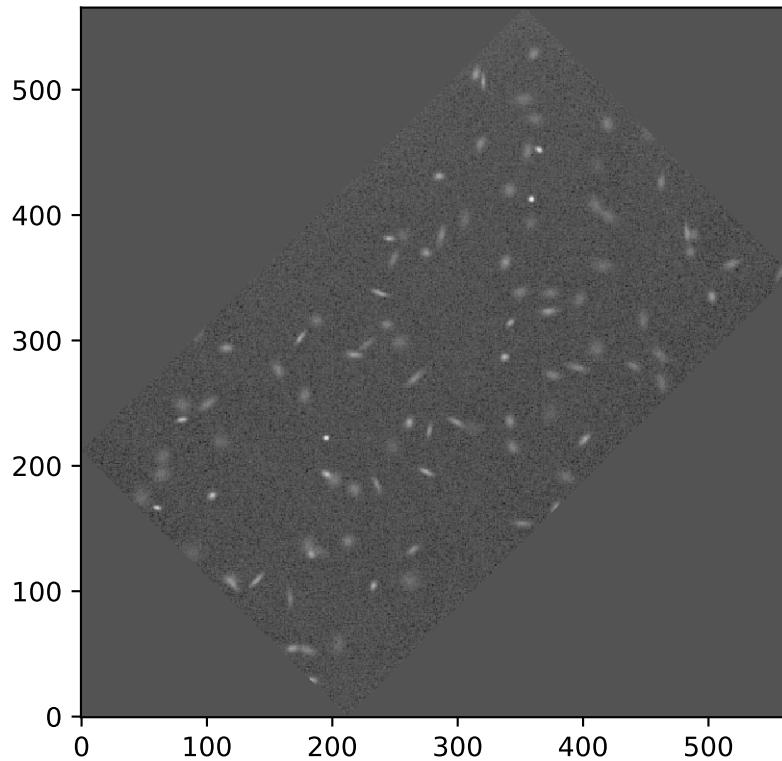
The input masks are never applied to the returned background image because the input mask can represent either a coverage mask or a source mask, or a combination of both. Therefore, we need to manually apply the coverage mask to the returned background image:

```
>>> back3 = bkg3.background * ~mask
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(back3, origin='lower', cmap='Greys_r', norm=norm)
```



Finally, let's subtract the background from the image and plot it:

```
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(data3 - back3, origin='lower', cmap='Greys_r', norm=norm)
```

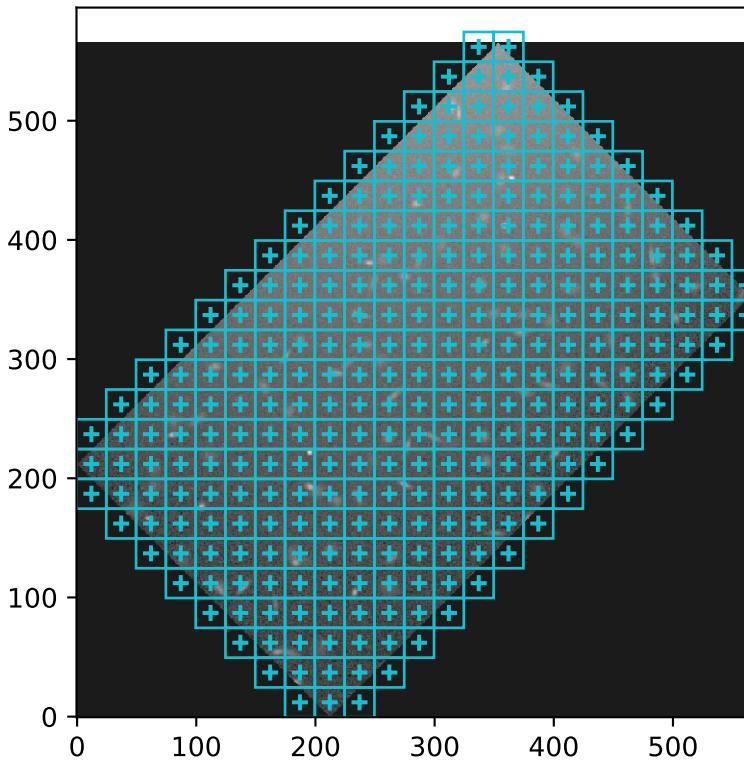


If there is any small residual background still present in the image, the background subtraction can be improved by masking the sources and/or through further iterations.

Plotting Meshes

Finally, the meshes that were used in generating the 2D background can be plotted on the original image using the `plot_meshes()` method:

```
>>> plt.imshow(data3, origin='lower', cmap='Greys_r', norm=norm)
>>> bkg3.plot_meshes(outlines=True, color='#1f77b4')
```



The meshes extended beyond the original image on the top and right because `Background2D`'s default `edge_method` is 'pad'.

Reference/API

This subpackage contains modules and packages for background and background rms estimation.

Classes

<code>Background2D(data, box_size[, mask, ...])</code>	Class to estimate a 2D background and background RMS noise in an image.
<code>BackgroundBase</code>	Base class for classes that estimate scalar background values.
<code>BackgroundRMSBase</code>	Base class for classes that estimate scalar background RMS values.
<code>BiweightLocationBackground</code>	Class to calculate the background in an array using the biweight location.
<code>BiweightMidvarianceBackgroundRMS</code>	Class to calculate the background RMS in an array as the (sigma-clipped) biweight midvariance.

Continued on next page

Table 5.1 – continued from previous page

BkgIDWInterpolator([leafsize, n_neighbors, ...])	This class generates full-sized background and background RMS images from lower-resolution mesh images using inverse-distance weighting (IDW) interpolation (ShepardIDWInterpolator).
BkgZoomInterpolator([order, mode, cval])	This class generates full-sized background and background RMS images from lower-resolution mesh images using the zoom (spline) interpolator.
MADStdBackgroundRMS	Class to calculate the background RMS in an array as using the median absolute deviation (MAD) .
MMBackground	Class to calculate the background in an array using the DAOPHOT MMM algorithm.
MeanBackground	Class to calculate the background in an array as the (sigma-clipped) mean.
MedianBackground	Class to calculate the background in an array as the (sigma-clipped) median.
ModeEstimatorBackground	Class to calculate the background in an array using a mode estimator of the form <code>(median_factor * median) - (mean_factor * mean)</code> .
SExtractorBackground	Class to calculate the background in an array using the SExtractor algorithm.
SigmaClip([sigma, sigma_lower, sigma_upper, ...])	Class to perform sigma clipping.
StdBackgroundRMS	Class to calculate the background RMS in an array as the (sigma-clipped) standard deviation.

Background2D

```
class photutils.background.Background2D(data,          box_size,          mask=None,          ex-
                                         exclude_mesh_method='threshold',      ex-
                                         exclude_mesh_percentile=10.0,        filter_size=(3,      3),
                                         filter_threshold=None,            edge_method='pad',
                                         sigma_clip=<photutils.background.core.SigmaClip object>, 
                                         bkg_estimator=<photutils.background.core.SExtractorBackground
                                         object>, bkgrms_estimator=<photutils.background.core.StdBackgroundRMS
                                         object>, interpolator=<photutils.background.background_2d.BkgZoomInterpolator
                                         object>)
```

Bases: `object`

Class to estimate a 2D background and background RMS noise in an image.

The background is estimated using sigma-clipped statistics in each mesh of a grid that covers the input data to create a low-resolution, and possibly irregularly-gridded, background map.

The final background map is calculated by interpolating the low-resolution background map.

Parameters

data : array_like

The 2D array from which to estimate the background and/or background RMS map.

box_size : int or array_like (int)

The box size along each axis. If `box_size` is a scalar then a square box of size `box_size` will be used. If `box_size` has two elements, they should be in (ny, nx) order. For best results, the box shape should be chosen such that the data are covered by an integer

number of boxes in both dimensions. When this is not the case, see the `edge_method` keyword for more options.

mask : array_like (bool), optional

A boolean mask, with the same shape as `data`, where a `True` value indicates the corresponding element of data is masked. Masked data are excluded from calculations.

exclude_mesh_method : {‘threshold’, ‘any’, ‘all’}, optional

The method used to determine whether to exclude a particular mesh based on the number of masked pixels it contains in the input (e.g. `source`) mask or padding mask (if `edge_method='pad'`):

- ‘threshold’: exclude meshes that contain greater than `exclude_mesh_percentile` percent masked pixels. This is the default.
- ‘any’: exclude meshes that contain any masked pixels.
- ‘all’: exclude meshes that are completely masked.

exclude_mesh_percentile : float in the range of [0, 100], optional

The percentile of masked pixels in a mesh used as a threshold for determining if the mesh is excluded. If `exclude_mesh_method='threshold'`, then meshes that contain greater than `exclude_mesh_percentile` percent masked pixels are excluded. This parameter is used only if `exclude_mesh_method='threshold'`. The default is 10. For best results, `exclude_mesh_percentile` should be kept as low as possible (i.e, as long as there are sufficient pixels for reasonable statistical estimates).

filter_size : int or array_like (int), optional

The window size of the 2D median filter to apply to the low-resolution background map. If `filter_size` is a scalar then a square box of size `filter_size` will be used. If `filter_size` has two elements, they should be in (ny, nx) order. A filter size of 1 (or (1, 1)) means no filtering.

filter_threshold : int, optional

The threshold value for used for selective median filtering of the low-resolution 2D background map. The median filter will be applied to only the background meshes with values larger than `filter_threshold`. Set to `None` to filter all meshes (default).

edge_method : {‘pad’, ‘crop’}, optional

The method used to determine how to handle the case where the image size is not an integer multiple of the `box_size` in either dimension. Both options will resize the image to give an exact multiple of `box_size` in both dimensions.

- ‘pad’: pad the image along the top and/or right edges. This is the default and recommended method.
- ‘crop’: crop the image along the top and/or right edges.

sigma_clip : `SigmaClip` instance, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3` and `iters=10`.

bkg_estimator : callable, optional

A callable object (a function or e.g., an instance of any `BackgroundBase` subclass) used to estimate the background in each of the meshes. The callable object must take in a 2D

`ndarray` or `MaskedArray` and have an `axis` keyword (internally, the background will be calculated along `axis=1`). The callable object must return a 1D `MaskedArray`. If `bkg_estimator` includes sigma clipping, it will be ignored (use the `sigma_clip` keyword to define sigma clipping). The default is an instance of `SExtractorBackground`.

bkgrms_estimator : callable, optional

A callable object (a function or e.g., an instance of any `BackgroundRMSBase` subclass) used to estimate the background RMS in each of the meshes. The callable object must take in a 2D `ndarray` or `MaskedArray` and have an `axis` keyword (internally, the background RMS will be calculated along `axis=1`). The callable object must return a 1D `MaskedArray`. If `bkgrms_estimator` includes sigma clipping, it will be ignored (use the `sigma_clip` keyword to define sigma clipping). The default is an instance of `StdBackgroundRMS`.

interpolator : callable, optional

A callable object (a function or object) used to interpolate the low-resolution background or background RMS mesh to the full-size background or background RMS maps. The default is an instance of `BkgZoomInterpolator`.

Notes

If there is only one background mesh element (i.e., `box_size` is the same size as the data), then the background map will simply be a constant image.

Attributes Summary

<code>background</code>	A 2D <code>ndarray</code> containing the background image.
<code>background_median</code>	The median value of the 2D low-resolution background map.
<code>background_mesh_ma</code>	The background 2D (masked) array mesh prior to any interpolation.
<code>background_rms</code>	A 2D <code>ndarray</code> containing the background RMS image.
<code>background_rms_median</code>	The median value of the low-resolution background RMS map.
<code>background_rms_mesh_ma</code>	The background RMS 2D (masked) array mesh prior to any interpolation.
<code>mesh_nmasks</code>	A 2D (masked) array of the number of masked pixels in each mesh.

Methods Summary

<code>plot_meshes([ax, marker, color, outlines])</code>	Plot the low-resolution mesh boxes on a matplotlib Axes instance.
---	---

Attributes Documentation

background

A 2D `ndarray` containing the background image.

background_median

The median value of the 2D low-resolution background map.

This is equivalent to the value SExtractor prints to stdout (i.e., “(M+D) Background: <value>”).

background_mesh_ma

The background 2D (masked) array mesh prior to any interpolation.

background_rms

A 2D `ndarray` containing the background RMS image.

background_rms_median

The median value of the low-resolution background RMS map.

This is equivalent to the value SExtractor prints to stdout (i.e., “(M+D) RMS: <value>”).

background_rms_mesh_ma

The background RMS 2D (masked) array mesh prior to any interpolation.

mesh_nmasked

A 2D (masked) array of the number of masked pixels in each mesh. Only meshes included in the background estimation are included. Excluded meshes will be masked in the image.

Methods Documentation

plot_meshes(ax=None, marker=u'+', color=u'blue', outlines=False, **kwargs)

Plot the low-resolution mesh boxes on a matplotlib Axes instance.

Parameters

ax : `matplotlib.axes.Axes` instance, optional

If `None`, then the current Axes instance is used.

marker : str, optional

The marker to use to mark the center of the boxes. Default is ‘+’.

color : str, optional

The color for the markers and the box outlines. Default is ‘blue’.

outlines : bool, optional

Whether or not to plot the box outlines in addition to the box centers.

kwargs

Any keyword arguments accepted by `matplotlib.patches.Patch`. Used only if `outlines` is True.

BackgroundBase

class photutils.background.BackgroundBase

Bases: `object`

Base class for classes that estimate scalar background values.

Parameters

sigma_clip : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Methods Summary

`__call__(...)` <=> `x(...)`

`calc_background(data[, axis])`

Calculate the background value.

Methods Documentation

`__call__(...)` <=> `x(...)`

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BackgroundRMSBase

`class photutils.background.BackgroundRMSBase`

Bases: `object`

Base class for classes that estimate scalar background RMS values.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Methods Summary

`__call__(...)` <=> `x(...)`

`calc_background_rms(data[, axis])`

Calculate the background RMS value.

Methods Documentation

`__call__(...) <==> x(...)`

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BiweightLocationBackground

`class photutils.background.BiweightLocationBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array using the biweight location.

Parameters

`c` : float, optional

Tuning constant for the biweight estimator. Default value is 6.0.

`M` : float, optional

Initial guess for the biweight location. Default value is `None`.

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, BiweightLocationBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = BiweightLocationBackground(sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BiweightMidvarianceBackgroundRMS

`class photutils.background.BiweightMidvarianceBackgroundRMS`

Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as the (sigma-clipped) biweight midvariance.

Parameters

`c` : float, optional

Tuning constant for the biweight estimator. Default value is 9.0.

`M` : float, optional

Initial guess for the biweight location. Default value is `None`.

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, BiweightMidvarianceBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = BiweightMidvarianceBackgroundRMS(sigma_clip=sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
30.094338485893392
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
30.094338485893392
```

Methods Summary

<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.
--	-------------------------------------

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BkgIDWInterpolator

```
class photutils.background.BkgIDWInterpolator(leafsize=10, n_neighbors=10, power=1.0, reg=0.0)
Bases: object
```

This class generates full-sized background and background RMS images from lower-resolution mesh images using inverse-distance weighting (IDW) interpolation (`ShepardIDWInterpolator`).

This class must be used in concert with the `Background2D` class.

Parameters

leafsize : float, optional

The number of points at which the k-d tree algorithm switches over to brute-force. leafsize must be positive. See `scipy.spatial.cKDTree` for further information.

n_neighbors : int, optional

The maximum number of nearest neighbors to use during the interpolation.

power : float, optional

The power of the inverse distance used for the interpolation weights.

reg : float, optional

The regularization parameter. It may be used to control the smoothness of the interpolator.

Methods Summary

<code>__call__(mesh, bkg2d_obj)</code>	Resize the 2D mesh array.
--	---------------------------

Methods Documentation

`__call__(mesh, bkg2d_obj)`

Resize the 2D mesh array.

Parameters

mesh : 2D `ndarray`

The low-resolution 2D mesh array.

bkg2d_obj : `Background2D` object

The `Background2D` object that prepared the `mesh` array.

Returns

result : 2D `ndarray`

The resized background or background RMS image.

BkgZoomInterpolator

```
class photutils.background.BkgZoomInterpolator(order=3, mode=u'reflect', cval=0.0)
Bases: object
```

This class generates full-sized background and background RMS images from lower-resolution mesh images using the `zoom` (spline) interpolator.

This class must be used in concert with the `Background2D` class.

Parameters

order : int, optional

The order of the spline interpolation used to resize the low-resolution background and background RMS mesh images. The value must be an integer in the range 0-5. The default is 3 (bicubic interpolation).

mode : {‘reflect’, ‘constant’, ‘nearest’, ‘wrap’}, optional

Points outside the boundaries of the input are filled according to the given mode. Default is ‘reflect’.

eval : float, optional

The value used for points outside the boundaries of the input if mode='constant'. Default is 0.0.

Methods Summary

<code>__call__(mesh, bkg2d_obj)</code>	Resize the 2D mesh array.
--	---------------------------

Methods Documentation

__call__(mesh, bkg2d_obj)

Resize the 2D mesh array.

Parameters

mesh : 2D `ndarray`

The low-resolution 2D mesh array.

bkg2d_obj : `Background2D` object

The `Background2D` object that prepared the mesh array.

Returns

result : 2D `ndarray`

The resized background or background RMS image.

MADStdBackgroundRMS

class photutils.background.MADStdBackgroundRMS

Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as using the median absolute deviation (MAD).

The standard deviation estimator is given by:

$$\sigma \approx \frac{\text{MAD}}{\Phi^{-1}(3/4)} \approx 1.4826 \text{ MAD}$$

where $\Phi^{-1}(P)$ is the normal inverse cumulative distribution function evaluated at probability $P = 3/4$.

Parameters

sigma_clip : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, MADStdBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = MADStdBackgroundRMS(sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
37.065055462640053
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
37.065055462640053
```

Methods Summary

<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.
--	-------------------------------------

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

MMMBBackground

`class photutils.background.MMMBackground`

Bases: `photutils.ModeEstimatorBackground`

Class to calculate the background in an array using the DAOPHOT MMM algorithm.

The background is calculated using a mode estimator of the form $(3 * \text{median}) - (2 * \text{mean})$.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MMMBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MMMBackground(sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

MeanBackground

`class photutils.background.MeanBackground`
Bases: `photutils.BackgroundBase`

Class to calculate the background in an array as the (sigma-clipped) mean.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MeanBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MeanBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

MedianBackground

`class photutils.background.MedianBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array as the (sigma-clipped) median.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MedianBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MedianBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

ModeEstimatorBackground

`class photutils.background.ModeEstimatorBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array using a mode estimator of the form $(\text{median_factor} * \text{median}) - (\text{mean_factor} * \text{mean})$.

Parameters

`median_factor` : float, optional

The multiplicative factor for the data median. Defaults to 3.

`mean_factor` : float, optional

The multiplicative factor for the data mean. Defaults to 2.

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, ModeEstimatorBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = ModeEstimatorBackground(median_factor=3., mean_factor=2.,
...                                 sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

SExtractorBackground

`class photutils.background.SExtractorBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array using the SExtractor algorithm.

The background is calculated using a mode estimator of the form $(2.5 * \text{median}) - (1.5 * \text{mean})$.

If $(\text{mean} - \text{median}) / \text{std} > 0.3$ then the median is used instead. Despite what the [SExtractor User's Manual](#) says, this is the method it *always* uses.

Parameters

`sigma_clip` : [SigmaClip](#) object, optional

A [SigmaClip](#) object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, SExtractorBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = SExtractorBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

`calc_background(data[, axis])`

Calculate the background value.

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or [MaskedArray](#)

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or [MaskedArray](#)

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a [MaskedArray](#) will be returned.

SigmaClip

```
class photutils.background.SigmaClip(sigma=3.0, sigma_lower=None, sigma_upper=None, iters=5,
                                     cenfunc=<function median>, stdfunc=<function std>)
```

Bases: `object`

Class to perform sigma clipping.

Parameters

`sigma` : float, optional

The number of standard deviations to use for both the lower and upper clipping limit.
These limits are overridden by `sigma_lower` and `sigma_upper`, if input. Defaults to 3.

`sigma_lower` : float or `None`, optional

The number of standard deviations to use as the lower bound for the clipping limit. If
`None` then the value of `sigma` is used. Defaults to `None`.

`sigma_upper` : float or `None`, optional

The number of standard deviations to use as the upper bound for the clipping limit. If
`None` then the value of `sigma` is used. Defaults to `None`.

`iters` : int or `None`, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence
is achieved (i.e., continue until the last iteration clips nothing). Defaults to 5.

`cenfunc` : callable, optional

The function used to compute the center for the clipping. Must be a callable that takes
in a masked array and outputs the central value. Defaults to the median (`numpy.ma.median`).

`stdfunc` : callable, optional

The function used to compute the standard deviation about the center. Must be a callable
that takes in a masked array and outputs a width estimator. Masked (rejected) pixels are
those where:

```
deviation < (-sigma_lower * stdfunc(deviation))
deviation > (sigma_upper * stdfunc(deviation))
```

where:

```
deviation = data - cenfunc(data [,axis=int])
```

Defaults to the standard deviation (`numpy.std`).

Methods Summary

`__call__(data[, axis, copy])`

Perform sigma clipping on the provided data.

Methods Documentation

`__call__(data, axis=None, copy=True)`

Perform sigma clipping on the provided data.

Parameters**data** : array-like

The data to be sigma clipped.

axis : int or `None`, optionalIf not `None`, clip along the given axis. For this case, `axis` will be passed on to `cenfunc` and `stdfunc`, which are expected to return an array with the axis dimension removed (like the numpy functions). If `None`, clip over all axes. Defaults to `None`.**copy** : bool, optionalIf `True`, the data array will be copied. If `False`, the returned masked array data will contain the same array as `data`. Defaults to `True`.**Returns****filtered_data** : `numpy.ma.MaskedArray`A masked array with the same shape as `data` input, where the points rejected by the algorithm have been masked.**StdBackgroundRMS****class photutils.background.StdBackgroundRMS**Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as the (sigma-clipped) standard deviation.

Parameters**sigma_clip** : `SigmaClip` object, optionalA `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`**Examples**

```
>>> from photutils import SigmaClip, StdBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = StdBackgroundRMS(sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
28.866070047722118
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
28.866070047722118
```

Methods Summary

<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.
--	-------------------------------------

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

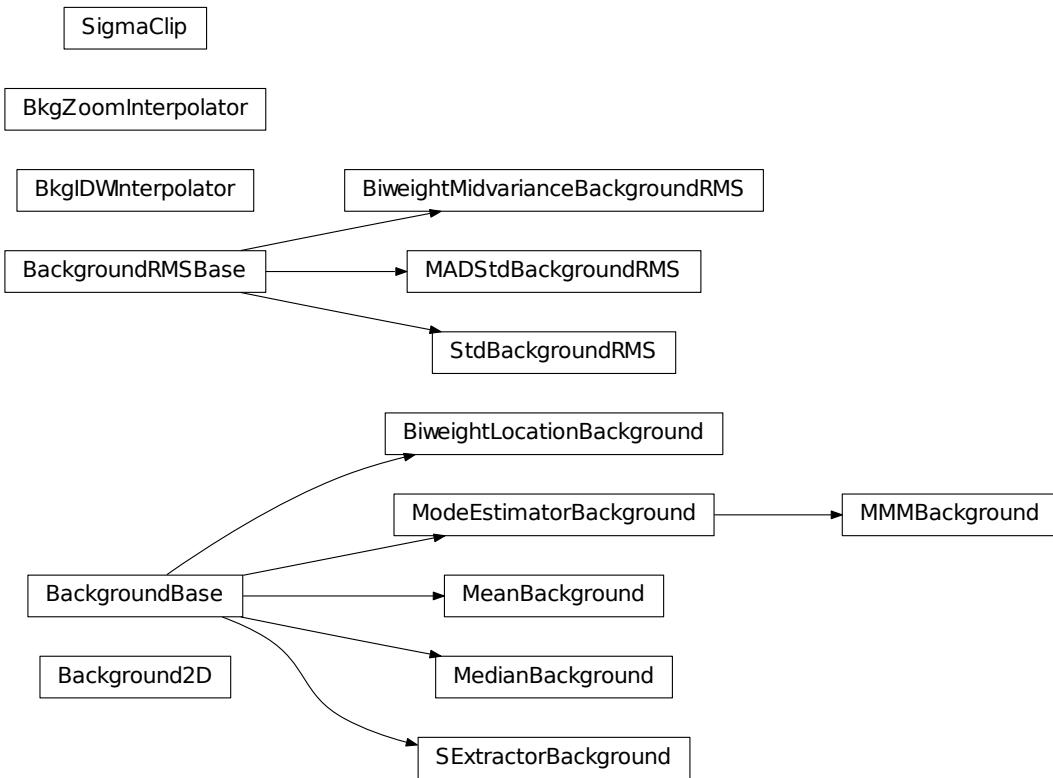
The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

Class Inheritance Diagram



CHAPTER 6

Source Detection (`photutils.detection`)

Introduction

One generally needs to identify astronomical sources in their data before they can perform photometry or morphological measurements. Photutils provides two functions designed specifically to detect point-like (stellar) sources in an astronomical image. Photutils also provides a function to identify local peaks in an image that are above a specified threshold value.

For general-use source detection and extraction of both point-like and extended sources, please see *Image Segmentation* (`photutils.segmentation`).

Detecting Stars

Photutils includes two widely-used tools that are used to detect stars in an image, `DAOFIND` and IRAF's `starfind`.

`DAOStarFinder` is a class that provides an implementation of the `DAOFIND` algorithm (Stetson 1987, PASP 99, 191). It searches images for local density maxima that have a peak amplitude greater than a specified threshold (the threshold is applied to a convolved image) and have a size and shape similar to a defined 2D Gaussian kernel. `DAOStarFinder` also provides an estimate of the objects' roundness and sharpness, whose lower and upper bounds can be specified.

`IRAFStarFinder` is a class that implements IRAF's `starfind` algorithm. It is fundamentally similar to `DAOStarFinder`, but `DAOStarFinder` can use an elliptical Gaussian kernel. One other difference in `IRAFStarFinder` is that it calculates the objects' centroid, roundness, and sharpness using image moments.

As an example, let's load an image from the bundled datasets and select a subset of the image. We will estimate the background and background noise using sigma-clipped statistics:

```
>>> from astropy.stats import sigma_clipped_stats
>>> from photutils import datasets
>>> hdu = datasets.load_star_image()
Downloading ...
>>> data = hdu.data[0:400, 0:400]
>>> mean, median, std = sigma_clipped_stats(data, sigma=3.0, iters=5)
```

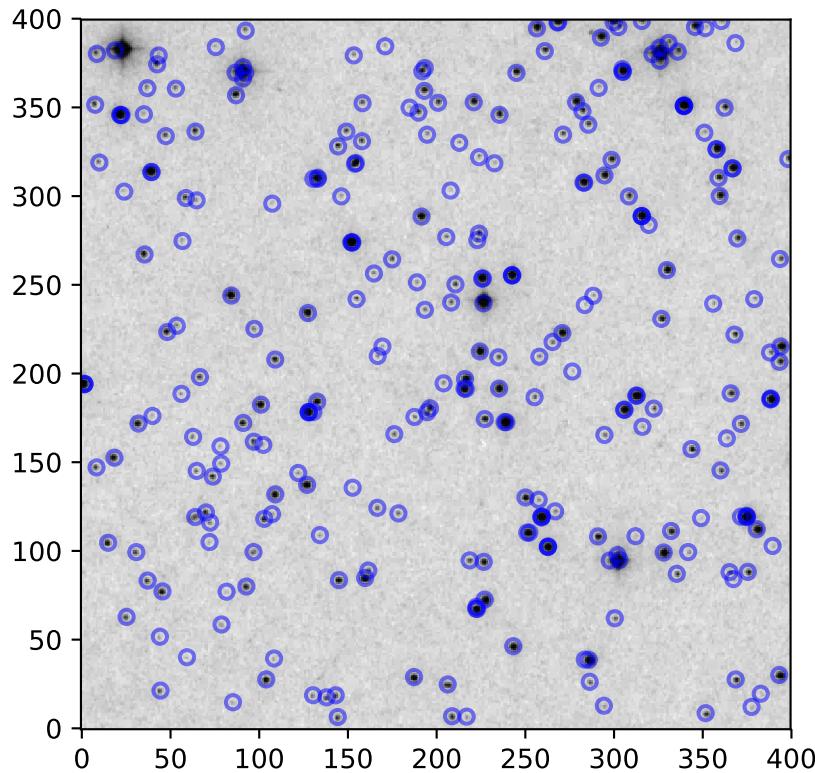
```
>>> print((mean, median, std))
(3667.7792400186008, 3649.0, 204.27923665845705)
```

Now we will subtract the background and use an instance of `DAOStarFinder` to find the stars in the image that have FWHMs of around 3 pixels and have peaks approximately 5-sigma above the background. Running this class on the data yields an astropy `Table` containing the results of the star finder:

```
>>> from photutils import DAOStarFinder
>>> daofind = DAOStarFinder(fwhm=3.0, threshold=5.*std)
>>> sources = daofind(data - median)
>>> print(sources)
   id    xcentroid    ycentroid ...   peak      flux       mag
---  -----  -----  ...  -----  -----  ...
  1  144.247567164  6.37979042704 ...  6903.0  5.70143033038 -1.88995955438
  2  208.669068628  6.82058053777 ...  7896.0  6.72306730455 -2.06891864748
  3  216.926136655  6.5775933198 ...  2195.0  1.66737467591 -0.555083002864
  4  351.625190383  8.5459013233 ...  6977.0  5.90092548147 -1.92730032571
  5  377.519909958  12.0655009987 ...  1260.0  1.11856203781 -0.121650189969
...
281 268.049236979  397.925371446 ...  9299.0  6.22022587541 -1.98451538884
282 268.475068392  398.020998272 ...  8754.0  6.05079160593 -1.95453048936
283 299.80943822  398.027911813 ...  8890.0  6.11853416663 -1.96661847383
284 315.689448343  398.70251891 ...  6485.0  5.55471107793 -1.86165368631
285 360.437243037  398.698539555 ...  8079.0  5.26549321379 -1.80359764345
Length = 285 rows
```

Let's plot the image and mark the location of detected sources:

```
>>> import matplotlib.pyplot as plt
>>> from astropy.visualization import SqrtStretch
>>> from astropy.visualization.mpl_normalize import ImageNormalize
>>> from photutils import CircularAperture
>>> positions = (sources['xcentroid'], sources['ycentroid'])
>>> apertures = CircularAperture(positions, r=4.)
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(data, cmap='Greys', origin='lower', norm=norm)
>>> apertures.plot(color='blue', lw=1.5, alpha=0.5)
```



Local Peak Detection

Photutils also includes a `find_peaks()` function to find local peaks in an image that are above a specified threshold value. Peaks are the local maxima above a specified threshold that are separated by a specified minimum number of pixels. By default, the returned pixel coordinates are always integer-valued (i.e., no centroiding is performed, only the peak pixel is identified). However, `find_peaks()` may be used to compute centroid coordinates with subpixel precision whenever the optional argument `subpixel` is set to `True`.

`find_peaks()` supports a number of additional options, including searching for peaks only within a segmentation image or a specified footprint. Please see the `find_peaks()` documentation for more options.

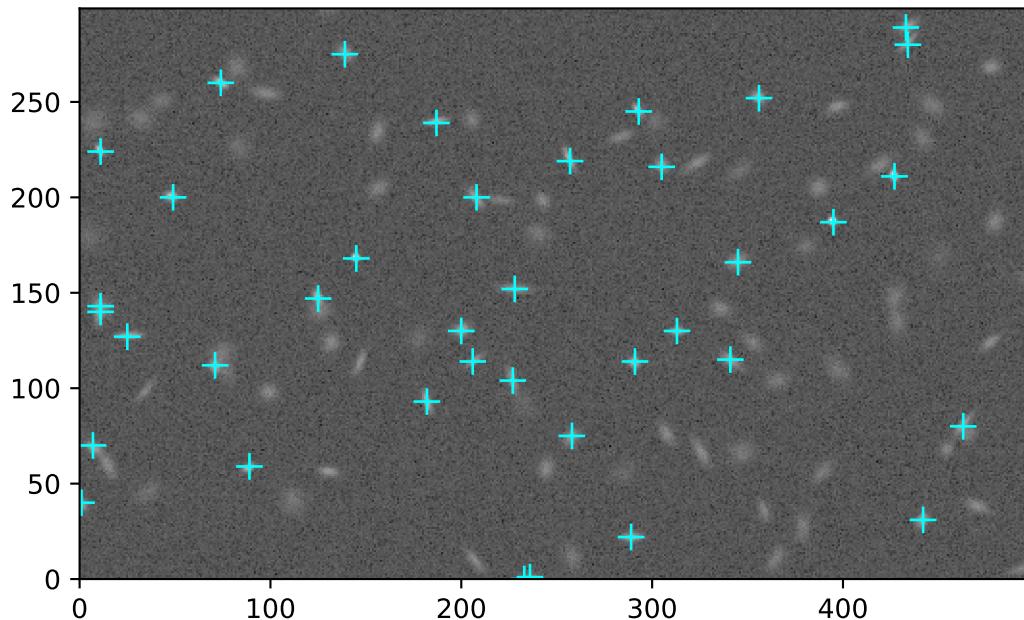
As simple example, let's find the local peaks in an image that are 10 sigma above the background and a separated by at least 2 pixels:

```
>>> from astropy.stats import sigma_clipped_stats
>>> from photutils.datasets import make_100gaussians_image
>>> from photutils import find_peaks
>>> data = make_100gaussians_image()
>>> mean, median, std = sigma_clipped_stats(data, sigma=3.0)
>>> threshold = median + (10.0 * std)
>>> tbl = find_peaks(data, threshold, box_size=5)
>>> print(tbl[:10])    # print only the first 10 peaks
x_peak y_peak  peak_value
-----
```

```
233      0 27.4778521972
236      1 27.339519624
289     22 35.8532759965
442     31 30.2399941373
1       40 35.5482863002
89      59 41.2190469279
7       70 33.2880647048
258     75 26.5624808518
463     80 28.7588206692
182    93 38.0885687202
```

And let's plot the location of the detected peaks in the image:

```
>>> import matplotlib.pyplot as plt
>>> from astropy.visualization import SqrtStretch
>>> from astropy.visualization.mpl_normalize import ImageNormalize
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> plt.imshow(data, cmap='Greys_r', origin='lower', norm=norm)
>>> plt.plot(tbl['x_peak'], tbl['y_peak'], ls='none', color='cyan',
...           marker='+', ms=10, lw=1.5)
>>> plt.xlim(0, data.shape[1]-1)
>>> plt.ylim(0, data.shape[0]-1)
```



Reference/API

This subpackage contains modules and packages for identifying sources in an astronomical image.

Functions

<code>daofind(*args, **kwargs)</code>	Deprecated since version 0.3.
<code>detect_threshold(data, snr[, background, ...])</code>	Calculate a pixel-wise threshold image that can be used to detect sources.
<code>find_peaks(data, threshold[, box_size, ...])</code>	Find local peaks in an image that are above a specified threshold value.
<code>irafstarfind(*args, **kwargs)</code>	Deprecated since version 0.3.

daofind

`photutils.detection.daofind(*args, **kwargs)`

Deprecated since version 0.3: The daofind function is deprecated and may be removed in a future version. Use DAOStarFinder instead.

detect_threshold

`photutils.detection.detect_threshold(data, snr, background=None, error=None, mask=None, mask_value=None, sigclip_sigma=3.0, sigclip_iters=None)`

Calculate a pixel-wise threshold image that can be used to detect sources.

Parameters

data : array_like

The 2D array of the image.

snr : float

The signal-to-noise ratio per pixel above the background for which to consider a pixel as possibly being part of a source.

background : float or array_like, optional

The background value(s) of the input data. background may either be a scalar value or a 2D image with the same shape as the input data. If the input data has been background-subtracted, then set background to 0.0. If `None`, then a scalar background value will be estimated using sigma-clipped statistics.

error : float or array_like, optional

The Gaussian 1-sigma standard deviation of the background noise in data. error should include all sources of “background” error, but *exclude* the Poisson error of the sources. If error is a 2D image, then it should represent the 1-sigma background error in each pixel of data. If `None`, then a scalar background rms value will be estimated using sigma-clipped statistics.

mask : array_like, bool, optional

A boolean mask with the same shape as data, where a `True` value indicates the corresponding element of data is masked. Masked pixels are ignored when computing the image background statistics.

mask_value : float, optional

An image data value (e.g., `0.0`) that is ignored when computing the image background statistics. `mask_value` will be ignored if `mask` is input.

sigclip_sigma : float, optional

The number of standard deviations to use as the clipping limit when calculating the image background statistics.

sigclip_iters : int, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence is achieved (i.e., continue until the last iteration clips nothing) when calculating the image background statistics.

Returns

threshold : 2D `ndarray`

A 2D image with the same shape as data containing the pixel-wise threshold values.

See also:

`photutils.segmentation.detect_sources()`

Notes

The `mask`, `mask_value`, `sigclip_sigma`, and `sigclip_iters` inputs are used only if it is necessary to estimate background or error using sigma-clipped background statistics. If `background` and `error` are both input, then `mask`, `mask_value`, `sigclip_sigma`, and `sigclip_iters` are ignored.

find_peaks

```
photutils.detection.find_peaks(data, threshold, box_size=3, footprint=None, mask=None, border_width=None, npeaks=inf, subpixel=False, error=None, wcs=None)
```

Find local peaks in an image that are above a specified threshold value.

Peaks are the maxima above the threshold within a local region. The regions are defined by either the `box_size` or `footprint` parameters. `box_size` defines the local region around each pixel as a square box. `footprint` is a boolean array where `True` values specify the region shape.

If multiple pixels within a local region have identical intensities, then the coordinates of all such pixels are returned. Otherwise, there will be only one peak pixel per local region. Thus, the defined region effectively imposes a minimum separation between peaks (unless there are identical peaks within the region).

When using subpixel precision (`subpixel=True`), then a cutout of the specified `box_size` or `footprint` will be taken centered on each peak and fit with a 2D Gaussian (plus a constant). In this case, the fitted local centroid and peak value (the Gaussian amplitude plus the background constant) will also be returned in the output table.

Parameters

data : array_like

The 2D array of the image.

threshold : float or array-like

The data value or pixel-wise data values to be used for the detection threshold. A 2D threshold must have the same shape as data. See [detect_threshold](#) for one way to create a threshold image.

box_size : scalar or tuple, optional

The size of the local region to search for peaks at every point in data. If `box_size` is a scalar, then the region shape will be `(box_size, box_size)`. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

footprint : `ndarray` of bools, optional

A boolean array where `True` values describe the local footprint region within which to search for peaks at every point in data. `box_size=(n, m)` is equivalent to `footprint=np.ones((n, m))`. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

mask : array_like, bool, optional

A boolean mask with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

border_width : bool, optional

The width in pixels to exclude around the border of the data.

npeaks : int, optional

The maximum number of peaks to return. When the number of detected peaks exceeds `npeaks`, the peaks with the highest peak intensities will be returned.

subpixel : bool, optional

If `True`, then a cutout of the specified `box_size` or `footprint` will be taken centered on each peak and fit with a 2D Gaussian (plus a constant). In this case, the fitted local centroid and peak value (the Gaussian amplitude plus the background constant) will also be returned in the output table.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data. `error` is used only to weight the 2D Gaussian fit performed when `subpixel=True`.

wcs : `WCS`

The WCS transformation to use to convert from pixel coordinates to ICRS world coordinates. If `None`, then the world coordinates will not be returned in the output `Table`.

Returns

output : `Table`

A table containing the x and y pixel location of the peaks and their values. If `subpixel=True`, then the table will also contain the local centroid and fitted peak value.

irafstarfind

`photutils.detection.irafstarfind(*args, **kwargs)`

Deprecated since version 0.3: The `irafstarfind` function is deprecated and may be removed in a future version. Use `IRAFStarFinder` instead.

Classes

DAOStarFinder	Detect stars in an image using the DAOFIND (Stetson 1987) algorithm.
IRAFStarFinder	Detect stars in an image using IRAF's "starfind" algorithm.
StarFinderBase	Abstract base class for Star Finders.

DAOStarFinder

class photutils.detection.DAOStarFinder

Bases: [photutils.StarFinderBase](#)

Detect stars in an image using the DAOFIND ([Stetson 1987](#)) algorithm.

DAOFIND ([Stetson 1987](#); [PASP 99, 191](#)) searches images for local density maxima that have a peak amplitude greater than `threshold` (approximately; `threshold` is applied to a convolved image) and have a size and shape similar to the defined 2D Gaussian kernel. The Gaussian kernel is defined by the `fw hm`, `ratio`, `theta`, and `sigma_radius` input parameters.

`DAOStarFinder` finds the object centroid by fitting the marginal x and y 1D distributions of the Gaussian kernel to the marginal x and y distributions of the input (unconvolved) data image.

`DAOStarFinder` calculates the object roundness using two methods. The `roundlo` and `roundhi` bounds are applied to both measures of roundness. The first method (`roundness1`; called `SROUND` in [DAOFIND](#)) is based on the source symmetry and is the ratio of a measure of the object's bilateral (2-fold) to four-fold symmetry. The second roundness statistic (`roundness2`; called `GROUND` in [DAOFIND](#)) measures the ratio of the difference in the height of the best fitting Gaussian function in x minus the best fitting Gaussian function in y, divided by the average of the best fitting Gaussian functions in x and y. A circular source will have a zero roundness. An source extended in x or y will have a negative or positive roundness, respectively.

The sharpness statistic measures the ratio of the difference between the height of the central pixel and the mean of the surrounding non-bad pixels in the convolved image, to the height of the best fitting Gaussian function at that point.

Parameters

threshold : float

The absolute image value above which to select sources.

fw hm : float

The full-width half-maximum (FWHM) of the major axis of the Gaussian kernel in units of pixels.

ratio : float, optional

The ratio of the minor to major axis standard deviations of the Gaussian kernel. `ratio` must be strictly positive and less than or equal to 1.0. The default is 1.0 (i.e., a circular Gaussian kernel).

theta : float, optional

The position angle (in degrees) of the major axis of the Gaussian kernel measured counter-clockwise from the positive x axis.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [1 $\text{sigma} = \text{FWHM} / (2.0 * \sqrt{2.0 * \log(2.0)}))$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

sky : float, optional

The background sky level of the image. Setting sky affects only the output values of the object peak, flux, and mag values. The default is 0.0, which should be used to replicate the results from [DAOFIND](#).

exclude_border : bool, optional

Set to [True](#) to exclude sources found within half the size of the convolution kernel from the image borders. The default is [False](#), which is the mode used by [DAOFIND](#).

See also:

[IRAFStarFinder](#)

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in [DAOFIND](#) are boundary='constant' and constant=0.0.

References

[\[R4\]](#), [\[R5\]](#)

Methods Summary

find_stars(data)

Find stars in an astronomical image.

Methods Documentation

find_stars(data)

Find stars in an astronomical image.

Parameters

data : array_like

The 2D image array.

Returns

table : [Table](#)

A table of found objects with the following parameters:

- `id`: unique object identification number.
- `xcentroid`, `ycentroid`: object centroid.
- `sharpness`: object sharpness.
- `roundness1`: object roundness based on symmetry.
- `roundness2`: object roundness based on marginal Gaussian fits.
- `npix`: number of pixels in the Gaussian kernel.
- `sky`: the input sky parameter.
- `peak`: the peak, sky-subtracted, pixel value of the object.
- `flux`: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of `DAOFIND` if `sky` is 0.0.
- `mag`: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of `DAOFIND` if `sky` is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF’s `starfind` are `boundary='constant'` and `constant=0.0`.

IRAF’s `starfind` uses `hwhm`, `fradius`, and `sepmin` as input parameters. The equivalent input values for `IRAFStarFinder` are:

- `FWHM` = `hwhm` * 2
- `Sigma_Radius` = `fradius` * $\sqrt{2.0 \times \log(2.0)}$
- `MinSep_FWHM` = 0.5 * `sepmin`

The main differences between `DAOStarFinder` and `IRAFStarFinder` are:

- `IRAFStarFinder` always uses a 2D circular Gaussian kernel, while `DAOStarFinder` can use an elliptical Gaussian kernel.
- `IRAFStarFinder` calculates the objects’ centroid, roundness, and sharpness using image moments.

IRAFStarFinder

`class photutils.detection.IRAFStarFinder`

Bases: `photutils.StarFinderBase`

Detect stars in an image using IRAF’s “starfind” algorithm.

`starfind` searches images for local density maxima that have a peak amplitude greater than `threshold` above the local background and have a PSF full-width half-maximum similar to the input `FWHM`. The objects’ centroid, roundness (ellipticity), and sharpness are calculated using image moments.

Parameters

`threshold` : float

The absolute image value above which to select sources.

`FWHM` : float

The full-width half-maximum (FWHM) of the 2D circular Gaussian kernel in units of pixels.

minsep_fwhm : float, optional

The minimum separation for detected objects in units of fwhm.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [1
 $\text{sigma} = \text{FWHM} / 2.0 * \sqrt{2.0 * \log(2.0)}$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

sky : float, optional

The background sky level of the image. Inputting a `sky` value will override the background sky estimate. Setting `sky` affects only the output values of the object peak, `flux`, and `mag` values. The default is `None`, which means the `sky` value will be estimated using the `starfind` method.

exclude_border : bool, optional

Set to `True` to exclude sources found within half the size of the convolution kernel from the image borders. The default is `False`, which is the mode used by `starfind`.

See also:

[DAOStarFinder](#)

References

[R6]

Methods Summary

`find_stars(data)`

Find stars in an astronomical image.

Methods Documentation

`find_stars(data)`

Find stars in an astronomical image.

Parameters

`data` : array_like

The 2D image array.

Returns

table : [Table](#)

A table of found objects with the following parameters:

- `id`: unique object identification number.
- `xcentroid`, `ycentroid`: object centroid.
- `sharpness`: object sharpness.
- `roundness1`: object roundness based on symmetry.
- `roundness2`: object roundness based on marginal Gaussian fits.
- `npix`: number of pixels in the Gaussian kernel.
- `sky`: the input sky parameter.
- `peak`: the peak, sky-subtracted, pixel value of the object.
- `flux`: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of [DAOFIND](#) if `sky` is 0.0.
- `mag`: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of [DAOFIND](#) if `sky` is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF's `starfind` are `boundary='constant'` and `constant=0.0`.

IRAF's `starfind` uses `hwhm`, `fradius`, and `sepmin` as input parameters. The equivalent input values for `IRAFStarFinder` are:

- `FWHM = hwhm * 2`
- `Sigma_Radius = fradius * sqrt(2.0*log(2.0))`
- `MinSep_FWHM = 0.5 * sepmin`

The main differences between `DAOStarFinder` and `IRAFStarFinder` are:

- `IRAFStarFinder` always uses a 2D circular Gaussian kernel, while `DAOStarFinder` can use an elliptical Gaussian kernel.
- `IRAFStarFinder` calculates the objects' centroid, roundness, and sharpness using image moments.

StarFinderBase

`class photutils.detection.StarFinderBase`

Bases: `object`

Abstract base class for Star Finders.

Methods Summary

<code>__call__(...)</code> <==> <code>x(...)</code>	
<code>find_stars(data)</code>	Find stars in an astronomical image.

Methods Documentation

`__call__(...) <==> x(...)`

find_stars(*data*)

Find stars in an astronomical image.

Parameters

data : array_like

The 2D image array.

Returns

table : `Table`

A table of found objects with the following parameters:

- **id**: unique object identification number.
- **xcentroid**, **ycentroid**: object centroid.
- **sharpness**: object sharpness.
- **roundness1**: object roundness based on symmetry.
- **roundness2**: object roundness based on marginal Gaussian fits.
- **npix**: number of pixels in the Gaussian kernel.
- **sky**: the input sky parameter.
- **peak**: the peak, sky-subtracted, pixel value of the object.
- **flux**: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of `DAOFIND` if sky is 0.0.
- **mag**: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of `DAOFIND` if sky is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF's `starfind` are `boundary='constant'` and `constant=0.0`.

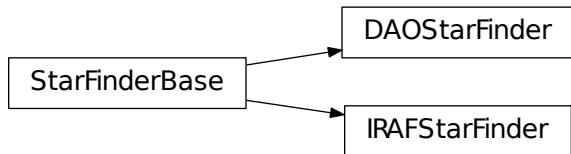
IRAF's `starfind` uses `hwhm`, `fradius`, and `sepmin` as input parameters. The equivalent input values for `IRAFStarFinder` are:

- `FWHM = hwhm * 2`
- `Sigma_Radius = fradius * sqrt(2.0*log(2.0))`
- `MinSep_FWHM = 0.5 * sepmin`

The main differences between `DAOStarFinder` and `IRAFStarFinder` are:

- `IRAFStarFinder` always uses a 2D circular Gaussian kernel, while `DAOStarFinder` can use an elliptical Gaussian kernel.
- `IRAFStarFinder` calculates the objects' centroid, roundness, and sharpness using image moments.

Class Inheritance Diagram



CHAPTER 7

Grouping Algorithms

Introduction

In Point Spread Function (PSF) photometry, a grouping algorithm is primarily used to divide a star list into optimum groups. More precisely, a grouping algorithm must be able to decide whether two or more stars belong to the same group, i. e., whether there are any pixels whose counts are due to the linear combination of counts from two or more sources.

DAOPHOT GROUP

Stetson, in his seminal paper (Stetson 1987, PASP 99, 191), provided a simple and powerful grouping algorithm to decide whether or not the profile of a given star extends into the fitting region around the centroid of any other star. This goal is achieved by means of a variable called “critical separation”, which is defined as the distance such that any two stars separated by less than it would be overlapping. Stetson also gives intuitive reasoning to suggest that the critical separation may be defined as the product of fwhm with some positive real number.

Grouping Sources

Photutils provides an implementation of DAOPHOT GROUP in the `DAOGroup` class. Let’s take a look at a simple example.

First, let’s make some Gaussian sources using `make_random_gaussians` and `make_gaussian_sources`. The former will return a `Table` containing parameters for 2D Gaussian sources and the latter will make an actual image using that table.

```
import numpy as np
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_random_gaussians
import matplotlib.pyplot as plt

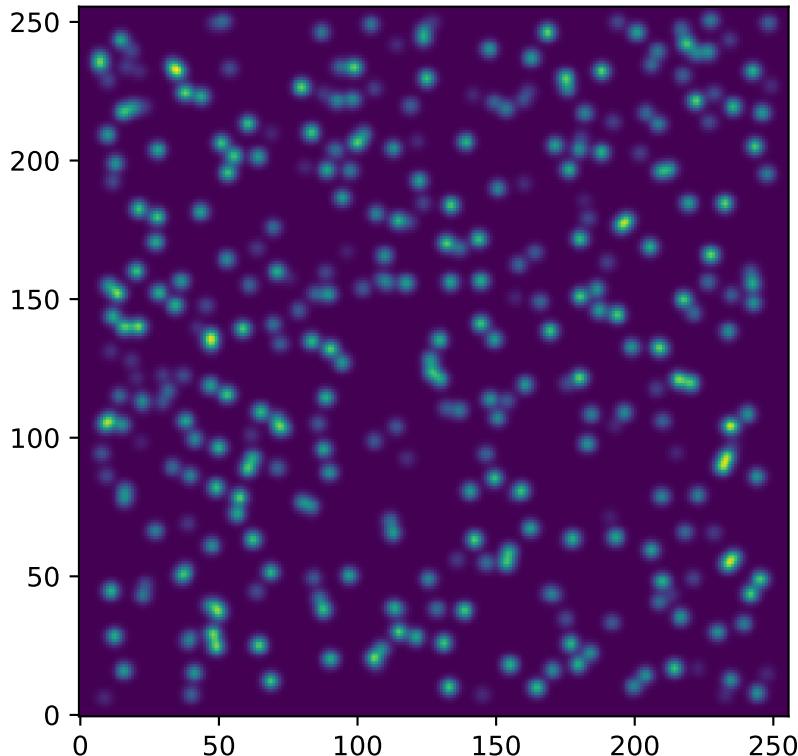
n_sources = 350
```

```
min_flux = 500
max_flux = 5000
min_xmean = min_ymean = 6
max_xmean = max_ymean = 250
sigma_psf = 2.0

starlist = make_random_gaussians(n_sources, [min_flux, max_flux], \
    [min_xmean, max_xmean], [min_ymean, max_ymean], \
    [sigma_psf, sigma_psf], [sigma_psf, sigma_psf], \
    random_state=1234)
shape = (256, 256)

sim_image = make_gaussian_sources(shape, starlist)

plt.imshow(sim_image, origin='lower', interpolation='nearest',
           cmap='viridis')
plt.show()
```

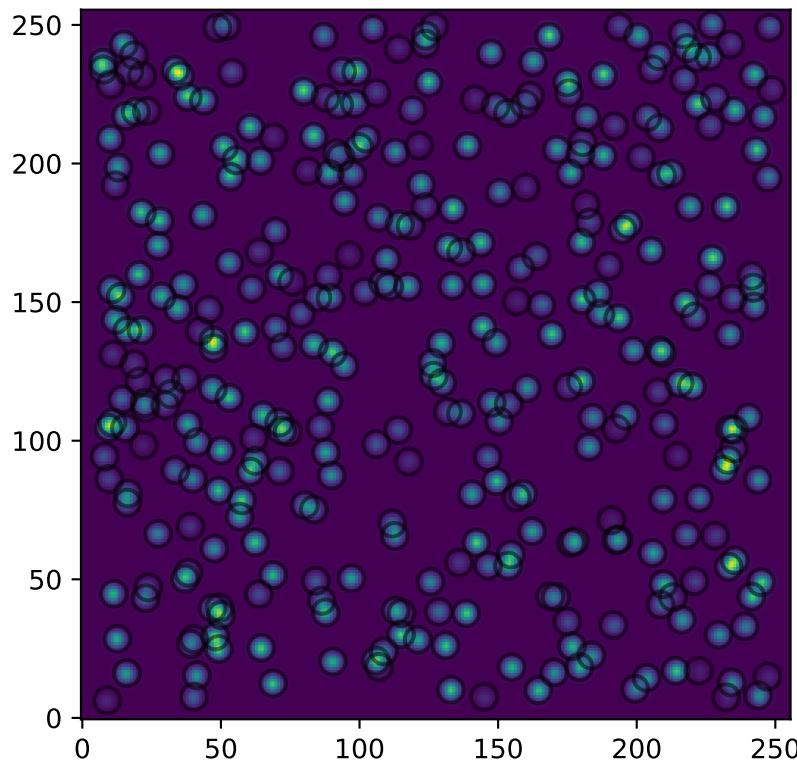


Now, we need to rename the columns of the centroid positions so that they agree with the names that DAOGroup expect:

```
starlist['x_mean'].name = 'x_0'
starlist['y_mean'].name = 'y_0'
```

Before finding groups, let's plot circular apertures around the sources.

```
from photutils import CircularAperture
from astropy.stats import gaussian_sigma_to_fwhm
circ_aperture = CircularAperture((starlist['x_0'], starlist['y_0']),
                                  r=sigma_psf*gaussian_sigma_to_fwhm)
plt.imshow(sim_image, origin='lower', interpolation='nearest',
           cmap='viridis')
circ_aperture.plot(lw=1.5, alpha=0.5)
plt.show()
```



Let's create a `DAOGroup` object and set its `crit_separation` attribute to $1.5 \times \text{fwhm}$:

```
from photutils.psf.groupstars import DAOGroup

fwhm = sigma_psf*gaussian_sigma_to_fwhm
daogroup = DAOGroup(crit_separation=1.5*fwhm)
```

Now, we can use the instance of `DAOGroup` as a function calling which receives as input our list of stars `starlist`:

```
star_groups = daogroup(starlist)
```

This procedure copies `starlist` into `star_groups` and adds a new column to `star_groups` called `group_id`. This column contains integer numbers which represent the group that the sources belong to.

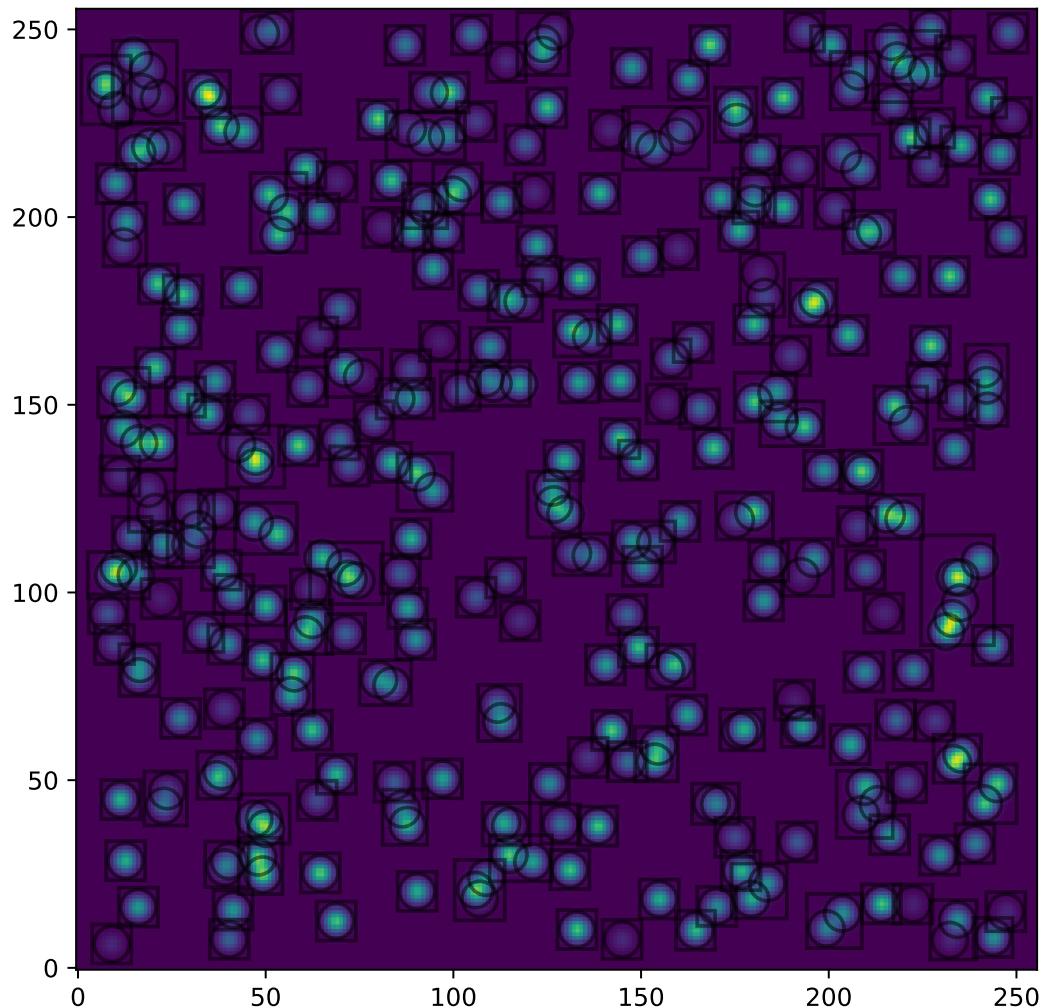
Finally, one can use the `group_by` functionality from `Table` to create groups according `group_id`:

```
star_groups = star_groups.groupby('group_id')
```

Now, let's plot rectangular apertures which cover each group:

```
from photutils import RectangularAperture

plt.imshow(sim_image, origin='lower', interpolation='nearest',
           cmap='viridis')
for group in star_groups.groups:
    group_center = (np.median(group['x_0']), np.median(group['y_0']))
    xmin = np.min(group['x_0']) - fwhm
    xmax = np.max(group['x_0']) + fwhm
    ymin = np.min(group['y_0']) - fwhm
    ymax = np.max(group['y_0']) + fwhm
    group_width = xmax - xmin + 1
    group_height = ymax - ymin + 1
    rect_aperture = RectangularAperture(group_center, group_width,
                                         group_height, theta=0)
    rect_aperture.plot(lw=1.5, alpha=0.5)
circ_aperture.plot(lw=1.5, alpha=0.5)
plt.show()
```



CHAPTER 8

Aperture Photometry (`photutils.aperture`)

Introduction

In Photutils, the `aperture_photometry()` function is the main tool to perform aperture photometry on an astronomical image for a given set of apertures.

Photutils provides several apertures defined in pixel or sky coordinates. The aperture classes that are defined in pixel coordinates are:

- `CircularAperture`
- `CircularAnnulus`
- `EllipticalAperture`
- `EllipticalAnnulus`
- `RectangularAperture`
- `RectangularAnnulus`

Each of these classes has a corresponding variant defined in celestial coordinates:

- `SkyCircularAperture`
- `SkyCircularAnnulus`
- `SkyEllipticalAperture`
- `SkyEllipticalAnnulus`
- `SkyRectangularAperture`
- `SkyRectangularAnnulus`

To perform aperture photometry with sky-based apertures, one will need to specify a WCS transformation.

Users can also create their own custom apertures (see [Defining Your Own Custom Apertures](#)).

Creating Aperture Objects

The first step in performing aperture photometry is to create an aperture object. An aperture object is defined by a position (or a list of positions) and parameters that define its size and possibly, orientation (e.g., an elliptical aperture).

We start with an example of creating a circular aperture in pixel coordinates using the `CircularAperture` class:

```
>>> from photutils import CircularAperture
>>> positions = [(30., 30.), (40., 40.)]
>>> apertures = CircularAperture(positions, r=3.)
```

The positions should be either a single tuple of (x, y), a list of (x, y) tuples, or an array with shape Nx2, where N is the number of positions. The above example defines two circular apertures located at pixel coordinates (30, 30) and (40, 40) with a radius of 3 pixels.

Creating an aperture object in celestial coordinates is similar. One first uses the `SkyCoord` class to define celestial coordinates and then the `SkyCircularAperture` class to define the aperture object:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> from photutils import SkyCircularAperture
>>> positions = SkyCoord(l=[1.2, 2.3] * u.deg, b=[0.1, 0.2] * u.deg,
...                         frame='galactic')
>>> apertures = SkyCircularAperture(positions, r=4. * u.arcsec)
```

Note: Sky apertures are not defined completely in celestial coordinates. They simply use celestial coordinates to define the central position, and the remaining parameters are converted to pixels using the pixel scale of the image at the central position. Projection distortions are not taken into account. If the apertures were defined completely in celestial coordinates, their shapes would not be preserved when converting to pixel coordinates.

Performing Aperture Photometry

After the aperture object is created, we can then perform the photometry using the `aperture_photometry()` function. We start by defining the apertures as described above:

```
>>> positions = [(30., 30.), (40., 40.)]
>>> apertures = CircularAperture(positions, r=3.)
```

and then we call the `aperture_photometry()` function with the data and the apertures:

```
>>> import numpy as np
>>> from photutils import aperture_photometry
>>> data = np.ones((100, 100))
>>> phot_table = aperture_photometry(data, apertures)
>>> print(phot_table)
   id  xcenter  ycenter  aperture_sum
     pix      pix
-----
 1    30.0    30.0  28.2743338823
 2    40.0    40.0  28.2743338823
```

This function returns the results of the photometry in an Astropy `QTable`. In this example, the table has four columns, named 'id', 'xcenter', 'ycenter', and 'aperture_sum'.

Since all the data values are 1.0, the aperture sums are equal to the area of a circle with a radius of 3:

```
>>> print(np.pi * 3. ** 2)
28.2743338823
```

Aperture and Pixel Overlap

The overlap of the apertures with the data pixels can be handled in different ways. For the default method (method='exact'), the exact intersection of the aperture with each pixel is calculated. The other options, 'center' and 'subpixel', are faster, but with the expense of less precision. For 'center', a pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. For 'subpixel', pixels are divided into a number of subpixels, which are in or out of the aperture based on their centers. For this method, the number of subpixels needs to be set with the subpixels keyword.

This example uses the 'subpixel' method where pixels are resampled by a factor of 5 (subpixels=5) in each dimension:

```
>>> phot_table = aperture_photometry(data, apertures, method='subpixel',
...                                     subpixels=5)
>>> print(phot_table)
   id  xcenter  ycenter  aperture_sum
      pix      pix
-----
 1    30.0     30.0      27.96
 2    40.0     40.0      27.96
```

Note that the results differ from the true value of 28.274333 (see above).

For the 'subpixel' method, the default value is subpixels=5, meaning that each pixel is equally divided into 25 smaller pixels (this is the method and subsampling factor used in [SourceExtractor](#)). The precision can be increased by increasing subpixels, but note that computation time will be increased.

Multiple Apertures at Each Position

While the [Aperture](#) objects support multiple positions, they must have a fixed shape, e.g. radius, size, and orientation.

To perform photometry in multiple apertures at each position, one may input a list of aperture objects to the [aperture_photometry\(\)](#) function.

Suppose that we wish to use three circular apertures, with radii of 3, 4, and 5 pixels, on each source:

```
>>> radii = [3., 4., 5.]
>>> apertures = [CircularAperture(positions, r=r) for r in radii]
>>> phot_table = aperture_photometry(data, apertures)
>>> print(phot_table)
   id  xcenter  ycenter  aperture_sum_0  aperture_sum_1  aperture_sum_2
      pix      pix
-----
 1    30.0     30.0    28.2743338823  50.2654824574  78.5398163397
 2    40.0     40.0    28.2743338823  50.2654824574  78.5398163397
```

For multiple apertures, the output table column names are appended with the positions index.

Other apertures have multiple parameters specifying the aperture size and orientation. For example, for elliptical apertures, one must specify a, b, and theta:

```
>>> from photutils import EllipticalAperture
>>> a = 5.
>>> b = 3.
>>> theta = np.pi / 4.
>>> apertures = EllipticalAperture(positions, a, b, theta)
>>> phot_table = aperture_photometry(data, apertures)
>>> print(phot_table)
   id xcenter ycenter  aperture_sum
      pix      pix
-----
 1    30.0    30.0  47.1238898038
 2    40.0    40.0  47.1238898038
```

Again, for multiple apertures one should input a list of aperture objects, each with identical positions:

```
>>> a = [5., 6., 7.]
>>> b = [3., 4., 5.]
>>> theta = np.pi / 4.
>>> apertures = [EllipticalAperture(positions, a=ai, b=bi, theta=theta)
...             for (ai, bi) in zip(a, b)]
>>> phot_table = aperture_photometry(data, apertures)
>>> print(phot_table)
   id xcenter ycenter aperture_sum_0 aperture_sum_1 aperture_sum_2
      pix      pix
-----
 1    30.0    30.0  47.1238898038  75.3982236862  109.955742876
 2    40.0    40.0  47.1238898038  75.3982236862  109.955742876
```

Background Subtraction

Global Background Subtraction

`aperture_photometry()` assumes that the data have been background-subtracted. If `bkg` is an array representing the background of the data (determined by `Background2D` or an external function), simply do:

```
>>> phot_table = aperture_photometry(data - bkg, apertures)
```

Local Background Subtraction

Suppose we want to perform the photometry in a circular aperture with a radius of 3 pixels and estimate the local background level around each source with a circular annulus of inner radius 6 pixels and outer radius 8 pixels. We start by defining the apertures:

```
>>> from photutils import CircularAnnulus
>>> apertures = CircularAperture(positions, r=3)
>>> annulus_apertures = CircularAnnulus(positions, r_in=6., r_out=8.)
```

We then perform the photometry in both apertures:

```
>>> apers = [apertures, annulus_apertures]
>>> phot_table = aperture_photometry(data, apers)
>>> print(phot_table)
   id xcenter ycenter aperture_sum_0 aperture_sum_1
```

	pix	pix		
1	30.0	30.0	28.2743338823	87.9645943005
2	40.0	40.0	28.2743338823	87.9645943005

Note that we cannot simply subtract the aperture sums because the apertures have different areas.

To calculate the mean local background within the circular annulus aperture, we need to divide its sum by its area, which can be calculated using the `area()` method:

```
>>> bkg_mean = phot_table['aperture_sum_1'] / annulus_apertures.area()
```

The background sum within the circular aperture is then the mean local background times the circular aperture area:

```
>>> bkg_sum = bkg_mean * apertures.area()
>>> final_sum = phot_table['aperture_sum_0'] - bkg_sum
>>> phot_table['residual_aperture_sum'] = final_sum
>>> print(phot_table['residual_aperture_sum'])
residual_aperture_sum
-----
-7.1054273576e-15
-7.1054273576e-15
```

The result here should be zero because all of the data values are 1.0 (the tiny difference from 0.0 is due to numerical precision).

Error Estimation

If and only if the `error` keyword is input to `aperture_photometry()`, the returned table will include a '`aperture_sum_err`' column in addition to '`aperture_sum`'. '`aperture_sum_err`' provides the propagated uncertainty associated with '`aperture_sum`'.

For example, suppose we have previously calculated the error on each pixel's value and saved it in the array `error`:

```
>>> error = 0.1 * data
>>> phot_table = aperture_photometry(data, apertures, error=error)
>>> print(phot_table)
   id  xcenter  ycenter  aperture_sum  aperture_sum_err
      pix      pix
-----
 1    30.0     30.0   28.2743338823   0.531736155272
 2    40.0     40.0   28.2743338823   0.531736155272
```

'`aperture_sum_err`' values are given by:

$$\Delta F = \sqrt{\sum_{i \in A} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, A are the non-masked pixels in the aperture, and $\sigma_{\text{tot},i}$ is the input `error` array.

In the example above, it is assumed that the `error` keyword specifies the *total* error – either it includes Poisson noise due to individual sources or such noise is irrelevant. However, it is often the case that one has calculated a smooth “background-only error” array, which by design doesn’t include increased noise on bright pixels. To include Poisson noise from the sources, we can use the `calc_total_error()` function.

Let's assume we have a background-only image called `bkg_error`. If our data are in units of electrons/s, we would use the exposure time as the effective gain:

```
>>> from photutils.utils import calc_total_error
>>> effective_gain = 500 # seconds
>>> error = calc_total_error(data, bkg_error, effective_gain)
>>> phot_table = aperture_photometry(data - bkg, apertures, error=error)
```

Note: In cases where the error array is slowly varying across the image, it is not necessary to sum the error from every pixel in the aperture individually. Instead, we can approximate the error as being roughly constant across the aperture and simply take the value of σ at the center of the aperture. This can be done by setting the keyword `pixelwise_errors=False`. In this case the flux error is

$$\Delta F = \sigma\sqrt{A}$$

where σ is the error at the center of the aperture and A is the area of the aperture.

Pixel Masking

Pixels can be ignored/excluded (e.g., bad pixels) from the aperture photometry by providing an image mask via the `mask` keyword:

```
>>> data = np.ones((5, 5))
>>> aperture = CircularAperture((2, 2), 2.)
>>> mask = np.zeros_like(data, dtype=bool)
>>> data[2, 2] = 100. # bad pixel
>>> mask[2, 2] = True
>>> t1 = aperture_photometry(data, aperture, mask=mask)
>>> print(t1['aperture_sum'])
aperture_sum
-----
11.5663706144
```

The result is very different if a mask image is not provided:

```
>>> t2 = aperture_photometry(data, aperture)
>>> print(t2['aperture_sum'])
aperture_sum
-----
111.566370614
```

Aperture Photometry Using Sky Coordinates

As mentioned in [Creating Aperture Objects](#), performing photometry using apertures defined in celestial coordinates simply requires defining a “sky” aperture at positions defined by a `SkyCoord` object. Here we show an example of photometry on real data using a `SkyCircularAperture`.

We start by loading a Spitzer 4.5 micron image of a region of the Galactic plane:

```
>>> from photutils import datasets
>>> hdu = datasets.load_spitzer_image()
Downloading http://data.astropy.org/photometry/spitzer_example_image.fits [Done]
>>> catalog = datasets.load_spitzer_catalog()
Downloading http://data.astropy.org/photometry/spitzer_example_catalog.xml [Done]
```

The catalog contains (among other things) the Galactic coordinates of the sources in the image as well as the PSF-fitted fluxes from the official Spitzer data reduction. We define the apertures positions based on the existing catalog positions:

```
>>> positions = SkyCoord(catalog['l'], catalog['b'], frame='galactic')
>>> apertures = SkyCircularAperture(positions, r=4.8 * u.arcsec)
```

Now perform the photometry in these apertures using the hdu. The hdu object is a FITS HDU that contains the data and a header describing the WCS transformation of the image. The WCS includes the coordinate frame of the image and the projection from celestial to pixel coordinates. The `aperture_photometry` function uses the WCS information to automatically convert the apertures defined in celestial coordinates into pixel coordinates:

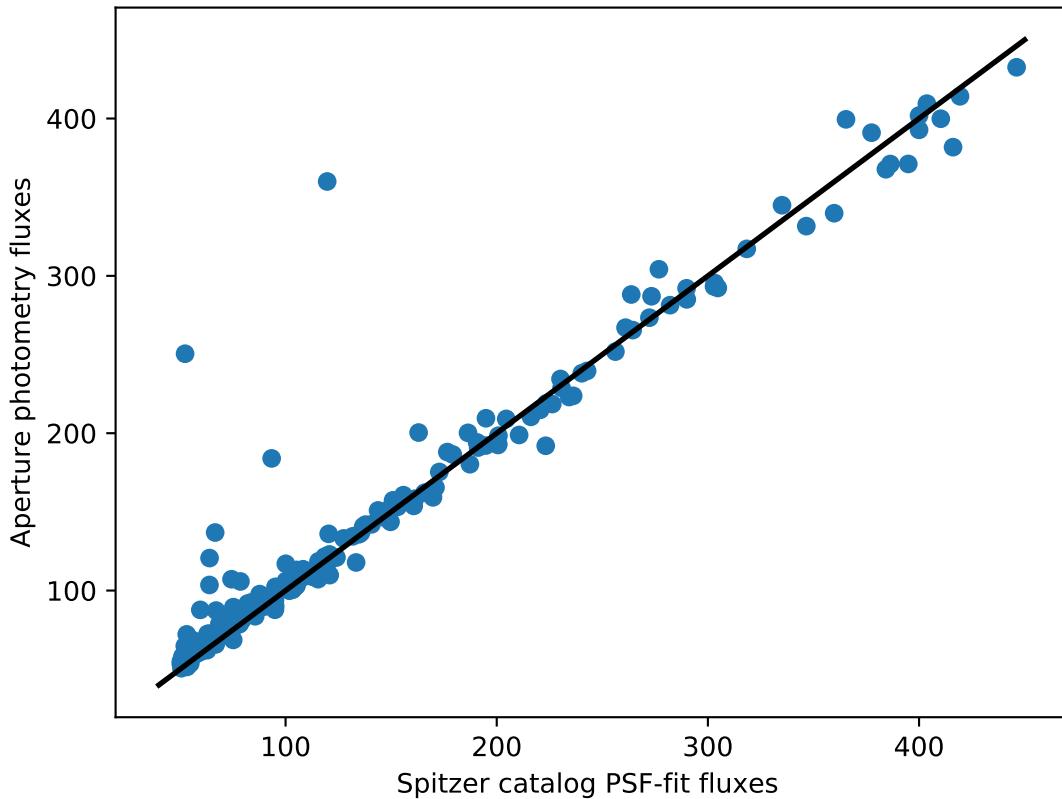
```
>>> phot_table = aperture_photometry(hdu, apertures)
```

The Spitzer catalog also contains the official fluxes for the sources, so we can compare to our fluxes. Because the Spitzer catalog fluxes are in units of mJy and the data are in units of MJy/sr, we need to convert units before comparing the results. The image data have a pixel scale of 1.2 arcsec/pixel.

```
>>> import astropy.units as u
>>> factor = (1.2 * u.arcsec) ** 2 / u.pixel
>>> fluxes_catalog = catalog['f4_5']
>>> converted_aperture_sum = (phot_table['aperture_sum'] *
...                               factor).to(u.mJy / u.pixel)
```

Finally, we can plot the comparison of the photometry:

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(fluxes_catalog, converted_aperture_sum.value)
>>> plt.xlabel('Spitzer catalog PSF-fit fluxes ')
>>> plt.ylabel('Aperture photometry fluxes')
```



Despite using different methods, the two catalogs are in good agreement. The aperture photometry fluxes are based on a circular aperture with a radius of 4.8 arcsec. The Spitzer catalog fluxes were computed using PSF photometry. Therefore, differences are expected between the two measurements.

Aperture Masks

All `PixelAperture` objects have a `to_mask()` method that returns a list of `ApertureMask` objects, one for each aperture position. The `ApertureMask` object contains a cutout of the aperture mask and a `slices` object that provides the locations where the mask is to be applied. It also provides a `to_image()` method to obtain an image of the mask in a 2D array of the given shape, a `cutout()` method to create a cutout from the input data over the mask bounding box, and an `apply()` method to apply the aperture mask to the input data to create a mask-weighted data cutout. All of these methods properly handle the cases of partial or no overlap of the aperture mask with the data.

Let's start by creating an aperture object:

```
>>> from photutils import CircularAperture  
>>> positions = [(30., 30.), (40., 40.)]  
>>> apertures = CircularAperture(positions, r=3.)
```

Now let's create a list of `ApertureMask` objects using the `to_mask()` method:

```
>>> masks = aperture.to_mask(method='center')
```

We can now create an image with of the first aperture mask at its position:

```
>>> mask = masks[0]
>>> image = mask.to_image(shape=(200, 200))
```

We can also create a cutout from a data image over the mask domain:

```
>>> data_cutout = mask.cutout(data)
```

We can also create a mask-weighted cutout from the data. Here the circular aperture mask has been applied to the data:

```
>>> data_cutout_aper = mask.apply(data)
```

Defining Your Own Custom Apertures

The `aperture_photometry()` function can perform aperture photometry in arbitrary apertures. This function accepts any `Aperture`-derived objects, such as `CircularAperture`. This makes it simple to extend functionality: a new type of aperture photometry simply requires the definition of a new `Aperture` subclass.

All `PixelAperture` subclasses must define a `_slices` property, `to_mask()` and `plot()` methods, and optionally an `area()` method. All `SkyAperture` subclasses must implement only a `to_pixel()` method.

- `_slices`: A property defining the minimal bounding box slices for the aperture at each position.
- `to_mask()`: A method to return a list of `ApertureMask` objects, one for each aperture position.
- `area()`: A method to return the exact analytical area (in pixels**2) of the aperture.
- `plot()`: A method to plot the aperture on a `matplotlib.axes.Axes` instance.

See Also

1. [IRAF's APPHOT specification \[PDF\]](#) (Sec. 3.3.5.8 - 3.3.5.9)
2. [SourceExtractor Manual \[PDF\]](#) (Sec. 9.4 p. 36)

Reference/API

This subpackage contains modules and packages for identifying sources in an astronomical image.

Functions

<code>aperture_photometry(data, apertures[, ...])</code>	Perform aperture photometry on the input data by summing the flux within the given aperture(s).
--	---

`aperture_photometry`

```
photutils.aperture.aperture_photometry(data, apertures, error=None, pixelwise_error=True,
                                        mask=None, method=u'exact', subpixels=5, unit=None,
                                        wcs=None)
```

Perform aperture photometry on the input data by summing the flux within the given aperture(s).

Parameters**data** : array_like, [Quantity](#), [ImageHDU](#), or [HDUList](#)

The 2D array on which to perform photometry. `data` should be background-subtracted. Units can be used during the photometry, either provided with the data (i.e. a [Quantity](#) array) or the `unit` keyword. If `data` is an [ImageHDU](#) or [HDUList](#), the unit is determined from the '`BUNIT`' header keyword.

apertures : [Aperture](#)

The aperture(s) to use for the photometry.

error : array_like or [Quantity](#), optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see `calc_total_error`). `.error` must have the same shape as the input data.

pixelwise_error : bool, optional

If `True` (default), the photometric error is calculated using the `error` values from each pixel within the aperture. If `False`, the `error` value at the center of the aperture is used for the entire aperture.

mask : array_like (bool), optional

A boolean mask with the same shape as `data` where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

• **‘exact’ (default):**

The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.

• **‘center’:**

A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).

• **‘subpixel’:**

A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels` ** 2 subpixels.

unit : [UnitBase](#) object or str, optional

An object that represents the unit associated with the input data and `error` arrays. Must be a [UnitBase](#) object or a string parseable by the [units](#) package. If `data` or `error` already have a different unit, the input `unit` will not be used and a warning will be raised. If `data` is an [ImageHDU](#) or [HDUList](#), `unit` will override the ‘`BUNIT`’ header keyword.

wcs : [WCS](#), optional

The WCS transformation to use if the input apertures is a [SkyAperture](#) object. If data is an [ImageHDU](#) or [HDUList](#), wcs overrides any WCS transformation present in the header.

Returns

table : [QTable](#)

A table of the photometry with the following columns:

- 'id': The source ID.
- 'xcenter', 'ycenter': The x and y pixel coordinates of the input aperture center(s).
- 'celestial_center': The celestial coordinates of the input aperture center(s). Returned only if the input apertures is a [SkyAperture](#) object.
- 'aperture_sum': The sum of the values within the aperture.
- 'aperture_sum_err': The corresponding uncertainty in the 'aperture_sum' values. Returned only if the input error is not [None](#).

The table metadata includes the Astropy and Photutils version numbers and the [aperture_photometry](#) calling arguments.

Notes

This function is decorated with [support_nddata](#) and thus supports [NDData](#) objects as input.

Classes

Aperture	Abstract base class for all apertures.
ApertureMask (mask, bbox_slice)	Class for an aperture mask.
CircularAnnulus	Circular annulus aperture(s), defined in pixel coordinates.
CircularAperture	Circular aperture(s), defined in pixel coordinates.
CircularMaskMixin	Mixin class to create masks for circular and circular-annulus aperture objects.
EllipticalAnnulus	Elliptical annulus aperture(s), defined in pixel coordinates.
EllipticalAperture	Elliptical aperture(s), defined in pixel coordinates.
EllipticalMaskMixin	Mixin class to create masks for elliptical and elliptical-annulus aperture objects.
PixelAperture	Abstract base class for 2D apertures defined in pixel coordinates.
RectangularAnnulus	Rectangular annulus aperture(s), defined in pixel coordinates.
RectangularAperture	Rectangular aperture(s), defined in pixel coordinates.
RectangularMaskMixin	Mixin class to create masks for rectangular or rectangular-annulus aperture objects.
SkyAperture	Abstract base class for 2D apertures defined in celestial coordinates.
SkyCircularAnnulus	Circular annulus aperture(s), defined in sky coordinates.
SkyCircularAperture	Circular aperture(s), defined in sky coordinates.
SkyEllipticalAnnulus	Elliptical annulus aperture(s), defined in sky coordinates.

Continued on next page

Table 8.2 – continued from previous page

SkyEllipticalAperture	Elliptical aperture(s), defined in sky coordinates.
SkyRectangularAnnulus	Rectangular annulus aperture(s), defined in sky coordinates.
SkyRectangularAperture	Rectangular aperture(s), defined in sky coordinates.

Aperture

class photutils.aperture.Aperture

Bases: `object`

Abstract base class for all apertures.

ApertureMask

class photutils.aperture.ApertureMask(mask, bbox_slice)

Bases: `object`

Class for an aperture mask.

Parameters

`mask` : array_like

A 2D array of an aperture mask representing the fractional overlap of the aperture on the pixel grid. This should be the full-sized (i.e. not truncated) array that is the direct output of one of the low-level `photutils.geometry` functions.

`bbox_slice` : tuple of slice objects

A tuple of (y, x) numpy slice objects defining the aperture minimal bounding box.

Attributes Summary

<code>array</code>	The 2D mask array.
--------------------	--------------------

Methods Summary

<code>apply(data[, fill_value])</code>	Apply the aperture mask to the input data, taking any edge effects into account.
<code>cutout(data[, fill_value])</code>	Create a cutout from the input data over the mask bounding box, taking any edge effects into account.
<code>to_image(shape)</code>	Return an image of the mask in a 2D array of the given shape, taking any edge effects into account.

Attributes Documentation

array

The 2D mask array.

Methods Documentation

`apply(data, fill_value=0.0)`

Apply the aperture mask to the input data, taking any edge effects into account.

The result is a mask-weighted cutout from the data.

Parameters

`data` : array_like or `Quantity`

A 2D array on which to apply the aperture mask.

`fill_value` : float, optional

The value is used to fill pixels where the aperture mask does not overlap with the input data. The default is 0.

Returns

`result` : `ndarray`

A 2D mask-weighted cutout from the input data. If there is a partial overlap of the aperture mask with the input data, pixels outside of the data will be assigned to `fill_value` before being multiplied with the mask. `None` is returned if there is no overlap of the aperture with the input data.

`cutout(data, fill_value=0.0)`

Create a cutout from the input data over the mask bounding box, taking any edge effects into account.

Parameters

`data` : array_like or `Quantity`

A 2D array on which to apply the aperture mask.

`fill_value` : float, optional

The value is used to fill pixels where the aperture mask does not overlap with the input data. The default is 0.

Returns

`result` : `ndarray`

A 2D array cut out from the input data representing the same cutout region as the aperture mask. If there is a partial overlap of the aperture mask with the input data, pixels outside of the data will be assigned to `fill_value`. `None` is returned if there is no overlap of the aperture with the input data.

`to_image(shape)`

Return an image of the mask in a 2D array of the given shape, taking any edge effects into account.

Parameters

`shape` : tuple of int

The (ny, nx) shape of the output array.

Returns

`result` : `ndarray`

A 2D array of the mask.

CircularAnnulus

```
class photutils.aperture.CircularAnnulus
    Bases: photutils.CircularMaskMixin, photutils.PixelAperture
```

Circular annulus aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or [Quantity](#)

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN [ndarray](#)
- Nx2 or 2xN [Quantity](#) in pixel units

Note that a 2x2 [ndarray](#) or [Quantity](#) is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

r_in : float

The inner radius of the annulus.

r_out : float

The outer radius of the annulus.

Raises

ValueError : [ValueError](#)

If inner radius (**r_in**) is greater than outer radius (**r_out**).

ValueError : [ValueError](#)

If inner radius (**r_in**) is negative.

Methods Summary

area()	Return the exact area of the aperture shape.
plot([origin, indices, ax, fill])	Plot the aperture(s) on a matplotlib Axes instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

area : float

The aperture area.

plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)

Plot the aperture(s) on a matplotlib [Axes](#) instance.

Parameters

origin : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional
The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional
If `None`, then the current `Axes` instance is used.

fill : bool, optional
Set whether to fill the aperture patch. The default is `False`.

kwargs
Any keyword arguments accepted by `matplotlib.patches.Patch`.

CircularAperture

class photutils.aperture.CircularAperture
Bases: `photutils.CircularMaskMixin`, `photutils.PixelAperture`

Circular aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN ndarray
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

r : float

The radius of the aperture(s), in pixels.

Raises

`ValueError` : `ValueError`

If the input radius, `r`, is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

```
plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)
```

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

kwargs

Any keyword arguments accepted by `matplotlib.patches.Patch`.

CircularMaskMixin

```
class photutils.aperture.CircularMaskMixin
```

Bases: `object`

Mixin class to create masks for circular and circular-annulus aperture objects.

Methods Summary

`to_mask([method, subpixels])`

Return a list of `ApertureMask` objects, one for each aperture position.

Methods Documentation

```
to_mask(method=u'exact', subpixels=5)
```

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether

its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

mask : list of `ApertureMask`

A list of aperture mask objects.

EllipticalAnnulus

class photutils.aperture.EllipticalAnnulus

Bases: `photutils.EllipticalMaskMixin`, `photutils.PixelAperture`

Elliptical annulus aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

a_in : float

The inner semimajor axis.

a_out : float

The outer semimajor axis.

b_out : float

The outer semiminor axis. The inner semiminor axis is calculated as:

$$b_{in} = b_{out} \left(\frac{a_{in}}{a_{out}} \right)$$

theta : float

The rotation angle in radians of the semimajor axis from the positive x axis. The rotation angle increases counterclockwise.

Raises

ValueError : `ValueError`

If inner semimajor axis (`a_in`) is greater than outer semimajor axis (`a_out`).

ValueError : `ValueError`

If either the inner semimajor axis (`a_in`) or the outer semiminor axis (`b_out`) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

`area()`

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

`plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)`

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

EllipticalAperture

`class photutils.aperture.EllipticalAperture`

Bases: `photutils.EllipticalMaskMixin`, `photutils.PixelAperture`

Elliptical aperture(s), defined in pixel coordinates.

Parameters

`positions` : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

a : float

The semimajor axis.

b : float

The semiminor axis.

theta : float

The rotation angle in radians of the semimajor axis from the positive x axis. The rotation angle increases counterclockwise.

Raises

ValueError : `ValueError`

If either axis (a or b) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

area : float

The aperture area.

plot(*origin*=(0, 0), *indices*=None, *ax*=None, *fill*=False, ***kwargs*)

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

origin : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

fill : bool, optional

Set whether to fill the aperture patch. The default is `False`.

kwargs

Any keyword arguments accepted by `matplotlib.patches.Patch`.

EllipticalMaskMixin

`class photutils.aperture.EllipticalMaskMixin`

Bases: `object`

Mixin class to create masks for elliptical and elliptical-annulus aperture objects.

Methods Summary

<code>to_mask([method, subpixels])</code>	Return a list of <code>ApertureMask</code> objects, one for each aperture position.
---	---

Methods Documentation

`to_mask(method=u'exact', subpixels=5)`

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

`subpixels` : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels` $\star\star$ 2 subpixels.

Returns

`mask` : list of `ApertureMask`

A list of aperture mask objects.

PixelAperture

`class photutils.aperture.PixelAperture`

Bases: `photutils.Aperture`

Abstract base class for 2D apertures defined in pixel coordinates.

Derived classes must define a `_slices` property, `to_mask` and `plot` methods, and optionally an `area` method.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape. Continued on next page
---------------------	--

Table 8.11 – continued from previous page

<code>do_photometry(data[, error, ...])</code>	Perform aperture photometry on the input data.
<code>mask_area([method, subpixels])</code>	Return the area of the aperture(s) mask.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.
<code>to_mask([method, subpixels])</code>	Return a list of <code>ApertureMask</code> objects, one for each aperture position.

Methods Documentation

`area()`

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

`do_photometry(data, error=None, pixelwise_error=True, mask=None, method=u'exact', subpixels=5, unit=None)`

Perform aperture photometry on the input data.

Parameters

`data` : array_like or `Quantity` instance

The 2D array on which to perform photometry. `data` should be background subtracted.

`error` : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include all sources of error, including the Poisson error of the sources (see `calc_total_error`). `.error` must have the same shape as the input data.

`pixelwise_error` : bool, optional

If `True` (default), the photometric error is calculated using the `error` values from each pixel within the aperture. If `False`, the `error` value at the center of the aperture is used for the entire aperture.

`mask` : array_like (bool), optional

A boolean mask with the same shape as `data` where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’ A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

`subpixels` : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels` $\star\star$ 2 subpixels.

unit : `UnitBase` object or str, optional

An object that represents the unit associated with the input data and error arrays. Must be a `UnitBase` object or a string parseable by the `units` package. If data or error already have a different unit, the input unit will not be used and a warning will be raised.

Returns

aperture_sums : `ndarray` or `Quantity`

The sums within each aperture.

aperture_sum_errs : `ndarray` or `Quantity`

The errors on the sums within each aperture.

mask_area(*method=u'exact'*, *subpixels*=5)

Return the area of the aperture(s) mask.

For *method* other than 'exact', this area will be less than the exact analytical area (e.g. the `area` method). Note that for these methods, the values can also differ because of fractional pixel positions.

Parameters

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- 'exact' (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- 'center': A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- 'subpixel': A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels`=1, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels` $\star\star$ 2 subpixels.

Returns

area : float

A list of the mask area of the aperture(s).

plot(*origin*=(0, 0), *indices*=None, *ax*=None, *fill*=False, ***kwargs*)

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

origin : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

fill : bool, optional

Set whether to fill the aperture patch. The default is `False`.

kwargs

Any keyword arguments accepted by `matplotlib.patches.Patch`.

to_mask(*method=u'exact'*, *subpixels=5*)

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

mask : list of `ApertureMask`

A list of aperture mask objects.

RectangularAnnulus

class photutils.aperture.RectangularAnnulus

Bases: `photutils.RectangularMaskMixin`, `photutils.PixelAperture`

Rectangular annulus aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN ndarray
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2×2 `ndarray` or `Quantity` is interpreted as $N \times 2$, i.e. two rows of (x, y) coordinates.

w_in : float

The inner full width of the aperture. For $\text{theta}=0$ the width side is along the x axis.

w_out : float

The outer full width of the aperture. For $\text{theta}=0$ the width side is along the x axis.

h_out : float

The outer full height of the aperture. The inner full height is calculated as:

$$h_{in} = h_{out} \left(\frac{w_{in}}{w_{out}} \right)$$

For $\text{theta}=0$ the height side is along the y axis.

theta : float

The rotation angle in radians of the width side from the positive x axis. The rotation angle increases counterclockwise.

Raises

ValueError : `ValueError`

If inner width (`w_in`) is greater than outer width (`w_out`).

ValueError : `ValueError`

If either the inner width (`w_in`) or the outer height (`h_out`) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

plot(`origin=(0, 0)`, `indices=None`, `ax=None`, `fill=False`, `kwargs`)**

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional
If `None`, then the current `Axes` instance is used.

fill : bool, optional
Set whether to fill the aperture patch. The default is `False`.

kwargs
Any keyword arguments accepted by `matplotlib.patches.Patch`.

RectangularAperture

class photutils.aperture.RectangularAperture

Bases: `photutils.RectangularMaskMixin`, `photutils.PixelAperture`

Rectangular aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

w : float

The full width of the aperture. For `theta=0` the width side is along the x axis.

h : float

The full height of the aperture. For `theta=0` the height side is along the y axis.

theta : float

The rotation angle in radians of the width (w) side from the positive x axis. The rotation angle increases counterclockwise.

Raises

`ValueError` : `ValueError`

If either width (w) or height (h) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

`area()`

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

`plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)`

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : matplotlib.axes.Axes instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

RectangularMaskMixin

`class photutils.aperture.RectangularMaskMixin`

Bases: `object`

Mixin class to create masks for rectangular or rectangular-annulus aperture objects.

Methods Summary

`to_mask([method, subpixels])`

Return a list of `ApertureMask` objects, one for each aperture position.

Methods Documentation

`to_mask(method=u'exact', subpixels=5)`

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- 'exact' (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- 'center': A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- 'subpixel': A pixel is divided into subpixels (see the subpixels keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If subpixels=1, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into subpixels $\star\star 2$ subpixels.

Returns

mask : list of `ApertureMask`

A list of aperture mask objects.

SkyAperture

`class photutils.aperture.SkyAperture`

Bases: `photutils.Aperture`

Abstract base class for 2D apertures defined in celestial coordinates.

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>PixelAperture</code> object in pixel coordinates.
------------------------------------	---

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a `PixelAperture` object in pixel coordinates.

Parameters

wcs : `WCS`

The WCS transformation to use.

mode : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

aperture : `PixelAperture` object

A `PixelAperture` object.

SkyCircularAnnulus

class photutils.aperture.SkyCircularAnnulus

Bases: photutils.SkyAperture

Circular annulus aperture(s), defined in sky coordinates.

Parameters

positions : SkyCoord

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

r_in : Quantity

The inner radius of the annulus, either in angular or pixel units.

r_out : Quantity

The outer radius of the annulus, either in angular or pixel units.

Methods Summary

to_pixel(wcs[, mode])

Convert the aperture to a CircularAnnulus instance in pixel coordinates.

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a CircularAnnulus instance in pixel coordinates.

Parameters

wcs : WCS

The WCS transformation to use.

mode : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions (‘all’; default) or only including only the core WCS transformation (‘wcs’).

Returns

aperture : CircularAnnulus object

A CircularAnnulus object.

SkyCircularAperture

class photutils.aperture.SkyCircularAperture

Bases: photutils.SkyAperture

Circular aperture(s), defined in sky coordinates.

Parameters

positions : SkyCoord

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

r : `Quantity`

The radius of the aperture(s), either in angular or pixel units.

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>CircularAperture</code> instance in pixel coordinates.
------------------------------------	--

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a `CircularAperture` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

`aperture` : `CircularAperture` object

A `CircularAperture` object.

SkyEllipticalAnnulus

`class photutils.aperture.SkyEllipticalAnnulus`

Bases: `photutils.SkyAperture`

Elliptical annulus aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`a_in` : `Quantity`

The inner semimajor axis, either in angular or pixel units.

`a_out` : `Quantity`

The outer semimajor axis, either in angular or pixel units.

`b_out` : float

The outer semiminor axis, either in angular or pixel units. The inner semiminor axis is calculated as:

$$b_{in} = b_{out} \left(\frac{a_{in}}{a_{out}} \right)$$

theta : `Quantity`

The position angle (in angular units) of the semimajor axis. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to an <code>EllipticalAnnulus</code> instance in pixel coordinates.
------------------------------------	--

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to an `EllipticalAnnulus` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

`aperture` : `EllipticalAnnulus` object

An `EllipticalAnnulus` object.

SkyEllipticalAperture

`class photutils.aperture.SkyEllipticalAperture`

Bases: `photutils.SkyAperture`

Elliptical aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`a` : `Quantity`

The semimajor axis, either in angular or pixel units.

`b` : `Quantity`

The semiminor axis, either in angular or pixel units.

`theta` : `Quantity`

The position angle (in angular units) of the semimajor axis. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to an <code>EllipticalAperture</code> instance in pixel coordinates.
------------------------------------	---

Methods Documentation

`to_pixel(wcs, mode=u'all')`

Convert the aperture to an `EllipticalAperture` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions (‘all’; default) or only including only the core WCS transformation (‘wcs’).

Returns

`aperture` : `EllipticalAperture` object

An `EllipticalAperture` object.

SkyRectangularAnnulus

`class photutils.aperture.SkyRectangularAnnulus`

Bases: `photutils.SkyAperture`

Rectangular annulus aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`w_in` : `Quantity`

The inner full width of the aperture, either in angular or pixel units. For `theta=0` the width side is along the North-South axis.

`w_out` : `Quantity`

The outer full width of the aperture, either in angular or pixel units. For `theta=0` the width side is along the North-South axis.

`h_out` : `Quantity`

The outer full height of the aperture, either in angular or pixel units. The inner full height is calculated as:

$$h_{in} = h_{out} \left(\frac{w_{in}}{w_{out}} \right)$$

For `theta=0` the height side is along the East-West axis.

`theta` : `Quantity`

The position angle (in angular units) of the width side. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>RectangularAnnulus</code> instance in pixel coordinates.
------------------------------------	--

Methods Documentation

`to_pixel(wcs, mode=u'all')`

Convert the aperture to a `RectangularAnnulus` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

`aperture` : `RectangularAnnulus` object

A `RectangularAnnulus` object.

SkyRectangularAperture

`class photutils.aperture.SkyRectangularAperture`

Bases: `photutils.SkyAperture`

Rectangular aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`w` : `Quantity`

The full width of the aperture, either in angular or pixel units. For `theta=0` the width side is along the North-South axis.

`h` : `Quantity`

The full height of the aperture, either in angular or pixel units. For `theta=0` the height side is along the East-West axis.

`theta` : `Quantity`

The position angle (in angular units) of the width side. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>RectangularAperture</code> instance in pixel coordinates.
------------------------------------	---

Methods Documentation

`to_pixel(wcs, mode=u'all')`

Convert the aperture to a `RectangularAperture` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : { ‘all’, ‘wcs’ }, optional

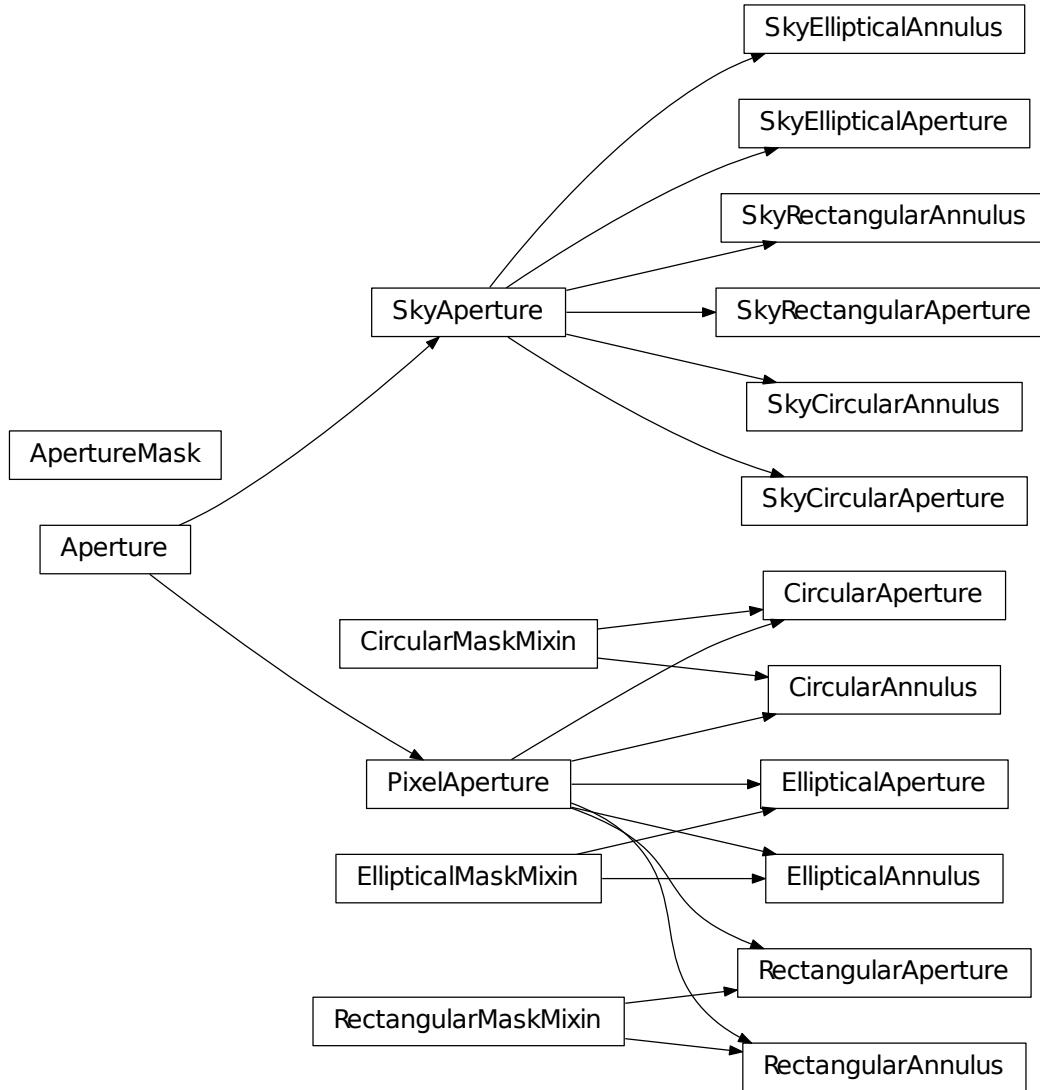
Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

`aperture` : `RectangularAperture` object

A `RectangularAperture` object.

Class Inheritance Diagram



CHAPTER 9

PSF Photometry (`photutils.psf`)

The `photutils.psf` module contains tools for model-fitting photometry, often called “PSF photometry”.

Warning: The PSF photometry API is currently considered *experimental* and may change in the future. We will aim to keep compatibility where practical, but will not finalize the API until sufficient user feedback has been accumulated.

Terminology

Different astronomy sub-fields use the terms Point Spread Function (PSF) and Point Response Function (PRF) somewhat differently, especially when colloquial usage is taken into account. For this module we assume that the PRF is an image of a point source *after discretization* e.g., onto a rectilinear CCD grid. This is the definition used by [Spitzer](#). Where relevant, we use this terminology for this sort of model, and consider “PSF” to refer to the underlying model. In many cases this distinction is unimportant, but can be critical when dealing with undersampled data.

Despite this, in colloquial usage “PSF photometry” often means the same sort of model-fitting analysis, regardless to exactly what kind of model is actually being fit. We take this road, using “PSF photometry” as shorthand for the general approach.

PSF Photometry

Photutils provides a modular set of tools to perform PSF photometry for different science cases. These are implemented as separate classes to do sub-tasks of PSF photometry. It also provides high-level classes that connect these pieces together. In particular, it contains an implementation of the DAOPHOT algorithm ([DAOPhotPSFPhotometry](#)) proposed by [Stetson in his seminal paper](#) for crowded-field stellar photometry.

The DAOPHOT algorithm consists in applying the loop FIND, GROUP, NSTAR, SUBTRACT, FIND until no more stars are detected or a given number of iterations is reached. Basically, [DAOPhotPSFPhotometry](#) works as follows. The first step is to estimate the sky background. For this task, photutils provides several classes to compute scalar and 2D backgrounds, see [background](#) for details. The next step is to find an initial estimate of the positions of potential sources. This can be accomplished by using source detection algorithms, which are implemented in [detection](#).

After finding sources one would apply a clustering algorithm in order to label the sources according to groups. Usually, those groups are formed by a distance criterion, which is the case of the grouping algorithm proposed by Stetson. In [DAOGroup](#), we provide an implementation of that algorithm. In addition, [DBSCANGroup](#) can also be used to group sources with more complex distance criteria. The reason behind the construction of groups is illustrated as follows: imagine that one would like to fit 300 stars and the model for each star has three parameters to be fitted. If one constructs a single model to fit the 300 stars simultaneously, then the optimization algorithm will have to search for the solution in a 900 dimensional space, which is computationally expensive and error-prone. Reducing the stars in groups effectively reduces the dimension of the parameter space, which facilitates the optimization process.

Provided that the groups are available, the next step is to fit the sources simultaneously for each group. This task can be done using an astropy fitter, for instance, [LevMarLSQFitter](#).

After sources are fitted, they are subtracted from the given image and, after fitting all sources, the residual image is analyzed by the finding routine again in order to check if there exist any source which has not been detected previously. This process goes on until no more sources are identified by the finding routine.

Note: It is important to note the conventions on the column names of the input/output astropy Tables which are passed along to the source detection and photometry objects. For instance, all source detection objects should output a table with columns named as `xcentroid` and `ycentroid` ([check_detection](#)). On the other hand, [DAOGroup](#) expects columns named as `x_0` and `y_0`, which represents the initial guesses on the sources' centroids. Finally, the output of the fitting process shows columns named as `x_fit`, `y_fit`, `flux_fit` for the optimum values and `x_0`, `y_0`, `flux_0` for the initial guesses. Although this convention implies that the columns have to be renamed along the process, it has the advantage of clarity so that one can keep track and easily differentiate where input/outputs came from.

High-Level Structure

Photutils provides three classes to perform PSF Photometry: [BasicPSFPhotometry](#), [IterativelySubtractedPSFPhotometry](#), and [DAOPhotPSFPhotometry](#). Together these provide the core workflow to make photometric measurements given an appropriate PSF (or other) model.

[BasicPSFPhotometry](#) implements the minimum tools for model-fitting photometry. At its core, this involves finding sources in an image, grouping overlapping sources into a single model, fitting the model to the sources, and subtracting the models from the image. In DAOPHOT parlance, this is essentially running the “FIND, GROUP, NSTAR, SUBTRACT” once. Because it is only a single cycle of that sequence, this class should be used when the degree of crowdedness of the field is not very high, for instance, when most stars are separated by a distance no less than one FWHM and their brightness are relatively uniform. It is critical to understand, though, that [BasicPSFPhotometry](#) does not actually contain the functionality to *do* all these steps - that is provided by other objects (or can be user-written) functions. Rather it provides the framework and data structures in which these operations run. Because of this, [BasicPSFPhotometry](#) is particularly useful for build more complex workflows, as all of the stages can be turned on or off or replaced with different implementations as the user desires.

[IterativelySubtractedPSFPhotometry](#) is similar to [BasicPSFPhotometry](#), but it adds a parameter called `n_iters` which is the number of iterations for which the loop “FIND, GROUP, NSTAR, SUBTRACT, FIND...” will be performed. This class enables photometry in a scenario where there exists significant overlap between stars that are of quite different brightness. For instance, the detection algorithm may not be able to detect a faint and bright star very close together in the first iteration, but they will be detected in the next iteration after the brighter stars have been fit and subtracted. Like [BasicPSFPhotometry](#), it does not include implementations of the stages of this process, but it provides the structure in which those stages run.

[DAOPhotPSFPhotometry](#) is a special case of [IterativelySubtractedPSFPhotometry](#). Unlike [IterativelySubtractedPSFPhotometry](#) and [BasicPSFPhotometry](#), the class includes specific implementations of the stages of the photometric measurements, tuned to reproduce the algorithms used for the DAOPHOT code. Specifically, the `finder`, `group_maker`, `bkg_estimator` attributes are set to the [DAOStarFinder](#), [DAOGroup](#), and [MMBackground](#), respectively. Therefore, users need to input the parameters of those classes to set up a [DAOPhotPSFPhotometry](#) object, rather than providing objects to do these stages (which is what the other classes require).

Those classes and all of the classes they *use* for the steps in the photometry process can always be replaced by user-supplied functions if you wish to customize any stage of the photometry process. This makes the machinery very flexible, while still providing a “batteries included” approach with a default implementation that’s suitable for many use cases.

Basic Usage

The basic usage of, e.g., `IterativelySubtractedPSFPhotometry` is as follows:

```
>>> # create an IterativelySubtractedPSFPhotometry object
>>> from photutils.psf import IterativelySubtractedPSFPhotometry
>>> my_photometry = IterativelySubtractedPSFPhotometry(finder=my_finder,
...                                         group_maker=my_group_maker,
...                                         bkg_estimator=my_bkg_estimator,
...                                         psf_model=my_psf_model,
...                                         fitter=my_fitter, niters=3,
...                                         fitshape=(7,7))
>>> # get photometry results
>>> photometry_results = my_photometry(image=my_image)
>>> # get residual image
>>> residual_image = my_photometry.get_residual_image()
```

Where `my_finder`, `my_group_maker`, and `my_bkg_estimator` may be any suitable class or callable function. This approach allows one to customize every part of the photometry process provided that their input/output are compatible with the input/output expected by `IterativelySubtractedPSFPhotometry`. `photutils.psf` provides all the necessary classes to reproduce the DAOPHOT algorithm, but any individual part of that algorithm can be swapped for a user-defined function. See the API documentation for precise details on what these classes or functions should look like.

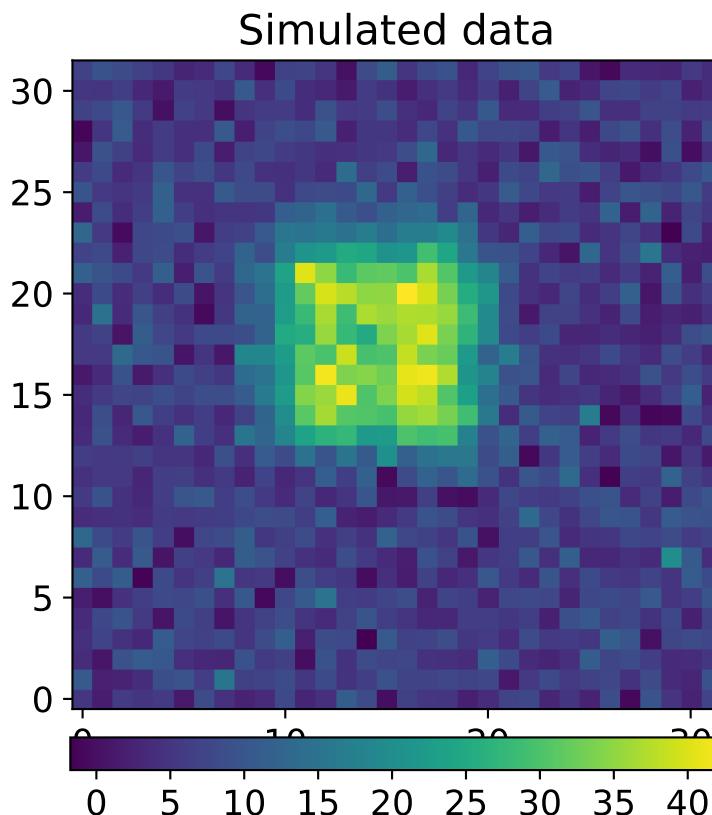
Performing PSF Photometry

Let’s take a look at a simple example with simulated stars whose PSF is assumed to be Gaussian.

First let’s create an image with four overlapping stars:

```
>>> import numpy as np
>>> from astropy.table import Table
>>> from photutils.datasets import make_random_gaussians, make_noise_image
>>> from photutils.datasets import make_gaussian_sources
>>> sigma_psf = 2.0
>>> sources = Table()
>>> sources['flux'] = [700, 800, 700, 800]
>>> sources['x_mean'] = [12, 17, 12, 17]
>>> sources['y_mean'] = [15, 15, 20, 20]
>>> sources['x_stddev'] = sigma_psf*np.ones(4)
>>> sources['y_stddev'] = sources['x_stddev']
>>> sources['theta'] = [0, 0, 0, 0]
>>> sources['id'] = [1, 2, 3, 4]
>>> tshape = (32, 32)
>>> image = (make_gaussian_sources(tshape, sources) +
...             make_noise_image(tshape, type='poisson', mean=6.,
...                               random_state=1) +
...             make_noise_image(tshape, type='gaussian', mean=0.,
...                               stddev=2., random_state=1))
```

```
>>> from matplotlib import rcParams
>>> rcParams['font.size'] = 13
>>> import matplotlib.pyplot as plt
>>> plt.imshow(image, cmap='viridis', aspect=1, interpolation='nearest',
...             origin='lower')
>>> plt.title('Simulated data')
>>> plt.colorbar(orientation='horizontal', fraction=0.046, pad=0.04)
```



Then let's import the required classes to set up a `IterativelySubtractedPSFPhotometry` object:

```
>>> from photutils.detection import IRAFStarFinder
>>> from photutils.psf import IntegratedGaussianPRF, DAOGroup
>>> from photutils.background import MMMBackground, MADStdBackgroundRMS
>>> from astropy.modeling.fitting import LevMarLSQFitter
>>> from astropy.stats import gaussian_sigma_to_fwhm
```

Let's then instantiate and use the objects:

```
>>> bkgrms = MADStdBackgroundRMS()
>>> std = bkgrms(image)
>>> iraffind = IRAFStarFinder(threshold=3.5*std,
...                             fwhm=sigma_psf*gaussian_sigma_to_fwhm,
...                             minsep_fwhm=0.01, roundhi=5.0, roundlo=-5.0,
...                             sharplow=0.0, sharphi=2.0)
>>> daogroup = DAOGroup(2.0*sigma_psf*gaussian_sigma_to_fwhm)
>>> mmm_bkg = MMMBackground()
```

```

>>> fitter = LevMarLSQFitter()
>>> psf_model = IntegratedGaussianPRF(sigma=sigma_psf)
>>> from photutils.psf import IterativelySubtractedPSFPhotometry
>>> photometry = IterativelySubtractedPSFPhotometry(finder=iraffind,
...                                         group_maker=daogroup,
...                                         bkg_estimator=mmm_bkg,
...                                         psf_model=psf_model,
...                                         fitter=LevMarLSQFitter(),
...                                         nitors=1, fitshape=(11,11))
>>> result_tab = photometry(image=image)
>>> residual_image = photometry.get_residual_image()

```

Note that the parameters values for the finder class, i.e., `IRAFStarFinder`, are completely chosen in an arbitrary manner and optimum values do vary according to the data.

As mentioned before, the way to actually do the photometry is by using `photometry` as a function-like call.

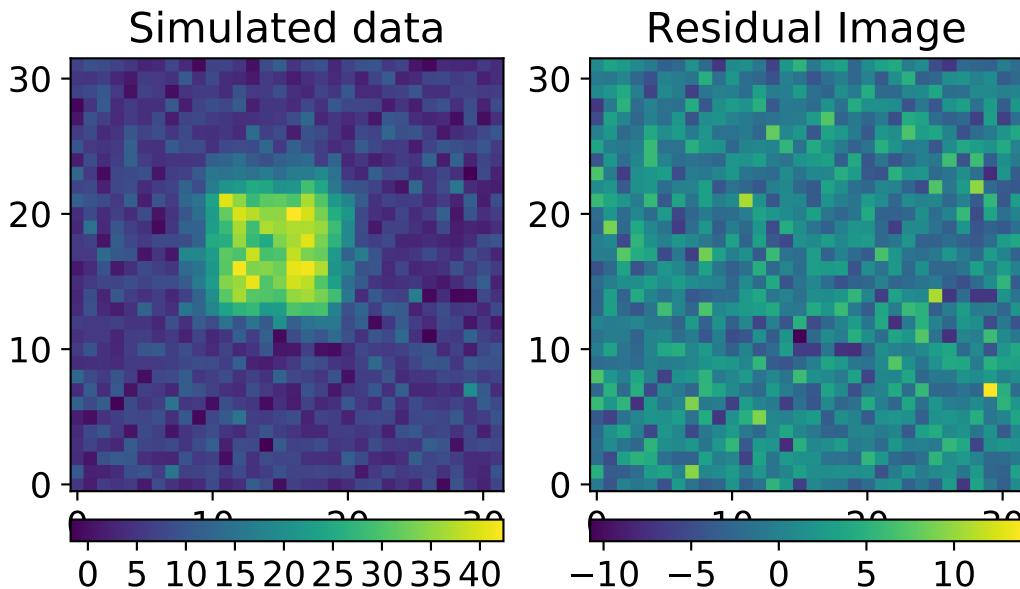
It's worth noting that `image` does not need to be background subtracted. The subtraction is done during the photometry process with the attribute `bkg` that was used to set up `photometry`.

Now, let's compare the simulated and the residual images:

```

>>> plt.subplot(1, 2, 1)
>>> plt.imshow(image, cmap='viridis', aspect=1, interpolation='nearest',
...             origin='lower')
>>> plt.title('Simulated data')
>>> plt.colorbar(orientation='horizontal', fraction=0.046, pad=0.04)
>>> plt.subplot(1, 2, 2)
>>> plt.imshow(residual_image, cmap='viridis', aspect=1,
...             interpolation='nearest', origin='lower')
>>> plt.title('Residual Image')
>>> plt.colorbar(orientation='horizontal', fraction=0.046, pad=0.04)
>>> plt.show()

```



Performing PSF Photometry with Fixed Centroids

In case that the centroids positions of the stars are known a priori, then they can be held fixed during the fitting process and the optimizer will only consider flux as a variable. To do that, one has to set the `fixed` attribute for the centroid parameters in `psf` as `True`.

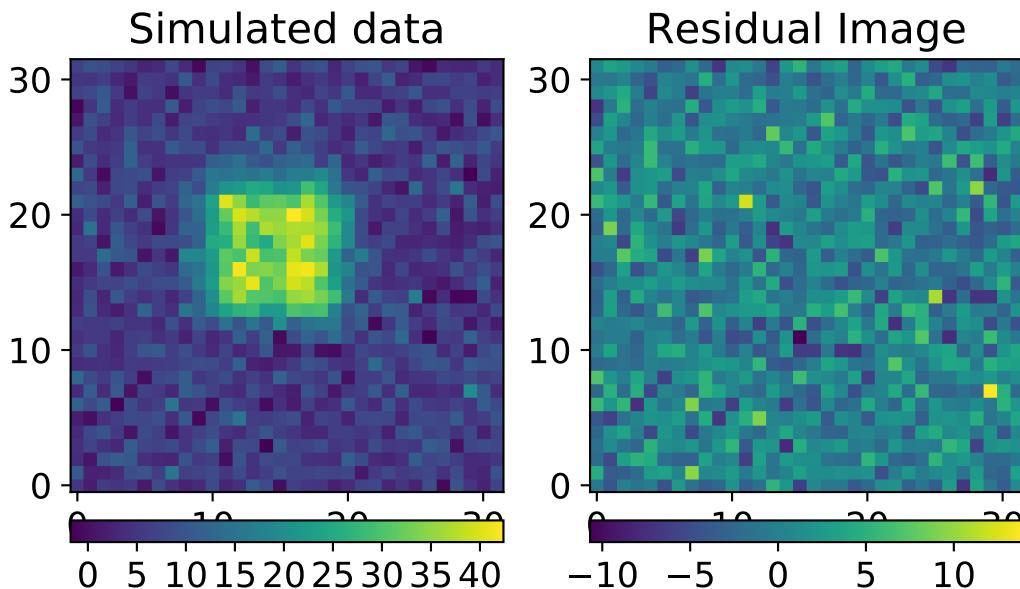
Consider the previous example after the line `psf_model = IntegratedGaussianPRF(sigma=sigma_psf)`:

```
>>> psf_model.x_0.fixed = True
>>> psf_model.y_0.fixed = True
>>> pos = Table(names=['x_0', 'y_0'], data=[sources['x_mean'],
...                                         sources['y_mean']])
```

```
>>> photometry = BasicPSFPhotometry(group_maker=daogroup,
...                                     bkg_estimator=mmm_bkg,
...                                     psf_model=psf_model,
...                                     fitter=LevMarLSQFitter(),
...                                     fitshape=(11,11))
>>> result_tab = photometry(image=image, positions=pos)
>>> residual_image = photometry.get_residual_image()
```

```
>>> plt.subplot(1, 2, 1)
>>> plt.imshow(image, cmap='viridis', aspect=1,
...             interpolation='nearest', origin='lower')
```

```
>>> plt.title('Simulated data')
>>> plt.colorbar(orientation='horizontal', fraction=0.046, pad=0.04)
>>> plt.subplot(1, 2, 2)
>>> plt.imshow(residual_image, cmap='viridis', aspect=1,
...             interpolation='nearest', origin='lower')
>>> plt.title('Residual Image')
>>> plt.colorbar(orientation='horizontal', fraction=0.046, pad=0.04)
```



For more examples, also check the online notebook in the next section.

Example Notebooks (online)

- PSF photometry on artificial Gaussian stars in crowded fields
- PSF photometry on artificial Gaussian stars
- PSF/PRF Photometry on Spitzer Data

References

Spitzer PSF vs. PRF

Kepler PSF calibration

The Kepler Pixel Response Function

Stetson, Astronomical Society of the Pacific, Publications, (ISSN 0004-6280), vol. 99, March 1987, p. 191-222.

Reference/API

This subpackage contains modules and packages for point spread function photometry.

Functions

<code>create_matching_kernel(source_psf, target_psf)</code>	Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.
<code>get_grouped_psf_model(template_psf_model, ...)</code>	Construct a joint PSF model which consists of a sum of PSF's templated on a specific model, but whose parameters are given by a table of objects.
<code>prepare_psf_model(psfmodel[, xname, yname, ...])</code>	Convert a 2D PSF model to one suitable for use with <code>BasicPSFPhotometry</code> or its subclasses.
<code>resize_psf(psf, input_pixel_scale, ...[, order])</code>	Resize a PSF using spline interpolation of the requested order.
<code>subtract_psf(data, psf, posflux[, subshape])</code>	Subtract PSF/PRFs from an image.

`create_matching_kernel`

`photutils.psf.create_matching_kernel(source_psf, target_psf, window=None)`

Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.

Parameters

`source_psf` : 2D `ndarray`

The source PSF. The source PSF should have higher resolution (i.e. narrower) than the target PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

`target_psf` : 2D `ndarray`

The target PSF. The target PSF should have lower resolution (i.e. broader) than the source PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

`window` : callable, optional

The window (or taper) function or callable class instance used to remove high frequency noise from the PSF matching kernel. Some examples include:

- [HanningWindow](#)
- [TukeyWindow](#)
- [CosineBellWindow](#)
- [SplitCosineBellWindow](#)
- [TopHatWindow](#)

For more information on window functions and example usage, see [PSF Matching \(photutils.psf.matching\)](#).

Returns

`kernel` : 2D `ndarray`

The matching kernel to go from source_psf to target_psf. The output matching kernel is normalized such that it sums to 1.

get_grouped_psf_model

`photutils.psf.get_grouped_psf_model(template_psf_model, star_group)`

Construct a joint PSF model which consists of a sum of PSF's templated on a specific model, but whose parameters are given by a table of objects.

Parameters

template_psf_model : `astropy.modeling.Fittable2DModel` instance

The model to use for *individual* objects. Must have parameters named `x_0`, `y_0`, and `flux`.

star_group : `Table`

Table of stars for which the compound PSF will be constructed. It must have columns named `x_0`, `y_0`, and `flux_0`.

Returns

`group_psf`

An `astropy.modeling.CompoundModel` instance which is a sum of the given PSF models.

prepare_psf_model

`photutils.psf.prepare_psf_model(psfmodel, xname=None, yname=None, fluxname=None, renormalize_psf=True)`

Convert a 2D PSF model to one suitable for use with `BasicPSFPhotometry` or its subclasses.

The resulting model may be a composite model, but should have only the x, y, and flux related parameters un-fixed.

Parameters

psfmodel : a 2D model

The model to assume as representative of the PSF.

xname : str or None

The name of the `psfmodel` parameter that corresponds to the x-axis center of the PSF. If None, the model will be assumed to be centered at `x=0`, and a new parameter will be added for the offset.

yname : str or None

The name of the `psfmodel` parameter that corresponds to the y-axis center of the PSF. If None, the model will be assumed to be centered at `x=0`, and a new parameter will be added for the offset.

fluxname : str or None

The name of the `psfmodel` parameter that corresponds to the total flux of the star. If None, a scaling factor will be added to the model.

renormalize_psf : bool

If True, the model will be integrated from -inf to inf and re-scaled so that the total integrates to 1. Note that this renormalization only occurs *once*, so if the total flux of `psfmodel` depends on position, this will *not* be correct.

Returns

`outmod` : a model

A new model ready to be passed into `BasicPSFPhotometry` or its subclasses.

resize_psf

`photutils.psf.resize_psf(psf, input_pixel_scale, output_pixel_scale, order=3)`

Resize a PSF using spline interpolation of the requested order.

Parameters

`psf` : 2D `ndarray`

The 2D data array of the PSF.

`input_pixel_scale` : float

The pixel scale of the input psf. The units must match `output_pixel_scale`.

`output_pixel_scale` : float

The pixel scale of the output psf. The units must match `input_pixel_scale`.

`order` : float, optional

The order of the spline interpolation (0-5). The default is 3.

Returns

`result` : 2D `ndarray`

The resampled/interpolated 2D data array.

subtract_psf

`photutils.psf.subtract_psf(data, psf, posflux, subshape=None)`

Subtract PSF/PRFs from an image.

Parameters

`data` : `NDData` or array (must be 2D)

Image data.

`psf` : `astropy.modeling.Fittable2DModel` instance

PSF/PRF model to be subtracted from the data.

`posflux` : Array-like of shape (3, N) or `Table`

Positions and fluxes for the objects to subtract. If an array, it is interpreted as (x, y, flux) If a table, the columns ‘x_fit’, ‘y_fit’, and ‘flux_fit’ must be present.

`subshape` : length-2 or None

The shape of the region around the center of the location to subtract the PSF from. If None, subtract from the whole image.

Returns

`subdata` : same shape and type as `data`

The image with the PSF subtracted

Classes

<code>BasicPSFPhotometry(group_maker, ...[, ...])</code>	This class implements a PSF photometry algorithm that can find sources in an image, group overlapping sources into a single model, fit the model to the sources, and subtracting the models from the image.
<code>CosineBellWindow(alpha)</code>	Class to define a 2D cosine bell window function.
<code>DAOGroup(crit_separation)</code>	This is class implements the DAOGROUP algorithm presented by Stetson (1987).
<code>DAOPhotPSFPhotometry(crit_separation, ...[, ...])</code>	This class implements an iterative algorithm based on the DAOPHOT algorithm presented by Stetson (1987) to perform point spread function photometry in crowded fields.
<code>DBSCANGroup(crit_separation[, min_samples, ...])</code>	Class to create star groups according to a distance criteria using the Density-based Spatial Clustering of Applications with Noise (DBSCAN) from scikit-learn.
<code>FittableImageModel</code>	A fittable 2D model of an image allowing for image intensity scaling and image translations.
<code>GroupStarsBase</code>	This base class provides the basic interface for subclasses that are capable of classifying stars in groups.
<code>HanningWindow()</code>	Class to define a 2D Hanning (or Hann) window function.
<code>IntegratedGaussianPRF</code>	Circular Gaussian model integrated over pixels.
<code>IterativelySubtractedPSFPhotometry(...[, ...])</code>	This class implements an iterative algorithm to perform point spread function photometry in crowded fields.
<code>NonNormalizable</code>	Used to indicate that a <code>FittableImageModel</code> model is non-normalizable.
<code>PRFAdapter</code>	A model that adapts a supplied PSF model to act as a PRF.
<code>SplitCosineBellWindow(alpha, beta)</code>	Class to define a 2D split cosine bell taper function.
<code>TopHatWindow(beta)</code>	Class to define a 2D top hat window function.
<code>TukeyWindow(alpha)</code>	Class to define a 2D Tukey window function.

BasicPSFPhotometry

```
class photutils.psf.BasicPSFPhotometry(group_maker, bkg_estimator, psf_model, fitshape, finder=None,
                                         fitter=<astropy.modeling.fitting.LevMarLSQFitter object>,
                                         aperture_radius=None)
```

Bases: `object`

This class implements a PSF photometry algorithm that can find sources in an image, group overlapping sources into a single model, fit the model to the sources, and subtracting the models from the image. This is roughly equivalent to the DAOPHOT routines FIND, GROUP, NSTAR, and SUBTRACT. This implementation allows a flexible and customizable interface to perform photometry. For instance, one is able to use different implementations for grouping and finding sources by using `group_maker` and `finder` respectively. In addition, sky background estimation is performed by `bkg_estimator`.

Parameters

`group_maker` : callable or `GroupStarsBase`

`group_maker` should be able to decide whether a given star overlaps with any other and label them as belonging to the same group. `group_maker` receives as input an `Table` object with columns named as `id`, `x_0`, `y_0`, in which `x_0` and `y_0` have the same meaning of `xcentroid` and `ycentroid`. This callable must return an `Table` with columns `id`, `x_0`, `y_0`, and `group_id`. The column `group_id` should contain integers starting from 1 that indicate which group a given source belongs to. See, e.g., `DAOGroup`.

bkg_estimator : callable, instance of any `BackgroundBase` subclass, or None

`bkg_estimator` should be able to compute either a scalar background or a 2D background of a given 2D image. See, e.g., `MedianBackground`. If None, no background subtraction is performed.

psf_model : `astropy.modeling.Fittable2DModel` instance

PSF or PRF model to fit the data. Could be one of the models in this package like `DiscretePRF`, `IntegratedGaussianPRF`, or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as `x_0`, `y_0` and `flux`. `prepare_psf_model` can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of `fitshape` must be an odd number.

finder : callable or instance of any `StarFinderBase` subclasses or None

`finder` should be able to identify stars, i.e. compute a rough estimate of the centroids, in a given 2D image. `finder` receives as input a 2D image and returns an `Table` object which contains columns with names: `id`, `xcentroid`, `ycentroid`, and `flux`. In which `id` is an integer-valued column starting from 1, `xcentroid` and `ycentroid` are center position estimates of the sources and `flux` contains flux estimates of the sources. See, e.g., `DAOStarFinder`. If `finder` is None, initial guesses for positions of objects must be provided.

fitter : `Fitter` instance

`Fitter` object used to compute the optimized centroid positions and/or flux of the identified sources. See `fitting` for more details on fitters.

aperture_radius : float or None

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `'psf_model'`.

Notes

Note that an ambiguity arises whenever `finder` and `positions` (keyword argument for `do_photometry`)` are both not ``None. In this case, `finder` is ignored and initial guesses are taken from `positions`. In addition, an warning is raised to remind the user about this behavior.

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a `star_groups` table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, *Astronomical Society of the Pacific, Publications*,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP...99..191S>

Attributes Summary

aperture_radius
fitshape

Methods Summary

<code>__call__(image[, positions])</code>	Performs PSF photometry.
<code>do_photometry(image[, positions])</code>	Perform PSF photometry in <code>image</code> .
<code>get_residual_image()</code>	Returns an image that is the result of the subtraction between the original image and the fitted sources.
<code>nstar(image, star_groups)</code>	Fit, as appropriate, a compound or single model to the given <code>star_groups</code> .

Attributes Documentation

`aperture_radius`

`fitshape`

Methods Documentation

`__call__(image, positions=None)`

Performs PSF photometry. See `do_photometry` for more details including the `__call__` signature.

`do_photometry(image, positions=None)`

Perform PSF photometry in `image`.

This method assumes that `psf_model` has centroids and flux parameters which will be fitted to the data provided in `image`. A compound model, in fact a sum of `psf_model`, will be fitted to groups of stars automatically identified by `group_maker`. Also, `image` is not assumed to be background subtracted. If `positions` are not `None` then this method uses `positions` as initial guesses for the centroids. If the centroid positions are set as fixed in the PSF model `psf_model`, then the optimizer will only consider the flux as a variable.

Parameters

`image` : 2D array-like, `ImageHDU`, `HDUList`

Image to perform photometry.

`positions`: ‘~astropy.table.Table‘

Positions (in pixel coordinates) at which to *start* the fit for each object. Columns ‘x_0’ and ‘y_0’ must be present. ‘flux_0’ can also be provided to set initial fluxes. If ‘flux_0’ is not provided, aperture photometry is used to estimate initial values for the fluxes.

Returns

`output_tab` : `Table` or `None`

Table with the photometry results, i.e., centroids and fluxes estimations and the initial estimates used to start the fitting process. `None` is returned if no sources are found in `image`.

get_residual_image()

Returns an image that is the result of the subtraction between the original image and the fitted sources.

Returns

residual_image : 2D array-like, [ImageHDU](#), [HDUList](#)

nstar(image, star_groups)

Fit, as appropriate, a compound or single model to the given `star_groups`. Groups are fitted sequentially from the smallest to the biggest. In each iteration, `image` is subtracted by the previous fitted group.

Parameters

image : numpy.ndarray

Background-subtracted image.

star_groups : [Table](#)

This table must contain the following columns: `id`, `group_id`, `x_0`, `y_0`, `flux_0`. `x_0` and `y_0` are initial estimates of the centroids and `flux_0` is an initial estimate of the flux.

Returns

result_tab : [Table](#)

Astropy table that contains photometry results.

image : numpy.ndarray

Residual image.

CosineBellWindow

class photutils.psf.CosineBellWindow(alpha)

Bases: [photutils.psf.matching.SplitCosineBellWindow](#)

Class to define a 2D cosine bell window function.

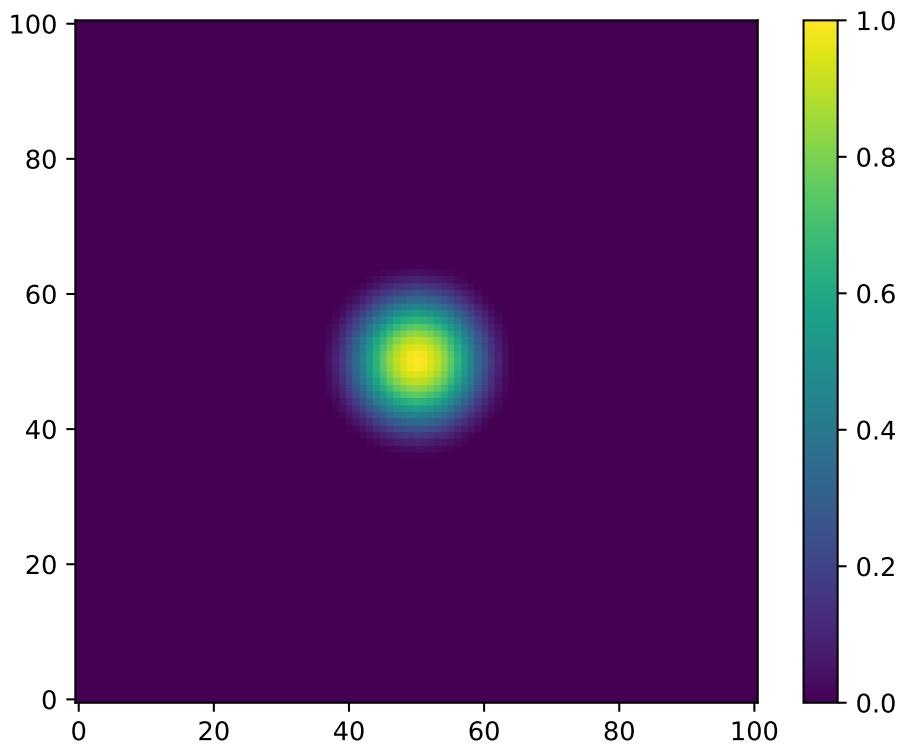
Parameters

alpha : float, optional

The percentage of array values that are tapered.

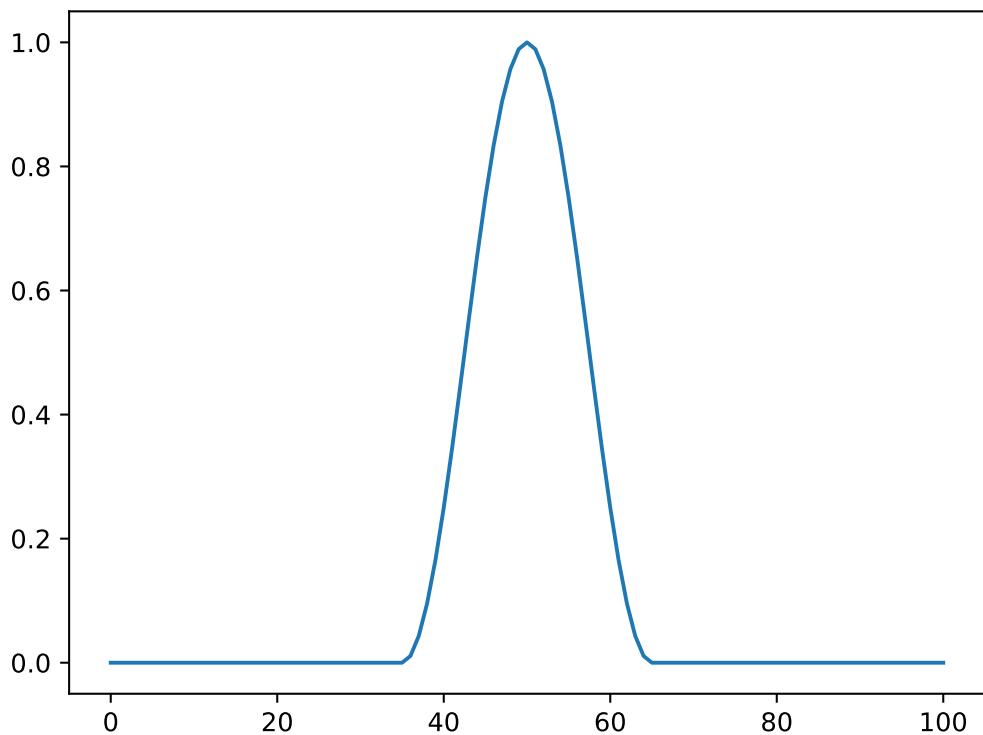
Examples

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



DAOGroup

class photutils.psf.**DAOGroup**(*crit_separation*)

Bases: photutils.psf.GroupStarsBase

This is class implements the DAOGROUP algorithm presented by Stetson (1987).

The method `group_stars` divides an entire starlist into sets of distinct, self-contained groups of mutually overlapping stars. It accepts as input a list of stars and determines which stars are close enough to be capable of adversely influencing each others' profile fits.

Parameters

`crit_separation` : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

See also:

[photutils.DAOStarFinder](#)

Notes

Assuming the psf fwhm to be known, `crit_separation` may be set to $k * \text{fwhm}$, for some positive real k .

References

- [1] Stetson, Astronomical Society of the Pacific, Publications,
(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP..99..191S>

Attributes Summary

`crit_separation`

Methods Summary

<code>find_group(star, starlist)</code>	Find the ids of those stars in <code>starlist</code> which are at a distance less than <code>crit_separation</code> from <code>star</code> .
<code>group_stars(starlist)</code>	Classify stars into groups.

Attributes Documentation

`crit_separation`

Methods Documentation

`find_group(star, starlist)`

Find the ids of those stars in `starlist` which are at a distance less than `crit_separation` from `star`.

Parameters

`star` : `Row`

Star which will be either the head of a cluster or an isolated one.

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

Array containing the ids of those stars which are at a distance less than `crit_separation` from `star`.

`group_stars(starlist)`

Classify stars into groups.

Parameters

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : `Table`

starlist with an additional column named group_id whose unique values represent groups of mutually overlapping stars.

DAOPhotPSFPhotometry

```
class photutils.psf.DAOPhotPSFPhotometry(crit_separation, threshold, fwhm, psf_model, fitshape,
                                         sigma=3.0, ratio=1.0, theta=0.0, sigma_radius=1.5,
                                         sharplo=0.2, sharphi=1.0, roundlo=-1.0, roundhi=1.0,
                                         fitter=<astropy.modeling.fitting.LevMarLSQFitter object>,
                                         niters=3, aperture_radius=None)
```

Bases: photutils.psf.IterativelySubtractedPSFPhotometry

This class implements an iterative algorithm based on the DAOPHOT algorithm presented by Stetson (1987) to perform point spread function photometry in crowded fields. This consists of applying a loop of find sources, make groups, fit groups, subtract groups, and then repeat until no more stars are detected or a given number of iterations is reached.

Basically, this classes uses IterativelySubtractedPSFPhotometry, but with grouping, finding, and background estimation routines defined a priori. More precisely, this class uses DAOGroup for grouping, DAOStarFinder for finding sources, and MMMBackground for background estimation. Those classes are based on GROUP, FIND, and SKY routines used in DAOPHOT, respectively.

The parameter crit_separation is associated with DAOGroup. sigma_clip is associated with MMMBackground. threshold and fwhm are associated with DAOStarFinder. Parameters from ratio to roundhi are also associated with DAOStarFinder.

Parameters

crit_separation : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

threshold : float

The absolute image value above which to select sources.

fwhm : float

The full-width half-maximum (FWHM) of the major axis of the Gaussian kernel in units of pixels.

psf_model : astropy.modeling.Fittable2DModel instance

PSF or PRF model to fit the data. Could be one of the models in this package like DiscretePRF, IntegratedGaussianPRF, or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as x_0, y_0 and flux. prepare_psf_model can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of fitshape must be an odd number.

sigma : float, optional

Number of standard deviations used to perform sigma clip with a SigmaClip object.

ratio : float, optional

The ratio of the minor to major axis standard deviations of the Gaussian kernel. **ratio** must be strictly positive and less than or equal to 1.0. The default is 1.0 (i.e., a circular Gaussian kernel).

theta : float, optional

The position angle (in degrees) of the major axis of the Gaussian kernel measured counter-clockwise from the positive x axis.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [1
 $\text{sigma} = \text{FWHM} / (2.0 * \sqrt{2.0 * \log(2.0)}))$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

fitter : `Fitter` instance

Fitter object used to compute the optimized centroid positions and/or flux of the identified sources. See `fitting` for more details on fitters.

niters : int or None

Number of iterations to perform of the loop FIND, GROUP, SUBTRACT, NSTAR. If None, iterations will proceed until no more stars remain. Note that in this case it is *possible* that the loop will never end if the PSF has structure that causes subtraction to create new sources infinitely.

aperture_radius : float

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `'psf_model'`.

Notes

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a `star_groups` table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, Astronomical Society of the Pacific, Publications,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP...99..191S>

DBSCANGroup

```
class photutils.psf.DBSCANGroup(crit_separation, min_samples=1, metric='euclidean', algorithm='auto',
                                 leaf_size=30)
```

Bases: `photutils.psf.GroupStarsBase`

Class to create star groups according to a distance criteria using the Density-based Spatial Clustering of Applications with Noise (DBSCAN) from scikit-learn.

Parameters

crit_separation : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

min_samples : int, optional (default=1)

Minimum number of stars necessary to form a group.

metric : string or callable (default='euclidean')

The metric to use when calculating distance between each pair of stars.

algorithm : { 'auto', 'ball_tree', 'kd_tree', 'brute' }, optional

The algorithm to be used to actually find nearest neighbors.

leaf_size : int, optional (default = 30)

Leaf size passed to BallTree or cKDTree.

Notes

- The attribute `crit_separation` corresponds to `eps` in `sklearn.cluster.DBSCAN`.
- This class provides more general algorithms than `photutils.psf.DAOGroup`. More precisely, `photutils.psf.DAOGroup` is a special case of `photutils.psf.DBSCANGroup` when `min_samples=1` and `metric=euclidean`. Additionally, `photutils.psf.DBSCANGroup` may be faster than `photutils.psf.DAOGroup`.

References

[1] Scikit Learn DBSCAN.

<http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN>

Methods Summary

<code>group_stars(starlist)</code>	Classify stars into groups.
------------------------------------	-----------------------------

Methods Documentation

group_stars(starlist)
Classify stars into groups.

Parameters

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : [Table](#)

`starlist` with an additional column named `group_id` whose unique values represent groups of mutually overlapping stars.

FittableImageModel

`class photutils.psf.FittableImageModel`

Bases: [astropy.modeling.Fittable2DModel](#)

A fittable 2D model of an image allowing for image intensity scaling and image translations.

This class takes 2D image data and computes the values of the model at arbitrary locations (including at intra-pixel, fractional positions) within this image using spline interpolation provided by [RectBivariateSpline](#).

The fittable model provided by this class has three model parameters: an image intensity scaling factor (`flux`) which is applied to (normalized) image, and two positional parameters (`x_0` and `y_0`) indicating the location of a feature in the coordinate grid on which the model is to be evaluated.

If this class is initialized with `flux` (intensity scaling factor) set to `None`, then `flux` is going to be estimated as `sum(data)`.

Parameters

`data` : [numpy.ndarray](#)

Array containing 2D image.

`origin` : tuple, `None`, optional

A reference point in the input image data array. When `origin` is `None`, `origin` will be set at the middle of the image array.

If `origin` represents the location of a feature (e.g., the position of an intensity peak) in the input data, then model parameters `x_0` and `y_0` show the location of this peak in an another target image to which this model was fitted. Fundamentally, it is the coordinate in the model's image data that should map to coordinate (`x_0`, `y_0`) of the output coordinate system on which the model is evaluated.

Alternatively, when `origin` is set to `(0, 0)`, then model parameters `x_0` and `y_0` are shifts by which model's image should be translated in order to match a target image.

`normalize` : bool, optional

Indicates whether or not the model should be build on normalized input image data. If true, then the normalization constant (`N`) is computed so that

$$N \cdot C \cdot \sum_{i,j} D_{i,j} = 1,$$

where `N` is the normalization constant, `C` is correction factor given by the parameter `normalization_correction`, and `Di,j` are the elements of the input image data array.

`normalization_correction` : float, optional

A strictly positive number that represents correction that needs to be applied to model's data normalization (see `C` in the equation in the comments to `normalize` for more details).

A possible application for this parameter is to account for aperture correction. Assuming model's data represent a PSF to be fitted to some target star, we set `normalization_correction` to the aperture correction that needs to be applied to the model. That is, `normalization_correction` in this case should be set to the ratio between the total flux of the PSF (including flux outside model's data) to the flux of model's data. Then, best fitted value of the `flux` model parameter will represent an aperture-corrected flux of the target star.

`fill_value` : float, optional

The value to be returned by the `evaluate` or `astropy.modeling.Model.__call__` methods when evaluation is performed outside the definition domain of the model.

`ikwargs` : dict, optional

Additional optional keyword arguments to be passed directly to the `compute_interpolator` method. See `compute_interpolator` for more details.

Attributes Summary

<code>data</code>	Get original image data.
<code>fill_value</code>	Fill value to be returned for coordinates outside of the domain of definition of the interpolator.
<code>flux</code>	Intensity scaling factor for image data.
<code>interpolator_kwarg</code> s	Get current interpolator's arguments used when interpolator was created.
<code>normalization_constant</code>	Get normalization constant.
<code>normalization_correction</code>	Set/Get flux correction factor.
<code>normalization_status</code>	Get normalization status.
<code>normalized_data</code>	Get normalized and/or intensity-corrected image data.
<code>nx</code>	Number of columns in the data array.
<code>ny</code>	Number of rows in the data array.
<code>origin</code>	A tuple of x and y coordinates of the origin of the coordinate system in terms of pixels of model's image.
<code>oversampling</code>	The factor by which the stored image is oversampled.
<code>param_names</code>	
<code>shape</code>	A tuple of dimensions of the data array in numpy style (ny, nx).
<code>x_0</code>	X-position of a feature in the image in the output coordinate grid on which the model is evaluated.
<code>x_origin</code>	X-coordinate of the origin of the coordinate system.
<code>y_0</code>	Y-position of a feature in the image in the output coordinate grid on which the model is evaluated.
<code>y_origin</code>	Y-coordinate of the origin of the coordinate system.

Methods Summary

<code>compute_interpolator([ikwargs])</code>	Compute/define the interpolating spline.
<code>evaluate(x, y, flux, x_0, y_0)</code>	Evaluate the model on some input variables and provided model parameters.

Attributes Documentation

`data`

Get original image data.

`fill_value`

Fill value to be returned for coordinates outside of the domain of definition of the interpolator. If `fill_value` is `None`, then values outside of the domain of definition are the ones returned by the interpolator.

`flux`

Intensity scaling factor for image data.

`interpolator_kwargs`

Get current interpolator's arguments used when interpolator was created.

`normalization_constant`

Get normalization constant.

`normalization_correction`

Set/Get flux correction factor.

Note: When setting correction factor, model's flux will be adjusted accordingly such that if this model was a good fit to some target image before, then it will remain a good fit after correction factor change.

`normalization_status`

Get normalization status. Possible status values are:

- 0: **Performed**. Model has been successfully normalized at user's request.
- 1: **Failed**. Attempt to normalize has failed.
- 2: **NotRequested**. User did not request model to be normalized.

`normalized_data`

Get normalized and/or intensity-corrected image data.

`nx`

Number of columns in the data array.

`ny`

Number of rows in the data array.

`origin`

A tuple of x and y coordinates of the origin of the coordinate system in terms of pixels of model's image.

When setting the coordinate system origin, a tuple of two `int` or `float` may be used. If `origin` is set to `None`, the origin of the coordinate system will be set to the middle of the data array ($(\text{npix}-1)/2.0$).

Warning: Modifying `origin` will not adjust (modify) model's parameters `x_0` and `y_0`.

`oversampling`

The factor by which the stored image is oversampled. I.e., an input to this model is multiplied by this factor to yield the index into the stored image.

`param_names = ('flux', 'x_0', 'y_0')`

`shape`

A tuple of dimensions of the data array in numpy style (ny, nx).

`x_0`

X-position of a feature in the image in the output coordinate grid on which the model is evaluated.

x_origin

X-coordinate of the origin of the coordinate system.

y_0

Y-position of a feature in the image in the output coordinate grid on which the model is evaluated.

y_origin

Y-coordinate of the origin of the coordinate system.

Methods Documentation

compute_interpolator(ikwargs=())

Compute/define the interpolating spline. This function can be overriden in a subclass to define custom interpolators.

Parameters

ikwargs : dict, optional

Additional optional keyword arguments. Possible values are:

•degree

[int, tuple, optional] Degree of the interpolating spline. A tuple can be used to provide different degrees for the X- and Y-axes. Default value is degree=3.

•s

[float, optional] Non-negative smoothing factor. Default value s=0 corresponds to interpolation. See `RectBivariateSpline` for more details.

Notes

- When subclassing `FittableImageModel` for the purpose of overriding `compute_interpolator()`, the `evaluate()` may need to be overriden as well depending on the behavior of the new interpolator. In addition, for improved future compatibility, make sure that the overriding method stores keyword arguments `ikwargs` by calling `_store_interpolator_kwarg` method.
- Use caution when modifying interpolator's degree or smoothness in a computationally intensive part of the code as it may decrease code performance due to the need to recompute interpolator.

evaluate(x, y, flux, x_0, y_0)

Evaluate the model on some input variables and provided model parameters.

GroupStarsBase

class photutils.psf.GroupStarsBase

Bases: `object`

This base class provides the basic interface for subclasses that are capable of classifying stars in groups.

Methods Summary

`__call__(starlist)`

Classify stars into groups.

Methods Documentation

`__call__(starlist)`

Classify stars into groups.

Parameters

`starlist` : [Table](#)

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : [Table](#)

`starlist` with an additional column named `group_id` whose unique values represent groups of mutually overlapping stars.

HanningWindow

`class photutils.psf.HanningWindow`

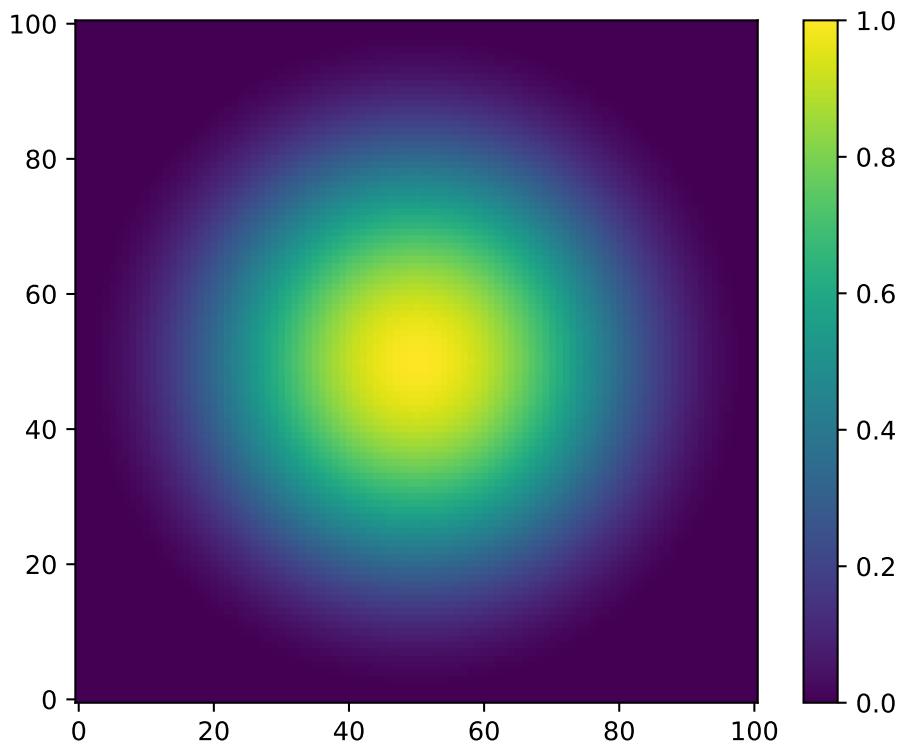
Bases: [photutils.psf.matching.SplitCosineBellWindow](#)

Class to define a 2D Hanning (or Hann) window function.

The Hann window is a taper formed by using a raised cosine with ends that touch zero.

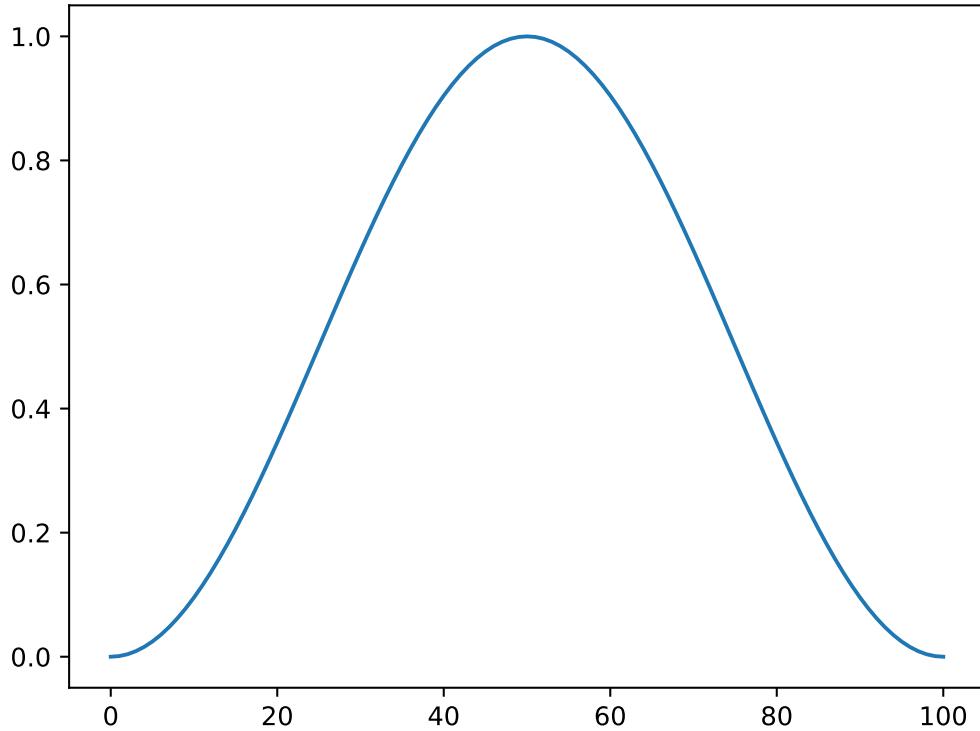
Examples

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.plot(data[50, :])
```



IntegratedGaussianPRF

```
class photutils.psf.IntegratedGaussianPRF
Bases: astropy.modeling.Fittable2DModel
```

Circular Gaussian model integrated over pixels. Because it is integrated, this model is considered a PRF, *not* a PSF (see [Terminology](#) for more about the terminology used here.)

This model is a Gaussian *integrated* over an area of 1 (in units of the model input coordinates, e.g. 1 pixel). This is in contrast to the apparently similar [astropy.modeling.functional_models.Gaussian2D](#), which is the value of a 2D Gaussian *at* the input coordinates, with no integration. So this model is equivalent to assuming the PSF is Gaussian at a *sub-pixel* level.

Parameters

sigma : float

Width of the Gaussian PSF.

flux : float (default 1)

Total integrated flux over the entire PSF

x_0 : float (default 0)

Position of the peak in x direction.

y_0 : float (default 0)

Position of the peak in y direction.

Notes

This model is evaluated according to the following formula:

$$f(x, y) = \frac{F}{4} \left[\operatorname{erf} \left(\frac{x - x_0 + 0.5}{\sqrt{2}\sigma} \right) - \operatorname{erf} \left(\frac{x - x_0 - 0.5}{\sqrt{2}\sigma} \right) \right] \left[\operatorname{erf} \left(\frac{y - y_0 + 0.5}{\sqrt{2}\sigma} \right) - \operatorname{erf} \left(\frac{y - y_0 - 0.5}{\sqrt{2}\sigma} \right) \right]$$

where erf denotes the error function and F the total integrated flux.

Attributes Summary

```
fit_deriv
flux
param_names
sigma
x_0
y_0
```

Methods Summary

`evaluate(x, y, flux, x_0, y_0, sigma)` Model function Gaussian PSF model.

Attributes Documentation

fit_deriv = None

flux

```
param_names = ('flux', 'x_0', 'y_0', 'sigma')
```

sigma

x_0

y_0

Methods Documentation

evaluate($x, y, flux, x_0, y_0, sigma$)
Model function Gaussian PSF model.

IterativelySubtractedPSFPhotometry

```
class photutils.psf.IterativelySubtractedPSFPhotometry(group_maker, bkg_estimator,
                                                       psf_model, fitshape, finder, fitter=<astropy.modeling.fitting.LevMarLSQFitter
                                                       object>, niters=3, aperture_radius=None)
```

Bases: photutils.psf.BasicPSFPhotometry

This class implements an iterative algorithm to perform point spread function photometry in crowded fields. This consists of applying a loop of find sources, make groups, fit groups, subtract groups, and then repeat until no more stars are detected or a given number of iterations is reached.

Parameters

group_maker : callable or `GroupStarsBase`

`group_maker` should be able to decide whether a given star overlaps with any other and label them as belonging to the same group. `group_maker` receives as input an `Table` object with columns named as `id`, `x_0`, `y_0`, in which `x_0` and `y_0` have the same meaning of `xcentroid` and `ycentroid`. This callable must return an `Table` with columns `id`, `x_0`, `y_0`, and `group_id`. The column `group_id` should contain integers starting from 1 that indicate which group a given source belongs to. See, e.g., `DAOGroup`.

bkg_estimator : callable, instance of any `BackgroundBase` subclass, or None

`bkg_estimator` should be able to compute either a scalar background or a 2D background of a given 2D image. See, e.g., `MedianBackground`. If None, no background subtraction is performed.

psf_model : `astropy.modeling.Fittable2DModel` instance

PSF or PRF model to fit the data. Could be one of the models in this package like `DiscretePRF`, `IntegratedGaussianPRF`, or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as `x_0`, `y_0` and `flux`. `prepare_psf_model` can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of `fitshape` must be an odd number.

finder : callable or instance of any `StarFinderBase` subclasses

`finder` should be able to identify stars, i.e. compute a rough estimate of the centroids, in a given 2D image. `finder` receives as input a 2D image and returns an `Table` object which contains columns with names: `id`, `xcentroid`, `ycentroid`, and `flux`. In which `id` is an integer-valued column starting from 1, `xcentroid` and `ycentroid` are center position estimates of the sources and `flux` contains flux estimates of the sources. See, e.g., `DAOStarFinder` or `IRAFStarFinder`.

fitter : `Fitter` instance

Fitter object used to compute the optimized centroid positions and/or flux of the identified sources. See `fitting` for more details on fitters.

aperture_radius : float

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `'psf_model'`.

niters : int or None

Number of iterations to perform of the loop FIND, GROUP, SUBTRACT, NSTAR. If None, iterations will proceed until no more stars remain. Note that in this case it is *possible* that the loop will never end if the PSF has structure that causes subtraction to create new sources infinitely.

Notes

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a star_groups table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, **Astronomical Society of the Pacific, Publications**,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP..99..191S>

Attributes Summary

finder

niters

Methods Summary

`do_photometry(image[, positions])`

Perform PSF photometry in `image`.

Attributes Documentation

finder

niters

Methods Documentation

`do_photometry(image, positions=None)`

Perform PSF photometry in `image`.

This method assumes that `psf_model` has centroids and flux parameters which will be fitted to the data provided in `image`. A compound model, in fact a sum of `psf_model`, will be fitted to groups of stars automatically identified by `group_maker`. Also, `image` is not assumed to be background subtracted. If `positions` are not `None` then this method uses `positions` as initial guesses for the centroids. If the centroid positions are set as `fixed` in the PSF model `psf_model`, then the optimizer will only consider the flux as a variable.

Parameters**image** : 2D array-like, `ImageHDU`, `HDUList`

Image to perform photometry.

positions: ‘`~astropy.table.Table`’Positions (in pixel coordinates) at which to *start* the fit for each object. Columns ‘x_0’ and ‘y_0’ must be present. ‘flux_0’ can also be provided to set initial fluxes. If ‘flux_0’ is not provided, aperture photometry is used to estimate initial values for the fluxes.**Returns****output_table** : `Table` or NoneTable with the photometry results, i.e., centroids and fluxes estimations and the initial estimates used to start the fitting process. None is returned if no sources are found in `image`.**NonNormalizable****exception** `photutils.psf.NonNormalizable`Used to indicate that a `FittableImageModel` model is non-normalizable.**PRFAdapter****class** `photutils.psf.PRFAdapter`Bases: `astropy.modeling.Fittable2DModel`A model that adapts a supplied PSF model to act as a PRF. It integrates the PSF model over pixel “boxes”. A critical built-in assumption is that the PSF model scale and location parameters are in *pixel* units.**Parameters****psfmodel** : a 2D model

The model to assume as representative of the PSF

renormalize_psf : boolIf True, the model will be integrated from -inf to inf and re-scaled so that the total integrates to 1. Note that this renormalization only occurs *once*, so if the total flux of `psfmodel` depends on position, this will *not* be correct.**xname** : str or NoneThe name of the `psfmodel` parameter that corresponds to the x-axis center of the PSF.
If None, the model will be assumed to be centered at x=0.**yname** : str or NoneThe name of the `psfmodel` parameter that corresponds to the y-axis center of the PSF.
If None, the model will be assumed to be centered at y=0.**fluxname** : str or NoneThe name of the `psfmodel` parameter that corresponds to the total flux of the star. If None, a scaling factor will be applied by the `PRFAdapter` instead of modifying the `psfmodel`.

Notes

This current implementation of this class (using numerical integration for each pixel) is extremely slow, and only suited for experimentation over relatively few small regions.

Attributes Summary

Methods Summary

`evaluate(x, y, flux, x_0, y_0)` The evaluation function for PRFAdapter.

Attributes Documentation

`flux`

`param_names = ('flux', 'x_0', 'y_0')`

`x_0`

`y_0`

Methods Documentation

`evaluate(x, y, flux, x_0, y_0)`

The evaluation function for PRFAdapter.

SplitCosineBellWindow

`class photutils.psf.SplitCosineBellWindow(alpha, beta)`
Bases: `object`

Class to define a 2D split cosine bell taper function.

Parameters

`alpha` : float, optional

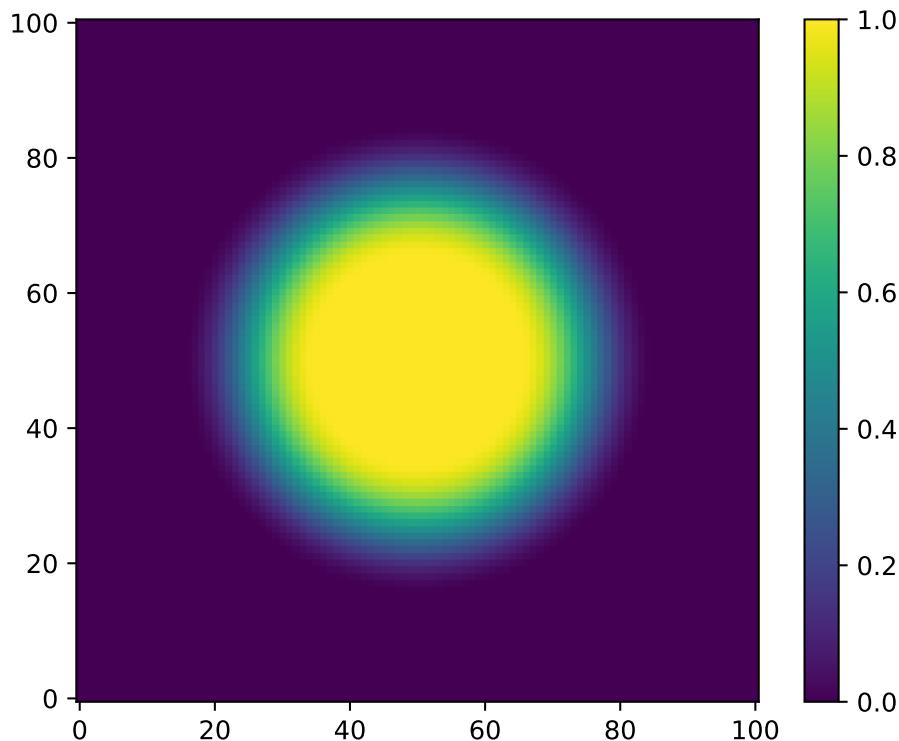
The percentage of array values that are tapered.

`beta` : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. `beta` must be less or equal to 1.0.

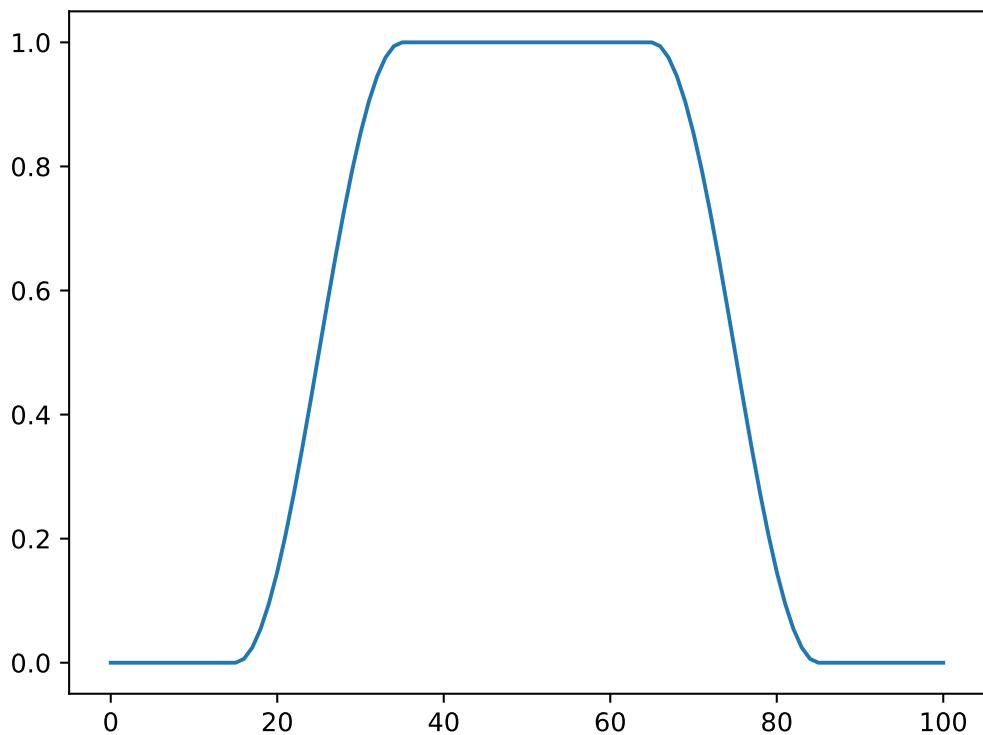
Examples

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



Methods Summary

<code>__call__(shape)</code>	Return a 2D split cosine bell.
------------------------------	--------------------------------

Methods Documentation

`__call__(shape)`

Return a 2D split cosine bell.

Parameters

`shape` : tuple of int

The size of the output array along each axis.

Returns

`result` : ndarray

A 2D array containing the cosine bell values.

TopHatWindow

```
class photutils.psf.TopHatWindow(beta)
    Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D top hat window function.

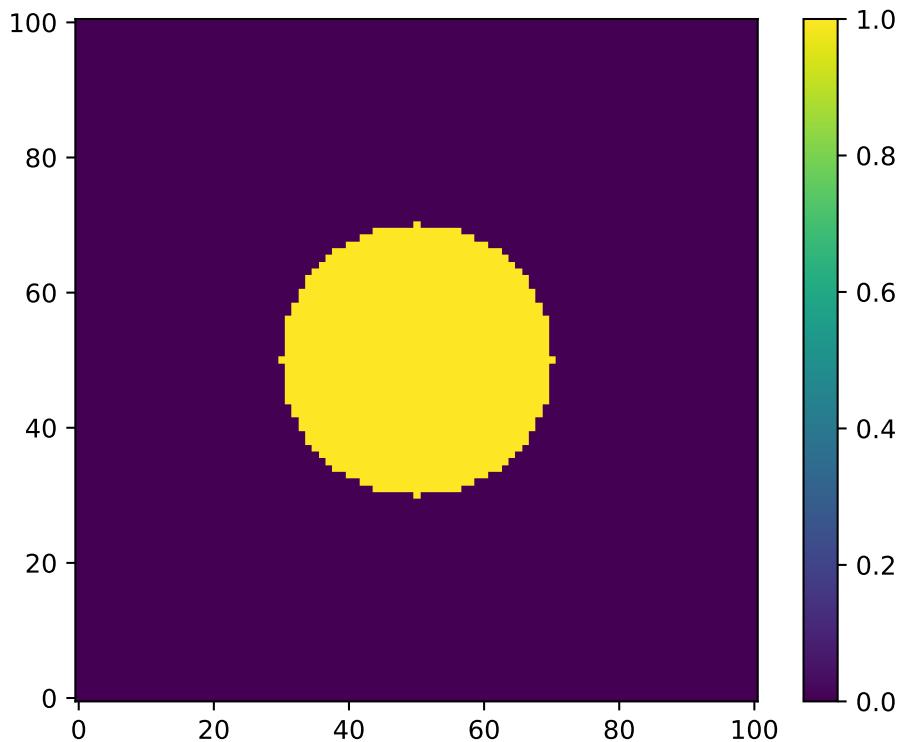
Parameters

beta : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. beta must be less or equal to 1.0.

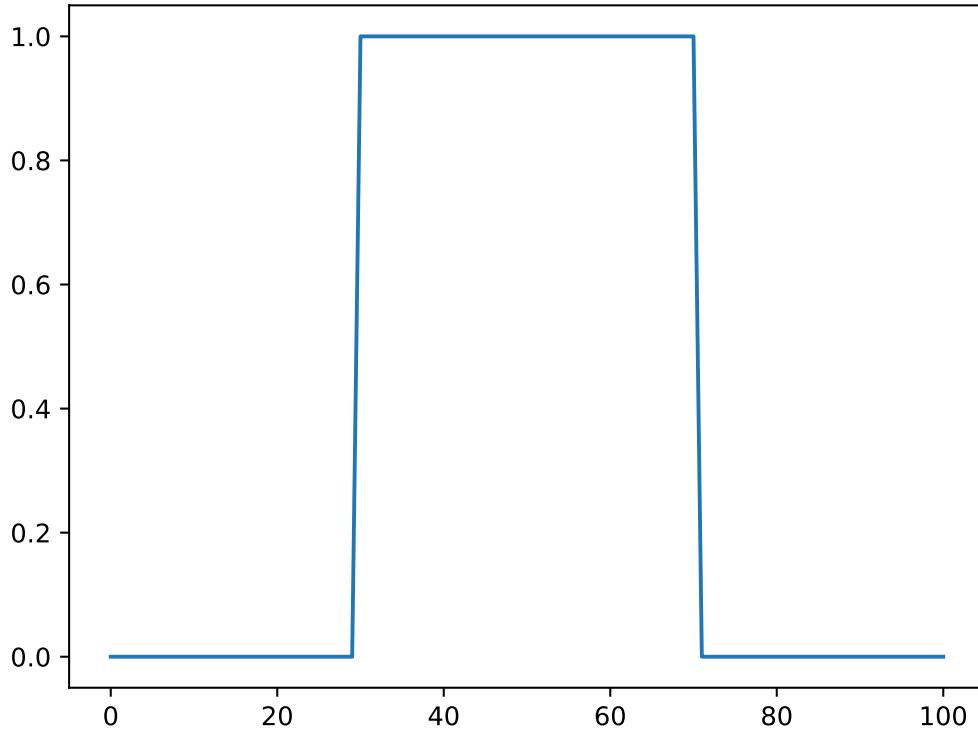
Examples

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower',
           interpolation='nearest')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



TukeyWindow

```
class photutils.psf.TukeyWindow(alpha)
Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D Tukey window function.

The Tukey window is a taper formed by using a split cosine bell function with ends that touch zero.

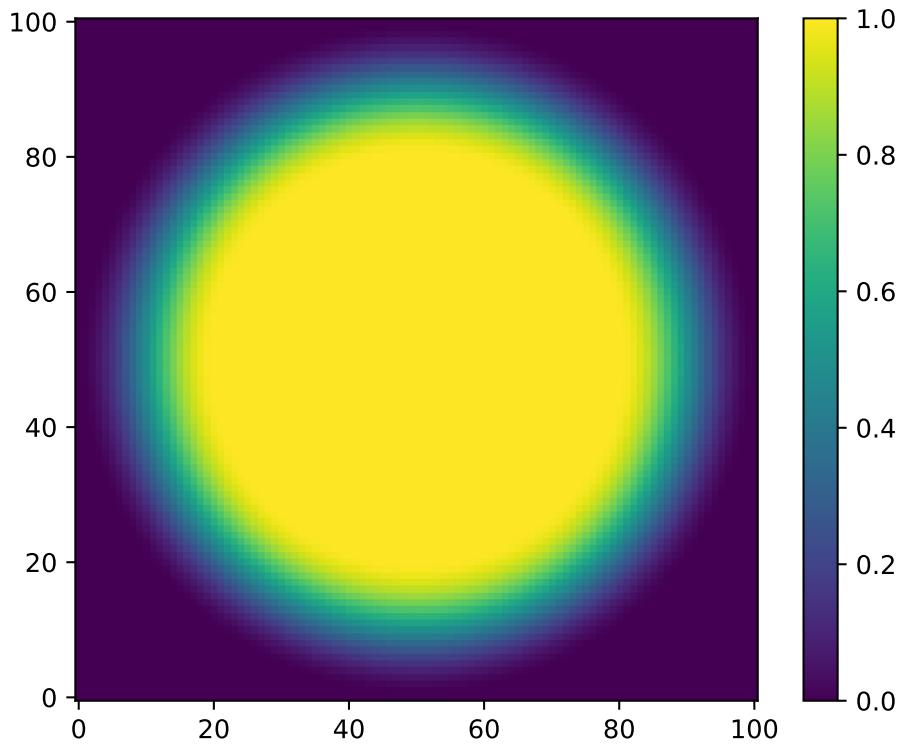
Parameters

alpha : float, optional

The percentage of array values that are tapered.

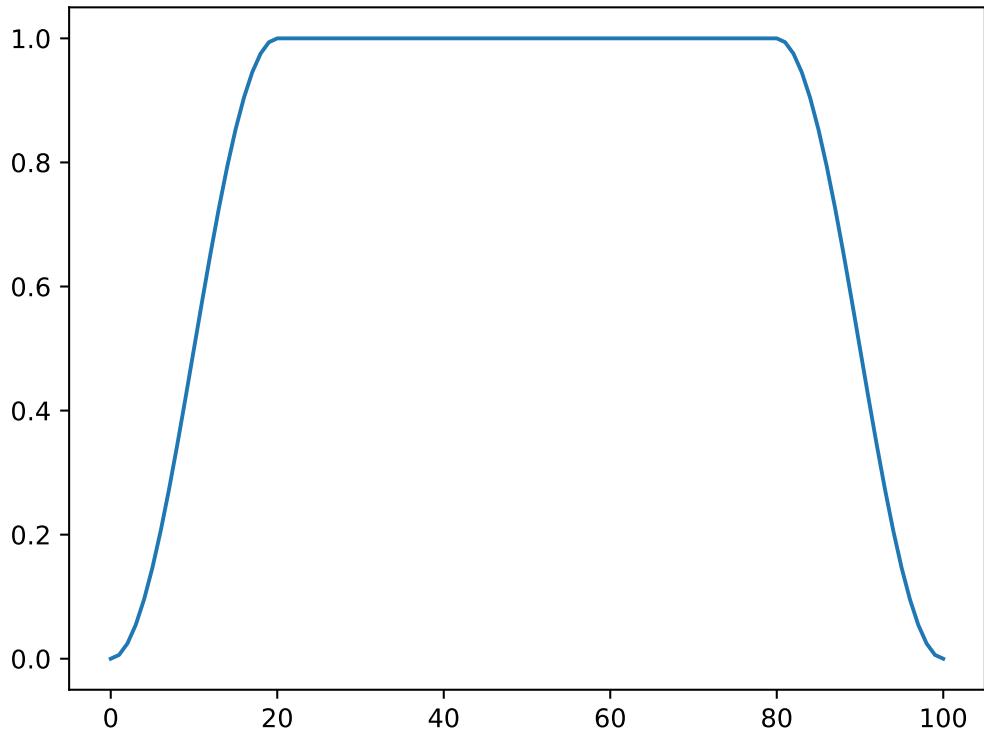
Examples

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```

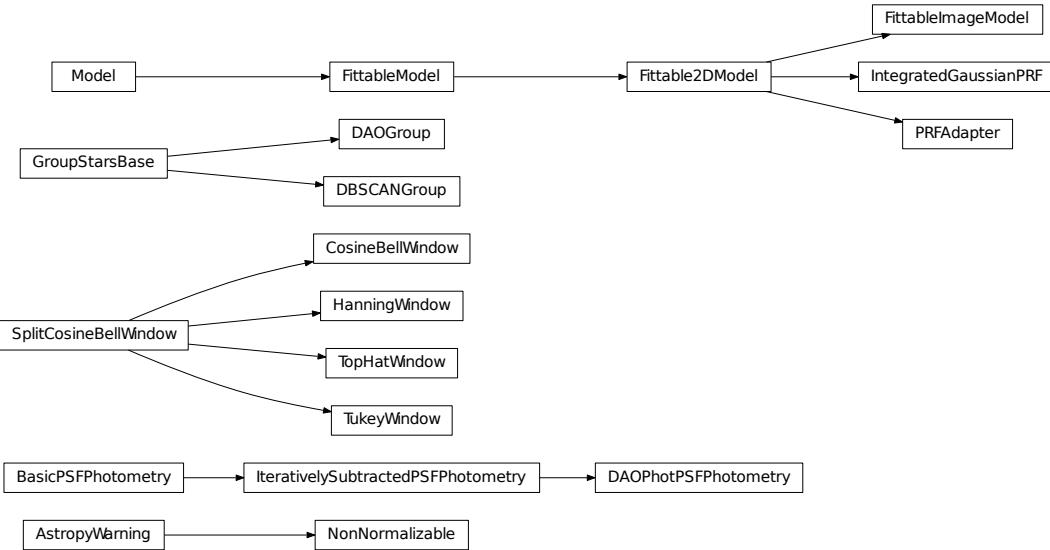


A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



Class Inheritance Diagram



photutils.psf.sandbox Module

This module stores work related to photutils.psf that is not quite ready for prime-time (i.e., is not considered a stable public API), but is included either for experimentation or as legacy code.

Classes

<code>DiscretePRF</code>	A discrete Pixel Response Function (PRF) model.
--------------------------	---

DiscretePRF

```
class photutils.psf.sandbox.DiscretePRF
Bases: astropy.modeling.Fittable2DModel
```

A discrete Pixel Response Function (PRF) model.

The discrete PRF model stores images of the PRF at different subpixel positions or offsets as a lookup table. The resolution is given by the subsampling parameter, which states in how many subpixels a pixel is divided.

In the typical case of wanting to create a PRF from an image with many point sources, use the `create_from_image` method, rather than directly initializing this class.

The discrete PRF model class is initialized with a 4 dimensional array, that contains the PRF images at different subpixel positions. The definition of the axes is as following:

- 1.Axis: y subpixel position

- 2.Axis: x subpixel position
- 3.Axis: y direction of the PRF image
- 4.Axis: x direction of the PRF image

The total array therefore has the following shape (subsampling, subsampling, prf_size, prf_size)

Parameters

prf_array : ndarray

Array containing PRF images.

normalize : bool

Normalize PRF images to unity. Equivalent to saying there is *no* flux outside the bounds of the PRF images.

subsampling : int, optional

Factor of subsampling. Default = 1.

Notes

See [Terminology](#) for more details on the distinction between PSF and PRF as used in this module.

Attributes Summary

<code>flux</code>	
<code>param_names</code>	
<code>prf_shape</code>	Shape of the PRF image.
<code>x_0</code>	
<code>y_0</code>	

Methods Summary

<code>create_from_image(imdata, positions, size[, ...])</code>	Create a discrete point response function (PRF) from image data.
<code>evaluate(x, y, flux, x_0, y_0)</code>	Discrete PRF model evaluation.

Attributes Documentation

flux

param_names = ('flux', 'x_0', 'y_0')

prf_shape

Shape of the PRF image.

x_0

y_0

Methods Documentation

classmethod **create_from_image**(*imdata*, *positions*, *size*, *fluxes=None*, *mask=None*, *mode='mean'*,
 subsampling=1, *fix_nan=False*)

Create a discrete point response function (PRF) from image data.

Given a list of positions and size this function estimates an image of the PRF by extracting and combining the individual PRFs from the given positions.

NaN values are either ignored by passing a mask or can be replaced by the mirrored value with respect to the center of the PRF.

Note that if fluxes are *not* specified explicitly, it will be flux estimated from an aperture of the same size as the PRF image. This does *not* account for aperture corrections so often will *not* be what you want for anything other than quick-look needs.

Parameters

imdata : array

Data array with the image to extract the PRF from

positions : List or array or [Table](#)

List of pixel coordinate source positions to use in creating the PRF. If this is a [Table](#) it must have columns called *x_0* and *y_0*.

size : odd int

Size of the quadratic PRF image in pixels.

mask : bool array, optional

Boolean array to mask out bad values.

fluxes : array, optional

Object fluxes to normalize extracted PRFs. If not given (or None), the flux is estimated from an aperture of the same size as the PRF image.

mode : {‘mean’, ‘median’}

One of the following modes to combine the extracted PRFs:

- ‘mean’: Take the pixelwise mean of the extracted PRFs.

- ‘median’: Take the pixelwise median of the extracted PRFs.

subsampling : int

Factor of subsampling of the PRF (default = 1).

fix_nan : bool

Fix NaN values in the data by replacing it with the mirrored value. Assuming that the PRF is symmetrical.

Returns

prf: [photutils.psf.sandbox.DiscretePRF](#)

Discrete PRF model estimated from data.

evaluate(*x*, *y*, *flux*, *x_0*, *y_0*)

Discrete PRF model evaluation.

Given a certain position and flux the corresponding image of the PSF is chosen and scaled to the flux. If x and y are outside the boundaries of the image, zero will be returned.

Parameters

x : float

x coordinate array in pixel coordinates.

y : float

y coordinate array in pixel coordinates.

flux : float

Model flux.

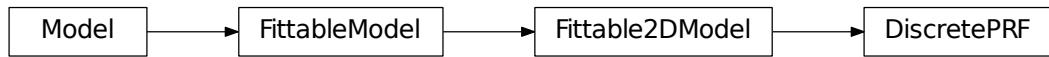
x_0 : float

x position of the center of the PRF.

y_0 : float

y position of the center of the PRF.

Class Inheritance Diagram



CHAPTER 10

PSF Matching (`photutils.psf.matching`)

Introduction

This subpackage contains tools to generate kernels for matching point spread functions (PSFs).

Matching PSFs

Photutils provides a function called `create_matching_kernel()` that generates a matching kernel between two PSFs using the ratio of Fourier transforms (see e.g., Gordon et al. 2008; Aniano et al. 2011).

For this first simple example, let's assume our source and target PSFs are noiseless 2D Gaussians. The “high-resolution” PSF will be a Gaussian with $\sigma = 3$. The “low-resolution” PSF will be a Gaussian with $\sigma = 5$:

```
>>> import numpy as np
>>> from astropy.modeling.models import Gaussian2D
>>> y, x = np.mgrid[0:51, 0:51]
>>> gm1 = Gaussian2D(100, 25, 25, 3, 3)
>>> gm2 = Gaussian2D(100, 25, 25, 5, 5)
>>> g1 = gm1(x, y)
>>> g2 = gm2(x, y)
>>> g1 /= g1.sum()
>>> g2 /= g2.sum()
```

For these 2D Gaussians, the matching kernel should be a 2D Gaussian with $\sigma = 4$ ($\sqrt{5^2 - 3^2}$). Let's create the matching kernel using a Fourier ratio method. Note that the input source and target PSFs must have the same shape and pixel scale.

```
>>> from photutils import create_matching_kernel
>>> kernel = create_matching_kernel(g1, g2)
```

Let's plot the result:

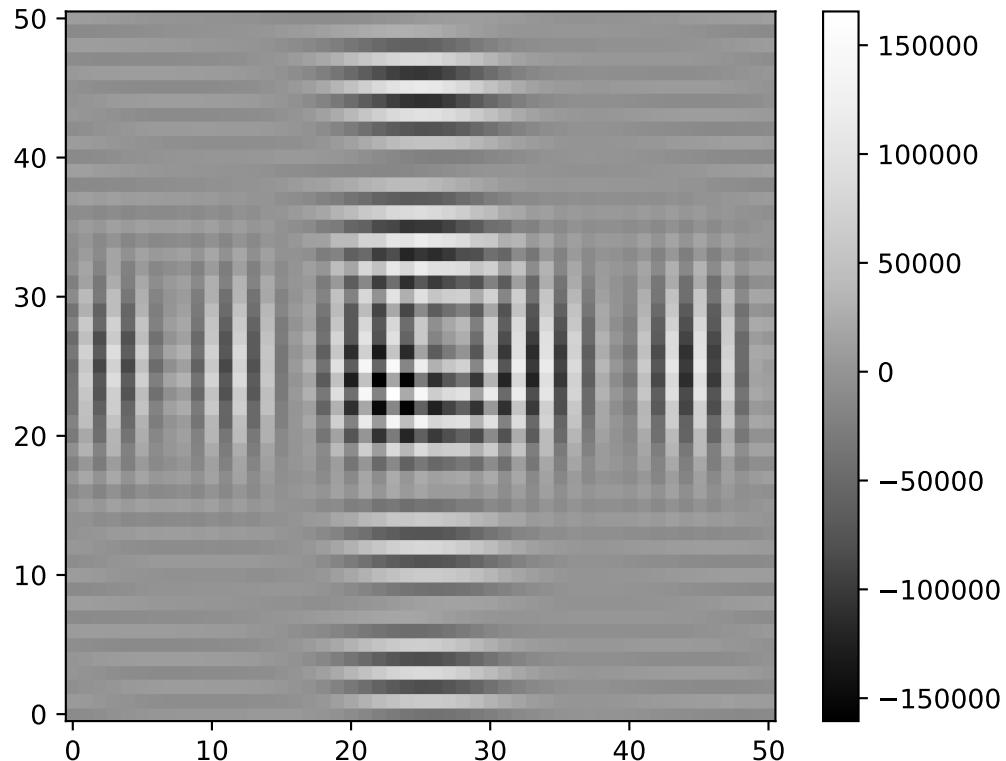
```

import numpy as np
from astropy.modeling.models import Gaussian2D
from photutils import create_matching_kernel
import matplotlib.pyplot as plt

y, x = np.mgrid[0:51, 0:51]
gm1 = Gaussian2D(100, 25, 25, 3, 3)
gm2 = Gaussian2D(100, 25, 25, 5, 5)
g1 = gm1(x, y)
g2 = gm2(x, y)
g1 /= g1.sum()
g2 /= g2.sum()

kernel = create_matching_kernel(g1, g2)
plt.imshow(kernel, cmap='Greys_r', origin='lower')
plt.colorbar()

```



We quickly observe that the result is not as expected. This is because of high-frequency noise in the Fourier transforms (even though these are noiseless PSFs, there is floating-point noise in the ratios). Using the Fourier ratio method, one must filter the high-frequency noise from the Fourier ratios. This is performed by inputting a [window function](#), which may be a function or a callable object. In general, the user will need to exercise some care when defining a window function. For more information, please see [Aniano et al. 2011](#).

Photutils provides the following window classes:

- [HanningWindow](#)

- [TukeyWindow](#)
- [CosineBellWindow](#)
- [SplitCosineBellWindow](#)
- [TopHatWindow](#)

Here are plots of 1D cuts across the center of the 2D window functions:

```
from photutils import (HanningWindow, TukeyWindow, CosineBellWindow,
                      SplitCosineBellWindow, TopHatWindow)
import matplotlib.pyplot as plt
w1 = HanningWindow()
w2 = TukeyWindow(alpha=0.5)
w3 = CosineBellWindow(alpha=0.5)
w4 = SplitCosineBellWindow(alpha=0.4, beta=0.3)
w5 = TopHatWindow(beta=0.4)
shape = (101, 101)
y0 = (shape[0] - 1) // 2

plt.figure()
plt.subplots_adjust(wspace=0.4, hspace=0.4)

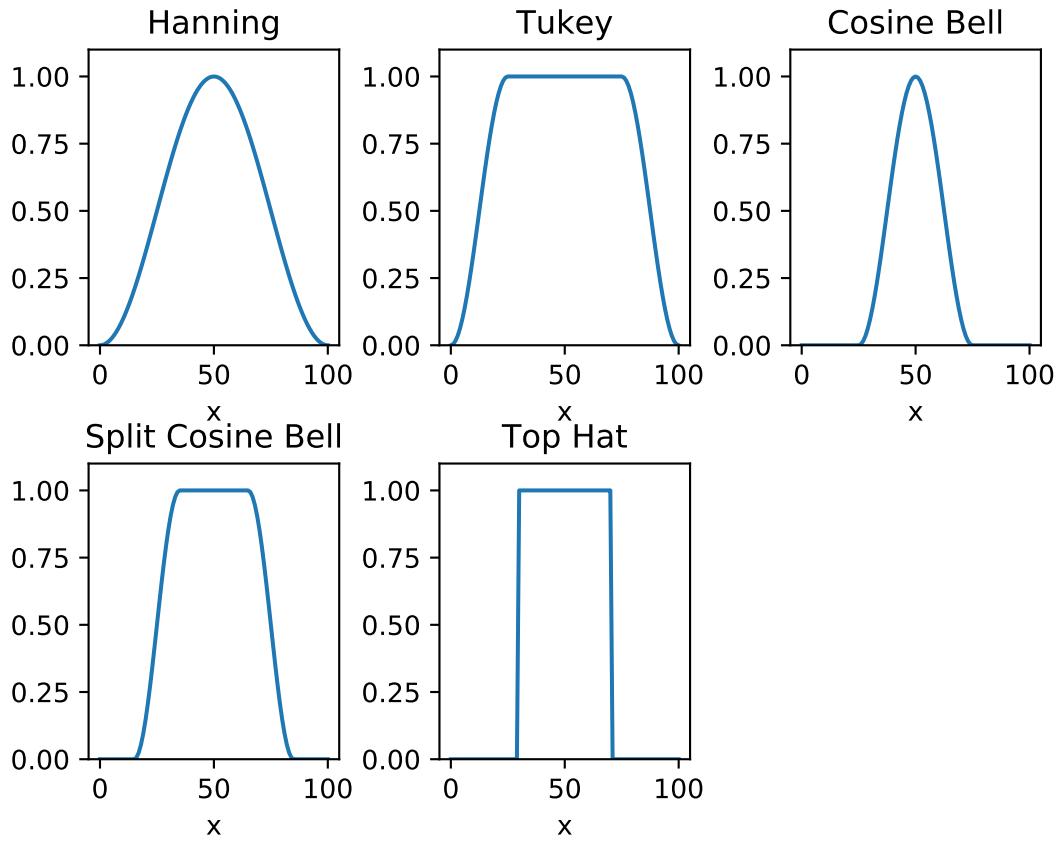
plt.subplot(2, 3, 1)
plt.plot(w1(shape)[y0, :])
plt.title('Hanning')
plt.xlabel('x')
plt.ylim(0, 1.1))

plt.subplot(2, 3, 2)
plt.plot(w2(shape)[y0, :])
plt.title('Tukey')
plt.xlabel('x')
plt.ylim(0, 1.1))

plt.subplot(2, 3, 3)
plt.plot(w3(shape)[y0, :])
plt.title('Cosine Bell')
plt.xlabel('x')
plt.ylim(0, 1.1))

plt.subplot(2, 3, 4)
plt.plot(w4(shape)[y0, :])
plt.title('Split Cosine Bell')
plt.xlabel('x')
plt.ylim(0, 1.1))

plt.subplot(2, 3, 5)
plt.plot(w5(shape)[y0, :], label='Top Hat')
plt.title('Top Hat')
plt.xlabel('x')
plt.ylim(0, 1.1))
```



However, the user may input any function or callable object to generate a custom window function.

In this example, because these are noiseless PSFs, we will use a `TopHatWindow` object as the low-pass filter:

```
>>> from photutils import TopHatWindow
>>> window = TopHatWindow(0.35)
>>> kernel = create_matching_kernel(g1, g2, window=window)
```

Note that the output matching kernel from `create_matching_kernel()` is always normalized such that the kernel array sums to 1:

```
>>> print(kernel.sum())
1.0
```

Let's display the new matching kernel:

```
import numpy as np
from astropy.modeling.models import Gaussian2D
from photutils import create_matching_kernel, TopHatWindow
import matplotlib.pyplot as plt

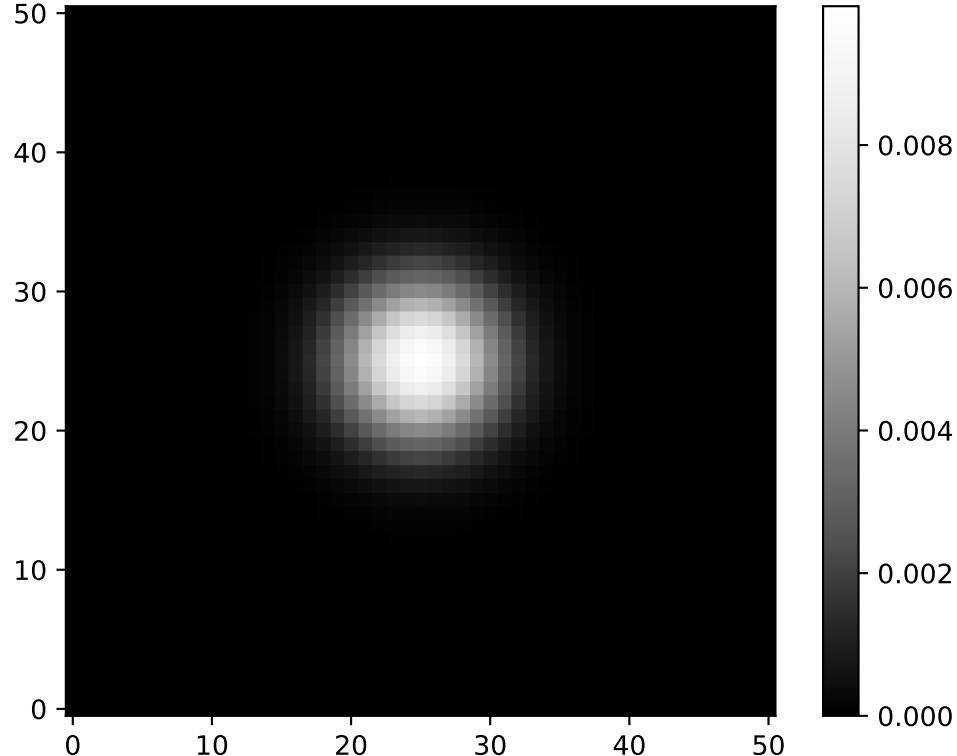
y, x = np.mgrid[0:51, 0:51]
gm1 = Gaussian2D(100, 25, 25, 3, 3)
gm2 = Gaussian2D(100, 25, 25, 5, 5)
g1 = gm1(x, y)
g2 = gm2(x, y)
g1 /= g1.sum()
```

```

g2 /= g2.sum()

window = TopHatWindow(0.35)
kernel = create_matching_kernel(g1, g2, window=window)
plt.imshow(kernel, cmap='Greys_r', origin='lower')
plt.colorbar()

```



As desired, the result is indeed a 2D Gaussian with a $\sigma = 4$. Here we will show 1D cuts across the center of the kernel images:

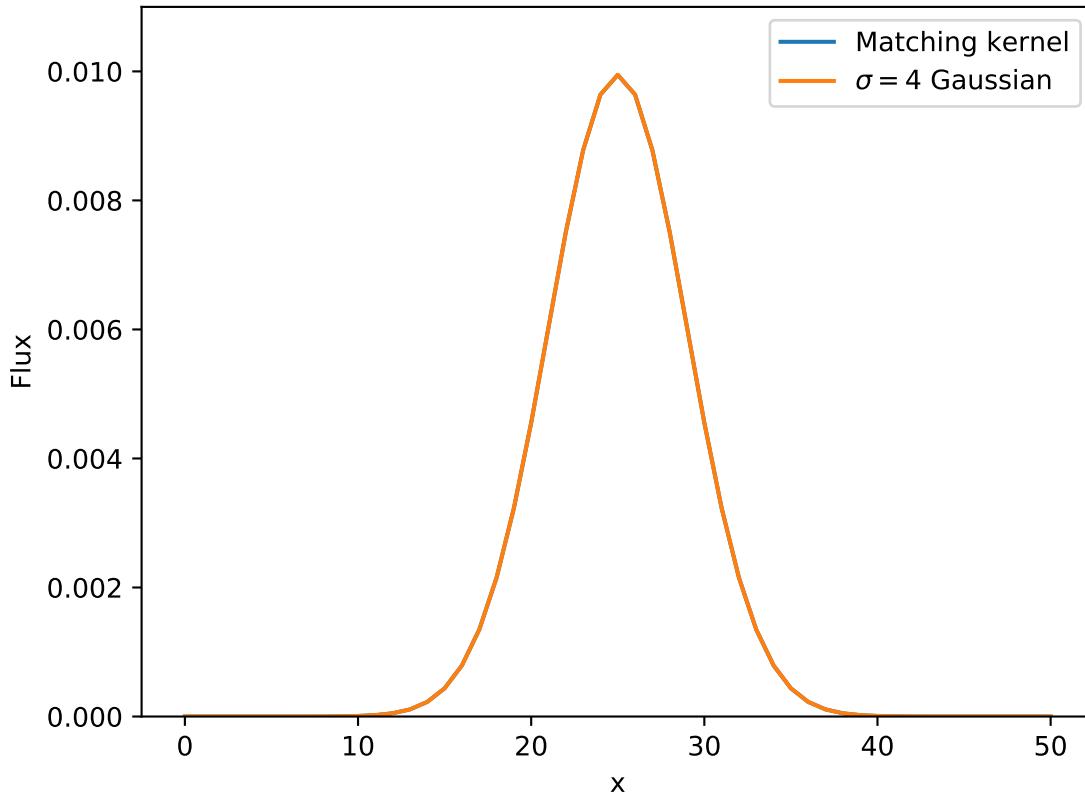
```

import numpy as np
from astropy.modeling.models import Gaussian2D
from photutils import create_matching_kernel, TopHatWindow
import matplotlib.pyplot as plt

y, x = np.mgrid[0:51, 0:51]
gm1 = Gaussian2D(100, 25, 25, 3, 3)
gm2 = Gaussian2D(100, 25, 25, 5, 5)
gm3 = Gaussian2D(100, 25, 25, 4, 4)
g1 = gm1(x, y)
g2 = gm2(x, y)
g3 = gm3(x, y)
g1 /= g1.sum()
g2 /= g2.sum()
g3 /= g3.sum()

```

```
window = TopHatWindow(0.35)
kernel = create_matching_kernel(g1, g2, window=window)
kernel /= kernel.sum()
plt.plot(kernel[25, :], label='Matching kernel')
plt.plot(g3[25, :], label='$\sigma=4$ Gaussian')
plt.xlabel('x')
plt.ylabel('Flux')
plt.legend()
plt.ylim((0.0, 0.011))
```



Matching IRAC PSFs

For this example, let's generate a matching kernel to go from the Spitzer/IRAC channel 1 (3.6 microns) PSF to the channel 4 (8.0 microns) PSF. We load the PSFs using the `load_irac_psf()` convenience function:

```
>>> from photutils.datasets import load_irac_psf
>>> ch1_hdu = load_irac_psf(channel=1)
Downloading http://data.astropy.org/photometry/irac_ch1_flight.fits [Done]
>>> ch4_hdu = load_irac_psf(channel=4)
Downloading http://data.astropy.org/photometry/irac_ch4_flight.fits [Done]
>>> ch1 = ch1_hdu.data
>>> ch4 = ch4_hdu.data
```

Let's display the images:

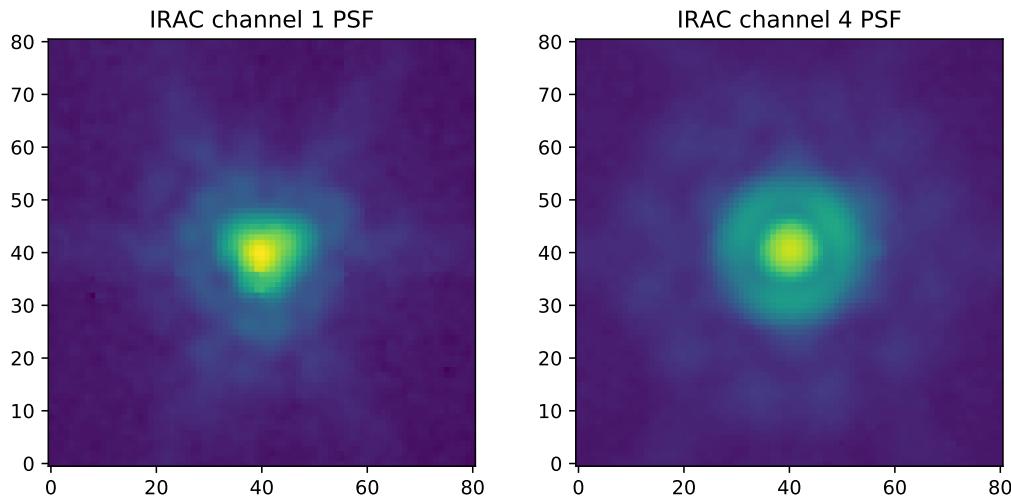
```
import matplotlib.pyplot as plt
from astropy.visualization import LogStretch
from astropy.visualization.mpl_normalize import ImageNormalize
from photutils.datasets import load_irac_psf

ch1_hdu = load_irac_psf(channel=1)
ch4_hdu = load_irac_psf(channel=4)
ch1 = ch1_hdu.data
ch4 = ch4_hdu.data
norm = ImageNormalize(stretch=LogStretch())

plt.figure(figsize=(9, 4))

plt.subplot(1, 2, 1)
plt.imshow(ch1, norm=norm, cmap='viridis', origin='lower')
plt.title('IRAC channel 1 PSF')

plt.subplot(1, 2, 2)
plt.imshow(ch4, norm=norm, cmap='viridis', origin='lower')
plt.title('IRAC channel 4 PSF')
```



For this example, we will use the `CosineBellWindow` for the low-pass window. Also note that these Spitzer/IRAC channel 1 and 4 PSFs have the same shape and pixel scale. If that is not the case, one can use the `resize_psf()` convenience function to resize a PSF image. Typically one would interpolate the lower-resolution PSF to the same size as the higher-resolution PSF.

```
>>> from photutils import CosineBellWindow, create_matching_kernel
>>> window = CosineBellWindow(alpha=0.35)
>>> kernel = create_matching_kernel(ch1, ch4, window=window)
```

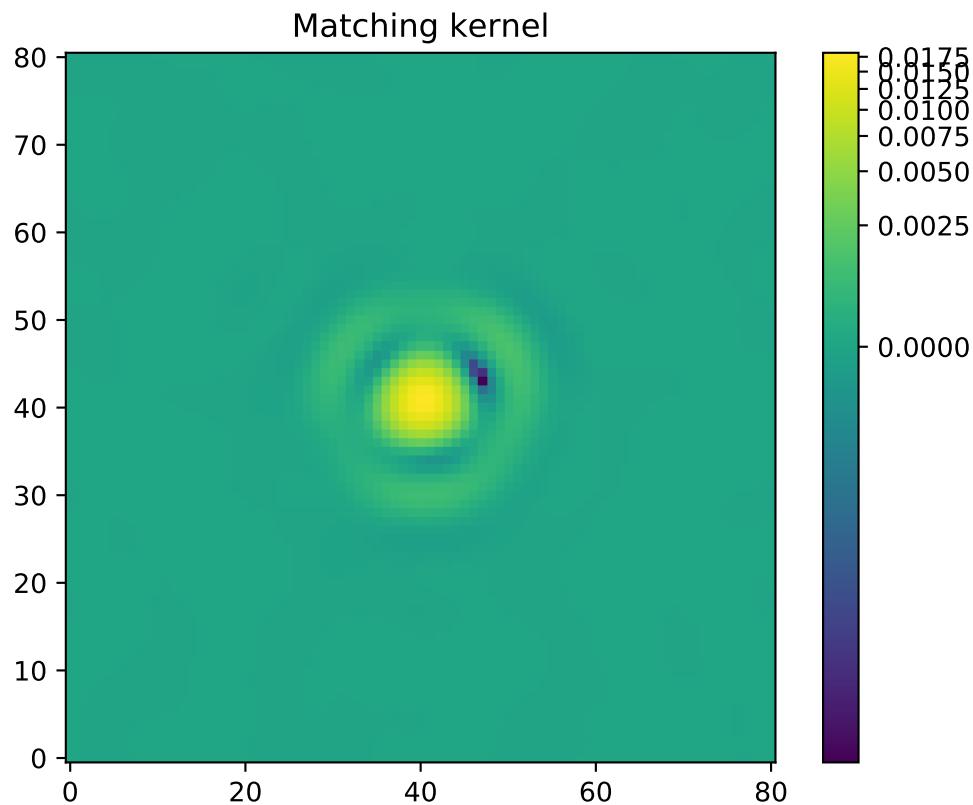
Let's display the matching kernel result:

```
import matplotlib.pyplot as plt
from astropy.visualization import LogStretch
from astropy.visualization.mpl_normalize import ImageNormalize
from photutils.datasets import load_irac_psf
from photutils import CosineBellWindow, create_matching_kernel
```

```
ch1_hdu = load_irac_psf(channel=1)
ch4_hdu = load_irac_psf(channel=4)
ch1 = ch1_hdu.data
ch4 = ch4_hdu.data
norm = ImageNormalize(stretch=LogStretch())

window = CosineBellWindow(alpha=0.35)
kernel = create_matching_kernel(ch1, ch4, window=window)

plt.imshow(kernel, norm=norm, cmap='viridis', origin='lower')
plt.colorbar()
plt.title('Matching kernel')
```



The Spitzer/IRAC channel 1 image could then be convolved with this matching kernel to produce an image with the same resolution as the channel 4 image.

Reference/API

This subpackage contains modules and packages to generate kernels for matching point spread functions.

Functions

<code>create_matching_kernel(source_psf, target_psf)</code>	Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.
<code>resize_psf(psf, input_pixel_scale, ..., [order])</code>	Resize a PSF using spline interpolation of the requested order.

create_matching_kernel

`photutils.psf.matching.create_matching_kernel(source_psf, target_psf, window=None)`
Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.

Parameters

source_psf : 2D `ndarray`

The source PSF. The source PSF should have higher resolution (i.e. narrower) than the target PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

target_psf : 2D `ndarray`

The target PSF. The target PSF should have lower resolution (i.e. broader) than the source PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

window : callable, optional

The window (or taper) function or callable class instance used to remove high frequency noise from the PSF matching kernel. Some examples include:

- [HanningWindow](#)
- [TukeyWindow](#)
- [CosineBellWindow](#)
- [SplitCosineBellWindow](#)
- [TopHatWindow](#)

For more information on window functions and example usage, see [PSF Matching \(photutils.psf.matching\)](#).

Returns

kernel : 2D `ndarray`

The matching kernel to go from `source_psf` to `target_psf`. The output matching kernel is normalized such that it sums to 1.

resize_psf

`photutils.psf.matching.resize_psf(psf, input_pixel_scale, output_pixel_scale, order=3)`
Resize a PSF using spline interpolation of the requested order.

Parameters

psf : 2D `ndarray`

The 2D data array of the PSF.

input_pixel_scale : float

The pixel scale of the input psf. The units must match `output_pixel_scale`.

output_pixel_scale : float

The pixel scale of the output psf. The units must match `input_pixel_scale`.

order : float, optional

The order of the spline interpolation (0-5). The default is 3.

Returns

result : 2D ndarray

The resampled/interpolated 2D data array.

Classes

<code>CosineBellWindow(alpha)</code>	Class to define a 2D cosine bell window function.
<code>HanningWindow()</code>	Class to define a 2D Hanning (or Hann) window function.
<code>SplitCosineBellWindow(alpha, beta)</code>	Class to define a 2D split cosine bell taper function.
<code>TopHatWindow(beta)</code>	Class to define a 2D top hat window function.
<code>TukeyWindow(alpha)</code>	Class to define a 2D Tukey window function.

CosineBellWindow

```
class photutils.psf.matching.CosineBellWindow(alpha)
    Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D cosine bell window function.

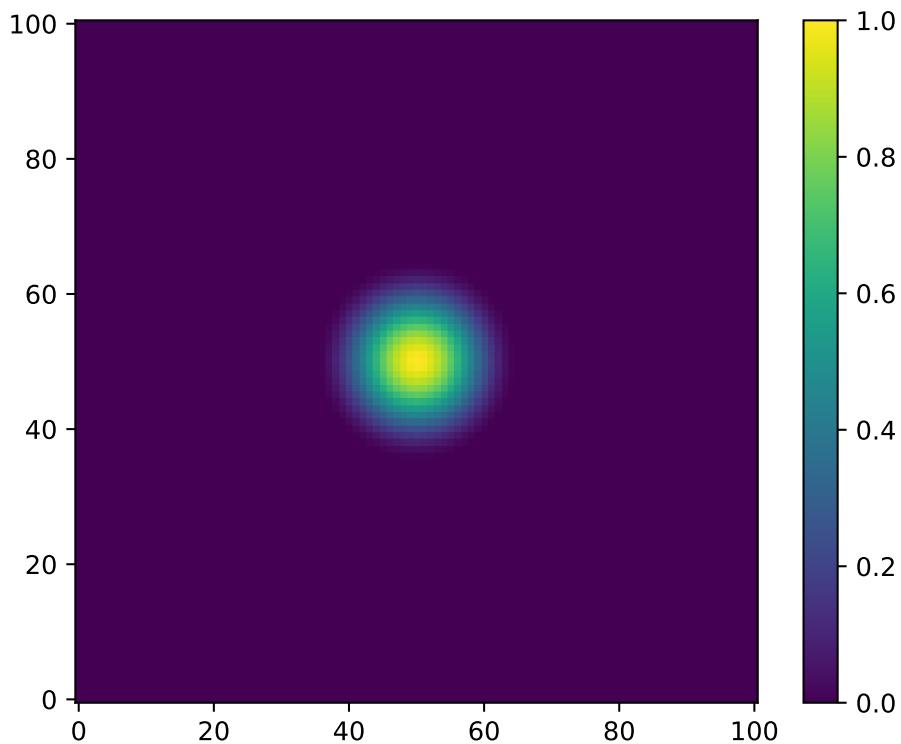
Parameters

alpha : float, optional

The percentage of array values that are tapered.

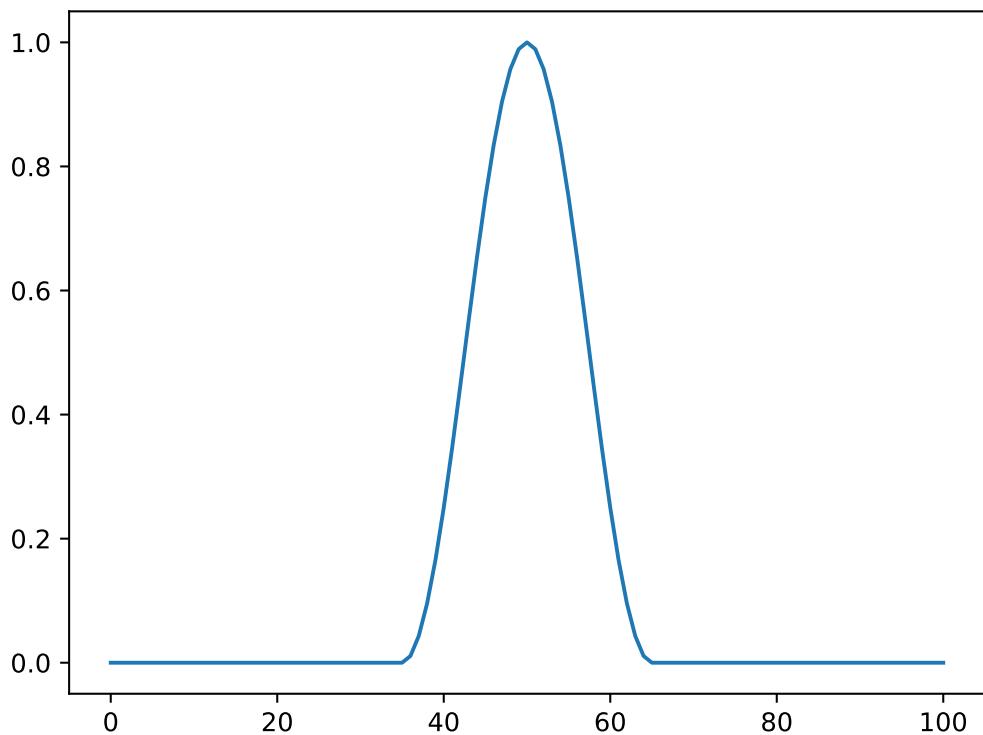
Examples

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



HanningWindow

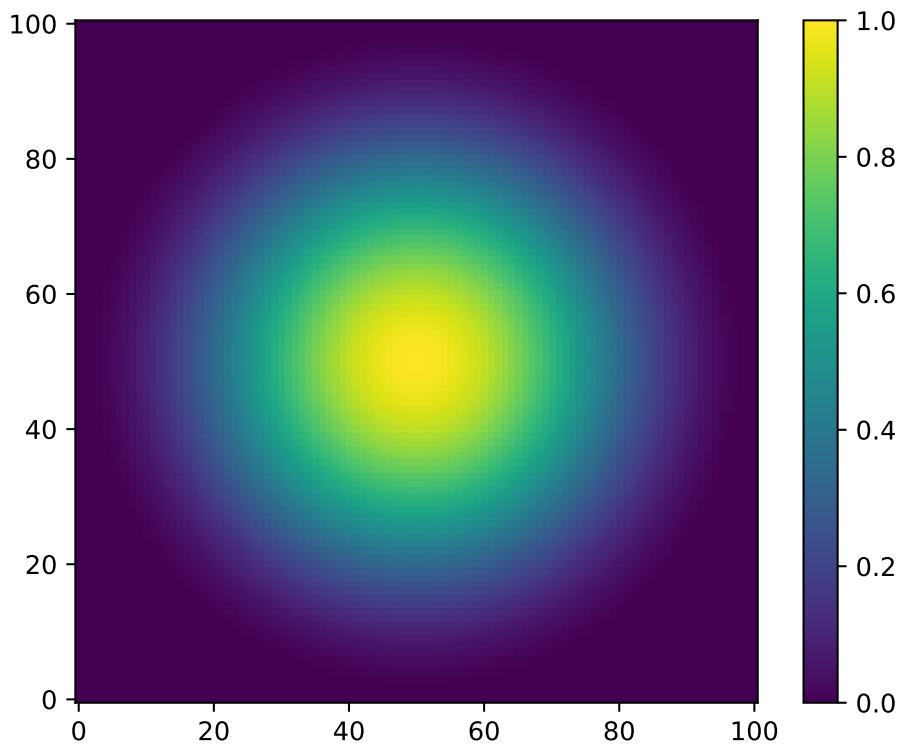
`class photutils.psf.matching.HanningWindow`
Bases: `photutils.psf.matching.SplitCosineBellWindow`

Class to define a 2D Hanning (or Hann) window function.

The Hann window is a taper formed by using a raised cosine with ends that touch zero.

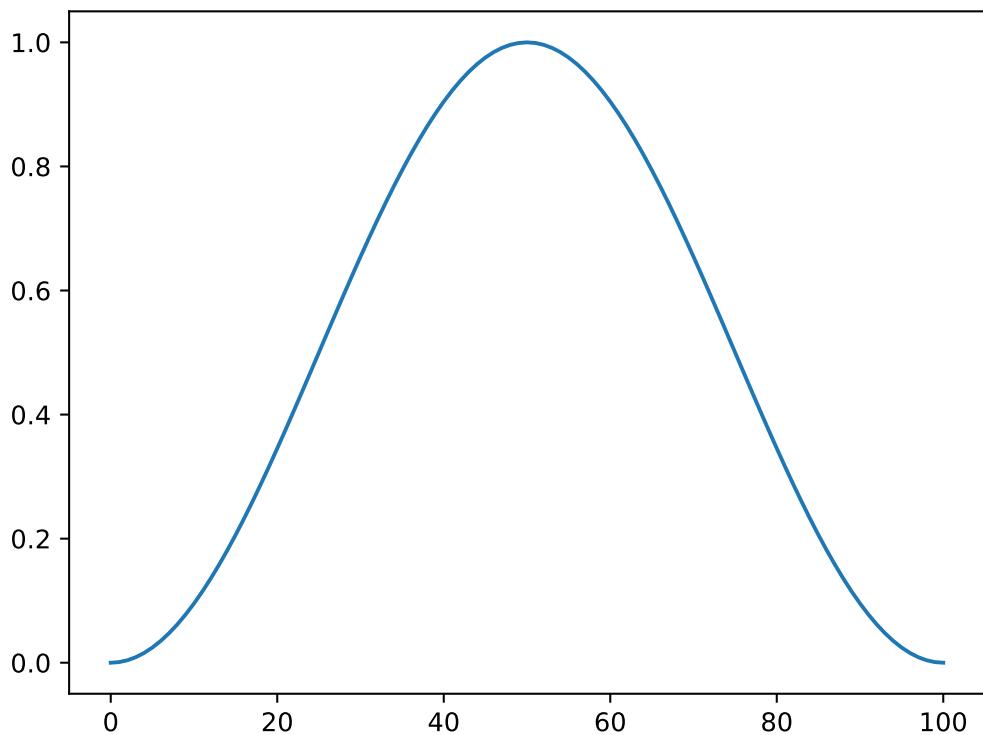
Examples

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.plot(data[50, :])
```



SplitCosineBellWindow

```
class photutils.psf.matching.SplitCosineBellWindow(alpha, beta)
Bases: object
```

Class to define a 2D split cosine bell taper function.

Parameters

alpha : float, optional

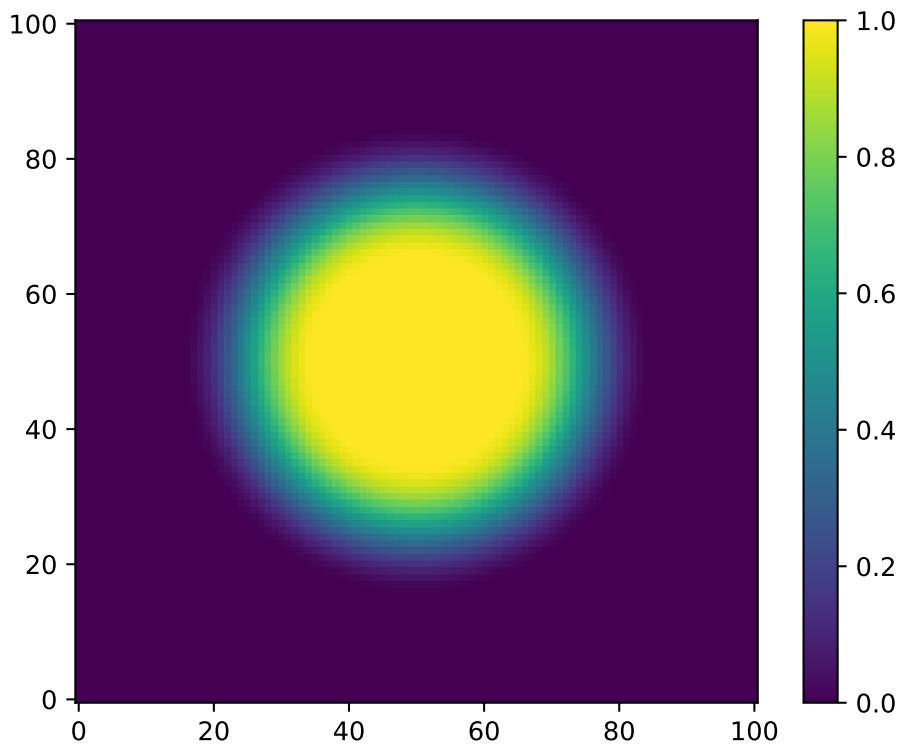
The percentage of array values that are tapered.

beta : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. beta must be less or equal to 1.0.

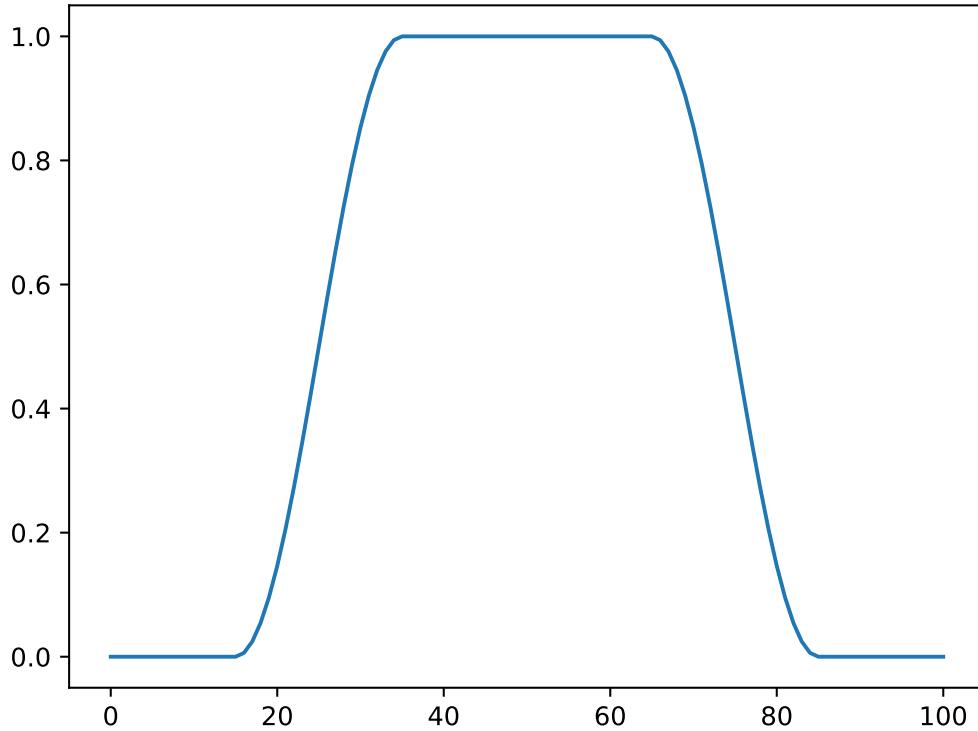
Examples

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



Methods Summary

<code>__call__(shape)</code>	Return a 2D split cosine bell.
------------------------------	--------------------------------

Methods Documentation

`__call__(shape)`
Return a 2D split cosine bell.

Parameters
`shape` : tuple of int

The size of the output array along each axis.

Returns
`result` : ndarray
A 2D array containing the cosine bell values.

TopHatWindow

`class photutils.psf.matching.TopHatWindow(beta)`
Bases: `photutils.psf.matching.SplitCosineBellWindow`

Class to define a 2D top hat window function.

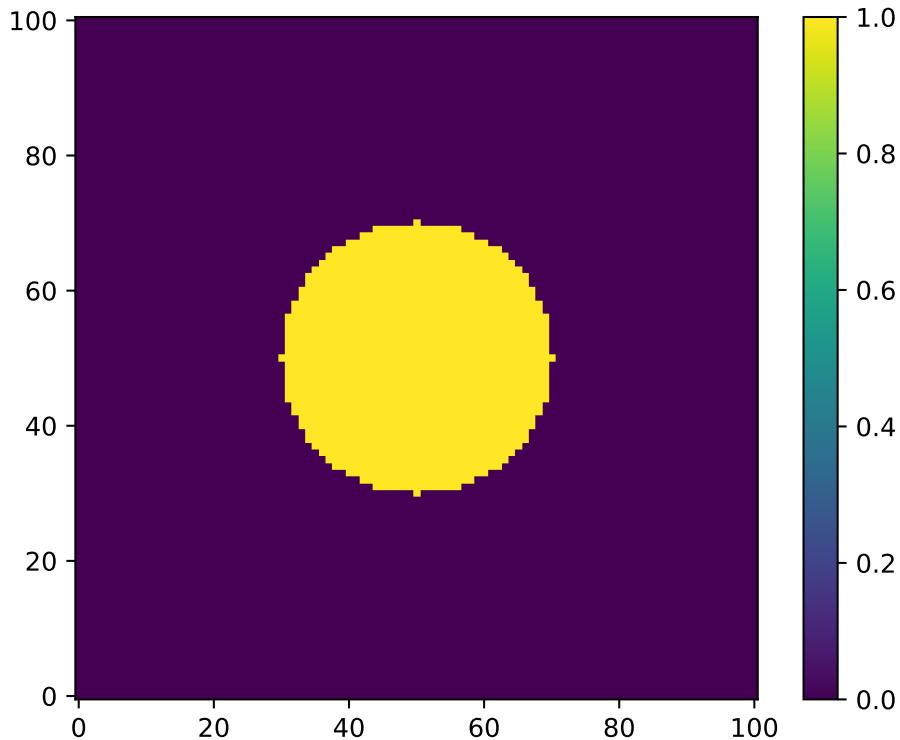
Parameters

beta : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. beta must be less or equal to 1.0.

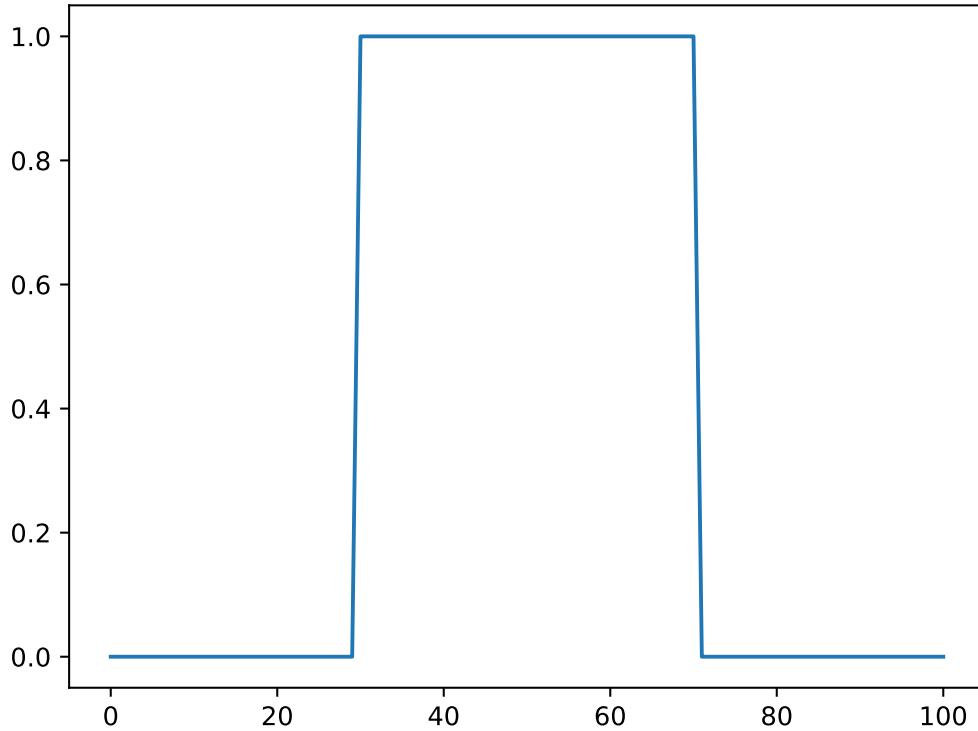
Examples

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower',
           interpolation='nearest')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



TukeyWindow

```
class photutils.psf.matching.TukeyWindow(alpha)
Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D Tukey window function.

The Tukey window is a taper formed by using a split cosine bell function with ends that touch zero.

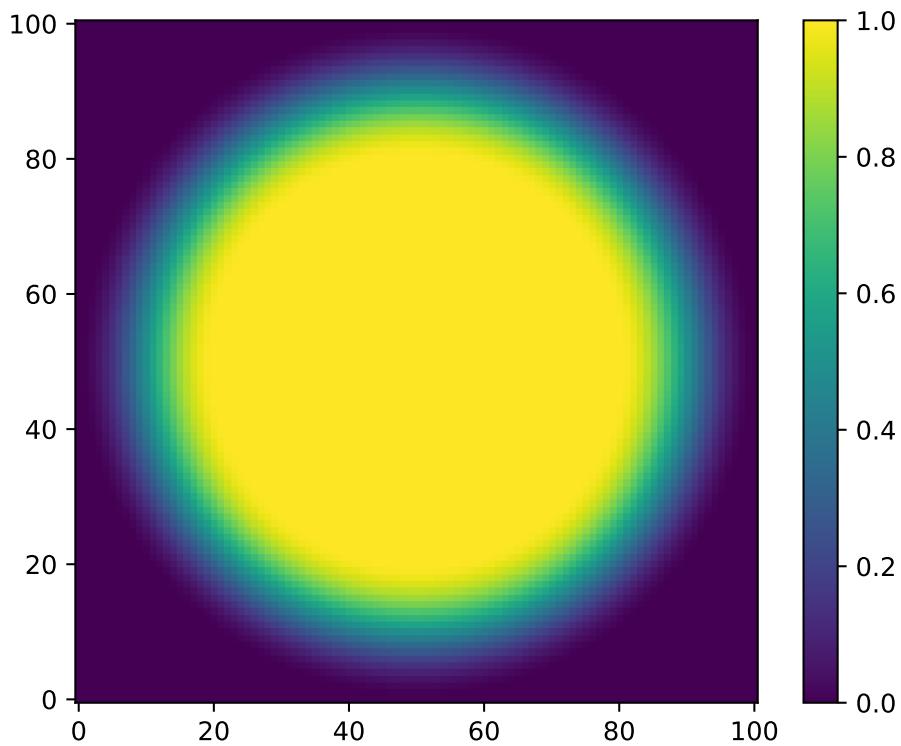
Parameters

alpha : float, optional

The percentage of array values that are tapered.

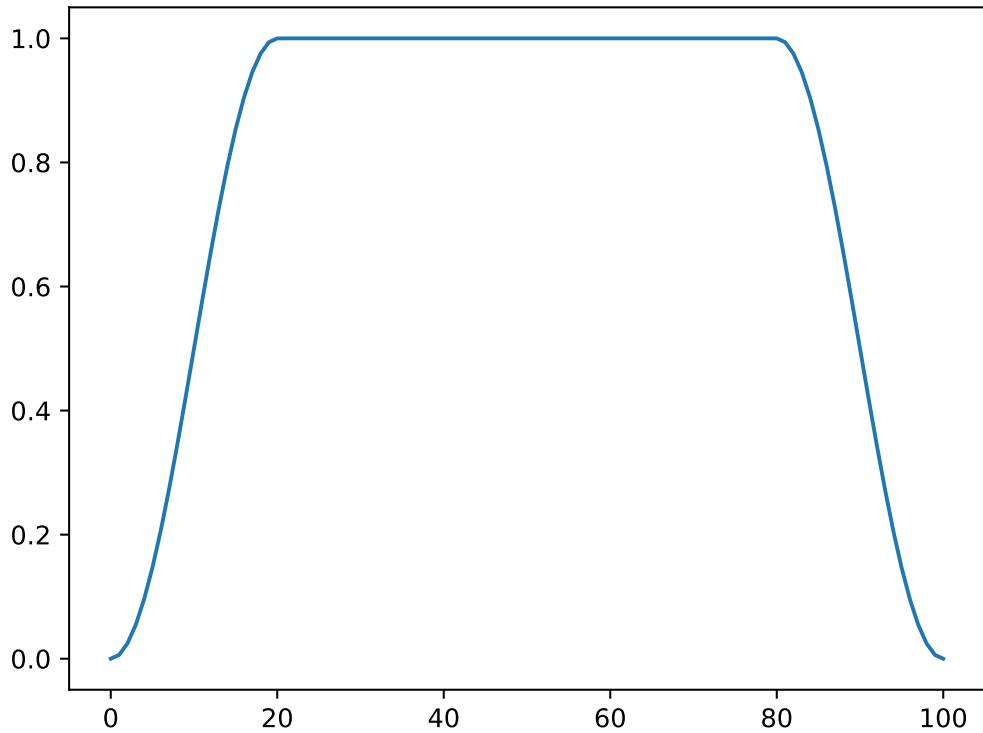
Examples

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



Class Inheritance Diagram

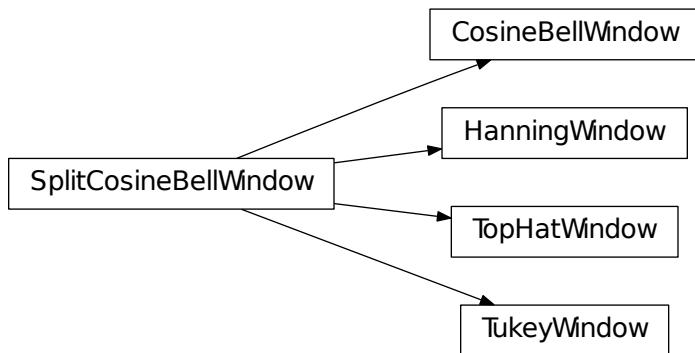


Image Segmentation (`photutils.segmentation`)

Introduction

Photutils includes a general-use function to detect sources (both point-like and extended) in an image using a process called `image segmentation` in the `computer vision` field. After detecting sources using `image segmentation`, we can then measure their photometry, centroids, and morphological properties by using additional tools in Photutils.

Source Extraction Using Image Segmentation

Photutils provides tools to detect astronomical sources using `image segmentation`, which is a process of assigning a label to every pixel in an image such that pixels with the same label are part of the same source. The segmentation procedure implemented in Photutils is called the `threshold` method, where detected sources must have a minimum number of connected pixels that are each greater than a specified threshold value in an image. The threshold level is usually defined at some multiple of the background standard deviation (sigma) above the background. The image can also be filtered before thresholding to smooth the noise and maximize the detectability of objects with a shape similar to the filter kernel.

In Photutils, source extraction is performed using the `detect_sources()` function. The `detect_threshold()` tool is a convenience function that generates a 2D detection threshold image using simple sigma-clipped statistics to estimate the background and background RMS.

For this example, let's detect sources in a synthetic image provided by the datasets module:

```
>>> from photutils.datasets import make_100gaussians_image  
>>> data = make_100gaussians_image()
```

We will use `detect_threshold()` to produce a detection threshold image. `detect_threshold()` will estimate the background and background RMS using sigma-clipped statistics, if they are not input. The threshold level is calculated using the `snr` input as the sigma level above the background. Here we generate a simple pixel-wise threshold at 3 sigma above the background:

```
>>> from photutils import detect_threshold  
>>> threshold = detect_threshold(data, snr=3.)
```

For more sophisticated analyses, one should generate a 2D background and background-only error image (e.g., from your data reduction or by using [Background2D](#)). In that case, a 3-sigma threshold image is simply:

```
>>> threshold = bkg + (3.0 * bkg_rms)
```

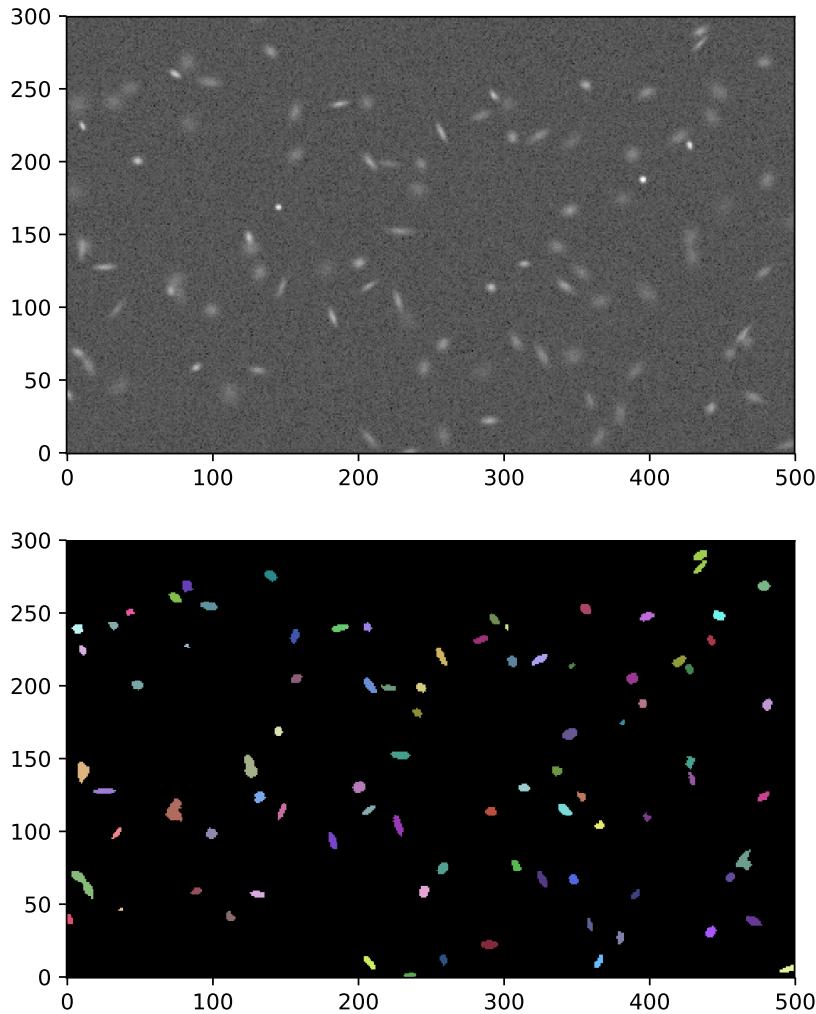
Note that if the threshold includes the background level (as above), then the image input into [detect_sources\(\)](#) should *not* be background subtracted.

Let's find sources that have 5 connected pixels that are each greater than the corresponding pixel-wise threshold level defined above. Because the threshold returned by [detect_threshold\(\)](#) includes the background, we do not subtract the background from the data here. We will also input a 2D circular Gaussian kernel with a FWHM of 2 pixels to filter the image prior to thresholding:

```
>>> from astropy.convolution import Gaussian2DKernel  
>>> from astropy.stats import gaussian_fwhm_to_sigma  
>>> from photutils import detect_sources  
>>> sigma = 2.0 * gaussian_fwhm_to_sigma      # FWHM = 2.  
>>> kernel = Gaussian2DKernel(sigma, x_size=3, y_size=3)  
>>> kernel.normalize()  
>>> segm = detect_sources(data, threshold, npixels=5, filter_kernel=kernel)
```

The result is a [SegmentationImage](#) object with the same shape as the data, where sources are labeled by different positive integer values. A value of zero is always reserved for the background. Let's plot both the image and the segmentation image showing the detected sources:

```
>>> import numpy as np  
>>> import matplotlib.pyplot as plt  
>>> from astropy.visualization import SqrtStretch  
>>> from astropy.visualization.mpl_normalize import ImageNormalize  
>>> from photutils.utils import random_cmap  
>>> rand_cmap = random_cmap(segm.max + 1, random_state=12345)  
>>> norm = ImageNormalize(stretch=SqrtStretch())  
>>> fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))  
>>> ax1.imshow(data, origin='lower', cmap='Greys_r', norm=norm)  
>>> ax2.imshow(segm, origin='lower', cmap=rand_cmap)
```



When the segmentation image is generated using image thresholding (e.g., using `detect_sources()`), the source segments effectively represent the isophotal footprint of each source.

Source Deblending

In the example above, overlapping sources are detected as single sources. Separating those sources requires a deblending procedure, such as a multi-thresholding technique used by `SExtractor`. Photutils provides an experimental `deblend_sources()` function that deblends sources uses a combination of multi-thresholding and `watershed segmentation`. Note that in order to deblend sources, they must be separated enough such that there is a saddle between them.

Here's a simple example of source deblending:

```
>>> from photutils import deblend_sources
>>> segm_deblend = deblend_sources(data, segm, npixels=5,
...                                filter_kernel=kernel)
```

where `segm` is the `SegmentationImage` that was generated by `detect_sources()`. Note that the `npixels` and `filter_kernel` input values should match those used in `detect_sources()`. The result is a `SegmentationImage` object containing the deblended segmentation image.

Modifying a Segmentation Image

The `SegmentationImage` object provides several methods that can be used to modify itself (e.g., combining labels, removing labels, removing border segments) prior to measuring source photometry and other source properties, including:

- `relabel()`: Relabel one or more label numbers.
- `relabel_sequential()`: Relable the label numbers sequentially.
- `keep_labels()`: Keep only certain label numbers.
- `remove_labels()`: Remove one or more label numbers.
- `remove_border_labels()`: Remove labeled segments near the image border.
- `remove_masked_labels()`: Remove labeled segments located within a masked region.
- `outline_segments()`: Outline the labeled segments for plotting.

Centroids, Photometry, and Morphological Properties

The `source_properties()` function is the primary tool for measuring the centroids, photometry, and morphological properties of sources defined in a segmentation image. When the segmentation image is generated using image thresholding (e.g., using `detect_sources()`), the source segments effectively represent the isophotal footprint of each source and the resulting photometry is effectively isophotal photometry.

`source_properties()` returns a list of `SourceProperties` objects, one for each segmented source (or a specified subset of sources). An Astropy `Table` of source properties can be generated using the `properties_table()` function. Please see `SourceProperties` for the list of the many properties that are calculated for each source. More properties are likely to be added in the future.

Let's detect sources and measure their properties in a synthetic image. For this example, we will use the `Background2D` class to produce a background and background noise image. We define a 2D detection threshold image using the background and background RMS images. We set the threshold at 3 sigma above the background:

```
>>> from astropy.convolution import Gaussian2DKernel
>>> from photutils.datasets import make_100gaussians_image
>>> from photutils import Background2D, MedianBackground
>>> from photutils import detect_threshold, detect_sources
>>> data = make_100gaussians_image()
>>> bkg_estimator = MedianBackground()
>>> bkg = Background2D(data, (50, 50), filter_size=(3, 3),
...                     bkg_estimator=bkg_estimator)
>>> threshold = bkg.background + (3. * bkg.background_rms)
```

Now we find sources that have 5 connected pixels that are each greater than the corresponding pixel-wise threshold image defined above. Because the threshold includes the background, we do not subtract the background from the data here. We also input a 2D circular Gaussian kernel with a FWHM of 2 pixels to filter the image prior to thresholding:

```
>>> from astropy.stats import gaussian_fwhm_to_sigma
>>> sigma = 2.0 * gaussian_fwhm_to_sigma      # FWHM = 2.
>>> kernel = Gaussian2DKernel(sigma, x_size=3, y_size=3)
>>> kernel.normalize()
>>> segm = detect_sources(data, threshold, npixels=5, filter_kernel=kernel)
```

The result is a `SegmentationImage` where sources are labeled by different positive integer values. Now let's measure the properties of the detected sources defined in the segmentation image with the minimum number of inputs to `source_properties()`:

```
>>> from photutils import source_properties, properties_table
>>> props = source_properties(data, segm)
>>> tbl = properties_table(props)
>>> print(tbl)
  id    xcentroid     ycentroid   ...       cxy           cyy
    pix          pix     ...  1 / pix2  1 / pix2
  ---  -----  -----  ...  -----  -----
  1  235.187719359 1.09919615282 ... -0.192074627794  1.2174907202
  2  494.139941114 5.77044246809 ... -0.541775595871  1.02440633651
  3  207.375726658 10.0753101977 ...  0.77640832975  0.465060945444
  4  364.689548633 10.8904591886 ... -0.547888762468  0.304081033554
  5  258.192771992 11.9617673653 ...  0.0443061872873 0.321833380384
  ...
  ...
  82 74.4566900469 259.833303502 ...  0.47891309336  0.565732743194
  83 82.5392499545 267.718933667 ...  0.067591261795  0.244881586651
  84 477.674384997 267.891446048 ... -0.02140562548  0.391914760017
  85 139.763784105 275.041398359 ...  0.232932536525 0.352391174432
  86 434.040665678 285.607027036 ... -0.0607421731445 0.0555135557551
Length = 86 rows
```

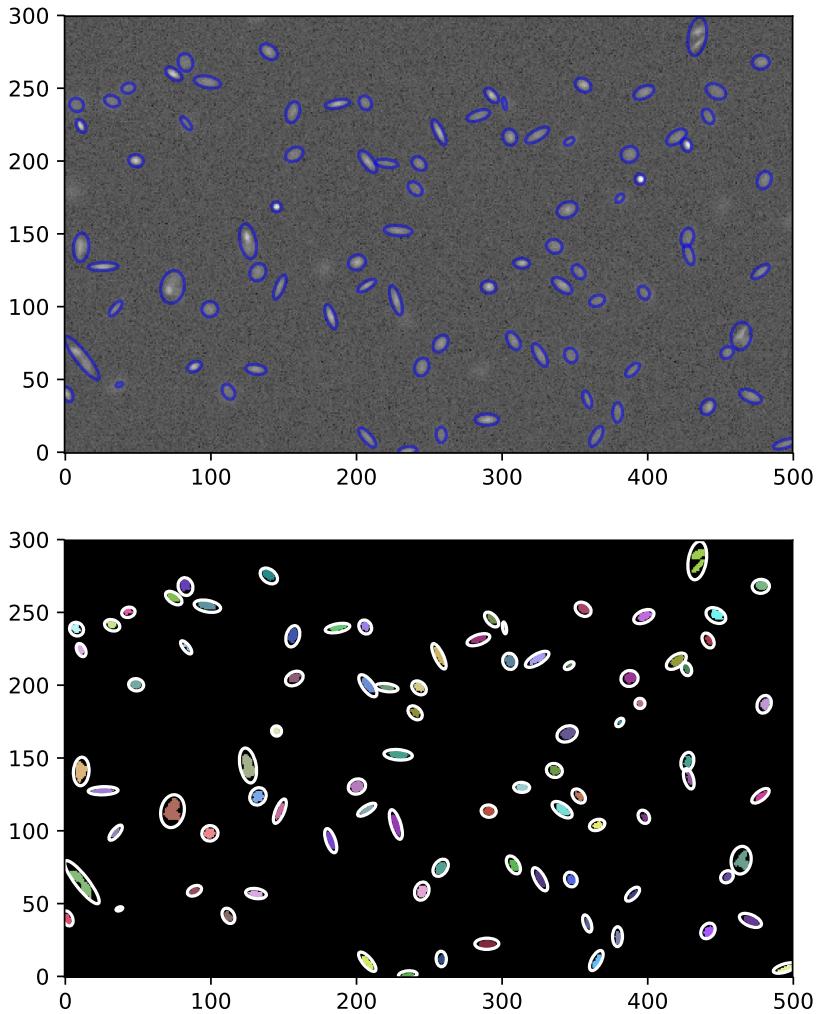
Let's use the measured morphological properties to define approximate isophotal ellipses for each source:

```
>>> from photutils import source_properties, properties_table
>>> from photutils import EllipticalAperture
>>> props = source_properties(data, segm)
>>> r = 3.      # approximate isophotal extent
>>> apertures = []
>>> for prop in props:
...     position = (prop.xcentroid.value, prop.ycentroid.value)
...     a = prop.semimajor_axis_sigma.value * r
...     b = prop.semiminor_axis_sigma.value * r
...     theta = prop.orientation.value
...     apertures.append(EllipticalAperture(position, a, b, theta=theta))
```

Now let's plot the results:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from astropy.visualization import SqrtStretch
>>> from astropy.visualization.mpl_normalize import ImageNormalize
>>> from photutils.utils import random_cmap
>>> rand_cmap = random_cmap(segm.max + 1, random_state=12345)
>>> norm = ImageNormalize(stretch=SqrtStretch())
>>> fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
```

```
>>> ax1.imshow(data, origin='lower', cmap='Greys_r', norm=norm)
>>> ax2.imshow(segm, origin='lower', cmap=rand_cmap)
>>> for aperture in apertures:
...     aperture.plot(color='blue', lw=1.5, alpha=0.5, ax=ax1)
...     aperture.plot(color='white', lw=1.5, alpha=1.0, ax=ax2)
```



We can also specify a specific subset of sources, defined by their labels in the segmentation image:

```
>>> labels = [1, 5, 20, 50, 75, 80]
>>> props = source_properties(data, segm, labels=labels)
>>> tbl = properties_table(props)
>>> print(tbl)
   id    xcentroid      ycentroid      ...       cxy          cyy
      pix         pix        ...  1 / pix2  1 / pix2
-----  ... -----

```

1	235.187719359	1.09919615282	...	-0.192074627794	1.2174907202
5	258.192771992	11.9617673653	...	0.0443061872873	0.321833380384
20	347.177561006	66.5509575226	...	0.115277391421	0.359564354573
50	380.796873199	174.418513707	...	-1.03058291289	1.2769245589
75	32.176218827	241.158486946	...	0.196860594008	0.601167034704
80	355.61483405	252.142253219	...	0.178598051026	0.400332492208

By default, `properties_table()` will include all scalar-valued properties from `SourceProperties`, but a subset of properties can also be specified (or excluded) in the `Table`:

```
>>> labels = [1, 5, 20, 50, 75, 80]
>>> props = source_properties(data, segm, labels=labels)
>>> columns = ['id', 'xcentroid', 'ycentroid', 'source_sum', 'area']
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
  id   xcentroid     ycentroid      source_sum    area
    pix         pix           pix2
--- -----
 1  235.187719359  1.09919615282  496.635623206  27.0
 5  258.192771992  11.9617673653  347.611342072  25.0
20  347.177561006  66.5509575226  415.992569678  31.0
50  380.796873199  174.418513707  145.726417518  11.0
75  32.176218827  241.158486946  398.411403711  29.0
80  355.61483405  252.142253219  906.422600037  45.0
```

A `WCS` transformation can also be input to `source_properties()` via the `wcs` keyword, in which case the International Celestial Reference System (ICRS) Right Ascension and Declination coordinates at the source centroids will be returned.

Background Properties

Like with `aperture_photometry()`, the `data` array that is input to `source_properties()` should be background subtracted. If you input your background image (which should have already been subtracted from the data) into the `background` keyword of `source_properties()`, the background properties for each source will also be calculated:

```
>>> labels = [1, 5, 20, 50, 75, 80]
>>> props = source_properties(data, segm, labels=labels,
...                             background=bkg.background)
>>> columns = ['id', 'background_at_centroid', 'background_mean',
...             'background_sum']
...
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
  id  background_at_centroid  background_mean  background_sum
--- -----
 1      5.20203264929      5.20212082569    140.457262294
 5      5.21378104221      5.213780145     130.344503625
20      5.27885243993      5.27877182437    163.641926556
50      5.19865041002      5.1986157424    57.1847731664
75      5.21062790873      5.21060573569   151.107566335
80      5.12491678472      5.12502080804    230.625936362
```

Photometric Errors

`source_properties()` requires inputting a *total* error array, i.e. the background-only error plus Poisson noise due

to individual sources. The `calc_total_error()` function can be used to calculate the total error array from a background-only error array and an effective gain.

The `effective_gain`, which is the ratio of counts (electrons or photons) to the units of the data, is used to include the Poisson noise from the sources. `effective_gain` can either be a scalar value or a 2D image with the same shape as the data. A 2D effective gain image is useful for mosaic images that have variable depths (i.e., exposure times) across the field. For example, one should use an exposure-time map as the `effective_gain` for a variable depth mosaic image in count-rate units.

Let's assume our synthetic data is in units of electrons per second. In that case, the `effective_gain` should be the exposure time (here we set it to 500 seconds):

```
>>> from photutils.utils import calc_total_error
>>> labels = [1, 5, 20, 50, 75, 80]
>>> effective_gain = 500.
>>> error = calc_total_error(data, bkg.background_rms, effective_gain)
>>> props = source_properties(data, segm, labels=labels, error=error)
>>> columns = ['id', 'xcentroid', 'ycentroid', 'source_sum',
...             'source_sum_err']
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
  id    xcentroid     ycentroid      source_sum  source_sum_err
    pix          pix
--- -----
  1  235.187719359  1.09919615282  496.635623206  11.0788667038
  5  258.192771992  11.9617673653   347.611342072  10.723068215
 20  347.177561006  66.5509575226   415.992569678  12.1782078398
 50  380.796873199  174.418513707  145.726417518   7.29536295106
 75  32.176218827  241.158486946  398.411403711  11.553412812
 80  355.61483405  252.142253219  906.422600037  13.7686828317
```

`source_sum` and `source_sum_err` are the instrumental flux and propagated flux error within the source segments.

Pixel Masking

Pixels can be completely ignored/excluded (e.g. bad pixels) when measuring the source properties by providing a boolean mask image via the `mask` keyword (`True` pixel values are masked) to the `source_properties()` function or `SourceProperties` class.

Filtering

`SExtractor`'s centroid and morphological parameters are always calculated from a filtered “detection” image. The usual downside of the filtering is the sources will be made more circular than they actually are. If you wish to reproduce `SExtractor` results, then use the `filter_kernel` keyword to `source_properties()` to filter the data prior to centroid and morphological measurements. The input kernel should be the same one used to define the source segments in `detect_sources()`. If `filter_kernel` is `None`, then the centroid and morphological measurements will be performed on the unfiltered data. Note that photometry is *always* performed on the unfiltered data.

Reference/API

This subpackage contains tools for detecting sources using image segmentation and measuring their centroids, photometry, and morphological properties.

Functions

<code>deblend_sources(data, segment_img, npixels)</code>	Deblend overlapping sources labeled in a segmentation image.
<code>detect_sources(data, threshold, npixels[, ...])</code>	Detect sources above a specified threshold value in an image and return a <code>SegmentationImage</code> object.
<code>make_source_mask(data, snr, npixels[, mask, ...])</code>	Make a source mask using source segmentation and binary dilation.
<code>properties_table(source_props[, columns, ...])</code>	Construct a <code>Table</code> of properties from a list of <code>SourceProperties</code> objects.
<code>source_properties(data, segment_img[, ...])</code>	Calculate photometry and morphological properties of sources defined by a labeled segmentation image.

`deblend_sources`

```
photutils.segmentation.deblend_sources(data, segment_img, npixels, filter_kernel=None, labels=None,
                                         nlevels=32, contrast=0.001, mode=u'exponential', connectivity=8, relabel=True)
```

Deblend overlapping sources labeled in a segmentation image.

Sources are deblended using a combination of multi-thresholding and watershed segmentation. In order to deblend sources, they must be separated enough such that there is a saddle between them.

Note: This function is experimental. Please report any issues on the [Photutils GitHub issue tracker](#)

Parameters

data : array_like

The 2D array of the image.

segment_img : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as data where sources are labeled by different positive integer values. A value of zero is reserved for the background.

npixels : int

The number of connected pixels, each greater than threshold, that an object must have to be detected. `npixels` must be a positive integer.

filter_kernel : array-like (2D) or `Kernel1D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel.

labels : int or array-like of int, optional

The label numbers to deblend. If `None` (default), then all labels in the segmentation image will be deblended.

nlevels : int, optional

The number of multi-thresholding levels to use. Each source will be re-thresholded at `nlevels`, spaced exponentially or linearly (see the `mode` keyword), between its minimum and maximum values within the source segment.

contrast : float, optional

The fraction of the total (blended) source flux that a local peak must have to be considered as a separate object. `contrast` must be between 0 and 1, inclusive. If `contrast` = 0 then every local peak will be made a separate object (maximum deblending). If `contrast` = 1 then no deblending will occur. The default is 0.001, which will deblend sources with a magnitude differences of about 7.5.

mode : {‘exponential’, ‘linear’}, optional

The mode used in defining the spacing between the multi-thresholding levels (see the `nlevels` keyword).

connectivity : {4, 8}, optional

The type of pixel connectivity used in determining how pixels are grouped into a detected source. The options are 4 or 8 (default). 4-connected pixels touch along their edges. 8-connected pixels touch along their edges or corners. For reference, SExtractor uses 8-connected pixels.

relabel : bool

If `True` (default), then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Returns

segment_image : `SegmentationImage`

A 2D segmentation image, with the same shape as `data`, where sources are marked by different positive integer values. A value of zero is reserved for the background.

See also:

`photutils.detect_sources()`

`detect_sources`

`photutils.segmentation.detect_sources(data, threshold, npixels, filter_kernel=None, connectivity=8)`

Detect sources above a specified threshold value in an image and return a `SegmentationImage` object.

Detected sources must have `npixels` connected pixels that are each greater than the `threshold` value. If the filtering option is used, then the `threshold` is applied to the filtered image.

This function does not deblend overlapping sources. First use this function to detect sources followed by `deblend_sources()` to deblend sources.

Parameters

data : array-like

The 2D array of the image.

threshold : float or array-like

The data value or pixel-wise data values to be used for the detection threshold. A 2D `threshold` must have the same shape as `data`. See `detect_threshold` for one way to create a `threshold` image.

npixels : int

The number of connected pixels, each greater than `threshold`, that an object must have to be detected. `npixels` must be a positive integer.

filter_kernel : array-like (2D) or `Kernel1D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel.

connectivity : {4, 8}, optional

The type of pixel connectivity used in determining how pixels are grouped into a detected source. The options are 4 or 8 (default). 4-connected pixels touch along their edges. 8-connected pixels touch along their edges or corners. For reference, SExtractor uses 8-connected pixels.

Returns

segment_image : SegmentationImage

A 2D segmentation image, with the same shape as data, where sources are marked by different positive integer values. A value of zero is reserved for the background.

See also:

`photutils.detection.detect_threshold()`, `photutils.segmentation.SegmentationImage`,
`photutils.segmentation.source_properties()`, `photutils.segmentation.deblend_sources()`

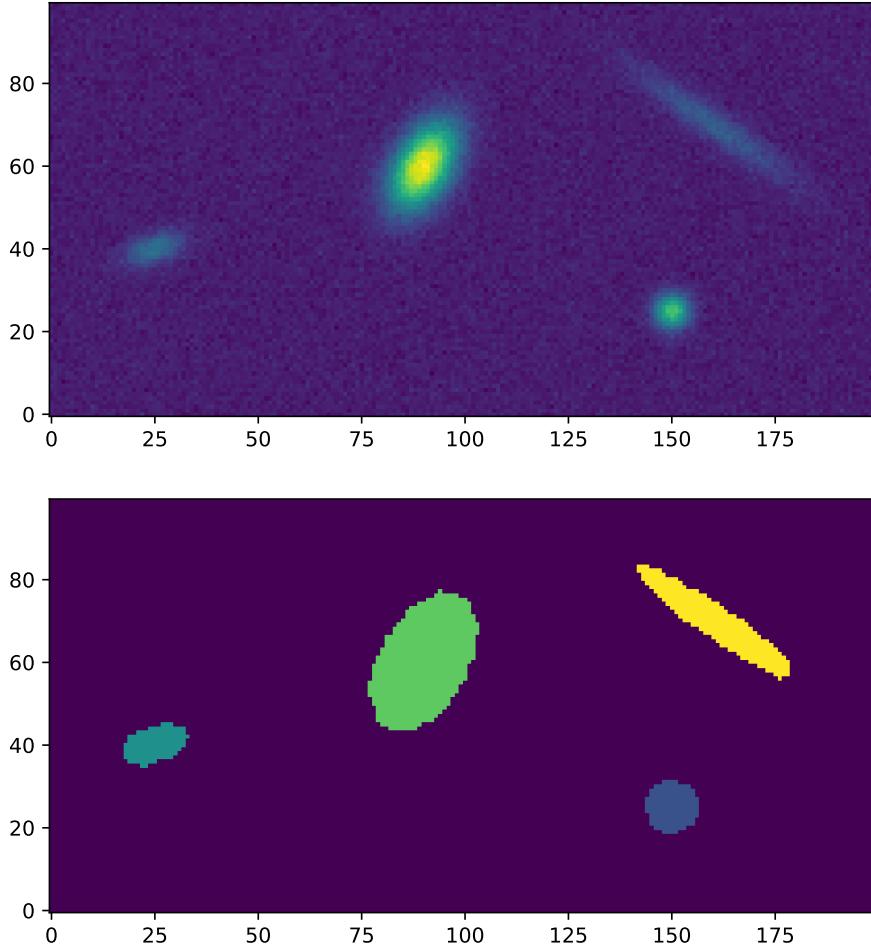
Examples

```
# make a table of Gaussian sources
from astropy.table import Table
table = Table()
table['amplitude'] = [50, 70, 150, 210]
table['x_mean'] = [160, 25, 150, 90]
table['y_mean'] = [70, 40, 25, 60]
table['x_stddev'] = [15.2, 5.1, 3., 8.1]
table['y_stddev'] = [2.6, 2.5, 3., 4.7]
table['theta'] = np.array([145., 20., 0., 60.]) * np.pi / 180.

# make an image of the sources with Gaussian noise
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_noise_image
shape = (100, 200)
sources = make_gaussian_sources(shape, table)
noise = make_noise_image(shape, type='gaussian', mean=0.,
                         stddev=5., random_state=12345)
image = sources + noise

# detect the sources
from photutils import detect_threshold, detect_sources
threshold = detect_threshold(image, snr=3)
from astropy.convolution import Gaussian2DKernel
sigma = 3.0 / (2.0 * np.sqrt(2.0 * np.log(2.0))) # FWHM = 3
kernel = Gaussian2DKernel(sigma, x_size=3, y_size=3)
kernel.normalize()
segm = detect_sources(image, threshold, npixels=5,
                      filter_kernel=kernel)

# plot the image and the segmentation image
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
ax1.imshow(image, origin='lower', interpolation='nearest')
ax2.imshow(segm.data, origin='lower', interpolation='nearest')
```



make_source_mask

```
photutils.segmentation.make_source_mask(data, snr, npixels, mask=None, mask_value=None, filter_fwhm=None, filter_size=3, filter_kernel=None, sigclip_sigma=3.0, sigclip_iters=5, dilate_size=11)
```

Make a source mask using source segmentation and binary dilation.

Parameters

data : array_like

The 2D array of the image.

snr : float

The signal-to-noise ratio per pixel above the background for which to consider a pixel as possibly being part of a source.

npixels : int

The number of connected pixels, each greater than `threshold`, that an object must have to be detected. `npixels` must be a positive integer.

mask : array_like, bool, optional

A boolean mask with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Masked pixels are ignored when computing the image background statistics.

mask_value : float, optional

An image data value (e.g., `0.0`) that is ignored when computing the image background statistics. `mask_value` will be ignored if `mask` is input.

filter_fwhm : float, optional

The full-width at half-maximum (FWHM) of the Gaussian kernel to filter the image before thresholding. `filter_fwhm` and `filter_size` are ignored if `filter_kernel` is defined.

filter_size : float, optional

The size of the square Gaussian kernel image. Used only if `filter_fwhm` is defined. `filter_fwhm` and `filter_size` are ignored if `filter_kernel` is defined.

filter_kernel : array-like (2D) or `Kerne12D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel. `filter_kernel` overrides `filter_fwhm` and `filter_size`.

sigclip_sigma : float, optional

The number of standard deviations to use as the clipping limit when calculating the image background statistics.

sigclip_iters : int, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence is achieved (i.e., continue until the last iteration clips nothing) when calculating the image background statistics.

dilate_size : int, optional

The size of the square array used to dilate the segmentation image.

Returns

mask : 2D `ndarray`, bool

A 2D boolean image containing the source mask.

properties_table

`photutils.segmentation.properties_table(source_props, columns=None, exclude_columns=None)`

Construct a `Table` of properties from a list of `SourceProperties` objects.

If `columns` or `exclude_columns` are not input, then the `Table` will include all scalar-valued properties. Multi-dimensional properties, e.g. `data_cutout`, can be included in the `columns` input.

Parameters

source_props : `SourceProperties` or list of `SourceProperties`

A `SourceProperties` object or list of `SourceProperties` objects, one for each source.

columns : str or list of str, optional

Names of columns, in order, to include in the output `Table`. The allowed column names are any of the attributes of `SourceProperties`.

exclude_columns : str or list of str, optional

Names of columns to exclude from the default properties list in the output `Table`. The default properties are those with scalar values.

Returns

table : `Table`

A table of properties of the segmented sources, one row per source.

See also:

`SegmentationImage`, `SourceProperties`, `source_properties`, `detect_sources`

Examples

```
>>> import numpy as np
>>> from photutils import source_properties, properties_table
>>> image = np.arange(16.).reshape(4, 4)
>>> print(image)
[[ 0.   1.   2.   3.]
 [ 4.   5.   6.   7.]
 [ 8.   9.  10.  11.]
 [12.  13.  14.  15.]]
>>> segm = SegmentationImage([[1, 1, 0, 0],
...                           [1, 0, 0, 2],
...                           [0, 0, 2, 2],
...                           [0, 2, 2, 0]])
>>> props = source_properties(image, segm)
>>> columns = ['id', 'xcentroid', 'ycentroid', 'source_sum']
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
      id    xcentroid     ycentroid  source_sum
      pix          pix
-----
 1       0.2        0.8       5.0
 2  2.09090909091  2.36363636364      55.0
```

source_properties

```
photutils.segmentation.source_properties(data, segment_img, error=None, mask=None, background=None, filter_kernel=None, wcs=None, labels=None)
```

Calculate photometry and morphological properties of sources defined by a labeled segmentation image.

Parameters

data : array_like or `Quantity`

The 2D array from which to calculate the source photometry and properties. `data` should be background-subtracted.

segment_img : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as data where sources are labeled by different positive integer values. A value of zero is reserved for the background.

error : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see `calc_total_error`). `.error` must have the same shape as the input data. See the Notes section below for details on the error propagation.

mask : array_like (bool), optional

A boolean mask with the same shape as data where a `True` value indicates the corresponding element of data is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was *previously* present in the input data. `background` may either be a scalar value or a 2D image with the same shape as the input data. Inputting the background merely allows for its properties to be measured within each source segment. The input background does *not* get subtracted from the input data, which should already be background-subtracted.

filter_kernel : array-like (2D) or `Kernel2D`, optional

The 2D array of the kernel used to filter the data prior to calculating the source centroid and morphological parameters. The kernel should be the same one used in defining the source segments (e.g., see `detect_sources()`). If `None`, then the unfiltered data will be used instead. Note that `SExtractor`'s centroid and morphological parameters are calculated from the filtered “detection” image.

wcs : `WCS`

The WCS transformation to use. If `None`, then `icrs_centroid`, `ra_icrs_centroid`, and `dec_icrs_centroid` will be `None`.

labels : int, array-like (1D, int)

Subset of segmentation labels for which to calculate the properties. If `None`, then the properties will be calculated for all labeled sources (the default).

Returns

output : list of `SourceProperties` objects

A list of `SourceProperties` objects, one for each source. The properties can be accessed as attributes or keys.

See also:

`SegmentationImage`, `SourceProperties`, `properties_table`, `detect_sources`

Notes

`SExtractor`'s centroid and morphological parameters are always calculated from the filtered “detection” image. The usual downside of the filtering is the sources will be made more circular than they actually are. If you wish to reproduce `SExtractor` results, then use the `filtered_data` input. If `filtered_data` is `None`, then the unfiltered data will be used for the source centroid and morphological parameters.

Negative (background-subtracted) data values within the source segment are set to zero when measuring morphological properties based on image moments. This could occur, for example, if the segmentation image was

defined from a different image (e.g., different bandpass) or if the background was oversubtracted. Note that `source_sum` includes the contribution of negative (background-subtracted) data values.

The input `error` is assumed to include *all* sources of error, including the Poisson error of the sources. `source_sum_err` is simply the quadrature sum of the pixel-wise total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, S are the non-masked pixels in the source segment, and $\sigma_{\text{tot},i}$ is the `error` array.

Examples

```
>>> import numpy as np
>>> from photutils import SegmentationImage, source_properties
>>> image = np.arange(16.).reshape(4, 4)
>>> print(image)
[[ 0.   1.   2.   3.]
 [ 4.   5.   6.   7.]
 [ 8.   9.  10.  11.]
 [12.  13.  14.  15.]]
>>> segm = SegmentationImage([[1, 1, 0, 0],
...                           [1, 0, 0, 2],
...                           [0, 0, 2, 2],
...                           [0, 2, 2, 0]])
>>> props = source_properties(image, segm)
```

Print some properties of the first object (labeled with 1 in the segmentation image):

```
>>> props[0].id      # id corresponds to segment label number
1
>>> props[0].centroid
<Quantity [ 0.8, 0.2] pix>
>>> props[0].source_sum
5.0
>>> props[0].area
<Quantity 3.0 pix2>
>>> props[0].max_value
4.0
```

Print some properties of the second object (labeled with 2 in the segmentation image):

```
>>> props[1].id      # id corresponds to segment label number
2
>>> props[1].centroid
<Quantity [ 2.36363636, 2.09090909] pix>
>>> props[1].perimeter
<Quantity 5.414213562373095 pix>
>>> props[1].orientation
<Quantity -0.7417593069227176 rad>
```

Classes

<code>SegmentationImage(data)</code>	Class for a segmentation image.
<code>SourceProperties(data, segment_img, label[, ...])</code>	Class to calculate photometry and morphological properties of a single labeled source.

SegmentationImage

```
class photutils.segmentation.SegmentationImage(data)
    Bases: object
```

Class for a segmentation image.

Parameters

`data` : array_like (int)

A 2D segmentation image where sources are labeled by different positive integer values.
A value of zero is reserved for the background.

Attributes Summary

<code>areas</code>	The areas (in pixel**2) of all labeled regions.
<code>array</code>	The 2D segmentation image.
<code>data</code>	The 2D segmentation image.
<code>data_masked</code>	A <code>MaskedArray</code> version of the segmentation image where the background (label = 0) has been masked.
<code>is_sequential</code>	Determine whether or not the non-zero labels in the segmentation image are sequential (with no missing values).
<code>labels</code>	The sorted non-zero labels in the segmentation image.
<code>max</code>	The maximum non-zero label in the segmentation image.
<code>nlabels</code>	The number of non-zero labels in the segmentation image.
<code>shape</code>	The shape of the 2D segmentation image.
<code>slices</code>	The minimal bounding box slices for each labeled region.

Methods Summary

<code>area(labels)</code>	The areas (in pixel**2) of the regions for the input labels.
<code>check_label(label[, allow_zero])</code>	Check for a valid label label number within the segmentation image.
<code>copy()</code>	Return a deep copy of this class instance.
<code>keep_labels(labels[, relabel])</code>	Keep only the specified label numbers.
<code>outline_segments([mask_background])</code>	Outline the labeled segments.
<code>relabel(labels, new_label)</code>	Relabel one or more label numbers.
<code>relabel_sequential([start_label])</code>	Relabel the label numbers sequentially, such that there are no missing label numbers (up to the maximum label number).
<code>remove_border_labels(border_width[, ...])</code>	Remove labeled segments near the image border.

Continued on next page

Table 11.4 – continued from previous page

<code>remove_labels(labels[, relabel])</code>	Remove one or more label numbers.
<code>remove_masked_labels(mask[, ...])</code>	Remove labeled segments located within a masked region.

Attributes Documentation

`areas`

The areas (in pixel $^{**}2$) of all labeled regions.

`array`

The 2D segmentation image.

`data`

The 2D segmentation image.

`data_masked`

A `MaskedArray` version of the segmentation image where the background (label = 0) has been masked.

`is_sequential`

Determine whether or not the non-zero labels in the segmenation image are sequential (with no missing values).

`labels`

The sorted non-zero labels in the segmentation image.

`max`

The maximum non-zero label in the segmentation image.

`nlabels`

The number of non-zero labels in the segmentation image.

`shape`

The shape of the 2D segmentation image.

`slices`

The minimal bounding box slices for each labeled region.

Methods Documentation

`area(labels)`

The areas (in pixel $^{**}2$) of the regions for the input labels.

Parameters

`labels` : int, array-like (1D, int)

The label(s) for which to return areas.

Returns

`areas` : `ndarray`

The areas of the labeled regions.

`check_label(label, allow_zero=False)`

Check for a valid label label number within the segmentation image.

Parameters

`label` : int

The label number to check.

allow_zero : bool

If `True` then a label of 0 is valid, otherwise 0 is invalid.

Raises

`ValueError`

If the input label is invalid.

`copy()`

Return a deep copy of this class instance.

Deep copy is used so that all attributes and values are copied.

`keep_labels(labels, relabel=False)`

Keep only the specified label numbers.

Parameters

`labels` : int, array-like (1D, int)

The label number(s) to keep. Labels of zero and those not in the segmentation image will be ignored.

`relabel` : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.keep_labels(labels=3)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.keep_labels(labels=[5, 3])
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [0, 0, 0, 0, 0, 5],
       [0, 0, 0, 5, 5, 5],
       [0, 0, 0, 0, 5, 5]])
```

```
outline_segments(mask_background=False)
    Outline the labeled segments.
```

The “outlines” represent the pixels *just inside* the segments, leaving the background pixels unmodified. This corresponds to the mode='inner' in `skimage.segmentation.find_boundaries`.

Parameters

mask_background : bool, optional

Set to `True` to mask the background pixels (labels = 0) in the returned image. This is useful for overplotting the segment outlines on an image. The default is `False`.

Returns

boundaries : 2D ndarray or MaskedArray

An image with the same shape of the segmentation image containing only the outlines of the labeled segments. The pixel values in the outlines correspond to the labels in the segmentation image. If `mask_background` is `True`, then a `MaskedArray` is returned.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[0, 0, 0, 0, 0, 0],
...                           [0, 2, 2, 2, 2, 0],
...                           [0, 2, 2, 2, 2, 0],
...                           [0, 2, 2, 2, 2, 0],
...                           [0, 2, 2, 2, 2, 0],
...                           [0, 0, 0, 0, 0, 0]])
>>> segm.outline_segments()
array([[0, 0, 0, 0, 0, 0],
       [0, 2, 2, 2, 2, 0],
       [0, 2, 0, 0, 2, 0],
       [0, 2, 0, 0, 2, 0],
       [0, 2, 2, 2, 2, 0],
       [0, 0, 0, 0, 0, 0]])
```

relabel(labels, new_label)

Relabel one or more label numbers.

The input `labels` will all be relabeled to `new_label`.

Parameters

labels : int, array-like (1D, int)

The label numbers(s) to relabel.

new_label : int

The relabeled label number.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
```

```

...
[7, 0, 0, 0, 0, 5],
...
[7, 7, 0, 5, 5, 5],
...
[7, 7, 0, 0, 5, 5])
>>> segm.relabel(labels=[1, 7], new_label=2)
>>> segm.data
array([[2, 2, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 3, 3, 0, 0],
       [2, 0, 0, 0, 0, 5],
       [2, 2, 0, 5, 5, 5],
       [2, 2, 0, 0, 5, 5]])

```

relabel_sequential(*start_label*=1)

Relabel the label numbers sequentially, such that there are no missing label numbers (up to the maximum label number).

Parameters

start_label : int, optional

The starting label number, which should be a positive integer. The default is 1.

Examples

```

>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.relabel_sequential()
>>> segm.data
array([[1, 1, 0, 0, 3, 3],
       [0, 0, 0, 0, 0, 3],
       [0, 0, 2, 2, 0, 0],
       [5, 0, 0, 0, 0, 4],
       [5, 5, 0, 4, 4, 4],
       [5, 5, 0, 0, 4, 4]])

```

remove_border_labels(*border_width*, *partial_overlap*=True, *relabel*=False)

Remove labeled segments near the image border.

Labels within the defined border region will be removed.

Parameters

border_width : int

The width of the border region in pixels.

partial_overlap : bool, optional

If this is set to `True` (the default), a segment that partially extends into the border region will be removed. Segments that are completely within the border region are always removed.

relabel : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_border_labels(border_width=1)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_border_labels(border_width=1,
...                             partial_overlap=False)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [7, 0, 0, 0, 0, 5],
       [7, 7, 0, 5, 5, 5],
       [7, 7, 0, 0, 5, 5]])
```

`remove_labels(labels, relabel=False)`

Remove one or more label numbers.

Parameters

`labels` : int, array-like (1D, int)

The label number(s) to remove. Labels of zero and those not in the segmentation image will be ignored.

`relabel` : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
```

```
>>> segm.remove_labels(labels=5)
>>> segm.data
array([[1, 1, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 3, 3, 0, 0],
       [7, 0, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_labels(labels=[5, 3])
>>> segm.data
array([[1, 1, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 0, 0, 0, 0],
       [7, 0, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0]])
```

`remove_masked_labels(mask, partial_overlap=True, relabel=False)`

Remove labeled segments located within a masked region.

Parameters

`mask` : array_like (bool)

A boolean mask, with the same shape as the segmentation image (`.data`), where `True` values indicate masked pixels.

`partial_overlap` : bool, optional

If this is set to `True` (the default), a segment that partially extends into a masked region will also be removed. Segments that are completely within a masked region are always removed.

`relabel` : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> mask = np.zeros_like(segm.data, dtype=np.bool)
>>> mask[0, :] = True      # mask the first row
>>> segm.remove_masked_labels(mask)
>>> segm.data
```

```
array([[0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0],  
       [0, 0, 3, 3, 0, 0],  
       [7, 0, 0, 0, 0, 5],  
       [7, 7, 0, 5, 5, 5],  
       [7, 7, 0, 0, 5, 5]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],  
...                                [0, 0, 0, 0, 0, 4],  
...                                [0, 0, 3, 3, 0, 0],  
...                                [7, 0, 0, 0, 0, 5],  
...                                [7, 7, 0, 5, 5, 5],  
...                                [7, 7, 0, 0, 5, 5]])  
>>> segm.remove_masked_labels(mask, partial_overlap=False)  
>>> segm.data  
array([[0, 0, 0, 0, 4, 4],  
       [0, 0, 0, 0, 0, 4],  
       [0, 0, 3, 3, 0, 0],  
       [7, 0, 0, 0, 0, 5],  
       [7, 7, 0, 5, 5, 5],  
       [7, 7, 0, 0, 5, 5]])
```

SourceProperties

```
class photutils.segmentation.SourceProperties(data, segment_img, label, filtered_data=None,  
                                              error=None, mask=None, background=None,  
                                              wcs=None)
```

Bases: object

Class to calculate photometry and morphological properties of a single labeled source.

Parameters

data : array_like or `Quantity`

The 2D array from which to calculate the source photometry and properties. If `filtered_data` is input, then it will be used instead of `data` to calculate the source centroid and morphological properties. Source photometry is always measured from `data`. For accurate source properties and photometry, `data` should be background-subtracted.

segment_img : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as `data` where sources are labeled by different positive integer values. A value of zero is reserved for the background.

label : int

The label number of the source whose properties to calculate.

filtered_data : array-like or `Quantity`, optional

The filtered version of the background-subtracted data from which to calculate the source centroid and morphological properties. The kernel used to perform the filtering should be the same one used in defining the source segments (e.g., see `detect_sources()`). If `None`, then the unfiltered data will be used instead. Note that SExtractor's centroid and morphological parameters are calculated from the filtered "detection" image.

error : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see `calc_total_error`). `.error` must have the same shape as the input data. See the Notes section below for details on the error propagation.

mask : array_like (bool), optional

A boolean mask with the same shape as `data` where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was *previously* present in the input data. `background` may either be a scalar value or a 2D image with the same shape as the input data. Inputting the background merely allows for its properties to be measured within each source segment. The input background does *not* get subtracted from the input data, which should already be background-subtracted.

wcs : `WCS`

The WCS transformation to use. If `None`, then `icrs_centroid`, `ra_icrs_centroid`, and `dec_icrs_centroid` will be `None`.

Notes

`SExtractor`'s centroid and morphological parameters are always calculated from the filtered “detection” image. The usual downside of the filtering is the sources will be made more circular than they actually are. If you wish to reproduce `SExtractor` results, then use the `filtered_data` input. If `filtered_data` is `None`, then the unfiltered data will be used for the source centroid and morphological parameters.

Negative (background-subtracted) data values within the source segment are set to zero when measuring morphological properties based on image moments. This could occur, for example, if the segmentation image was defined from a different image (e.g., different bandpass) or if the background was oversubtracted. Note that `source_sum` includes the contribution of negative (background-subtracted) data values.

The input `error` is assumed to include *all* sources of error, including the Poisson error of the sources. `source_sum_err` is simply the quadrature sum of the pixel-wise total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, S are the non-masked pixels in the source segment, and $\sigma_{\text{tot},i}$ is the input error array.

Custom errors for source segments can be calculated using the `error_cutout_ma` and `background_cutout_ma` properties, which are 2D `MaskedArray` cutout versions of the input `error` and `background`. The mask is `True` for both pixels outside of the source segment and masked pixels from the `mask` input.

Attributes Summary

<code>area</code>	The area of the source segment in units of pixels**2.
<code>background_at_centroid</code>	The value of the background at the position of the source centroid.

Continued on next page

Table 11.5 – continued from previous page

<code>background_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the input background, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.
<code>background_mean</code>	The mean of background values within the source segment.
<code>background_sum</code>	The sum of background values within the source segment.
<code>bbox</code>	The bounding box (<code>ymin</code> , <code>xmin</code> , <code>ymax</code> , <code>xmax</code>) of the minimal rectangular region containing the source segment.
<code>centroid</code>	The (<code>y</code> , <code>x</code>) coordinate of the centroid within the source segment.
<code>coords</code>	A tuple of <code>ndarrays</code> containing the <code>y</code> and <code>x</code> pixel coordinates of the source segment.
<code>covar_sigx2</code>	The $(0, 0)$ element of the <code>covariance</code> matrix, representing σ_x^2 , in units of <code>pixel**2</code> .
<code>covar_sigxy</code>	The $(0, 1)$ and $(1, 0)$ elements of the <code>covariance</code> matrix, representing $\sigma_x \sigma_y$, in units of <code>pixel**2</code> .
<code>covar_sigy2</code>	The $(1, 1)$ element of the <code>covariance</code> matrix, representing σ_y^2 , in units of <code>pixel**2</code> .
<code>covariance</code>	The covariance matrix of the 2D Gaussian function that has the same second-order moments as the source.
<code>covariance_eigvals</code>	The two eigenvalues of the <code>covariance</code> matrix in decreasing order.
<code>cutout_centroid</code>	The (<code>y</code> , <code>x</code>) coordinate, relative to the <code>data_cutout</code> , of the centroid within the source segment.
<code>cxx</code>	<code>SExtractor</code> 's CXX ellipse parameter in units of <code>pixel**(-2)</code> .
<code>cxy</code>	<code>SExtractor</code> 's CXY ellipse parameter in units of <code>pixel**(-2)</code> .
<code>cyy</code>	<code>SExtractor</code> 's CYY ellipse parameter in units of <code>pixel**(-2)</code> .
<code>data_cutout</code>	A 2D cutout from the (background-subtracted) data of the source segment.
<code>data_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the (background-subtracted) data, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.
<code>dec_icrs_centroid</code>	The ICRS Declination coordinate (in degrees) of the centroid within the source segment.
<code>eccentricity</code>	The eccentricity of the 2D Gaussian function that has the same second-order moments as the source.
<code>ellipticity</code>	1 minus the ratio of the lengths of the semimajor and
<code>elongation</code>	The ratio of the lengths of the semimajor and semiminor axes:
<code>equivalent_radius</code>	The radius of a circle with the same <code>area</code> as the source segment.
<code>error_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the input error image, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.

Continued on next page

Table 11.5 – continued from previous page

<code>icrs_centroid</code>	The International Celestial Reference System (ICRS) coordinates of the centroid within the source segment, returned as a <code>SkyCoord</code> object.
<code>id</code>	The source identification number corresponding to the object label in the segmentation image.
<code>inertia_tensor</code>	The inertia tensor of the source for the rotation around its center of mass.
<code>max_value</code>	The maximum pixel value of the (background-subtracted) data within the source segment.
<code>maxval_cutout_pos</code>	The (y, x) coordinate, relative to the <code>data_cutout</code> , of the maximum pixel value of the (background-subtracted) data.
<code>maxval_pos</code>	The (y, x) coordinate of the maximum pixel value of the (background-subtracted) data.
<code>maxval_xpos</code>	The x coordinate of the maximum pixel value of the (background-subtracted) data.
<code>maxval_ypos</code>	The y coordinate of the maximum pixel value of the (background-subtracted) data.
<code>min_value</code>	The minimum pixel value of the (background-subtracted) data within the source segment.
<code>minval_cutout_pos</code>	The (y, x) coordinate, relative to the <code>data_cutout</code> , of the minimum pixel value of the (background-subtracted) data.
<code>minval_pos</code>	The (y, x) coordinate of the minimum pixel value of the (background-subtracted) data.
<code>minval_xpos</code>	The x coordinate of the minimum pixel value of the (background-subtracted) data.
<code>minval_ypos</code>	The y coordinate of the minimum pixel value of the (background-subtracted) data.
<code>moments</code>	Spatial moments up to 3rd order of the source.
<code>moments_central</code>	Central moments (translation invariant) of the source up to 3rd order.
<code>orientation</code>	The angle in radians between the x axis and the major axis of the 2D Gaussian function that has the same second-order moments as the source.
<code>perimeter</code>	The perimeter of the source segment, approximated lines through the centers of the border pixels using a 4-connectivity.
<code>ra_icrs_centroid</code>	The ICRS Right Ascension coordinate (in degrees) of the centroid within the source segment.
<code>semimajor_axis_sigma</code>	The 1-sigma standard deviation along the semimajor axis of the 2D Gaussian function that has the same second-order central moments as the source.
<code>semiminor_axis_sigma</code>	The 1-sigma standard deviation along the semiminor axis of the 2D Gaussian function that has the same second-order central moments as the source.
<code>source_sum</code>	The sum of the non-masked (background-subtracted) data values within the source segment.
<code>source_sum_err</code>	The uncertainty of <code>source_sum</code> , propagated from the input error array.

Continued on next page

Table 11.5 – continued from previous page

values	A <code>ndarray</code> of the (background-subtracted) pixel values within the source segment.
xcentroid	The x coordinate of the centroid within the source segment.
xmax	The maximum x pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
xmin	The minimum x pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
ycentroid	The y coordinate of the centroid within the source segment.
ymax	The maximum y pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
ymin	The minimum y pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.

Methods Summary

<code>make_cutout(data[, masked_array])</code>	Create a (masked) cutout array from the input data using the minimal bounding box of the source segment.
<code>to_table([columns, exclude_columns])</code>	Create a <code>Table</code> of properties.

Attributes Documentation

`area`

The area of the source segment in units of pixels**2.

`background_at_centroid`

The value of the background at the position of the source centroid. Fractional position values are determined using bilinear interpolation.

`background_cutout_ma`

A 2D `MaskedArray` cutout from the input background, where the mask is `True` for both pixels outside of the source segment and masked pixels. If `background` is `None`, then `background_cutout_ma` is also `None`.

`background_mean`

The mean of background values within the source segment.

`background_sum`

The sum of background values within the source segment.

`bbox`

The bounding box (`ymin`, `xmin`, `ymax`, `xmax`) of the minimal rectangular region containing the source segment.

`centroid`

The (`y`, `x`) coordinate of the centroid within the source segment.

`coords`

A tuple of `ndarrays` containing the `y` and `x` pixel coordinates of the source segment. Masked pixels are not included.

`covar_sigx2`

The $(0, 0)$ element of the `covariance` matrix, representing σ_x^2 , in units of pixel**2.

Note that this is the same as SExtractor’s X2 parameter.

covar_sigxy

The (0, 1) and (1, 0) elements of the covariance matrix, representing $\sigma_x\sigma_y$, in units of pixel**2.

Note that this is the same as SExtractor’s XY parameter.

covar_sigy2

The (1, 1) element of the covariance matrix, representing σ_y^2 , in units of pixel**2.

Note that this is the same as SExtractor’s Y2 parameter.

covariance

The covariance matrix of the 2D Gaussian function that has the same second-order moments as the source.

covariance_eigvals

The two eigenvalues of the covariance matrix in decreasing order.

cutout_centroid

The (y, x) coordinate, relative to the data_cutout, of the centroid within the source segment.

cxx

SExtractor’s CXX ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$cxx(x - \bar{x})^2 + cxy(x - \bar{x})(y - \bar{y}) + cyy(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). SExtractor reports that the isophotal limit of a source is well represented by $R \approx 3$.

cxy

SExtractor’s CXY ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$cxx(x - \bar{x})^2 + cxy(x - \bar{x})(y - \bar{y}) + cyy(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). SExtractor reports that the isophotal limit of a source is well represented by $R \approx 3$.

cyy

SExtractor’s CYY ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$cxx(x - \bar{x})^2 + cxy(x - \bar{x})(y - \bar{y}) + cyy(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). SExtractor reports that the isophotal limit of a source is well represented by $R \approx 3$.

data_cutout

A 2D cutout from the (background-subtracted) data of the source segment.

data_cutout_ma

A 2D MaskedArray cutout from the (background-subtracted) data, where the mask is True for both pixels outside of the source segment and masked pixels.

dec_icrs_centroid

The ICRS Declination coordinate (in degrees) of the centroid within the source segment.

eccentricity

The eccentricity of the 2D Gaussian function that has the same second-order moments as the source.

The eccentricity is the fraction of the distance along the semimajor axis at which the focus lies.

$$e = \sqrt{1 - \frac{b^2}{a^2}}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

ellipticity

1 minus the ratio of the lengths of the semimajor and semiminor axes (or 1 minus the `elongation`):

$$\text{ellipticity} = 1 - \frac{b}{a}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

Note that this is the same as `SExtractor`'s ellipticity parameter.

elongation

The ratio of the lengths of the semimajor and semiminor axes:

$$\text{elongation} = \frac{a}{b}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

Note that this is the same as `SExtractor`'s elongation parameter.

equivalent_radius

The radius of a circle with the same `area` as the source segment.

error_cutout_ma

A 2D `MaskedArray` cutout from the input error image, where the mask is `True` for both pixels outside of the source segment and masked pixels. If `error` is `None`, then `error_cutout_ma` is also `None`.

icrs_centroid

The International Celestial Reference System (ICRS) coordinates of the centroid within the source segment, returned as a `SkyCoord` object.

id

The source identification number corresponding to the object label in the segmentation image.

inertia_tensor

The inertia tensor of the source for the rotation around its center of mass.

max_value

The maximum pixel value of the (background-subtracted) data within the source segment.

maxval_cutout_pos

The (y , x) coordinate, relative to the `data_cutout`, of the maximum pixel value of the (background-subtracted) data.

maxval_pos

The (y , x) coordinate of the maximum pixel value of the (background-subtracted) data.

maxval_xpos

The x coordinate of the maximum pixel value of the (background-subtracted) data.

maxval_ypos

The y coordinate of the maximum pixel value of the (background-subtracted) data.

min_value

The minimum pixel value of the (background-subtracted) data within the source segment.

minval_cutout_pos

The (y , x) coordinate, relative to the `data_cutout`, of the minimum pixel value of the (background-subtracted) data.

minval_pos

The (y , x) coordinate of the minimum pixel value of the (background-subtracted) data.

minval_xpos

The x coordinate of the minimum pixel value of the (background-subtracted) data.

minval_ypos

The y coordinate of the minimum pixel value of the (background-subtracted) data.

moments

Spatial moments up to 3rd order of the source.

moments_central

Central moments (translation invariant) of the source up to 3rd order.

orientation

The angle in radians between the x axis and the major axis of the 2D Gaussian function that has the same second-order moments as the source. The angle increases in the counter-clockwise direction.

perimeter

The perimeter of the source segment, approximated lines through the centers of the border pixels using a 4-connectivity.

ra_icrs_centroid

The ICRS Right Ascension coordinate (in degrees) of the centroid within the source segment.

semimajor_axis_sigma

The 1-sigma standard deviation along the semimajor axis of the 2D Gaussian function that has the same second-order central moments as the source.

semiminor_axis_sigma

The 1-sigma standard deviation along the semiminor axis of the 2D Gaussian function that has the same second-order central moments as the source.

source_sum

The sum of the non-masked (background-subtracted) data values within the source segment.

$$F = \sum_{i \in S} (I_i - B_i)$$

where F is `source_sum`, $(I_i - B_i)$ is the background-subtracted input data, and S are the non-masked pixels in the source segment.

source_sum_err

The uncertainty of `source_sum`, propagated from the input `error` array.

`source_sum_err` is the quadrature sum of the total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, $\sigma_{\text{tot},i}$ are the pixel-wise total errors, and S are the non-masked pixels in the source segment.

values

A `ndarray` of the (background-subtracted) pixel values within the source segment. Masked pixels are not included.

xcentroid

The x coordinate of the centroid within the source segment.

xmax

The maximum x pixel location of the minimal bounding box (`bbox`) of the source segment.

xmin

The minimum x pixel location of the minimal bounding box (`bbox`) of the source segment.

ycentroid

The y coordinate of the centroid within the source segment.

ymax

The maximum y pixel location of the minimal bounding box (`bbox`) of the source segment.

ymin

The minimum y pixel location of the minimal bounding box (`bbox`) of the source segment.

Methods Documentation

make_cutout(data, masked_array=False)

Create a (masked) cutout array from the input data using the minimal bounding box of the source segment.

Parameters

data : array-like (2D)

The data array from which to create the masked cutout array. `data` must have the same shape as the segmentation image input into `SourceProperties`.

masked_array : bool, optional

If `True` then a `MaskedArray` will be created where the mask is `True` for both pixels outside of the source segment and any masked pixels. If `False`, then a `ndarray` will be generated.

Returns

result : `ndarray` or `MaskedArray` (2D)

The 2D cutout array or masked array.

to_table(columns=None, exclude_columns=None)

Create a `Table` of properties.

If `columns` or `exclude_columns` are not input, then the `Table` will include all scalar-valued properties. Multi-dimensional properties, e.g. `data_cutout`, can be included in the `columns` input.

Parameters

columns : str or list of str, optional

Names of columns, in order, to include in the output `Table`. The allowed column names are any of the attributes of `SourceProperties`.

exclude_columns : str or list of str, optional

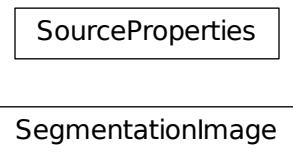
Names of columns to exclude from the default properties list in the output `Table`. The default properties are those with scalar values.

Returns

`table` : [Table](#)

A single-row table of properties of the source.

Class Inheritance Diagram



CHAPTER 12

Centroids (`photutils.centroids`)

Introduction

`photutils.centroids` provides several functions to calculate the centroid of a single source. The centroid methods are:

- `centroid_com()`: Calculates the object “center of mass” from 2D image moments.
- `centroid_1dg()`: Calculates the centroid by fitting 1D Gaussians to the marginal x and y distributions of the data.
- `centroid_2dg()`: Calculates the centroid by fitting a 2D Gaussian to the 2D distribution of the data.

Masks can be input into each of these functions to mask bad pixels. Error arrays can be input into the two fitting methods to weight the fits.

Getting Started

Let’s extract a single object from a synthetic dataset and find its centroid with each of these methods. For this simple example we will not subtract the background from the data (but in practice, one should subtract the background):

```
>>> from photutils.datasets import make_4gaussians_image
>>> from photutils import centroid_com, centroid_1dg, centroid_2dg
>>> data = make_4gaussians_image()[43:79, 76:104]
```

```
>>> x1, y1 = centroid_com(data)
>>> print((x1, y1))
(13.93157998341213, 17.051234441067088)
```

```
>>> x2, y2 = centroid_1dg(data)
>>> print((x2, y2))
(14.040352707371396, 16.962306463644801)
```

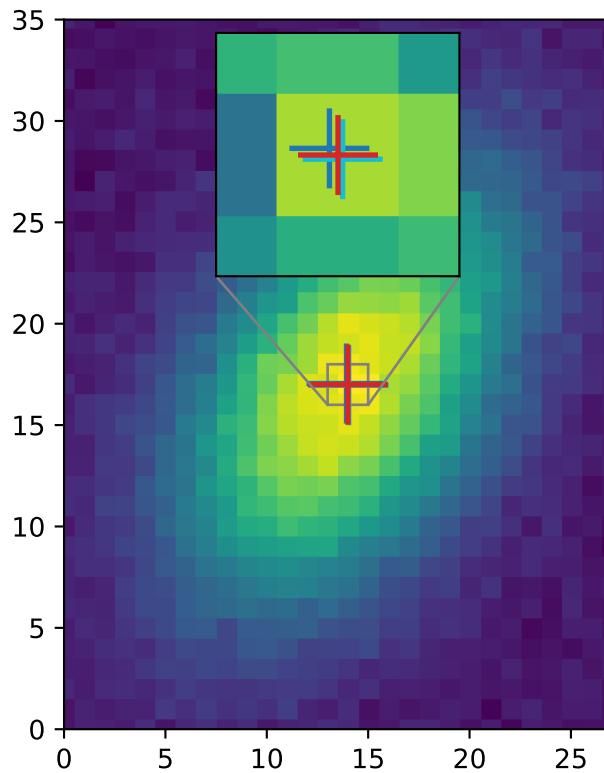
```
>>> x3, y3 = centroid_2dg(data)
>>> print((x3, y3))
(14.002212073733611, 16.996134592982017)
```

Now let's plot the results. Because the centroids are all very similar, we also include an inset plot zoomed in near the centroid:

```
from photutils.datasets import make_4gaussians_image
from photutils import centroid_com, centroid_1dg, centroid_2dg
import matplotlib.pyplot as plt

data = make_4gaussians_image()[43:79, 76:104]      # extract single object
x1, y1 = centroid_com(data)
x2, y2 = centroid_1dg(data)
x3, y3 = centroid_2dg(data)
fig, ax = plt.subplots(1, 1)
ax.imshow(data, origin='lower', interpolation='nearest', cmap='viridis')
marker = '+'
ms, mew = 30, 2.
plt.plot(x1, y1, color='#1f77b4', marker=marker, ms=ms, mew=mew)
plt.plot(x2, y2, color='#17becf', marker=marker, ms=ms, mew=mew)
plt.plot(x3, y3, color='#d62728', marker=marker, ms=ms, mew=mew)

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
ax2 = zoomed_inset_axes(ax, zoom=6, loc=9)
ax2.imshow(data, interpolation='nearest', origin='lower',
           cmap='viridis', vmin=190, vmax=220)
ax2.plot(x1, y1, color='#1f77b4', marker=marker, ms=ms, mew=mew)
ax2.plot(x2, y2, color='#17becf', marker=marker, ms=ms, mew=mew)
ax2.plot(x3, y3, color='#d62728', marker=marker, ms=ms, mew=mew)
ax2.set_xlim(13, 15)
ax2.set_ylim(16, 18)
mark_inset(ax, ax2, loc1=3, loc2=4, fc='none', ec='0.5')
ax2.axes.get_xaxis().set_visible(False)
ax2.axes.get_yaxis().set_visible(False)
ax.set_xlim(0, data.shape[1]-1)
ax.set_ylim(0, data.shape[0]-1)
```



Reference/API

This subpackage contains tools for centroiding objects in an astronomical image.

Functions

<code>centroid_1dg(data[, error, mask])</code>	Calculate the centroid of a 2D array by fitting 1D Gaussians to the marginal x and y distributions of the array.
<code>centroid_2dg(data[, error, mask])</code>	Calculate the centroid of a 2D array by fitting a 2D Gaussian (plus a constant) to the array.
<code>centroid_com(data[, mask])</code>	Calculate the centroid of a 2D array as its “center of mass” determined from image moments.
<code>fit_2dgaussian(data[, error, mask])</code>	Fit a 2D Gaussian plus a constant to a 2D image.
<code>gaussian1d_moments(data[, mask])</code>	Estimate 1D Gaussian parameters from the moments of 1D data.

`centroid_1dg`

`photutils.centroids.centroid_1dg(data, error=None, mask=None)`

Calculate the centroid of a 2D array by fitting 1D Gaussians to the marginal x and y distributions of the array.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters

data : array_like

The 2D data array.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : ndarray

The x, y coordinates of the centroid.

centroid_2dg

`photutils.centroids.centroid_2dg(data, error=None, mask=None)`

Calculate the centroid of a 2D array by fitting a 2D Gaussian (plus a constant) to the array.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters

data : array_like

The 2D data array.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : ndarray

The x, y coordinates of the centroid.

centroid_com

`photutils.centroids.centroid_com(data, mask=None)`

Calculate the centroid of a 2D array as its “center of mass” determined from image moments.

Invalid values (e.g. NaNs or infs) in the data array are automatically masked.

Parameters

data : array_like

The 2D array of the image.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : `ndarray`

The x, y coordinates of the centroid.

fit_2dgaussian

`photutils.centroids.fit_2dgaussian(data, error=None, mask=None)`

Fit a 2D Gaussian plus a constant to a 2D image.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters

data : array_like

The 2D array of the image.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

result : A `GaussianConst2D` model instance.

The best-fitting Gaussian 2D model.

gaussian1d_moments

`photutils.centroids.gaussian1d_moments(data, mask=None)`

Estimate 1D Gaussian parameters from the moments of 1D data.

This function can be useful for providing initial parameter values when fitting a 1D Gaussian to the data.

Parameters

data : array_like (1D)

The 1D array.

mask : array_like (1D bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

amplitude, mean, stddev : float

The estimated parameters of a 1D Gaussian.

Classes

GaussianConst2D

A model for a 2D Gaussian plus a constant.

GaussianConst2D

class photutils.centroids.**GaussianConst2D**
Bases: astropy.modeling.Fittable2DModel

A model for a 2D Gaussian plus a constant.

Parameters

constant : float

Value of the constant.

amplitude : float

Amplitude of the Gaussian.

x_mean : float

Mean of the Gaussian in x.

y_mean : float

Mean of the Gaussian in y.

x_stddev : float

Standard deviation of the Gaussian in x. **x_stddev** and **y_stddev** must be specified unless a covariance matrix (**cov_matrix**) is input.

y_stddev : float

Standard deviation of the Gaussian in y. **x_stddev** and **y_stddev** must be specified unless a covariance matrix (**cov_matrix**) is input.

theta : float, optional

Rotation angle in radians. The rotation angle increases counterclockwise.

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the **fixed** property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the **tied** property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the **min** and **max** properties of a parameter may be used.

eqcons : list

A list of functions of length n such that eqcons[j](x0,*args) == 0.0 in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that `ineqcons[j](x0,*args) >= 0.0` is a successfully optimized problem.

Attributes Summary

`amplitude`

`constant`

`param_names`

`theta`

`x_mean`

`x_stddev`

`y_mean`

`y_stddev`

Methods Summary

`evaluate(x, y, constant, amplitude, x_mean, ...)`

Two dimensional Gaussian plus constant function.

Attributes Documentation

amplitude

constant

param_names = ('constant', 'amplitude', 'x_mean', 'y_mean', 'x_stddev', 'y_stddev', 'theta')

theta

x_mean

x_stddev

y_mean

y_stddev

Methods Documentation

static evaluate(x, y, constant, amplitude, x_mean, y_mean, x_stddev, y_stddev, theta)

Two dimensional Gaussian plus constant function.

Class Inheritance Diagram



CHAPTER 13

Morphological Properties (`photutils.morphology`)

Introduction

The `data_properties()` function can be used to calculate the morphological properties of a single source in a cutout image. `data_properties` returns a `SourceProperties` object. Please see [SourceProperties](#) for the list of the many properties that are calculated. Even more properties are likely to be added in the future.

If you have a segmentation image, the `source_properties()` function can be used to calculate the properties for all (or a specified subset) of the segmented sources. Please see [Source Photometry and Properties from Image Segmentation](#) for more details.

Getting Started

Let's extract a single object from a synthetic dataset and find calculate its morphological properties. For this example, we will subtract the background using simple sigma-clipped statistics.

First, we create the source image and subtract its background:

```
>>> from photutils.datasets import make_4gaussians_image
>>> from astropy.stats import sigma_clipped_stats
>>> data = make_4gaussians_image()[43:79, 76:104]
>>> mean, median, std = sigma_clipped_stats(data, sigma=3.0, iters=5)
>>> data -= median    # subtract background
```

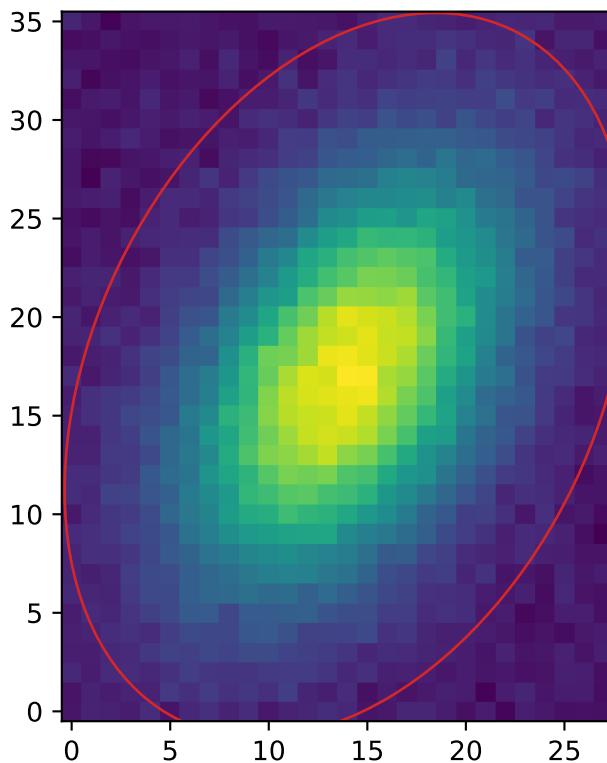
Then, calculate its properties:

```
>>> from photutils import data_properties, properties_table
>>> props = data_properties(data)
>>> columns = ['id', 'xcentroid', 'ycentroid', 'semimajor_axis_sigma',
...             'semiminor_axis_sigma', 'orientation']
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
   id      xcentroid     ycentroid ... semiminor_axis_sigma  orientation
```

	pix	pix	...	pix	rad
1	14.0225090502	16.9901801466	...	3.69777618702	1.04943689372

Now let's use the measured morphological properties to define an approximate isophotal ellipse for the source:

```
>>> from photutils import properties_table, EllipticalAperture
>>> position = (props.xcentroid.value, props.ycentroid.value)
>>> r = 3.0      # approximate isophotal extent
>>> a = props.semimajor_axis_sigma.value * r
>>> b = props.semiminor_axis_sigma.value * r
>>> theta = props.orientation.value
>>> apertures = EllipticalAperture(position, a, b, theta=theta)
>>> plt.imshow(data, origin='lower', cmap='viridis',
...             interpolation='nearest')
>>> apertures.plot(color='#d62728')
```



Reference/API

This subpackage contains tools for measuring morphological properties of objects in an astronomical image.

Functions

<code>data_properties(data[, mask, background])</code>	Calculate the morphological properties (and centroid) of a 2D array (e.g.
<code>gini(data)</code>	Calculate the Gini coefficient of a 2D array.

`data_properties`

`photutils.morphology.data_properties(data, mask=None, background=None)`

Calculate the morphological properties (and centroid) of a 2D array (e.g. an image cutout of an object) using image moments.

Parameters

data : array_like or `Quantity`

The 2D array of the image.

mask : array_like (bool), optional

A boolean mask, with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was previously present in the input data. `background` may either be a scalar value or a 2D image with the same shape as the input data. Inputting the background merely allows for its properties to be measured within each source segment. The input background does *not* get subtracted from the input data, which should already be background-subtracted.

Returns

result : `SourceProperties` instance

A `SourceProperties` object.

`gini`

`photutils.morphology.gini(data)`

Calculate the Gini coefficient of a 2D array.

The Gini coefficient is calculated using the prescription from Lotz et al. 2004 as:

$$G = \frac{1}{|\bar{x}| n(n-1)} \sum_i^n (2i - n - 1) |x_i|$$

where \bar{x} is the mean over all pixel values x_i .

The Gini coefficient is a way of measuring the inequality in a given set of values. In the context of galaxy morphology, it measures how the light of a galaxy image is distributed among its pixels. A G value of 0 corresponds to a galaxy image with the light evenly distributed over all pixels while a G value of 1 represents a galaxy image with all its light concentrated in just one pixel.

Usually Gini's measurement needs some sort of preprocessing for defining the galaxy region in the image based on the quality of the input data. As there is not a general standard for doing this, this is left for the user.

Parameters

data : array-like

The 2D data array or object that can be converted to an array.

Returns

gini : float

The Gini coefficient of the input 2D array.

CHAPTER 14

Geometry Functions (`photutils.geometry`)

Introduction

The `photutils.geometry` package contains low-level geometry functions used mainly by `aperture_photometry`.

Reference/API

Geometry subpackage for low-level geometry functions.

Functions

<code>circular_overlap_grid(xmin, xmax, ymin, ...)</code>	Area of overlap between a circle and a pixel grid.
<code>elliptical_overlap_grid(xmin, xmax, ymin, ...)</code>	Area of overlap between an ellipse and a pixel grid.
<code>rectangular_overlap_grid(xmin, xmax, ymin, ...)</code>	Area of overlap between a rectangle and a pixel grid.

`circular_overlap_grid`

`photutils.geometry.circular_overlap_grid(xmin, xmax, ymin, ymax, nx, ny, r, use_exact, subpixels)`
Area of overlap between a circle and a pixel grid. The circle is centered on the origin.

Parameters

`xmin, xmax, ymin, ymax` : float

Extent of the grid in the x and y direction.

`nx, ny` : int

Grid dimensions.

`r` : float

The radius of the circle.

use_exact : 0 or 1

If 1 calculates exact overlap, if 0 uses subpixel number of subpixels to calculate the overlap.

subpixels : int

Each pixel resampled by this factor in each dimension, thus each pixel is divided into $\text{subpixels} \times 2$ subpixels.

Returns

frac : `ndarray` (float)

2-d array of shape (ny, nx) giving the fraction of the overlap.

`elliptical_overlap_grid`

```
photutils.geometry.elliptical_overlap_grid(xmin, xmax, ymin, ymax, nx, ny, rx, ry, use_exact, subpixels)
```

Area of overlap between an ellipse and a pixel grid. The ellipse is centered on the origin.

Parameters

xmin, xmax, ymin, ymax : float

Extent of the grid in the x and y direction.

nx, ny : int

Grid dimensions.

rx : float

The semimajor axis of the ellipse.

ry : float

The semiminor axis of the ellipse.

theta : float

The position angle of the semimajor axis in radians (counterclockwise).

use_exact : 0 or 1

If set to 1, calculates the exact overlap, while if set to 0, uses a subpixel sampling method with subpixel subpixels in each direction.

subpixels : int

If use_exact is 0, each pixel is resampled by this factor in each dimension. Thus, each pixel is divided into $\text{subpixels} \times 2$ subpixels.

Returns

frac : `ndarray`

2-d array giving the fraction of the overlap.

`rectangular_overlap_grid`

```
photutils.geometry.rectangular_overlap_grid(xmin, xmax, ymin, ymax, nx, ny, width, height, use_exact, subpixels)
```

Area of overlap between a rectangle and a pixel grid. The rectangle is centered on the origin.

Parameters**xmin, xmax, ymin, ymax** : float

Extent of the grid in the x and y direction.

nx, ny : int

Grid dimensions.

width : float

The width of the rectangle

height : float

The height of the rectangle

theta : float

The position angle of the rectangle in radians (counterclockwise).

use_exact : 0 or 1

If set to 1, calculates the exact overlap, while if set to 0, uses a subpixel sampling method with subpixel subpixels in each direction.

subpixels : intIf use_exact is 0, each pixel is resampled by this factor in each dimension. Thus, each pixel is divided into subpixels $\star\star$ 2 subpixels.**Returns****frac** : ndarray

2-d array giving the fraction of the overlap.

CHAPTER 15

Datasets (photutils.datasets)

Introduction

`photutils.datasets` gives easy access to a few example datasets (mostly images, but also e.g. source catalogs or PSF models).

This is useful for the Photutils documentation, tests, and benchmarks, but also for users that would like to try out Photutils functions or implement new methods for Photutils or their own scripts.

Functions that start with `load_*` load data files from disk. Very small data files are bundled in the Photutils code repository and are guaranteed to be available. Mid-sized data files are currently available from the `astropy-data` repository and loaded into the Astropy cache on the user's machine on first load.

Functions that start with `make_*` generate simple simulated data (e.g. Gaussian sources on flat background with Poisson or Gaussian noise). Note that there are other tools like `skymaker` that can simulate much more realistic astronomical images.

Getting Started

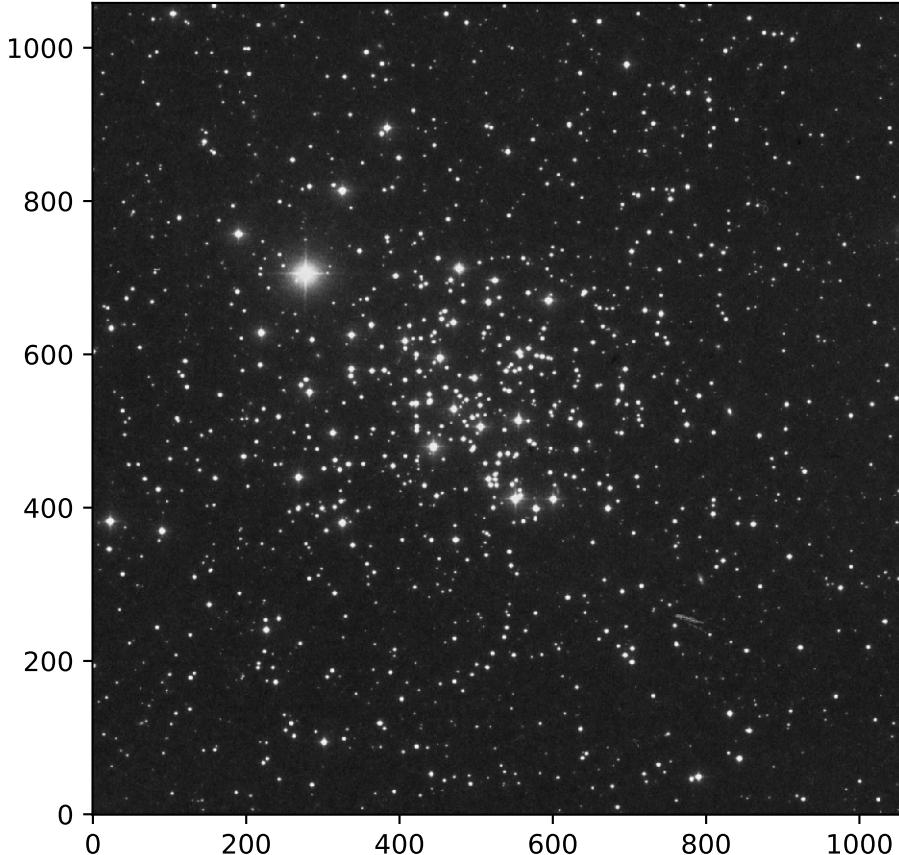
To load an example image with `load_star_image`:

```
>>> from photutils import datasets
>>> hdu = datasets.load_star_image()
Downloading http://data.astropy.org/photometry/M6707HH.fits [Done]
>>> print(hdu.data.shape)
(1059, 1059)
```

`hdu` is an `astropy.io.fits.ImageHDU` object and `hdu.data` is a `numpy.array` object that you can analyse with Photutils.

Let's plot the image:

```
from photutils import datasets
hdu = datasets.load_star_image()
plt.imshow(hdu.data, origin='lower', cmap='gray')
plt.tight_layout()
plt.show()
```



Reference/API

Load or make datasets for examples and tests.

Functions

get_path(filename[, location, cache])	Get path (location on your disk) for a given file.
load_fermi_image()	Load Fermi counts image for the Galactic center region.
load_irac_psf(channel)	Load Spitzer IRAC PSF.
load_spitzer_catalog()	Load 4.5 micron Spitzer catalog.
load_spitzer_image()	Load 4.5 micron Spitzer image.
load_star_image()	Load an optical image with stars.

Continued on next page

Table 15.1 – continued from previous page

<code>make_100gaussians_image()</code>	Make an example image containing 100 2D Gaussians plus Gaussian noise.
<code>make_4gaussians_image([hdu, wcs, wcsheader])</code>	Make an example image containing four 2D Gaussians plus Gaussian noise.
<code>make_gaussian_sources(image_shape, source_table)</code>	Make an image containing 2D Gaussian sources.
<code>make_noise_image(image_shape[, type, mean, ...])</code>	Make a noise image containing Gaussian or Poisson noise.
<code>make_poisson_noise(image[, random_state])</code>	Make a Poisson noise image from an image whose pixel values represent the expected number of counts (e.g., electrons or photons).
<code>make_random_gaussians(n_sources, flux_range, ...)</code>	Make a <code>Table</code> containing parameters for randomly generated 2D Gaussian sources.

get_path

`photutils.datasets.get_path(filename, location=u'local', cache=False)`

Get path (location on your disk) for a given file.

Parameters

`filename` : str

File name in the local or remote data folder

`location` : {‘local’, ‘remote’}

File location. ‘local’ means bundled with photutils. ‘remote’ means a server or the Astropy cache on your machine.

`cache` : bool, optional

Whether to cache the contents of remote URLs. Default is currently `False` because `True` will fail with Python 3.

Returns

`path` : str

Path (location on your disk) of the file.

Examples

```
>>> from astropy.io import fits
>>> from photutils import datasets
>>> hdu_list = fits.open(datasets.get_path('fermi_counts.fits.gz'))
```

load_fermi_image

`photutils.datasets.load_fermi_image()`

Load Fermi counts image for the Galactic center region.

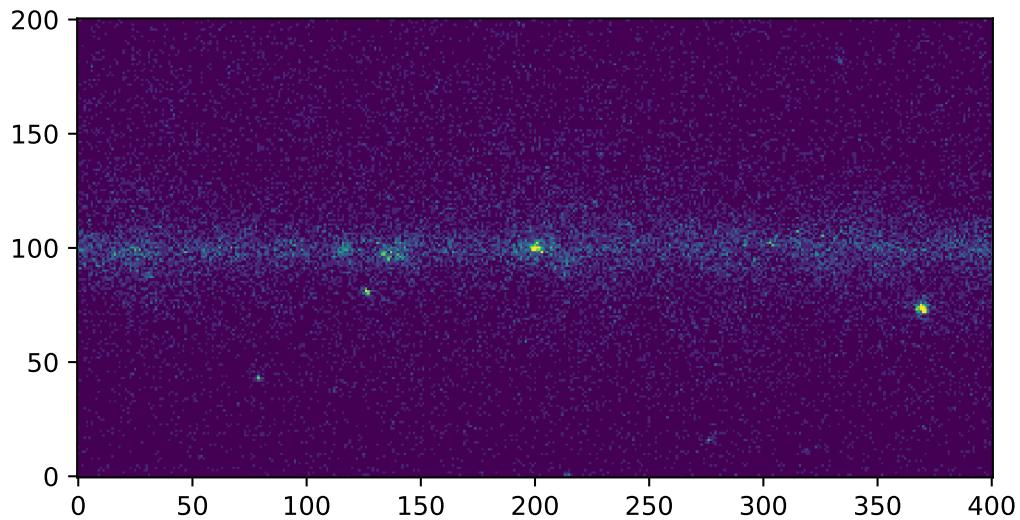
Returns

`hdu` : `ImageHDU`

Image HDU

Examples

```
from photutils import datasets
hdu = datasets.load_fermi_image()
plt.imshow(hdu.data, origin='lower', vmax=10)
```



load_irac_psf

```
photutils.datasets.load_irac_psf(channel)
Load Spitzer IRAC PSF.
```

Parameters

channel : int (1-4)

The IRAC channel number:

- Channel 1: 3.6 microns
- Channel 2: 4.5 microns
- Channel 3: 5.8 microns
- Channel 4: 8.0 microns

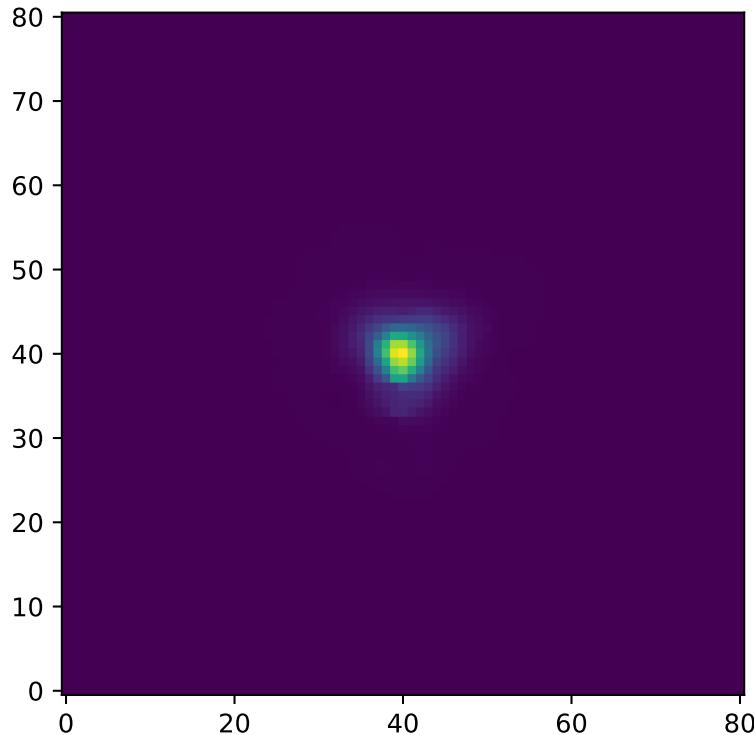
Returns

hdu : [ImageHDU](#)

The IRAC PSF.

Examples

```
from photutils.datasets import load_irac_psf
hdu = load_irac_psf(1)
plt.imshow(hdu.data, origin='lower')
```



load_spitzer_catalog

photutils.datasets.load_spitzer_catalog()

Load 4.5 micron Spitzer catalog.

This is the catalog corresponding to the image returned by `load_spitzer_image`.

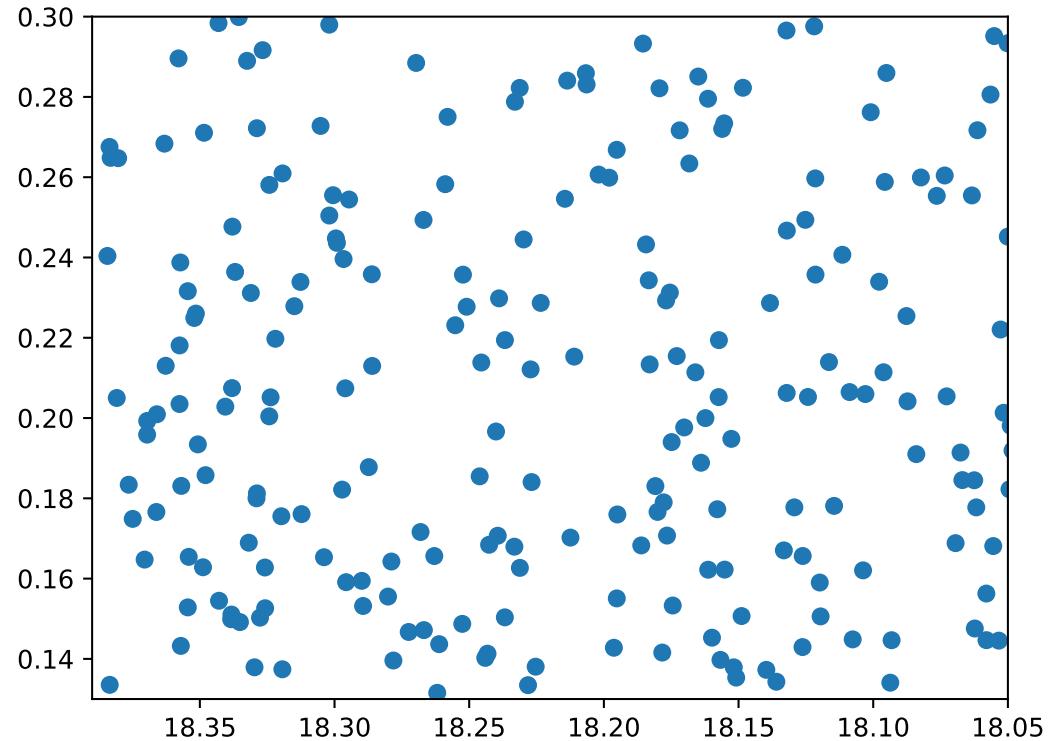
Returns

`catalog : Table`

The catalog of sources

Examples

```
from photutils import datasets
catalog = datasets.load_spitzer_catalog()
plt.scatter(catalog['l'], catalog['b'])
plt.xlim(18.39, 18.05)
plt.ylim(0.13, 0.30)
```



load_spitzer_image

`photutils.datasets.load_spitzer_image()`

Load 4.5 micron Spitzer image.

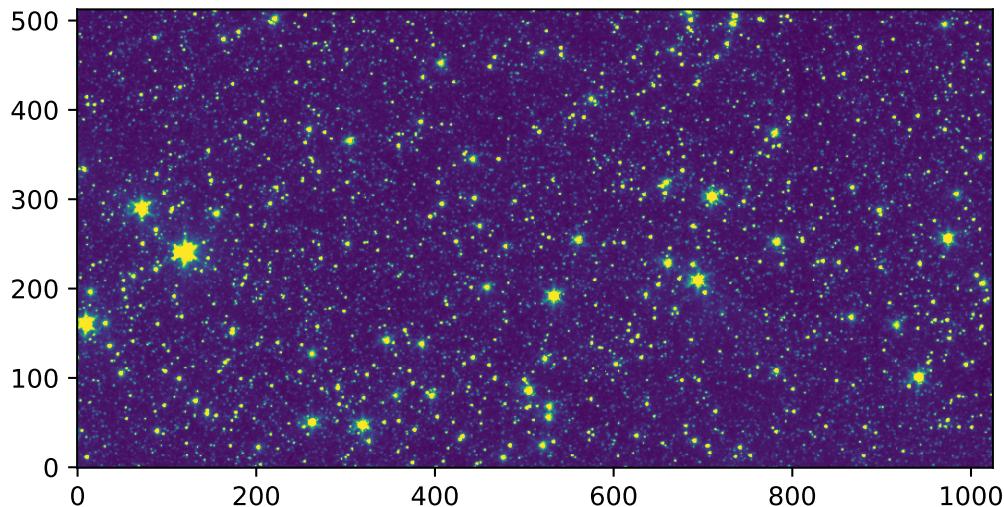
Returns

`hdu : ImageHDU`

The 4.5 micron image

Examples

```
from photutils import datasets
hdu = datasets.load_spitzer_image()
plt.imshow(hdu.data, origin='lower', vmax=50)
```



load_star_image

```
photutils.datasets.load_star_image()
```

Load an optical image with stars.

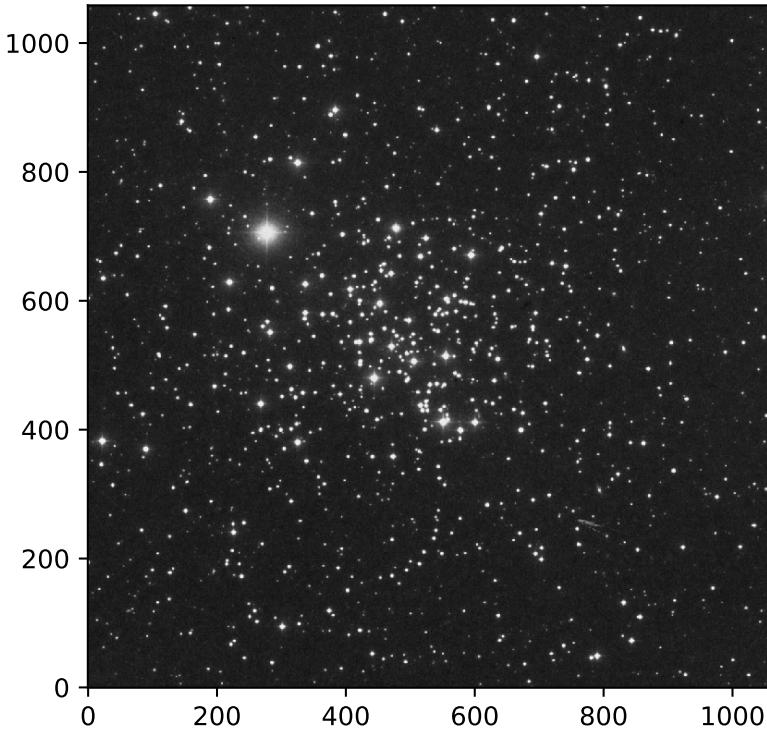
Returns

hdu : `ImageHDU`

Image HDU

Examples

```
from photutils import datasets
hdu = datasets.load_star_image()
plt.imshow(hdu.data, origin='lower', cmap='gray')
```



make_100gaussians_image

`photutils.datasets.make_100gaussians_image()`

Make an example image containing 100 2D Gaussians plus Gaussian noise.

The background has a mean of 5 and a standard deviation of 2.

Returns

`image : ndarray`

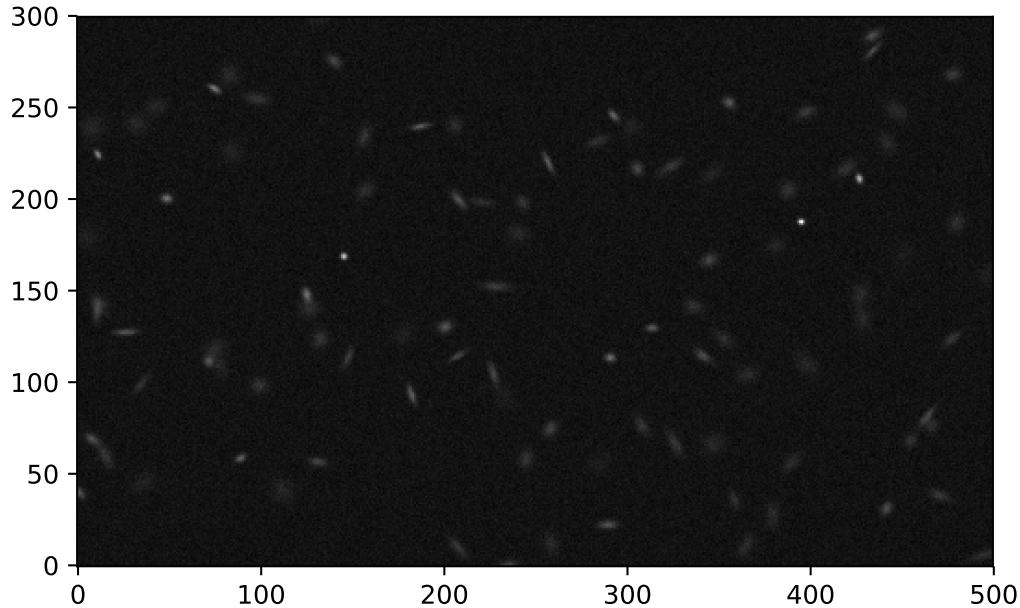
Image containing Gaussian sources.

See also:

[make_4gaussians_image](#)

Examples

```
from photutils import datasets
image = datasets.make_100gaussians_image()
plt.imshow(image, origin='lower', cmap='gray')
```



make_4gaussians_image

```
photutils.datasets.make_4gaussians_image(hdu=False, wcs=False, wcsheader=None)
```

Make an example image containing four 2D Gaussians plus Gaussian noise.

The background has a mean and standard deviation of 5.

Parameters

hdu : bool, optional

If `True` returns image as an `ImageHDU` object. To include WCS information in the header, use the `wcs` and `wcsheader` inputs. Otherwise the header will be minimal. Default is `False`.

wcs : bool, optional

If `True` and `hdu=True`, then a simple WCS will be included in the returned `ImageHDU` header. Default is `False`.

wcsheader : dict or `None`, optional

If `hdu` and `wcs` are `True`, this dictionary is passed to `WCS` to generate the returned `ImageHDU` header.

Returns

image : `ndarray` or `ImageHDU`

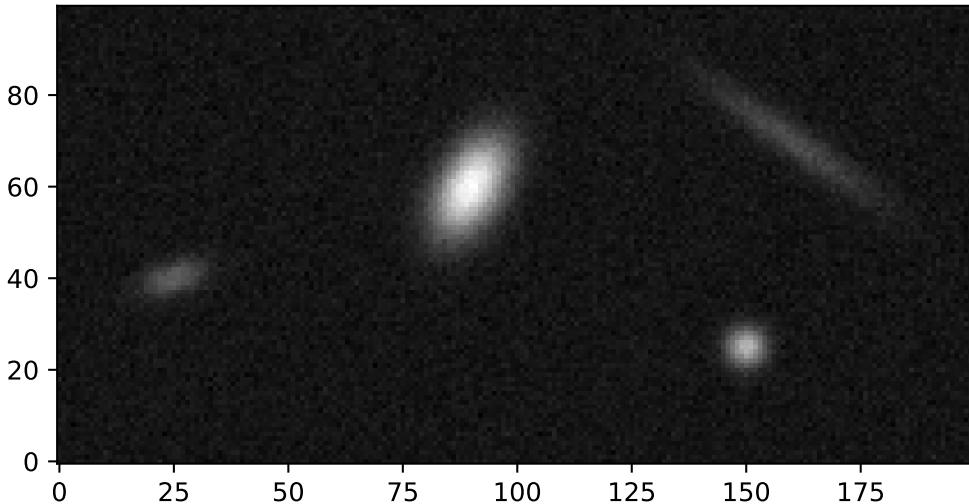
Image or `ImageHDU` containing Gaussian sources.

See also:

[make_100gaussians_image](#)

Examples

```
from photutils import datasets
image = datasets.make_4gaussians_image()
plt.imshow(image, origin='lower', cmap='gray')
```



[make_gaussian_sources](#)

```
photutils.datasets.make_gaussian_sources(image_shape, source_table, oversample=1, unit=None,  
hdu=False, wcs=False, wcsheader=None)
```

Make an image containing 2D Gaussian sources.

Parameters

image_shape : 2-tuple of int

Shape of the output 2D image.

source_table : [Table](#)

Table of parameters for the Gaussian sources. Each row of the table corresponds to a Gaussian source whose parameters are defined by the column names. The column names must include flux or amplitude, x_mean, y_mean, x_stddev, y_stddev, and

`theta` (see `Gaussian2D` for a description of most of these parameter names). If both `flux` and `amplitude` are present, then `amplitude` will be ignored.

oversample : float, optional

The sampling factor used to discretize the `Gaussian2D` models on a pixel grid.

If the value is 1.0 (the default), then the models will be discretized by taking the value at the center of the pixel bin. Note that this method will not preserve the total flux of very small sources.

Otherwise, the models will be discretized by taking the average over an oversampled grid. The pixels will be oversampled by the `oversample` factor.

unit : `UnitBase` instance, str, optional

An object that represents the unit desired for the output image. Must be an `UnitBase` object or a string parseable by the `units` package.

hdu : bool, optional

If `True` returns image as an `ImageHDU` object. To include WCS information in the header, use the `wcs` and `wcsheader` inputs. Otherwise the header will be minimal. Default is `False`.

wcs : bool, optional

If `True` and `hdu=True`, then a simple WCS will be included in the returned `ImageHDU` header. Default is `False`.

wcsheader : dict or `None`, optional

If `hdu` and `wcs` are `True`, this dictionary is passed to `WCS` to generate the returned `ImageHDU` header.

Returns

image : `ndarray` or `Quantity` or `ImageHDU`

Image or `ImageHDU` containing 2D Gaussian sources.

See also:

`make_random_gaussians`, `make_noise_image`, `make_poisson_noise`

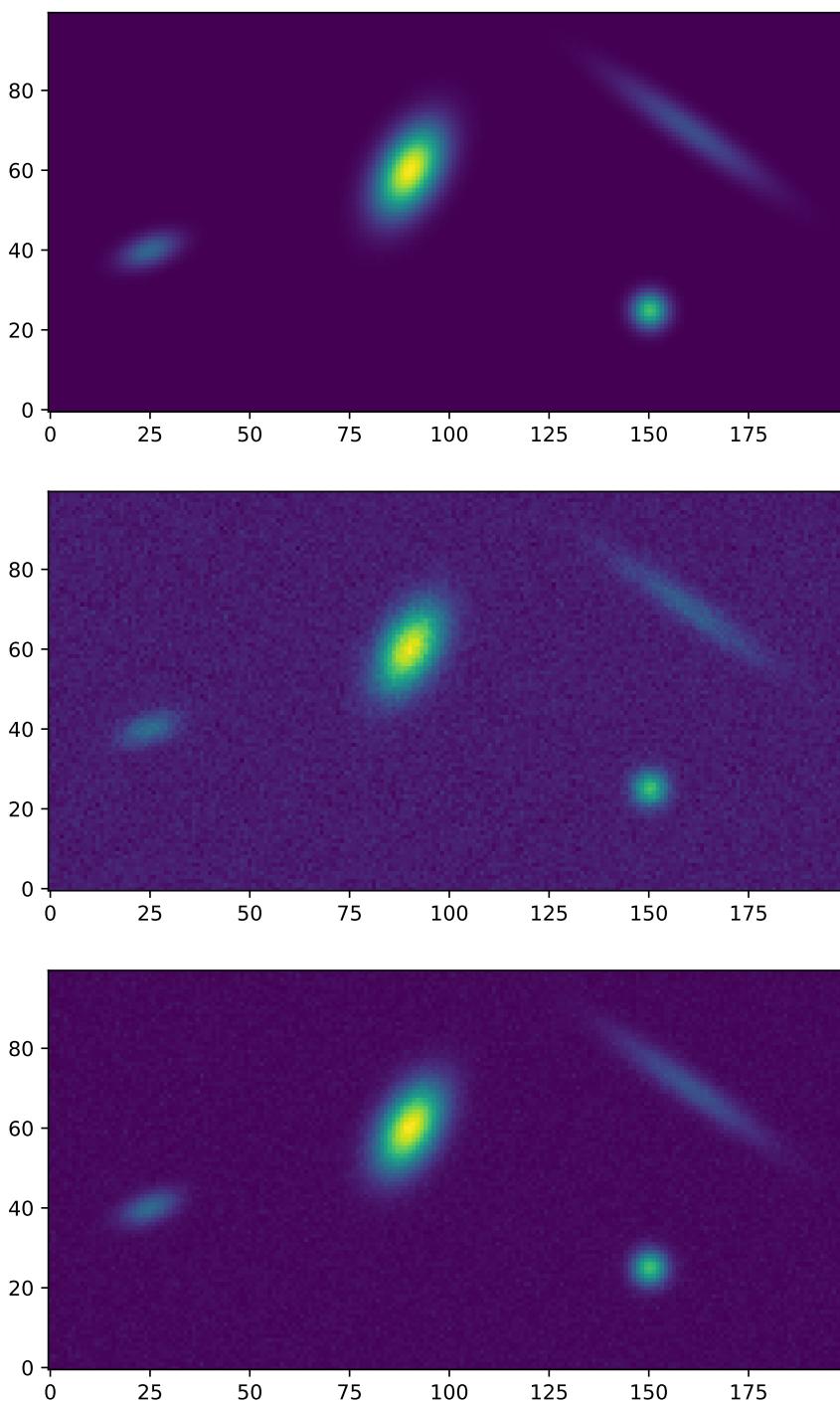
Examples

```
# make a table of Gaussian sources
from astropy.table import Table
table = Table()
table['amplitude'] = [50, 70, 150, 210]
table['x_mean'] = [160, 25, 150, 90]
table['y_mean'] = [70, 40, 25, 60]
table['x_stddev'] = [15.2, 5.1, 3., 8.1]
table['y_stddev'] = [2.6, 2.5, 3., 4.7]
table['theta'] = np.array([145., 20., 0., 60.]) * np.pi / 180.

# make an image of the sources without noise, with Gaussian
# noise, and with Poisson noise
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_noise_image
shape = (100, 200)
image1 = make_gaussian_sources(shape, table)
```

```
image2 = image1 + make_noise_image(shape, type='gaussian', mean=5.,
                                    stddev=5.)
image3 = image1 + make_noise_image(shape, type='poisson', mean=5.)

# plot the images
import matplotlib.pyplot as plt
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 12))
ax1.imshow(image1, origin='lower', interpolation='nearest')
ax2.imshow(image2, origin='lower', interpolation='nearest')
ax3.imshow(image3, origin='lower', interpolation='nearest')
```



make_noise_image

```
photutils.datasets.make_noise_image(image_shape, type=u'gaussian', mean=None, stddev=None, unit=None, random_state=None)
```

Make a noise image containing Gaussian or Poisson noise.

Parameters

image_shape : 2-tuple of int

Shape of the output 2D image.

type : {‘gaussian’, ‘poisson’}

The distribution used to generate the random noise.

- ‘gaussian’: Gaussian distributed noise.

- ‘poisson’: Poisson distributed noise.

mean : float

The mean of the random distribution. Required for both Gaussian and Poisson noise.

stddev : float, optional

The standard deviation of the Gaussian noise to add to the output image. Required for Gaussian noise and ignored for Poisson noise (the variance of the Poisson distribution is equal to its mean).

unit : [UnitBase](#) instance, str

An object that represents the unit desired for the output image. Must be an [UnitBase](#) object or a string parseable by the [units](#) package.

random_state : int or [RandomState](#), optional

Pseudo-random number generator state used for random sampling. Separate function calls with the same noise parameters and `random_state` will generate the identical noise image.

Returns

image : ndarray

Image containing random noise.

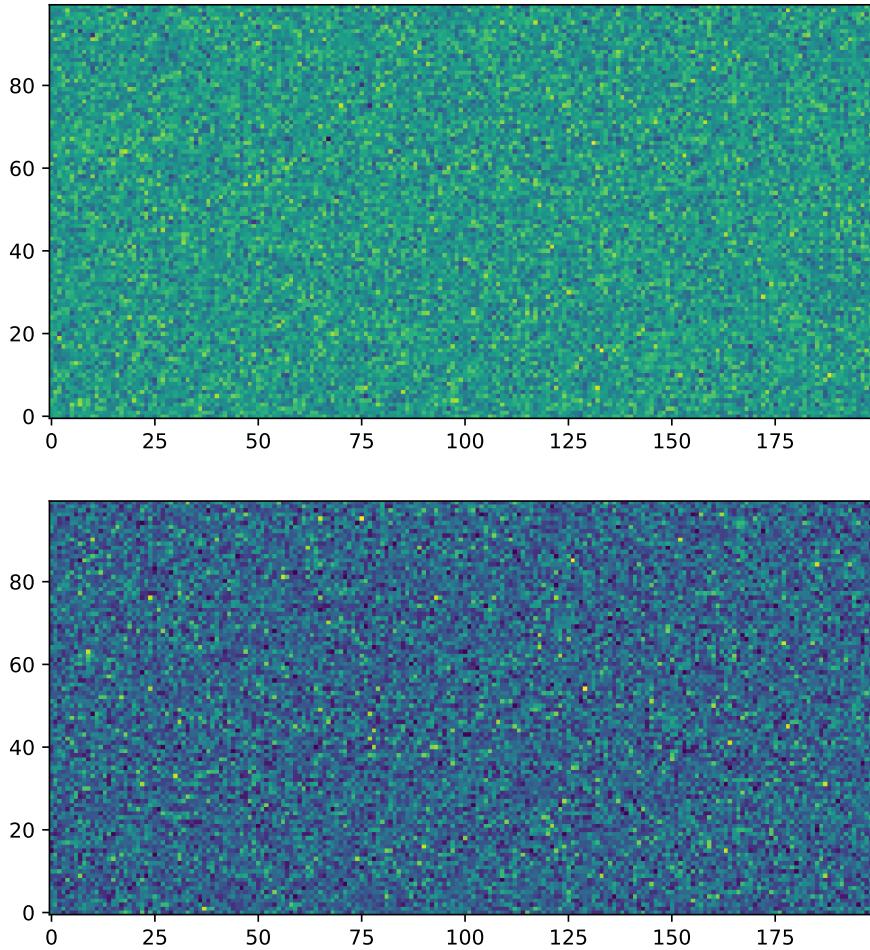
See also:

[make_poisson_noise](#), [make_gaussian_sources](#), [make_random_gaussians](#)

Examples

```
# make a Gaussian and Poisson noise image
from photutils.datasets import make_noise_image
shape = (100, 200)
image1 = make_noise_image(shape, type='gaussian', mean=0., stddev=5.)
image2 = make_noise_image(shape, type='poisson', mean=5.)

# plot the images
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
ax1.imshow(image1, origin='lower', interpolation='nearest')
ax2.imshow(image2, origin='lower', interpolation='nearest')
```



`make_poisson_noise`

`photutils.datasets.make_poisson_noise(image, random_state=None)`

Make a Poisson noise image from an image whose pixel values represent the expected number of counts (e.g., electrons or photons). Each pixel in the output noise image is generated by drawing a random sample from a Poisson distribution with expectation value given by the input `image`.

Parameters

`image` : `ndarray` or `Quantity`

The 2D image from which to make Poisson noise. Each pixel in the image must have a positive value (e.g., electron or photon counts).

`random_state` : `int` or `RandomState`, optional

Pseudo-random number generator state used for random sampling.

Returns**image** : ndarray or Quantity

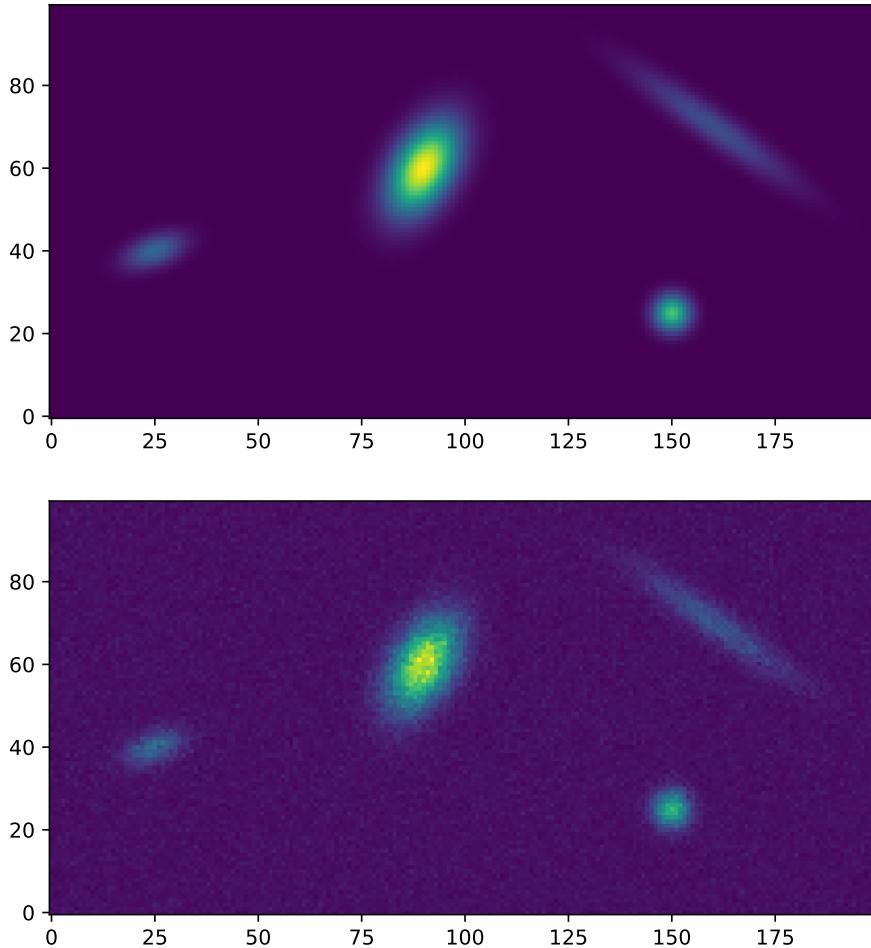
The 2D image of Poisson noise. The image is generated by drawing samples from a Poisson distribution with expectation values for each pixel given by the input `image`.

See also:`make_noise_image`, `make_gaussian_sources`, `make_random_gaussians`**Examples**

```
# make a table of Gaussian sources
from astropy.table import Table
table = Table()
table['amplitude'] = [50, 70, 150, 210]
table['x_mean'] = [160, 25, 150, 90]
table['y_mean'] = [70, 40, 25, 60]
table['x_stddev'] = [15.2, 5.1, 3., 8.1]
table['y_stddev'] = [2.6, 2.5, 3., 4.7]
table['theta'] = np.array([145., 20., 0., 60.]) * np.pi / 180.

# make an image of the sources and add a background level,
# then make the Poisson noise image.
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_poisson_noise
shape = (100, 200)
bkgrd = 10.
image1 = make_gaussian_sources(shape, table) + bkgrd
image2 = make_poisson_noise(image1, random_state=12345)

# plot the images
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
ax1.imshow(image1, origin='lower', interpolation='nearest')
ax2.imshow(image2, origin='lower', interpolation='nearest')
```



`make_random_gaussians`

```
photutils.datasets.make_random_gaussians(n_sources, flux_range, xmean_range, ymean_range, xstddev_range, ystddev_range, amplitude_range=None, random_state=None)
```

Make a [Table](#) containing parameters for randomly generated 2D Gaussian sources.

Each row of the table corresponds to a Gaussian source whose parameters are defined by the column names. The parameters are drawn from a uniform distribution over the specified input bounds.

The output table can be input into [`make_gaussian_sources`](#) to create an image containing the 2D Gaussian sources.

Parameters

n_sources : float

The number of random Gaussian sources to generate.

flux_range : array-like

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source fluxes. `flux_range` will be ignored if `amplitude_range` is input.

xmean_range : array-like

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source `x_mean`.

ymean_range : array-like

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source `y_mean`.

xstddev_range : array-like

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source `x_stddev`.

ystddev_range : array-like

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source `y_stddev`.

amplitude_range : array-like, optional

The lower and upper boundaries, (lower, upper), of the uniform distribution from which to draw source amplitudes. If `amplitude_range` is input, then `flux_range` will be ignored.

random_state : int or `RandomState`, optional

Pseudo-random number generator state used for random sampling. Separate function calls with the same parameters and `random_state` will generate the identical sources.

Returns**table** : `Table`

A table of parameters for the randomly generated Gaussian sources. Each row of the table corresponds to a Gaussian source whose parameters are defined by the column names. The column names will include `flux` or `amplitude`, `x_mean`, `y_mean`, `x_stddev`, `y_stddev`, and `theta` (see `Gaussian2D` for a description of most of these parameter names).

See also:

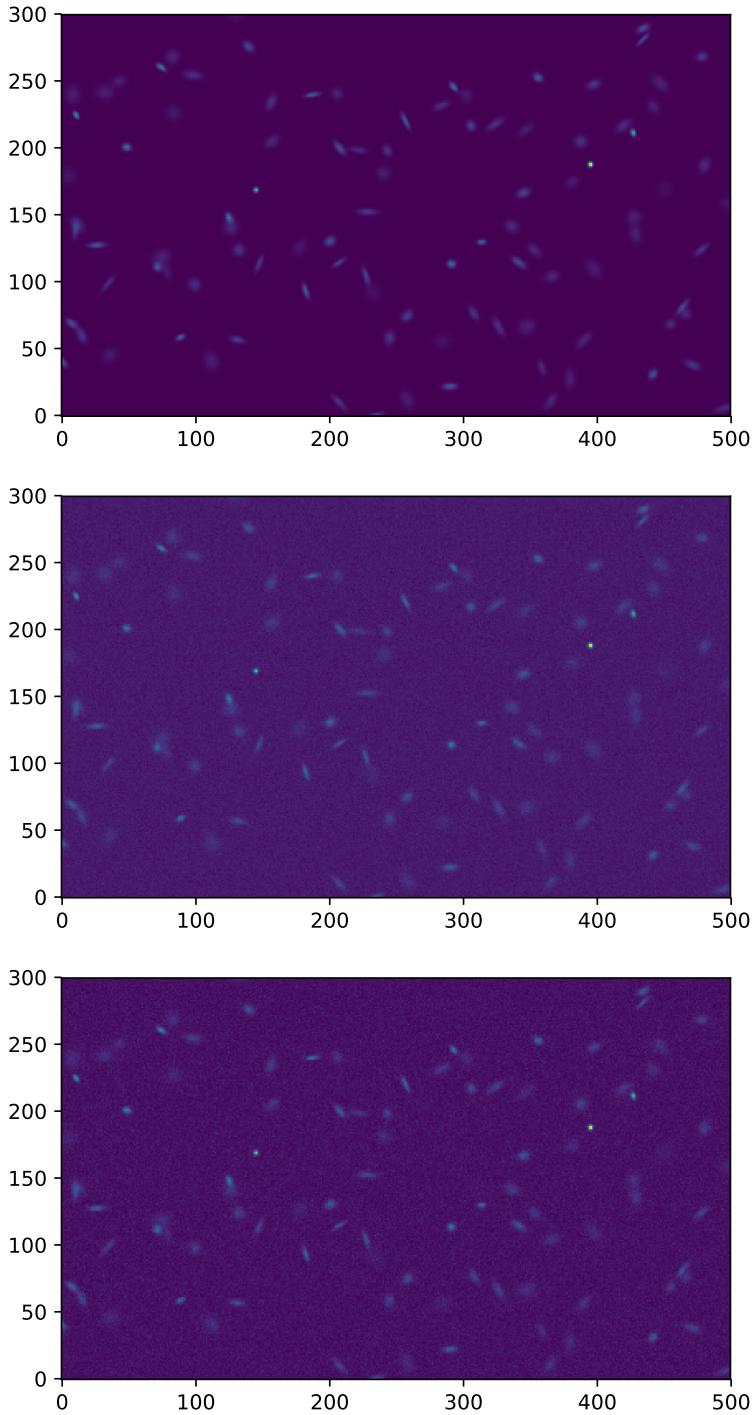
`make_gaussian_sources`, `make_noise_image`, `make_poisson_noise`

Examples

```
# create the random sources
from photutils.datasets import make_random_gaussians
n_sources = 100
flux_range = [500, 1000]
xmean_range = [0, 500]
ymean_range = [0, 300]
xstddev_range = [1, 5]
ystddev_range = [1, 5]
table = make_random_gaussians(n_sources, flux_range, xmean_range,
                               ymean_range, xstddev_range,
                               ystddev_range, random_state=12345)
```

```
# make an image of the random sources without noise, with
# Gaussian noise, and with Poisson noise
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_noise_image
shape = (300, 500)
image1 = make_gaussian_sources(shape, table)
image2 = image1 + make_noise_image(shape, type='gaussian', mean=5.,
                                   stddev=2.)
image3 = image1 + make_noise_image(shape, type='poisson', mean=5.)

# plot the images
import matplotlib.pyplot as plt
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 12))
ax1.imshow(image1, origin='lower', interpolation='nearest')
ax2.imshow(image2, origin='lower', interpolation='nearest')
ax3.imshow(image3, origin='lower', interpolation='nearest')
```



CHAPTER 16

Utility Functions (`photutils.utils`)

Introduction

The `photutils.utils` package contains general-purpose utility functions.

Reference/API

General-purpose utility functions.

Functions

<code>calc_total_error(data, bkg_error, effective_gain)</code>	Calculate a total error array, combining a background-only error array with the Poisson noise of sources.
<code>check_random_state(seed)</code>	Turn seed into a <code>numpy.random.RandomState</code> instance.
<code>cutout_footprint(data, position[, box_size, ...])</code>	Cut out a region from data (and optional mask and error) centered at specified (x, y) position.
<code>filter_data(data, kernel[, mode, ...])</code>	Convolve a 2D image with a 2D kernel.
<code>interpolate_masked_data(data, mask[, error, ...])</code>	Interpolate over masked pixels in data and optional error or background images.
<code>mask_to_mirrored_num(image, mask_image, ...)</code>	Replace masked pixels with the value of the pixel mirrored across a given <code>center_position</code> .
<code>random_cmap([ncolors, background_color, ...])</code>	Generate a <code>matplotlib</code> colormap consisting of random (muted) colors.
<code>std_blocksum(data, block_sizes[, mask])</code>	Calculate the standard deviation of block-summed data values at sizes of <code>block_sizes</code> .

calc_total_error

```
photutils.utils.calc_total_error(data, bkg_error, effective_gain)
```

Calculate a total error array, combining a background-only error array with the Poisson noise of sources.

Parameters

data : array_like or `Quantity`

The data array.

bkg_error : array_like or `Quantity`

The pixel-wise Gaussian 1-sigma background-only errors of the input data. `error` should include all sources of “background” error but *exclude* the Poisson error of the sources. `error` must have the same shape as `data`.

effective_gain : float, array-like, or `Quantity`

Ratio of counts (e.g., electrons or photons) to the units of data used to calculate the Poisson error of the sources.

Returns

total_error : `ndarray` or `Quantity`

The total error array. If `data`, `bkg_error`, and `effective_gain` are all `Quantity` objects, then `total_error` will also be returned as a `Quantity` object. Otherwise, a `ndarray` will be returned.

Notes

To use units, `data`, `bkg_error`, and `effective_gain` must *all* be `Quantity` objects. A `ValueError` will be raised if only some of the inputs are `Quantity` objects.

The total error array, σ_{tot} is:

$$\sigma_{\text{tot}} = \sqrt{\sigma_b^2 + \frac{I}{g}}$$

where σ_b , I , and g are the background `bkg_error` image, data image, and `effective_gain`, respectively.

Pixels where `data` (I_i) is negative do not contribute additional Poisson noise to the total error, i.e. $\sigma_{\text{tot},i} = \sigma_{b,i}$. Note that this is different from `SExtractor`, which sums the total variance in the segment, including pixels where I_i is negative. In such cases, `SExtractor` underestimates the total errors. Also note that data should be background-subtracted to match `SExtractor`’s errors.

`effective_gain` can either be a scalar value or a 2D image with the same shape as the data. A 2D image is useful with mosaic images that have variable depths (i.e., exposure times) across the field. For example, one should use an exposure-time map as the `effective_gain` for a variable depth mosaic image in count-rate units.

If your input data are in units of ADU, then `effective_gain` should represent electrons/ADU. If your input data are in units of electrons/s then `effective_gain` should be the exposure time or an exposure time map (e.g., for mosaics with non-uniform exposure times).

check_random_state

```
photutils.utils.check_random_state(seed)
```

Turn seed into a `numpy.random.RandomState` instance.

Parameters**seed** : `None`, `int`, or `numpy.random.RandomState`

If seed is `None`, return the `RandomState` singleton used by `numpy.random`. If seed is an `int`, return a new `RandomState` instance seeded with seed. If seed is already a `RandomState`, return it. Otherwise raise `ValueError`.

Returns**random_state** : `numpy.random.RandomState`

`RandomState` object.

Notes

This routine is from scikit-learn. See <http://scikit-learn.org/stable/developers/utilities.html#validation-tools>.

`cutout_footprint`

```
photutils.utils.cutout_footprint(data, position, box_size=3, footprint=None, mask=None, error=None)
```

Cut out a region from data (and optional mask and error) centered at specified (x, y) position.

The size of the region is specified via the `box_size` or `footprint` keywords. The output mask for the cutout region represents the combination of the input mask and footprint mask.

Parameters**data** : `array_like`

The 2D array of the image.

position : 2 tuple

The (x, y) pixel coordinate of the center of the region.

box_size : scalar or tuple, optional

The size of the region to cutout from data. If `box_size` is a scalar then a square box of size `box_size` will be used. If `box_size` has two elements, they should be in (ny, nx) order. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

footprint : `ndarray` of bools, optional

A boolean array where `True` values describe the local footprint region. `box_size=(n, m)` is equivalent to `footprint=np.ones((n, m))`. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

mask : `array_like`, `bool`, optional

A boolean mask with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

error : `array_like`, optional

The 2D array of the 1-sigma errors of the input data.

Returns**region_data** : `ndarray`

The data cutout.

region_mask : `ndarray`

The mask cutout.

region_error : `ndarray`

The error cutout.

slices : tuple of slices

Slices in each dimension of the data array used to define the cutout region.

filter_data

```
photutils.utils.filter_data(data, kernel, mode=u'constant', fill_value=0.0,  
                           check_normalization=False)
```

Convolve a 2D image with a 2D kernel.

The kernel may either be a 2D `ndarray` or a `Kernel2D` object.

Parameters

data : array_like

The 2D array of the image.

kernel : array-like (2D) or `Kernel2D`

The 2D kernel used to filter the input data. Filtering the data will smooth the noise and maximize detectability of objects with a shape similar to the kernel.

mode : {‘constant’, ‘reflect’, ‘nearest’, ‘mirror’, ‘wrap’}, optional

The mode determines how the array borders are handled. For the ‘constant’ mode, values outside the array borders are set to `fill_value`. The default is ‘constant’.

fill_value : scalar, optional

Value to fill data values beyond the array borders if mode is ‘constant’. The default is `0.0`.

check_normalization : bool, optional

If `True` then a warning will be issued if the kernel is not normalized to 1.

interpolate_masked_data

```
photutils.utils.interpolate_masked_data(data, mask, error=None, background=None)
```

Interpolate over masked pixels in data and optional error or background images.

The value of masked pixels are replaced by the mean value of the connected neighboring non-masked pixels. This function is intended for single, isolated masked pixels (e.g. hot/warm pixels).

Parameters

data : array_like or `Quantity`

The data array.

mask : array_like (bool)

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

error : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` must have the same shape as data.

background : array_like, or `Quantity`, optional

The pixel-wise background level of the input data. `background` must have the same shape as `data`.

Returns

data : `ndarray` or `Quantity`

Input data with interpolated masked pixels.

error : `ndarray` or `Quantity`

Input error with interpolated masked pixels. `None` if input `error` is not input.

background : `ndarray` or `Quantity`

Input background with interpolated masked pixels. `None` if input `background` is not input.

mask_to_mirrored_num

`photutils.utils.mask_to_mirrored_num(image, mask_image, center_position, bbox=None)`

Replace masked pixels with the value of the pixel mirrored across a given `center_position`. If the mirror pixel is unavailable (i.e. itself masked or outside of the image), then the masked pixel value is set to zero.

Parameters

image : `numpy.ndarray`, 2D

The 2D array of the image.

mask_image : array-like, bool

A boolean mask with the same shape as `image`, where a `True` value indicates the corresponding element of `image` is considered bad.

center_position : 2-tuple

(`x`, `y`) center coordinates around which masked pixels will be mirrored.

bbox : list, tuple, `numpy.ndarray`, optional

The bounding box (`x_min`, `x_max`, `y_min`, `y_max`) over which to replace masked pixels.

Returns

result : `numpy.ndarray`, 2D

A 2D array with replaced masked pixels.

Examples

```
>>> import numpy as np
>>> from photutils.utils import mask_to_mirrored_num
>>> image = np.arange(16).reshape(4, 4)
>>> mask = np.zeros_like(image, dtype=bool)
>>> mask[0, 0] = True
>>> mask[1, 1] = True
>>> mask_to_mirrored_num(image, mask, (1.5, 1.5))
array([[15,  1,  2,  3],
       [ 4, 10,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

random_cmap

```
photutils.utils.random_cmap(ncolors=256, background_color=u'black', random_state=None)
```

Generate a matplotlib colormap consisting of random (muted) colors.

A random colormap is very useful for plotting segmentation images.

Parameters

ncolors : int, optional

The number of colors in the colormap. For use with segmentation images, ncolors should be set to the number of labels. The default is 256.

background_color : str, optional

The name of the background (first) color in the colormap. Valid colors names are defined by `matplotlib.colors.cnames`. The default is 'black'.

random_state : int or `RandomState`, optional

The pseudo-random number generator state used for random sampling. Separate function calls with the same `random_state` will generate the same colormap.

Returns

cmap : `matplotlib.colors.Colormap`

The matplotlib colormap with random colors.

std_blocksum

```
photutils.utils.std_blocksum(data, block_sizes, mask=None)
```

Calculate the standard deviation of block-summed data values at sizes of `block_sizes`.

Values from incomplete blocks, either because of the image edges or masked pixels, are not included.

Parameters

data : array-like

The 2D array to block sum.

block_sizes : int, array-like of int

An array of integer (square) block sizes.

mask : array-like (bool), optional

A boolean mask, with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Blocks that contain *any* masked data are excluded from calculations.

Returns

result : `ndarray`

An array of the standard deviations of the block-summed array for the input `block_sizes`.

Classes

`ShepardIDWInterpolator(coordinates, values)`

Class to perform Inverse Distance Weighted (IDW) interpolation.

ShepardIDWInterpolator

```
class photutils.utils.ShepardIDWInterpolator(coordinates, values, weights=None, leafsize=10)
Bases: object
```

Class to perform Inverse Distance Weighted (IDW) interpolation.

This interpolator uses a modified version of [Shepard's method](#) (see the Notes section for details).

Parameters

coordinates : float, 1D array-like, or NxM-array-like

Coordinates of the known data points. In general, it is expected that these coordinates are in a form of a NxM-like array where N is the number of points and M is dimension of the coordinate space. When M=1 (1D space), then the coordinates parameter may be entered as a 1D array or, if only one data point is available, coordinates can be a scalar number representing the 1D coordinate of the data point.

Note: If the dimensionality of coordinates is larger than 2, e.g., if it is of the form $N_1 \times N_2 \times N_3 \times \dots \times N_n \times M$, then it will be flattened to form an array of size NxM where $N = N_1 * N_2 * \dots * N_n$.

values : float or 1D array-like

Values of the data points corresponding to each coordinate provided in coordinates. In general a 1D array is expected. When a single data point is available, then values can be a scalar number.

Note: If the dimensionality of values is larger than 1 then it will be flattened.

weights : float or 1D array-like, optional

Weights to be associated with each data value. These weights, if provided, will be combined with inverse distance weights (see the Notes section for details). When weights is `None` (default), then only inverse distance weights will be used. When provided, this input parameter must have the same form as values.

leafsize : float, optional

The number of points at which the k-d tree algorithm switches over to brute-force. `leafsize` must be positive. See [scipy.spatial.cKDTree](#) for further information.

Notes

This interpolator uses a slightly modified version of [Shepard's method](#). The essential difference is the introduction of a “regularization” parameter (`reg`) that is used when computing the inverse distance weights:

$$w_i = 1/(d(x, x_i)^{power} + r)$$

By supplying a positive regularization parameter one can avoid singularities at the locations of the data points as well as control the “smoothness” of the interpolation (e.g., make the weights of the neighbors less varied). The “smoothness” of interpolation can also be controlled by the power parameter (`power`).

Examples

This class can be instantiated using the following syntax:

```
>>> from photutils.utils import ShepardIDWInterpolator as idw
```

Example of interpolating 1D data:

```
>>> import numpy as np
>>> np.random.seed(123)
>>> x = np.random.random(100)
>>> y = np.sin(x)
>>> f = idw(x, y)
>>> f(0.4)
0.38862424043228855
>>> np.sin(0.4)
0.38941834230865052

>>> xi = np.random.random(4)
>>> xi
array([ 0.51312815,  0.66662455,  0.10590849,  0.13089495])
>>> f(xi)
array([ 0.49086423,  0.62647862,  0.1056854 ,  0.13048335])
>>> np.sin(xi)
array([ 0.49090493,  0.6183367 ,  0.10571061,  0.13052149])
```

NOTE: In the last example, `xi` may be a Nx1 array instead of a 1D vector.

Example of interpolating 2D data:

```
>>> pos = np.random.rand(1000, 2)
>>> val = np.sin(pos[:, 0] + pos[:, 1])
>>> f = idw(pos, val)
>>> f([0.5, 0.6])
0.89312649587405657
>>> np.sin(0.5 + 0.6)
0.89120736006143542
```

Methods Summary

<code>__call__(positions[, n_neighbors, eps, ...])</code>	Evaluate the interpolator at the given positions.
---	---

Methods Documentation

`__call__(positions, n_neighbors=8, eps=0.0, power=1.0, reg=0.0, conf_dist=1e-12, dtype=<type 'float'>)`

Evaluate the interpolator at the given positions.

Parameters

`positions` : float, 1D array-like, or NxM-array-like

Coordinates of the position(s) at which the interpolator should be evaluated. In general, it is expected that these coordinates are in a form of a NxM-like array where N is the number of points and M is dimension of the coordinate space. When M=1 (1D space), then the `positions` parameter may be input as a 1D-like array or, if only one data point is available, `positions` can be a scalar number representing the 1D coordinate of the data point.

Note: If the dimensionality of the `positions` argument is larger than 2, e.g., if it is of

the form $N_1 \times N_2 \times N_3 \times \dots \times N_n \times M$, then it will be flattened to form an array of size $N \times M$ where $N = N_1 * N_2 * \dots * N_n$.

Warning: The dimensionality of positions must match the dimensionality of the coordinates used during the initialization of the interpolator.

n_neighbors : int, optional

The maximum number of nearest neighbors to use during the interpolation.

eps : float, optional

Set to use approximate nearest neighbors; the k th neighbor is guaranteed to be no further than $(1 + \text{eps})$ times the distance to the real k -th nearest neighbor. See `scipy.spatial.KDTree.query` for further information.

power : float, optional

The power of the inverse distance used for the interpolation weights. See the Notes section for more details.

reg : float, optional

The regularization parameter. It may be used to control the smoothness of the interpolator. See the Notes section for more details.

conf_dist : float, optional

The confusion distance below which the interpolator should use the value of the closest data point instead of attempting to interpolate. This is used to avoid singularities at the known data points, especially if `reg` is 0.0.

dtype : data-type

The data type of the output interpolated values. If `None` then the type will be inferred from the type of the values parameter used during the initialization of the interpolator.

Class Inheritance Diagram

ShepardIDWInterpolator

CHAPTER 17

API Reference

These are the functions and classes available in the top-level `photutils` namespace. Other functionality is available by importing specific sub-packages (e.g. `photutils.utils`). Photutils is an Astropy affiliated package to provide tools for detecting and performing photometry of astronomical sources. It also has tools for background estimation, PSF matching, centroiding, and morphological measurements.

Functions

<code>aperture_photometry(data, apertures[, ...])</code>	Perform aperture photometry on the input data by summing the flux within the given aperture(s).
<code>centroid_1dg(data[, error, mask])</code>	Calculate the centroid of a 2D array by fitting 1D Gaussians to the marginal x and y distributions of the array.
<code>centroid_2dg(data[, error, mask])</code>	Calculate the centroid of a 2D array by fitting a 2D Gaussian (plus a constant) to the array.
<code>centroid_com(data[, mask])</code>	Calculate the centroid of a 2D array as its “center of mass” determined from image moments.
<code>create_matching_kernel(source_psf, target_psf)</code>	Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.
<code>daofind(*args, **kwargs)</code>	Deprecated since version 0.3.
<code>data_properties(data[, mask, background])</code>	Calculate the morphological properties (and centroid) of a 2D array (e.g.
<code>deblend_sources(data, segment_img, npixels)</code>	Deblend overlapping sources labeled in a segmentation image.
<code>detect_sources(data, threshold, npixels[, ...])</code>	Detect sources above a specified threshold value in an image and return a <code>SegmentationImage</code> object.
<code>detect_threshold(data, snr[, background, ...])</code>	Calculate a pixel-wise threshold image that can be used to detect sources.
<code>find_peaks(data, threshold[, box_size, ...])</code>	Find local peaks in an image that are above above a specified threshold value.

Continued on next page

Table 17.1 – continued from previous page

<code>fit_2dgaussian(data[, error, mask])</code>	Fit a 2D Gaussian plus a constant to a 2D image.
<code>gaussian1d_moments(data[, mask])</code>	Estimate 1D Gaussian parameters from the moments of 1D data.
<code>get_grouped_psf_model(template_psf_model, ...)</code>	Construct a joint PSF model which consists of a sum of PSF's templated on a specific model, but whose parameters are given by a table of objects.
<code>gini(data)</code>	Calculate the Gini coefficient of a 2D array.
<code>irafstarfind(*args, **kwargs)</code>	Deprecated since version 0.3.
<code>make_source_mask(data, snr, npixels[, mask, ...])</code>	Make a source mask using source segmentation and binary dilation.
<code>prepare_psf_model(psfmodel[, xname, yname, ...])</code>	Convert a 2D PSF model to one suitable for use with BasicPSFPhotometry or its subclasses.
<code>properties_table(source_props[, columns, ...])</code>	Construct a Table of properties from a list of SourceProperties objects.
<code>resize_psf(psf, input_pixel_scale, ...[, order])</code>	Resize a PSF using spline interpolation of the requested order.
<code>source_properties(data, segment_img[, ...])</code>	Calculate photometry and morphological properties of sources defined by a labeled segmentation image.
<code>subtract_psf(data, psf, posflux[, subshape])</code>	Subtract PSF/PRFs from an image.
<code>test([package, test_path, args, plugins, ...])</code>	Run the tests using <code>py.test</code> .

aperture_photometry

```
photutils.aperture_photometry(data, apertures, error=None, pixelwise_error=True, mask=None,  
                               method=u'exact', subpixels=5, unit=None, wcs=None)
```

Perform aperture photometry on the input data by summing the flux within the given aperture(s).

Parameters

data : array_like, [Quantity](#), [ImageHDU](#), or [HDUList](#)

The 2D array on which to perform photometry. `data` should be background-subtracted. Units can be used during the photometry, either provided with the data (i.e. a [Quantity](#) array) or the `unit` keyword. If `data` is an [ImageHDU](#) or [HDUList](#), the unit is determined from the 'BUNIT' header keyword.

apertures : [Aperture](#)

The aperture(s) to use for the photometry.

error : array_like or [Quantity](#), optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see [calc_total_error](#)). `error` must have the same shape as the input data.

pixelwise_error : bool, optional

If `True` (default), the photometric error is calculated using the `error` values from each pixel within the aperture. If `False`, the `error` value at the center of the aperture is used for the entire aperture.

mask : array_like (bool), optional

A boolean mask with the same shape as `data` where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- **‘exact’ (default):**

The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.

- **‘center’:**

A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).

- **‘subpixel’:**

A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

unit : `UnitBase` object or str, optional

An object that represents the unit associated with the input data and error arrays. Must be a `UnitBase` object or a string parseable by the `units` package. If data or error already have a different unit, the input unit will not be used and a warning will be raised. If data is an `ImageHDU` or `HDULList`, unit will override the ‘BUNIT’ header keyword.

wcs : `WCS`, optional

The WCS transformation to use if the input apertures is a `SkyAperture` object. If data is an `ImageHDU` or `HDULList`, wcs overrides any WCS transformation present in the header.

Returns

table : `QTable`

A table of the photometry with the following columns:

- ‘id’: The source ID.
- ‘xcenter’, ‘ycenter’: The x and y pixel coordinates of the input aperture center(s).
- ‘celestial_center’: The celestial coordinates of the input aperture center(s). Returned only if the input apertures is a `SkyAperture` object.
- ‘aperture_sum’: The sum of the values within the aperture.
- ‘aperture_sum_err’: The corresponding uncertainty in the ‘aperture_sum’ values. Returned only if the input error is not `None`.

The table metadata includes the Astropy and Photutils version numbers and the `aperture_photometry` calling arguments.

Notes

This function is decorated with `support_nddata` and thus supports `NDData` objects as input.

centroid_1dg

`photutils.centroid_1dg(data, error=None, mask=None)`

Calculate the centroid of a 2D array by fitting 1D Gaussians to the marginal x and y distributions of the array.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters

data : array_like

The 2D data array.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : ndarray

The x, y coordinates of the centroid.

centroid_2dg

`photutils.centroid_2dg(data, error=None, mask=None)`

Calculate the centroid of a 2D array by fitting a 2D Gaussian (plus a constant) to the array.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters

data : array_like

The 2D data array.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : ndarray

The x, y coordinates of the centroid.

centroid_com

```
photutils.centroid_com(data, mask=None)
```

Calculate the centroid of a 2D array as its “center of mass” determined from image moments.

Invalid values (e.g. NaNs or infs) in the data array are automatically masked.

Parameters

data : array_like

The 2D array of the image.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns

centroid : ndarray

The x, y coordinates of the centroid.

create_matching_kernel

```
photutils.create_matching_kernel(source_psf, target_psf, window=None)
```

Create a kernel to match 2D point spread functions (PSF) using the ratio of Fourier transforms.

Parameters

source_psf : 2D ndarray

The source PSF. The source PSF should have higher resolution (i.e. narrower) than the target PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

target_psf : 2D ndarray

The target PSF. The target PSF should have lower resolution (i.e. broader) than the source PSF. `source_psf` and `target_psf` must have the same shape and pixel scale.

window : callable, optional

The window (or taper) function or callable class instance used to remove high frequency noise from the PSF matching kernel. Some examples include:

- [HanningWindow](#)
- [TukeyWindow](#)
- [CosineBellWindow](#)
- [SplitCosineBellWindow](#)
- [TopHatWindow](#)

For more information on window functions and example usage, see *PSF Matching (photutils.psf.matching)*.

Returns

kernel : 2D ndarray

The matching kernel to go from `source_psf` to `target_psf`. The output matching kernel is normalized such that it sums to 1.

daofind

```
photutils.daofind(*args, **kwargs)
```

Deprecated since version 0.3: The daofind function is deprecated and may be removed in a future version. Use DAOStarFinder instead.

data_properties

```
photutils.data_properties(data, mask=None, background=None)
```

Calculate the morphological properties (and centroid) of a 2D array (e.g. an image cutout of an object) using image moments.

Parameters

data : array_like or `Quantity`

The 2D array of the image.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was previously present in the input data. background may either be a scalar value or a 2D image with the same shape as the input data. Inputting the background merely allows for its properties to be measured within each source segment. The input background does *not* get subtracted from the input data, which should already be background-subtracted.

Returns

result : `SourceProperties` instance

A `SourceProperties` object.

deblend_sources

```
photutils.deblend_sources(data, segment_img, npixels, filter_kernel=None, labels=None, nlevels=32, contrast=0.001, mode=u'exponential', connectivity=8, relabel=True)
```

Deblend overlapping sources labeled in a segmentation image.

Sources are deblended using a combination of multi-thresholding and `watershed segmentation`. In order to deblend sources, they must be separated enough such that there is a saddle between them.

Note: This function is experimental. Please report any issues on the [Photutils GitHub issue tracker](#)

Parameters

data : array_like

The 2D array of the image.

segment_img : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as data where sources are labeled by different positive integer values. A value of zero is reserved for the background.

npixels : int

The number of connected pixels, each greater than `threshold`, that an object must have to be detected. `npixels` must be a positive integer.

filter_kernel : array-like (2D) or `Kerne12D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel.

labels : int or array-like of int, optional

The label numbers to deblend. If `None` (default), then all labels in the segmentation image will be deblended.

nlevels : int, optional

The number of multi-thresholding levels to use. Each source will be re-thresholded at `nlevels`, spaced exponentially or linearly (see the `mode` keyword), between its minimum and maximum values within the source segment.

contrast : float, optional

The fraction of the total (blended) source flux that a local peak must have to be considered as a separate object. `contrast` must be between 0 and 1, inclusive. If `contrast = 0` then every local peak will be made a separate object (maximum deblending). If `contrast = 1` then no deblending will occur. The default is 0.001, which will deblend sources with a magnitude differences of about 7.5.

mode : {‘exponential’, ‘linear’}, optional

The mode used in defining the spacing between the multi-thresholding levels (see the `nlevels` keyword).

connectivity : {4, 8}, optional

The type of pixel connectivity used in determining how pixels are grouped into a detected source. The options are 4 or 8 (default). 4-connected pixels touch along their edges. 8-connected pixels touch along their edges or corners. For reference, SExtractor uses 8-connected pixels.

relabel : bool

If `True` (default), then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Returns

segment_image : `SegmentationImage`

A 2D segmentation image, with the same shape as `data`, where sources are marked by different positive integer values. A value of zero is reserved for the background.

See also:

`photutils.detect_sources()`

detect_sources

`photutils.detect_sources(data, threshold, npixels, filter_kernel=None, connectivity=8)`

Detect sources above a specified threshold value in an image and return a `SegmentationImage` object.

Detected sources must have `npixels` connected pixels that are each greater than the `threshold` value. If the filtering option is used, then the `threshold` is applied to the filtered image.

This function does not deblend overlapping sources. First use this function to detect sources followed by `deblend_sources()` to deblend sources.

Parameters

data : array_like

The 2D array of the image.

threshold : float or array-like

The data value or pixel-wise data values to be used for the detection threshold. A 2D threshold must have the same shape as `data`. See `detect_threshold` for one way to create a threshold image.

npixels : int

The number of connected pixels, each greater than `threshold`, that an object must have to be detected. `npixels` must be a positive integer.

filter_kernel : array-like (2D) or `Kernel2D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel.

connectivity : {4, 8}, optional

The type of pixel connectivity used in determining how pixels are grouped into a detected source. The options are 4 or 8 (default). 4-connected pixels touch along their edges. 8-connected pixels touch along their edges or corners. For reference, SExtractor uses 8-connected pixels.

Returns

segment_image : SegmentationImage

A 2D segmentation image, with the same shape as `data`, where sources are marked by different positive integer values. A value of zero is reserved for the background.

See also:

`photutils.detection.detect_threshold()`, `photutils.segmentation.SegmentationImage`,
`photutils.segmentation.source_properties()`, `photutils.segmentation.deblend_sources()`

Examples

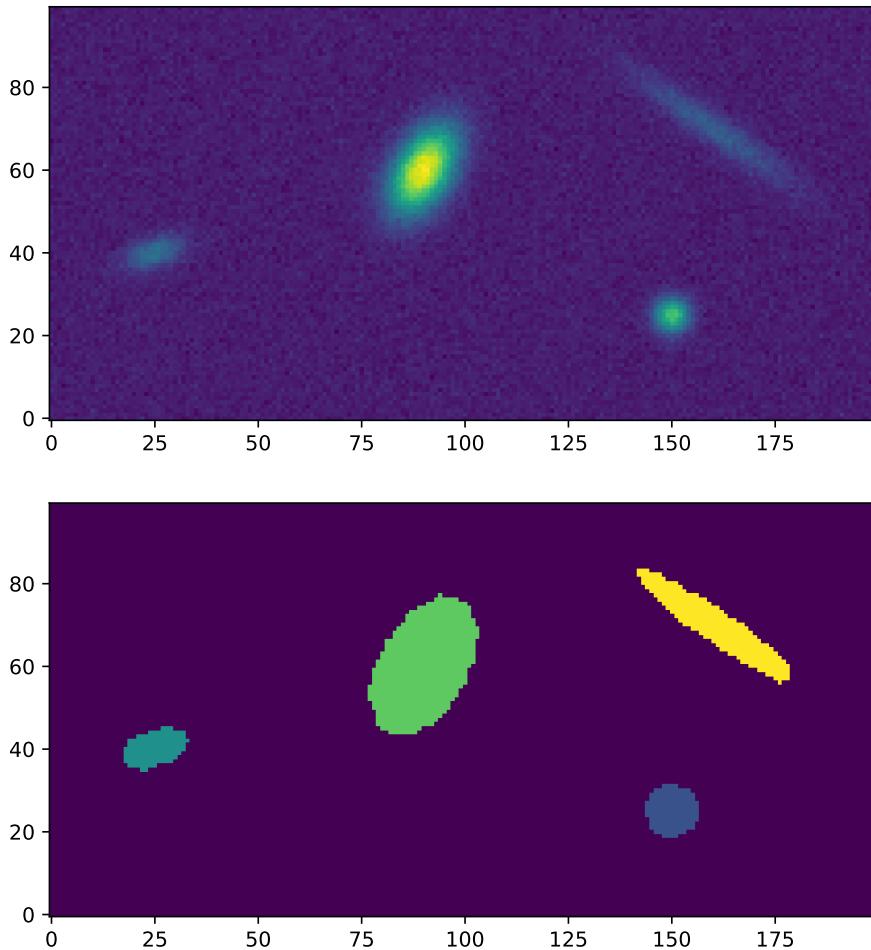
```
# make a table of Gaussian sources
from astropy.table import Table
table = Table()
table['amplitude'] = [50, 70, 150, 210]
table['x_mean'] = [160, 25, 150, 90]
table['y_mean'] = [70, 40, 25, 60]
table['x_stddev'] = [15.2, 5.1, 3., 8.1]
table['y_stddev'] = [2.6, 2.5, 3., 4.7]
table['theta'] = np.array([145., 20., 0., 60.]) * np.pi / 180.

# make an image of the sources with Gaussian noise
from photutils.datasets import make_gaussian_sources
from photutils.datasets import make_noise_image
shape = (100, 200)
sources = make_gaussian_sources(shape, table)
noise = make_noise_image(shape, type='gaussian', mean=0.,
```

```
        stddev=5., random_state=12345)
image = sources + noise

# detect the sources
from photutils import detect_threshold, detect_sources
threshold = detect_threshold(image, snr=3)
from astropy.convolution import Gaussian2DKernel
sigma = 3.0 / (2.0 * np.sqrt(2.0 * np.log(2.0)))    # FWHM = 3
kernel = Gaussian2DKernel(sigma, x_size=3, y_size=3)
kernel.normalize()
segm = detect_sources(image, threshold, npixels=5,
                      filter_kernel=kernel)

# plot the image and the segmentation image
import matplotlib.pyplot as plt
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
ax1.imshow(image, origin='lower', interpolation='nearest')
ax2.imshow(segm.data, origin='lower', interpolation='nearest')
```



detect_threshold

```
photutils.detect_threshold(data, snr, background=None, error=None, mask=None, mask_value=None,  
                           sigclip_sigma=3.0, sigclip_iters=None)
```

Calculate a pixel-wise threshold image that can be used to detect sources.

Parameters

data : array_like

The 2D array of the image.

snr : float

The signal-to-noise ratio per pixel above the background for which to consider a pixel as possibly being part of a source.

background : float or array_like, optional

The background value(s) of the input data. `background` may either be a scalar value or a 2D image with the same shape as the input data. If the input data has been background-subtracted, then set `background` to `0.0`. If `None`, then a scalar background value will be estimated using sigma-clipped statistics.

error : float or array_like, optional

The Gaussian 1-sigma standard deviation of the background noise in `data`. `error` should include all sources of “background” error, but *exclude* the Poisson error of the sources. If `error` is a 2D image, then it should represent the 1-sigma background error in each pixel of `data`. If `None`, then a scalar background rms value will be estimated using sigma-clipped statistics.

mask : array_like, bool, optional

A boolean mask with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Masked pixels are ignored when computing the image background statistics.

mask_value : float, optional

An image data value (e.g., `0.0`) that is ignored when computing the image background statistics. `mask_value` will be ignored if `mask` is input.

sigclip_sigma : float, optional

The number of standard deviations to use as the clipping limit when calculating the image background statistics.

sigclip_iters : int, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence is achieved (i.e., continue until the last iteration clips nothing) when calculating the image background statistics.

Returns

threshold : 2D ndarray

A 2D image with the same shape as `data` containing the pixel-wise threshold values.

See also:

`photutils.segmentation.detect_sources()`

Notes

The `mask`, `mask_value`, `sigclip_sigma`, and `sigclip_iters` inputs are used only if it is necessary to estimate background or error using sigma-clipped background statistics. If `background` and `error` are both input, then `mask`, `mask_value`, `sigclip_sigma`, and `sigclip_iters` are ignored.

find_peaks

```
photutils.find_peaks(data, threshold, box_size=3, footprint=None, mask=None, border_width=None,  
                      npeaks=inf, subpixel=False, error=None, wcs=None)
```

Find local peaks in an image that are above a specified threshold value.

Peaks are the maxima above the threshold within a local region. The regions are defined by either the `box_size` or `footprint` parameters. `box_size` defines the local region around each pixel as a square box. `footprint` is a boolean array where `True` values specify the region shape.

If multiple pixels within a local region have identical intensities, then the coordinates of all such pixels are returned. Otherwise, there will be only one peak pixel per local region. Thus, the defined region effectively imposes a minimum separation between peaks (unless there are identical peaks within the region).

When using subpixel precision (`subpixel=True`), then a cutout of the specified `box_size` or `footprint` will be taken centered on each peak and fit with a 2D Gaussian (plus a constant). In this case, the fitted local centroid and peak value (the Gaussian amplitude plus the background constant) will also be returned in the output table.

Parameters

data : array_like

The 2D array of the image.

threshold : float or array-like

The data value or pixel-wise data values to be used for the detection threshold. A 2D threshold must have the same shape as `data`. See [detect_threshold](#) for one way to create a `threshold` image.

box_size : scalar or tuple, optional

The size of the local region to search for peaks at every point in `data`. If `box_size` is a scalar, then the region shape will be `(box_size, box_size)`. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

footprint : ndarray of bools, optional

A boolean array where `True` values describe the local footprint region within which to search for peaks at every point in `data`. `box_size=(n, m)` is equivalent to `footprint=np.ones((n, m))`. Either `box_size` or `footprint` must be defined. If they are both defined, then `footprint` overrides `box_size`.

mask : array_like, bool, optional

A boolean mask with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked.

border_width : bool, optional

The width in pixels to exclude around the border of the data.

npeaks : int, optional

The maximum number of peaks to return. When the number of detected peaks exceeds `npeaks`, the peaks with the highest peak intensities will be returned.

subpixel : bool, optional

If `True`, then a cutout of the specified `box_size` or `footprint` will be taken centered on each peak and fit with a 2D Gaussian (plus a constant). In this case, the fitted local centroid and peak value (the Gaussian amplitude plus the background constant) will also be returned in the output table.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data. `error` is used only to weight the 2D Gaussian fit performed when `subpixel=True`.

wcs : WCS

The WCS transformation to use to convert from pixel coordinates to ICRS world coordinates. If `None`, then the world coordinates will not be returned in the output Table.

Returns**output** : `Table`

A table containing the x and y pixel location of the peaks and their values. If `subpixel=True`, then the table will also contain the local centroid and fitted peak value.

fit_2dgaussian

```
photutils.fit_2dgaussian(data, error=None, mask=None)
```

Fit a 2D Gaussian plus a constant to a 2D image.

Invalid values (e.g. NaNs or infs) in the data or error arrays are automatically masked. The mask for invalid values represents the combination of the invalid-value masks for the data and error arrays.

Parameters**data** : array_like

The 2D array of the image.

error : array_like, optional

The 2D array of the 1-sigma errors of the input data.

mask : array_like (bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns**result** : A `GaussianConst2D` model instance.

The best-fitting Gaussian 2D model.

gaussian1d_moments

```
photutils.gaussian1d_moments(data, mask=None)
```

Estimate 1D Gaussian parameters from the moments of 1D data.

This function can be useful for providing initial parameter values when fitting a 1D Gaussian to the data.

Parameters**data** : array_like (1D)

The 1D array.

mask : array_like (1D bool), optional

A boolean mask, with the same shape as data, where a `True` value indicates the corresponding element of data is masked.

Returns**amplitude, mean, stddev** : float

The estimated parameters of a 1D Gaussian.

get_grouped_psf_model

```
photutils.get_grouped_psf_model(template_psf_model, star_group)
```

Construct a joint PSF model which consists of a sum of PSF's templated on a specific model, but whose parameters are given by a table of objects.

Parameters**template_psf_model** : `astropy.modeling.Fittable2DModel` instance

The model to use for *individual* objects. Must have parameters named `x_0`, `y_0`, and `flux`.

star_group : `Table`

Table of stars for which the compound PSF will be constructed. It must have columns named `x_0`, `y_0`, and `flux_0`.

Returns`group_psf`

An `astropy.modeling.CompoundModel` instance which is a sum of the given PSF models.

gini

`photutils.gini(data)`

Calculate the [Gini coefficient](#) of a 2D array.

The Gini coefficient is calculated using the prescription from [Lotz et al. 2004](#) as:

$$G = \frac{1}{|\bar{x}| n(n-1)} \sum_i^n (2i - n - 1) |x_i|$$

where \bar{x} is the mean over all pixel values x_i .

The Gini coefficient is a way of measuring the inequality in a given set of values. In the context of galaxy morphology, it measures how the light of a galaxy image is distributed among its pixels. A G value of 0 corresponds to a galaxy image with the light evenly distributed over all pixels while a G value of 1 represents a galaxy image with all its light concentrated in just one pixel.

Usually Gini's measurement needs some sort of preprocessing for defining the galaxy region in the image based on the quality of the input data. As there is not a general standard for doing this, this is left for the user.

Parameters**data** : array-like

The 2D data array or object that can be converted to an array.

Returns**gini** : float

The Gini coefficient of the input 2D array.

irafstarfind

`photutils.irafstarfind(*args, **kwargs)`

Deprecated since version 0.3: The irafstarfind function is deprecated and may be removed in a future version. Use IRAFStarFinder instead.

make_source_mask

```
photutils.make_source_mask(data, snr, npixels, mask=None, mask_value=None, filter_fwhm=None,
                           filter_size=3, filter_kernel=None, sigclip_sigma=3.0, sigclip_iters=5, dilate_size=11)
```

Make a source mask using source segmentation and binary dilation.

Parameters

data : array_like

The 2D array of the image.

snr : float

The signal-to-noise ratio per pixel above the background for which to consider a pixel as possibly being part of a source.

npixels : int

The number of connected pixels, each greater than threshold, that an object must have to be detected. `npixels` must be a positive integer.

mask : array_like, bool, optional

A boolean mask with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Masked pixels are ignored when computing the image background statistics.

mask_value : float, optional

An image data value (e.g., `0.0`) that is ignored when computing the image background statistics. `mask_value` will be ignored if `mask` is input.

filter_fwhm : float, optional

The full-width at half-maximum (FWHM) of the Gaussian kernel to filter the image before thresholding. `filter_fwhm` and `filter_size` are ignored if `filter_kernel` is defined.

filter_size : float, optional

The size of the square Gaussian kernel image. Used only if `filter_fwhm` is defined. `filter_fwhm` and `filter_size` are ignored if `filter_kernel` is defined.

filter_kernel : array-like (2D) or `Kerne12D`, optional

The 2D array of the kernel used to filter the image before thresholding. Filtering the image will smooth the noise and maximize detectability of objects with a shape similar to the kernel. `filter_kernel` overrides `filter_fwhm` and `filter_size`.

sigclip_sigma : float, optional

The number of standard deviations to use as the clipping limit when calculating the image background statistics.

sigclip_iters : int, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence is achieved (i.e., continue until the last iteration clips nothing) when calculating the image background statistics.

dilate_size : int, optional

The size of the square array used to dilate the segmentation image.

Returns

mask : 2D `ndarray`, bool

A 2D boolean image containing the source mask.

`prepare_psf_model`

`photutils.prepare_psf_model(psfmodel, xname=None, yname=None, fluxname=None, renormalize_psf=True)`

Convert a 2D PSF model to one suitable for use with `BasicPSFPhotometry` or its subclasses.

The resulting model may be a composite model, but should have only the x, y, and flux related parameters un-fixed.

Parameters

psfmodel : a 2D model

The model to assume as representative of the PSF.

xname : str or None

The name of the `psfmodel` parameter that corresponds to the x-axis center of the PSF. If None, the model will be assumed to be centered at x=0, and a new parameter will be added for the offset.

yname : str or None

The name of the `psfmodel` parameter that corresponds to the y-axis center of the PSF. If None, the model will be assumed to be centered at x=0, and a new parameter will be added for the offset.

fluxname : str or None

The name of the `psfmodel` parameter that corresponds to the total flux of the star. If None, a scaling factor will be added to the model.

renormalize_psf : bool

If True, the model will be integrated from -inf to inf and re-scaled so that the total integrates to 1. Note that this renormalization only occurs *once*, so if the total flux of `psfmodel` depends on position, this will *not* be correct.

Returns

outmod : a model

A new model ready to be passed into `BasicPSFPhotometry` or its subclasses.

`properties_table`

`photutils.properties_table(source_props, columns=None, exclude_columns=None)`

Construct a `Table` of properties from a list of `SourceProperties` objects.

If `columns` or `exclude_columns` are not input, then the `Table` will include all scalar-valued properties. Multi-dimensional properties, e.g. `data_cutout`, can be included in the `columns` input.

Parameters

source_props : `SourceProperties` or list of `SourceProperties`

A `SourceProperties` object or list of `SourceProperties` objects, one for each source.

columns : str or list of str, optional

Names of columns, in order, to include in the output `Table`. The allowed column names are any of the attributes of `SourceProperties`.

exclude_columns : str or list of str, optional

Names of columns to exclude from the default properties list in the output `Table`. The default properties are those with scalar values.

Returns

table : `Table`

A table of properties of the segmented sources, one row per source.

See also:

`SegmentationImage`, `SourceProperties`, `source_properties`, `detect_sources`

Examples

```
>>> import numpy as np
>>> from photutils import source_properties, properties_table
>>> image = np.arange(16.).reshape(4, 4)
>>> print(image)
[[ 0.   1.   2.   3.]
 [ 4.   5.   6.   7.]
 [ 8.   9.  10.  11.]
 [12.  13.  14.  15.]]
>>> segm = SegmentationImage([[1, 1, 0, 0],
...                           [1, 0, 0, 2],
...                           [0, 0, 2, 2],
...                           [0, 2, 2, 0]])
>>> props = source_properties(image, segm)
>>> columns = ['id', 'xcentroid', 'ycentroid', 'source_sum']
>>> tbl = properties_table(props, columns=columns)
>>> print(tbl)
      id    xcentroid     ycentroid   source_sum
      pix          pix
-----
 1       0.2        0.8       5.0
 2  2.09090909091  2.36363636364      55.0
```

resize_psf

`photutils.resize_psf(psf, input_pixel_scale, output_pixel_scale, order=3)`

Resize a PSF using spline interpolation of the requested order.

Parameters

psf : 2D `ndarray`

The 2D data array of the PSF.

input_pixel_scale : float

The pixel scale of the input psf. The units must match `output_pixel_scale`.

output_pixel_scale : float

The pixel scale of the output psf. The units must match `input_pixel_scale`.

order : float, optional

The order of the spline interpolation (0-5). The default is 3.

Returns

result : 2D `ndarray`

The resampled/interpolated 2D data array.

source_properties

```
photutils.source_properties(data, segment_img, error=None, mask=None, background=None, filter_kernel=None, wcs=None, labels=None)
```

Calculate photometry and morphological properties of sources defined by a labeled segmentation image.

Parameters

data : array_like or `Quantity`

The 2D array from which to calculate the source photometry and properties. `data` should be background-subtracted.

segment_img : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as `data` where sources are labeled by different positive integer values. A value of zero is reserved for the background.

error : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input `data`. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see `calc_total_error`). `error` must have the same shape as the input `data`. See the Notes section below for details on the error propagation.

mask : array_like (bool), optional

A boolean mask with the same shape as `data` where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was *previously* present in the input `data`. `background` may either be a scalar value or a 2D image with the same shape as the input `data`. Inputting the `background` merely allows for its properties to be measured within each source segment. The input `background` does *not* get subtracted from the input `data`, which should already be background-subtracted.

filter_kernel : array-like (2D) or `Kernel2D`, optional

The 2D array of the kernel used to filter the data prior to calculating the source centroid and morphological parameters. The kernel should be the same one used in defining the source segments (e.g., see `detect_sources()`). If `None`, then the unfiltered data will be used instead. Note that `SExtractor`'s centroid and morphological parameters are calculated from the filtered “detection” image.

wcs : `WCS`

The WCS transformation to use. If `None`, then `icrs_centroid`, `ra_icrs_centroid`, and `dec_icrs_centroid` will be `None`.

labels : int, array-like (1D, int)

Subset of segmentation labels for which to calculate the properties. If `None`, then the properties will be calculated for all labeled sources (the default).

Returns

output : list of `SourceProperties` objects

A list of `SourceProperties` objects, one for each source. The properties can be accessed as attributes or keys.

See also:

`SegmentationImage`, `SourceProperties`, `properties_table`, `detect_sources`

Notes

`SExtractor`'s centroid and morphological parameters are always calculated from the filtered “detection” image. The usual downside of the filtering is the sources will be made more circular than they actually are. If you wish to reproduce `SExtractor` results, then use the `filtered_data` input. If `filtered_data` is `None`, then the unfiltered data will be used for the source centroid and morphological parameters.

Negative (background-subtracted) data values within the source segment are set to zero when measuring morphological properties based on image moments. This could occur, for example, if the segmentation image was defined from a different image (e.g., different bandpass) or if the background was oversubtracted. Note that `source_sum` includes the contribution of negative (background-subtracted) data values.

The input error is assumed to include *all* sources of error, including the Poisson error of the sources. `source_sum_err` is simply the quadrature sum of the pixel-wise total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, S are the non-masked pixels in the source segment, and $\sigma_{\text{tot},i}$ is the input error array.

Examples

```
>>> import numpy as np
>>> from photutils import SegmentationImage, source_properties
>>> image = np.arange(16.).reshape(4, 4)
>>> print(image)
[[ 0.   1.   2.   3.]
 [ 4.   5.   6.   7.]
 [ 8.   9.  10.  11.]
 [12.  13.  14.  15.]]
>>> segm = SegmentationImage([[1, 1, 0, 0],
...                           [1, 0, 0, 2],
...                           [0, 0, 2, 2],
...                           [0, 2, 2, 0]])
>>> props = source_properties(image, segm)
```

Print some properties of the first object (labeled with 1 in the segmentation image):

```
>>> props[0].id    # id corresponds to segment label number
1
>>> props[0].centroid
<Quantity [ 0.8, 0.2] pix>
>>> props[0].source_sum
5.0
```

```
>>> props[0].area  
<Quantity 3.0 pix2>  
>>> props[0].max_value  
4.0
```

Print some properties of the second object (labeled with 2 in the segmentation image):

```
>>> props[1].id      # id corresponds to segment label number  
2  
>>> props[1].centroid  
<Quantity [ 2.36363636, 2.09090909] pix>  
>>> props[1].perimeter  
<Quantity 5.414213562373095 pix>  
>>> props[1].orientation  
<Quantity -0.7417593069227176 rad>
```

subtract_psf

`photutils.subtract_psf(data, psf, posflux, subshape=None)`

Subtract PSF/PRFs from an image.

Parameters

data : `NDData` or array (must be 2D)

Image data.

psf : `astropy.modeling.Fittable2DModel` instance

PSF/PRF model to be subtracted from the data.

posflux : Array-like of shape (3, N) or `Table`

Positions and fluxes for the objects to subtract. If an array, it is interpreted as (x, y, flux) If a table, the columns ‘x_fit’, ‘y_fit’, and ‘flux_fit’ must be present.

subshape : length-2 or None

The shape of the region around the center of the location to subtract the PSF from. If None, subtract from the whole image.

Returns

subdata : same shape and type as data

The image with the PSF subtracted

test

`photutils.test(package=None, test_path=None, args=None, plugins=None, verbose=False, paste-bin=None, remote_data=False, pep8=False, pdb=False, coverage=False, open_files=False, **kwargs)`

Run the tests using `py.test`. A proper set of arguments is constructed and passed to `pytest.main`.

Parameters

package : str, optional

The name of a specific package to test, e.g. ‘io.fits’ or ‘utils’. If nothing is specified all default tests are run.

test_path : str, optional

Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.

args : str, optional

Additional arguments to be passed to `pytest.main` in the `args` keyword argument.

plugins : list, optional

Plugins to be passed to `pytest.main` in the `plugins` keyword argument.

verbose : bool, optional

Convenience option to turn on verbose output from `py.test`. Passing True is the same as specifying '`-v`' in `args`.

pastebin : {'failed', 'all', None}, optional

Convenience option for turning on `py.test` pastebin output. Set to 'failed' to upload info for failed tests, or 'all' to upload info for all tests.

remote_data : bool, optional

Controls whether to run tests marked with `@remote_data`. These tests use online data and are not run by default. Set to True to run these tests.

pep8 : bool, optional

Turn on PEP8 checking via the `pytest-pep8` plugin and disable normal tests. Same as specifying '`--pep8 -k pep8`' in `args`.

pdb : bool, optional

Turn on PDB post-mortem analysis for failing tests. Same as specifying '`--pdb`' in `args`.

coverage : bool, optional

Generate a test coverage report. The result will be placed in the directory `htmlcov`.

open_files : bool, optional

Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the `psutil` package.

parallel : int, optional

When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use all the cores on the machine. Requires the `pytest-xdist` plugin installed. Only available when using Astropy 0.3 or later.

kwargs

Any additional keywords passed into this function will be passed on to the astropy test runner. This allows use of test-related functionality implemented in later versions of astropy without explicitly updating the package template.

Classes

<code>Aperture</code>	Abstract base class for all apertures.
<code>ApertureMask</code> (mask, bbox_slice)	Class for an aperture mask.
Continued on next page	

Table 17.2 – continued from previous page

<code>Background2D(data, box_size[, mask, ...])</code>	Class to estimate a 2D background and background RMS noise in an image.
<code>BackgroundBase</code>	Base class for classes that estimate scalar background values.
<code>BackgroundRMSBase</code>	Base class for classes that estimate scalar background RMS values.
<code>BasicPSFPhotometry(group_maker, ...[, ...])</code>	This class implements a PSF photometry algorithm that can find sources in an image, group overlapping sources into a single model, fit the model to the sources, and subtracting the models from the image.
<code>BiweightLocationBackground</code>	Class to calculate the background in an array using the biweight location.
<code>BiweightMidvarianceBackgroundRMS</code>	Class to calculate the background RMS in an array as the (sigma-clipped) biweight midvariance.
<code>BkgIDWInterpolator([leafsize, n_neighbors, ...])</code>	This class generates full-sized background and background RMS images from lower-resolution mesh images using inverse-distance weighting (IDW) interpolation (ShepardIDWInterpolator).
<code>BkgZoomInterpolator([order, mode, cval])</code>	This class generates full-sized background and background RMS images from lower-resolution mesh images using the <code>zoom</code> (spline) interpolator.
<code>CircularAnnulus</code>	Circular annulus aperture(s), defined in pixel coordinates.
<code>CircularAperture</code>	Circular aperture(s), defined in pixel coordinates.
<code>CircularMaskMixin</code>	Mixin class to create masks for circular and circular-annulus aperture objects.
<code>CosineBellWindow(alpha)</code>	Class to define a 2D cosine bell window function.
<code>DAOGroup(crit_separation)</code>	This is class implements the DAOGROUP algorithm presented by Stetson (1987).
<code>DAOPhotPSFPhotometry(crit_separation, ...[, ...])</code>	This class implements an iterative algorithm based on the DAOPHOT algorithm presented by Stetson (1987) to perform point spread function photometry in crowded fields.
<code>DAOStarFinder</code>	Detect stars in an image using the DAOIND (Stetson 1987) algorithm.
<code>DBSCANGroup(crit_separation[, min_samples, ...])</code>	Class to create star groups according to a distance criteria using the Density-based Spatial Clustering of Applications with Noise (DBSCAN) from scikit-learn.
<code>EllipticalAnnulus</code>	Elliptical annulus aperture(s), defined in pixel coordinates.
<code>EllipticalAperture</code>	Elliptical aperture(s), defined in pixel coordinates.
<code>EllipticalMaskMixin</code>	Mixin class to create masks for elliptical and elliptical-annulus aperture objects.
<code>FittableImageModel</code>	A fittable 2D model of an image allowing for image intensity scaling and image translations.
<code>GaussianConst2D</code>	A model for a 2D Gaussian plus a constant.
<code>GroupStarsBase</code>	This base class provides the basic interface for subclasses that are capable of classifying stars in groups.
<code>HanningWindow()</code>	Class to define a 2D Hanning (or Hann) window function.
<code>IRAFStarFinder</code>	Detect stars in an image using IRAF's "starfind" algorithm.
<code>IntegratedGaussianPRF</code>	Circular Gaussian model integrated over pixels.
<code>IterativelySubtractedPSFPhotometry(...[, ...])</code>	This class implements an iterative algorithm to perform point spread function photometry in crowded fields.

Continued on next page

Table 17.2 – continued from previous page

MADStdBackgroundRMS	Class to calculate the background RMS in an array as using the median absolute deviation (MAD) .
MMMBBackground	Class to calculate the background in an array using the DAOPHOT MMM algorithm.
MeanBackground	Class to calculate the background in an array as the (sigma-clipped) mean.
MedianBackground	Class to calculate the background in an array as the (sigma-clipped) median.
ModeEstimatorBackground	Class to calculate the background in an array using a mode estimator of the form (<code>median_factor * median</code>) - (<code>mean_factor * mean</code>).
NonNormalizable	Used to indicate that a FittableImageModel model is non-normalizable.
PRFAdapter	A model that adapts a supplied PSF model to act as a PRF.
PixelAperture	Abstract base class for 2D apertures defined in pixel coordinates.
RectangularAnnulus	Rectangular annulus aperture(s), defined in pixel coordinates.
RectangularAperture	Rectangular aperture(s), defined in pixel coordinates.
RectangularMaskMixin	Mixin class to create masks for rectangular or rectangular-annulus aperture objects.
SExtractorBackground	Class to calculate the background in an array using the SExtractor algorithm.
SegmentationImage(data)	Class for a segmentation image.
SigmaClip([sigma, sigma_lower, sigma_upper, ...])	Class to perform sigma clipping.
SkyAperture	Abstract base class for 2D apertures defined in celestial coordinates.
SkyCircularAnnulus	Circular annulus aperture(s), defined in sky coordinates.
SkyCircularAperture	Circular aperture(s), defined in sky coordinates.
SkyEllipticalAnnulus	Elliptical annulus aperture(s), defined in sky coordinates.
SkyEllipticalAperture	Elliptical aperture(s), defined in sky coordinates.
SkyRectangularAnnulus	Rectangular annulus aperture(s), defined in sky coordinates.
SkyRectangularAperture	Rectangular aperture(s), defined in sky coordinates.
SourceProperties(data, segment_img, label[, ...])	Class to calculate photometry and morphological properties of a single labeled source.
SplitCosineBellWindow(alpha, beta)	Class to define a 2D split cosine bell taper function.
StarFinderBase	Abstract base class for Star Finders.
StdBackgroundRMS	Class to calculate the background RMS in an array as the (sigma-clipped) standard deviation.
TopHatWindow(beta)	Class to define a 2D top hat window function.
TukeyWindow(alpha)	Class to define a 2D Tukey window function.

Aperture

```
class photutils.Aperture
    Bases: object
```

Abstract base class for all apertures.

ApertureMask

```
class photutils.ApertureMask(mask, bbox_slice)
    Bases: object
```

Class for an aperture mask.

Parameters

mask : array_like

A 2D array of an aperture mask representing the fractional overlap of the aperture on the pixel grid. This should be the full-sized (i.e. not truncated) array that is the direct output of one of the low-level `photutils.geometry` functions.

bbox_slice : tuple of slice objects

A tuple of (y, x) numpy slice objects defining the aperture minimal bounding box.

Attributes Summary

<code>array</code>	The 2D mask array.
--------------------	--------------------

Methods Summary

<code>apply(data[, fill_value])</code>	Apply the aperture mask to the input data, taking any edge effects into account.
<code>cutout(data[, fill_value])</code>	Create a cutout from the input data over the mask bounding box, taking any edge effects into account.
<code>to_image(shape)</code>	Return an image of the mask in a 2D array of the given shape, taking any edge effects into account.

Attributes Documentation

array

The 2D mask array.

Methods Documentation

apply(*data*, *fill_value*=0.0)

Apply the aperture mask to the input data, taking any edge effects into account.

The result is a mask-weighted cutout from the data.

Parameters

data : array_like or `Quantity`

A 2D array on which to apply the aperture mask.

fill_value : float, optional

The value is used to fill pixels where the aperture mask does not overlap with the input data. The default is 0.

Returns**result** : `ndarray`

A 2D mask-weighted cutout from the input data. If there is a partial overlap of the aperture mask with the input data, pixels outside of the data will be assigned to `fill_value` before being multiplied with the mask. `None` is returned if there is no overlap of the aperture with the input data.

cutout(*data*, *fill_value*=0.0)

Create a cutout from the input data over the mask bounding box, taking any edge effects into account.

Parameters**data** : array_like or `Quantity`

A 2D array on which to apply the aperture mask.

fill_value : float, optional

The value is used to fill pixels where the aperture mask does not overlap with the input data. The default is 0.

Returns**result** : `ndarray`

A 2D array cut out from the input data representing the same cutout region as the aperture mask. If there is a partial overlap of the aperture mask with the input data, pixels outside of the data will be assigned to `fill_value`. `None` is returned if there is no overlap of the aperture with the input data.

to_image(*shape*)

Return an image of the mask in a 2D array of the given shape, taking any edge effects into account.

Parameters**shape** : tuple of int

The (ny, nx) shape of the output array.

Returns**result** : `ndarray`

A 2D array of the mask.

Background2D

```
class photutils.Background2D(data,    box_size,    mask=None,    exclude_mesh_method=u'threshold',
                           exclude_mesh_percentile=10.0,                                filter_size=(3,
                           3),                                filter_threshold=None,          edge_method=u'pad',
                           sigma_clip=<photutils.background.core.SigmaClip      object>, 
                           bkg_estimator=<photutils.background.core.SExtractorBackground object>,
                           bkgrms_estimator=<photutils.background.core.StdBackgroundRMS      ob-
                           ject>, interpolator=<photutils.background.background_2d.BkgZoomInterpolator
                           object>)
```

Bases: `object`

Class to estimate a 2D background and background RMS noise in an image.

The background is estimated using sigma-clipped statistics in each mesh of a grid that covers the input data to create a low-resolution, and possibly irregularly-gridded, background map.

The final background map is calculated by interpolating the low-resolution background map.

Parameters**data** : array_like

The 2D array from which to estimate the background and/or background RMS map.

box_size : int or array_like (int)

The box size along each axis. If `box_size` is a scalar then a square box of size `box_size` will be used. If `box_size` has two elements, they should be in (ny, nx) order. For best results, the box shape should be chosen such that the data are covered by an integer number of boxes in both dimensions. When this is not the case, see the `edge_method` keyword for more options.

mask : array_like (bool), optional

A boolean mask, with the same shape as `data`, where a `True` value indicates the corresponding element of `data` is masked. Masked data are excluded from calculations.

exclude_mesh_method : {'threshold', 'any', 'all'}, optional

The method used to determine whether to exclude a particular mesh based on the number of masked pixels it contains in the input (e.g. source) `mask` or padding `mask` (if `edge_method='pad'`):

- 'threshold': exclude meshes that contain greater than `exclude_mesh_percentile` percent masked pixels. This is the default.
- 'any': exclude meshes that contain any masked pixels.
- 'all': exclude meshes that are completely masked.

exclude_mesh_percentile : float in the range of [0, 100], optional

The percentile of masked pixels in a mesh used as a threshold for determining if the mesh is excluded. If `exclude_mesh_method='threshold'`, then meshes that contain greater than `exclude_mesh_percentile` percent masked pixels are excluded. This parameter is used only if `exclude_mesh_method='threshold'`. The default is 10. For best results, `exclude_mesh_percentile` should be kept as low as possible (i.e., as long as there are sufficient pixels for reasonable statistical estimates).

filter_size : int or array_like (int), optional

The window size of the 2D median filter to apply to the low-resolution background map. If `filter_size` is a scalar then a square box of size `filter_size` will be used. If `filter_size` has two elements, they should be in (ny, nx) order. A filter size of 1 (or (1, 1)) means no filtering.

filter_threshold : int, optional

The threshold value for used for selective median filtering of the low-resolution 2D background map. The median filter will be applied to only the background meshes with values larger than `filter_threshold`. Set to `None` to filter all meshes (default).

edge_method : {'pad', 'crop'}, optional

The method used to determine how to handle the case where the image size is not an integer multiple of the `box_size` in either dimension. Both options will resize the image to give an exact multiple of `box_size` in both dimensions.

- 'pad': pad the image along the top and/or right edges. This is the default and recommended method.
- 'crop': crop the image along the top and/or right edges.

sigma_clip : `SigmaClip` instance, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=10`.

bkg_estimator : callable, optional

A callable object (a function or e.g., an instance of any `BackgroundBase` subclass) used to estimate the background in each of the meshes. The callable object must take in a 2D `ndarray` or `MaskedArray` and have an `axis` keyword (internally, the background will be calculated along `axis=1`). The callable object must return a 1D `MaskedArray`. If `bkg_estimator` includes sigma clipping, it will be ignored (use the `sigma_clip` keyword to define sigma clipping). The default is an instance of `SExtractorBackground`.

bkgrms_estimator : callable, optional

A callable object (a function or e.g., an instance of any `BackgroundRMSBase` subclass) used to estimate the background RMS in each of the meshes. The callable object must take in a 2D `ndarray` or `MaskedArray` and have an `axis` keyword (internally, the background RMS will be calculated along `axis=1`). The callable object must return a 1D `MaskedArray`. If `bkgrms_estimator` includes sigma clipping, it will be ignored (use the `sigma_clip` keyword to define sigma clipping). The default is an instance of `StdBackgroundRMS`.

interpolator : callable, optional

A callable object (a function or object) used to interpolate the low-resolution background or background RMS mesh to the full-size background or background RMS maps. The default is an instance of `BkgZoomInterpolator`.

Notes

If there is only one background mesh element (i.e., `box_size` is the same size as the `data`), then the background map will simply be a constant image.

Attributes Summary

<code>background</code>	A 2D <code>ndarray</code> containing the background image.
<code>background_median</code>	The median value of the 2D low-resolution background map.
<code>background_mesh_ma</code>	The background 2D (masked) array mesh prior to any interpolation.
<code>background_rms</code>	A 2D <code>ndarray</code> containing the background RMS image.
<code>background_rms_median</code>	The median value of the low-resolution background RMS map.
<code>background_rms_mesh_ma</code>	The background RMS 2D (masked) array mesh prior to any interpolation.
<code>mesh_nmasks</code>	A 2D (masked) array of the number of masked pixels in each mesh.

Methods Summary

<code>plot_meshes([ax, marker, color, outlines])</code>	Plot the low-resolution mesh boxes on a matplotlib Axes instance.
---	---

Attributes Documentation

`background`

A 2D `ndarray` containing the background image.

`background_median`

The median value of the 2D low-resolution background map.

This is equivalent to the value SExtractor prints to stdout (i.e., “(M+D) Background: <value>”).

`background_mesh_ma`

The background 2D (masked) array mesh prior to any interpolation.

`background_rms`

A 2D `ndarray` containing the background RMS image.

`background_rms_median`

The median value of the low-resolution background RMS map.

This is equivalent to the value SExtractor prints to stdout (i.e., “(M+D) RMS: <value>”).

`background_rms_mesh_ma`

The background RMS 2D (masked) array mesh prior to any interpolation.

`mesh_nmasks`

A 2D (masked) array of the number of masked pixels in each mesh. Only meshes included in the background estimation are included. Excluded meshes will be masked in the image.

Methods Documentation

`plot_meshes(ax=None, marker=u'+', color=u'blue', outlines=False, **kwargs)`

Plot the low-resolution mesh boxes on a matplotlib Axes instance.

Parameters

`ax` : `matplotlib.axes.Axes` instance, optional

If `None`, then the current Axes instance is used.

`marker` : str, optional

The marker to use to mark the center of the boxes. Default is ‘+’.

`color` : str, optional

The color for the markers and the box outlines. Default is ‘blue’.

`outlines` : bool, optional

Whether or not to plot the box outlines in addition to the box centers.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`. Used only if `outlines` is True.

BackgroundBase

class photutils.BackgroundBase
Bases: `object`

Base class for classes that estimate scalar background values.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Methods Summary

`__call__(...) <==> x(...)`

`calc_background(data[, axis])`

Calculate the background value.

Methods Documentation

`__call__(...) <==> x(...)`

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BackgroundRMSBase

class photutils.BackgroundRMSBase
Bases: `object`

Base class for classes that estimate scalar background RMS values.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Methods Summary

<code>__call__(...) <==> x(...)</code>	
<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.

Methods Documentation

`__call__(...) <==> x(...)`

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BasicPSFPhotometry

```
class photutils.BasicPSFPhotometry(group_maker, bkg_estimator, psf_model, fitshape, finder=None,
                                    fitter=<astropy.modeling.fitting.LevMarLSQFitter object>, aperture_radius=None)
```

Bases: `object`

This class implements a PSF photometry algorithm that can find sources in an image, group overlapping sources into a single model, fit the model to the sources, and subtracting the models from the image. This is roughly equivalent to the DAOPHOT routines FIND, GROUP, NSTAR, and SUBTRACT. This implementation allows a flexible and customizable interface to perform photometry. For instance, one is able to use different implementations for grouping and finding sources by using `group_maker` and `finder` respectively. In addition, sky background estimation is performed by `bkg_estimator`.

Parameters

`group_maker` : callable or `GroupStarsBase`

`group_maker` should be able to decide whether a given star overlaps with any other and label them as belonging to the same group. `group_maker` receives as input an `Table` object with columns named as `id`, `x_0`, `y_0`, in which `x_0` and `y_0` have the same meaning of `xcentroid` and `ycentroid`. This callable must return an `Table` with columns `id`, `x_0`, `y_0`, and `group_id`. The column `group_id` should contain integers starting from 1 that indicate which group a given source belongs to. See, e.g., `DAOGroup`.

`bkg_estimator` : callable, instance of any `BackgroundBase` subclass, or `None`

bkg_estimator should be able to compute either a scalar background or a 2D background of a given 2D image. See, e.g., [MedianBackground](#). If None, no background subtraction is performed.

psf_model : [astropy.modeling.Fittable2DModel](#) instance

PSF or PRF model to fit the data. Could be one of the models in this package like [DiscretePRF](#), [IntegratedGaussianPRF](#), or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as x_0, y_0 and flux. [prepare_psf_model](#) can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of fitshape must be an odd number.

finder : callable or instance of any [StarFinderBase](#) subclasses or None

finder should be able to identify stars, i.e. compute a rough estimate of the centroids, in a given 2D image. finder receives as input a 2D image and returns an [Table](#) object which contains columns with names: id, xcentroid, ycentroid, and flux. In which id is an integer-valued column starting from 1, xcentroid and ycentroid are center position estimates of the sources and flux contains flux estimates of the sources. See, e.g., [DAOStarFinder](#). If finder is None, initial guesses for positions of objects must be provided.

fitter : [Fitter](#) instance

Fitter object used to compute the optimized centroid positions and/or flux of the identified sources. See [fitting](#) for more details on fitters.

aperture_radius : float or None

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `psf_model`.

Notes

Note that an ambiguity arises whenever finder and positions (keyword argument for do_photometry)` are both not ``None. In this case, finder is ignored and initial guesses are taken from positions. In addition, an warning is raised to remind the user about this behavior.

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a star_groups table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, **Astronomical Society of the Pacific, Publications**,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP...99..191S>

Attributes Summary

aperture_radius
fitshape

Methods Summary

__call__(image[, positions])	Performs PSF photometry.
do_photometry(image[, positions])	Perform PSF photometry in <code>image</code> .
get_residual_image()	Returns an image that is the result of the subtraction between the original image and the fitted sources.
nstar(image, star_groups)	Fit, as appropriate, a compound or single model to the given <code>star_groups</code> .

Attributes Documentation

`aperture_radius`

`fitshape`

Methods Documentation

`__call__(image, positions=None)`

Performs PSF photometry. See [do_photometry](#) for more details including the `__call__` signature.

`do_photometry(image, positions=None)`

Perform PSF photometry in `image`.

This method assumes that `psf_model` has centroids and flux parameters which will be fitted to the data provided in `image`. A compound model, in fact a sum of `psf_model`, will be fitted to groups of stars automatically identified by `group_maker`. Also, `image` is not assumed to be background subtracted. If `positions` are not `None` then this method uses `positions` as initial guesses for the centroids. If the centroid positions are set as fixed in the PSF model `psf_model`, then the optimizer will only consider the flux as a variable.

Parameters

`image` : 2D array-like, [ImageHDU](#), [HDUList](#)

Image to perform photometry.

`positions`: ‘[~astropy.table.Table](#)‘

Positions (in pixel coordinates) at which to *start* the fit for each object. Columns ‘`x_0`’ and ‘`y_0`’ must be present. ‘`flux_0`’ can also be provided to set initial fluxes. If ‘`flux_0`’ is not provided, aperture photometry is used to estimate initial values for the fluxes.

Returns

`output_tab` : [Table](#) or `None`

Table with the photometry results, i.e., centroids and fluxes estimations and the initial estimates used to start the fitting process. `None` is returned if no sources are found in `image`.

get_residual_image()

Returns an image that is the result of the subtraction between the original image and the fitted sources.

Returns

residual_image : 2D array-like, [ImageHDU](#), [HDUList](#)

nstar(image, star_groups)

Fit, as appropriate, a compound or single model to the given `star_groups`. Groups are fitted sequentially from the smallest to the biggest. In each iteration, `image` is subtracted by the previous fitted group.

Parameters

image : numpy.ndarray

Background-subtracted image.

star_groups : [Table](#)

This table must contain the following columns: `id`, `group_id`, `x_0`, `y_0`, `flux_0`. `x_0` and `y_0` are initial estimates of the centroids and `flux_0` is an initial estimate of the flux.

Returns

result_tab : [Table](#)

Astropy table that contains photometry results.

image : numpy.ndarray

Residual image.

BiweightLocationBackground

class photutils.BiweightLocationBackground

Bases: [photutils.BackgroundBase](#)

Class to calculate the background in an array using the biweight location.

Parameters

c : float, optional

Tuning constant for the biweight estimator. Default value is 6.0.

M : float, optional

Initial guess for the biweight location. Default value is `None`.

sigma_clip : [SigmaClip](#) object, optional

A [SigmaClip](#) object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, BiweightLocationBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = BiweightLocationBackground(sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BiweightMidvarianceBackgroundRMS

`class photutils.BiweightMidvarianceBackgroundRMS`

Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as the (sigma-clipped) biweight midvariance.

Parameters

`c` : float, optional

Tuning constant for the biweight estimator. Default value is 9.0.

`M` : float, optional

Initial guess for the biweight location. Default value is `None`.

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, BiweightMidvarianceBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = BiweightMidvarianceBackgroundRMS(sigma_clip=sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
30.094338485893392
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
30.094338485893392
```

Methods Summary

<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.
--	-------------------------------------

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

BkgIDWInterpolator

`class photutils.BkgIDWInterpolator(leafsize=10, n_neighbors=10, power=1.0, reg=0.0)`
Bases: `object`

This class generates full-sized background and background RMS images from lower-resolution mesh images using inverse-distance weighting (IDW) interpolation (`ShepardIDWInterpolator`).

This class must be used in concert with the `Background2D` class.

Parameters

leafsize : float, optional

The number of points at which the k-d tree algorithm switches over to brute-force. leafsize must be positive. See [scipy.spatial.cKDTree](#) for further information.

n_neighbors : int, optional

The maximum number of nearest neighbors to use during the interpolation.

power : float, optional

The power of the inverse distance used for the interpolation weights.

reg : float, optional

The regularization parameter. It may be used to control the smoothness of the interpolator.

Methods Summary

<code>__call__(mesh, bkg2d_obj)</code>	Resize the 2D mesh array.
--	---------------------------

Methods Documentation

`__call__(mesh, bkg2d_obj)`

Resize the 2D mesh array.

Parameters

mesh : 2D `ndarray`

The low-resolution 2D mesh array.

bkg2d_obj : `Background2D` object

The `Background2D` object that prepared the `mesh` array.

Returns

result : 2D `ndarray`

The resized background or background RMS image.

BkgZoomInterpolator

`class photutils.BkgZoomInterpolator(order=3, mode=u'reflect', cval=0.0)`

Bases: `object`

This class generates full-sized background and background RMS images from lower-resolution mesh images using the `zoom` (spline) interpolator.

This class must be used in concert with the `Background2D` class.

Parameters

order : int, optional

The order of the spline interpolation used to resize the low-resolution background and background RMS mesh images. The value must be an integer in the range 0-5. The default is 3 (bicubic interpolation).

mode : {‘reflect’, ‘constant’, ‘nearest’, ‘wrap’}, optional

Points outside the boundaries of the input are filled according to the given mode. Default is ‘reflect’.

eval : float, optional

The value used for points outside the boundaries of the input if mode='constant'. Default is 0.0.

Methods Summary

<code>__call__(mesh, bkg2d_obj)</code>	Resize the 2D mesh array.
--	---------------------------

Methods Documentation

`__call__(mesh, bkg2d_obj)`

Resize the 2D mesh array.

Parameters

mesh : 2D `ndarray`

The low-resolution 2D mesh array.

bkg2d_obj : `Background2D` object

The `Background2D` object that prepared the mesh array.

Returns

result : 2D `ndarray`

The resized background or background RMS image.

CircularAnnulus

`class photutils.CircularAnnulus`

Bases: `photutils.CircularMaskMixin, photutils.PixelAperture`

Circular annulus aperture(s), defined in pixel coordinates.

Parameters

positions : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

r_in : float

The inner radius of the annulus.

r_out : float

The outer radius of the annulus.

Raises

ValueError : `ValueError`

If inner radius (`r_in`) is greater than outer radius (`r_out`).

ValueError : `ValueError`

If inner radius (`r_in`) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

plot(`origin=(0, 0)`, `indices=None`, `ax=None`, `fill=False`, `kwargs`)**

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

CircularAperture

`class photutils.CircularAperture`

Bases: `photutils.CircularMaskMixin`, `photutils.PixelAperture`

Circular aperture(s), defined in pixel coordinates.

Parameters**positions** : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.**r** : float

The radius of the aperture(s), in pixels.

Raises**ValueError** : `ValueError`If the input radius, `r`, is negative.**Methods Summary**

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation**area()**

Return the exact area of the aperture shape.

Returns**area** : float

The aperture area.

plot(`origin=(0, 0)`, `indices=None`, `ax=None`, `fill=False`, `**kwargs`)Plot the aperture(s) on a matplotlib `Axes` instance.**Parameters****origin** : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optionalIf `None`, then the current `Axes` instance is used.**fill** : bool, optionalSet whether to fill the aperture patch. The default is `False`.**kwargs**Any keyword arguments accepted by `matplotlib.patches.Patch`.

CircularMaskMixin

```
class photutils.CircularMaskMixin
    Bases: object
```

Mixin class to create masks for circular and circular-annulus aperture objects.

Methods Summary

<code>to_mask([method, subpixels])</code>	Return a list of <code>ApertureMask</code> objects, one for each aperture position.
---	---

Methods Documentation

`to_mask(method=u'exact', subpixels=5)`

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

`subpixels` : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels` $\star\star$ 2 subpixels.

Returns

`mask` : list of `ApertureMask`

A list of aperture mask objects.

CosineBellWindow

```
class photutils.CosineBellWindow(alpha)
    Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D cosine bell window function.

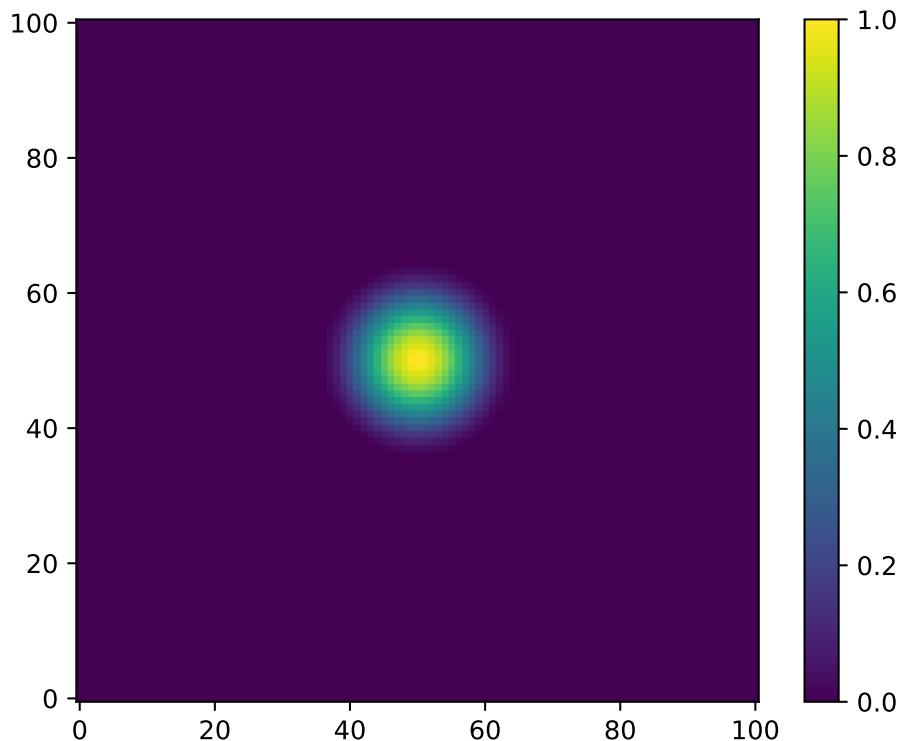
Parameters

`alpha` : float, optional

The percentage of array values that are tapered.

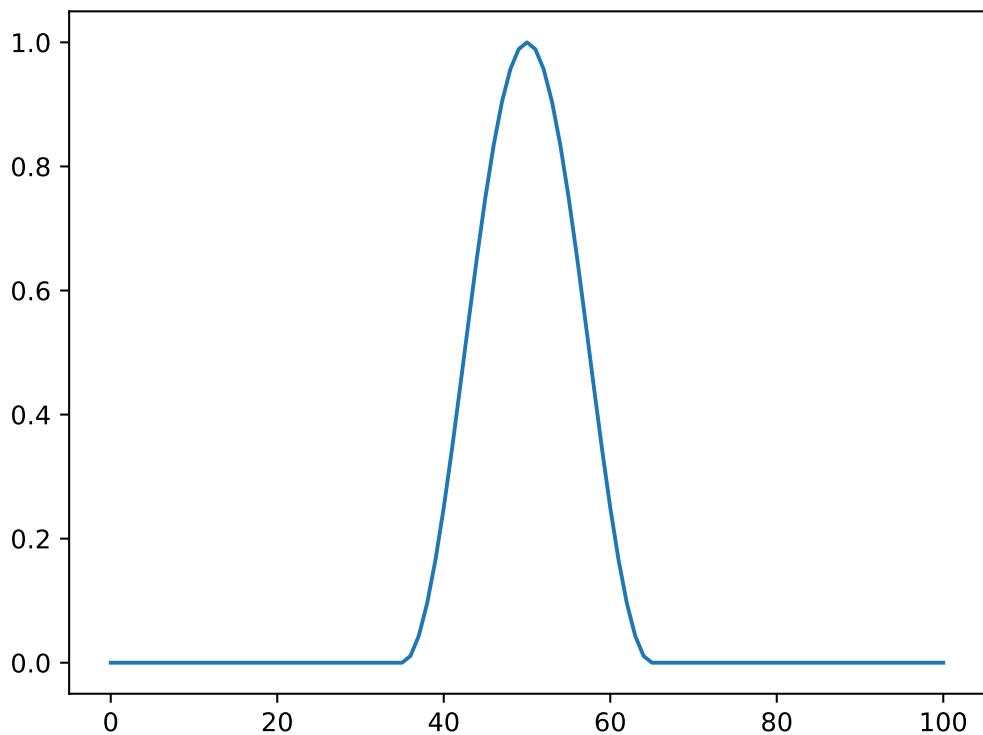
Examples

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import CosineBellWindow
taper = CosineBellWindow(alpha=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



DAOGroup

class photutils.**DAOGroup**(*crit_separation*)
Bases: photutils.psf.GroupStarsBase

This class implements the DAOGROUP algorithm presented by Stetson (1987).

The method `group_stars` divides an entire starlist into sets of distinct, self-contained groups of mutually overlapping stars. It accepts as input a list of stars and determines which stars are close enough to be capable of adversely influencing each others' profile fits.

Parameters

crit_separation : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

See also:

photutils.DAOStarFinder

Notes

Assuming the psf fwhm to be known, `crit_separation` may be set to $k * \text{fwhm}$, for some positive real k .

References

- [1] Stetson, Astronomical Society of the Pacific, Publications,
(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP..99..191S>

Attributes Summary

`crit_separation`

Methods Summary

<code>find_group(star, starlist)</code>	Find the ids of those stars in <code>starlist</code> which are at a distance less than <code>crit_separation</code> from <code>star</code> .
<code>group_stars(starlist)</code>	Classify stars into groups.

Attributes Documentation

`crit_separation`

Methods Documentation

`find_group(star, starlist)`

Find the ids of those stars in `starlist` which are at a distance less than `crit_separation` from `star`.

Parameters

`star` : `Row`

Star which will be either the head of a cluster or an isolated one.

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

Array containing the ids of those stars which are at a distance less than `crit_separation` from `star`.

`group_stars(starlist)`

Classify stars into groups.

Parameters

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : `Table`

starlist with an additional column named group_id whose unique values represent groups of mutually overlapping stars.

DAOPhotPSFPhotometry

```
class photutils.DAOPhotPSFPhotometry(crit_separation, threshold, fwhm, psf_model, fitshape,
                                       sigma=3.0, ratio=1.0, theta=0.0, sigma_radius=1.5,
                                       sharpl0=0.2, sharphi=1.0, roundlo=-1.0, roundhi=1.0, fit-
                                       ter=<astropy.modeling.fitting.LevMarLSQFitter object>,
                                       niters=3, aperture_radius=None)
```

Bases: photutils.psf.IterativelySubtractedPSFPhotometry

This class implements an iterative algorithm based on the DAOPHOT algorithm presented by Stetson (1987) to perform point spread function photometry in crowded fields. This consists of applying a loop of find sources, make groups, fit groups, subtract groups, and then repeat until no more stars are detected or a given number of iterations is reached.

Basically, this classes uses [IterativelySubtractedPSFPhotometry](#), but with grouping, finding, and background estimation routines defined a priori. More precisely, this class uses [DAOGroup](#) for grouping, [DAOStarFinder](#) for finding sources, and [MMMBoundary](#) for background estimation. Those classes are based on GROUP, FIND, and SKY routines used in DAOPHOT, respectively.

The parameter crit_separation is associated with [DAOGroup](#). sigma_clip is associated with [MMMBoundary](#). threshold and fwhm are associated with [DAOStarFinder](#). Parameters from ratio to roundhi are also associated with [DAOStarFinder](#).

Parameters

crit_separation : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

threshold : float

The absolute image value above which to select sources.

fwhm : float

The full-width half-maximum (FWHM) of the major axis of the Gaussian kernel in units of pixels.

psf_model : [astropy.modeling.Fittable2DModel](#) instance

PSF or PRF model to fit the data. Could be one of the models in this package like [DiscretePRF](#), [IntegratedGaussianPRF](#), or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as x_0, y_0 and flux. [prepare_psf_model](#) can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of fitshape must be an odd number.

sigma : float, optional

Number of standard deviations used to perform sigma clip with a [SigmaClip](#) object.

ratio : float, optional

The ratio of the minor to major axis standard deviations of the Gaussian kernel. **ratio** must be strictly positive and less than or equal to 1.0. The default is 1.0 (i.e., a circular Gaussian kernel).

theta : float, optional

The position angle (in degrees) of the major axis of the Gaussian kernel measured counter-clockwise from the positive x axis.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [1
 $\text{sigma} = \text{FWHM} / (2.0 * \sqrt{2.0 * \log(2.0)}))$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

fitter : `Fitter` instance

Fitter object used to compute the optimized centroid positions and/or flux of the identified sources. See `fitting` for more details on fitters.

niters : int or None

Number of iterations to perform of the loop FIND, GROUP, SUBTRACT, NSTAR. If None, iterations will proceed until no more stars remain. Note that in this case it is *possible* that the loop will never end if the PSF has structure that causes subtraction to create new sources infinitely.

aperture_radius : float

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `'psf_model'`.

Notes

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a `star_groups` table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, Astronomical Society of the Pacific, Publications,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP...99..191S>

DAOStarFinder

class photutils.DAOStarFinder
Bases: photutils.StarFinderBase

Detect stars in an image using the DAOFIND (Stetson 1987) algorithm.

DAOFIND (Stetson 1987; PASP 99, 191) searches images for local density maxima that have a peak amplitude greater than threshold (approximately; threshold is applied to a convolved image) and have a size and shape similar to the defined 2D Gaussian kernel. The Gaussian kernel is defined by the fwhm, ratio, theta, and sigma_radius input parameters.

DAOStarFinder finds the object centroid by fitting the marginal x and y 1D distributions of the Gaussian kernel to the marginal x and y distributions of the input (unconvolved) data image.

DAOStarFinder calculates the object roundness using two methods. The roundlo and roundhi bounds are applied to both measures of roundness. The first method (roundness1; called SROUND in DAOFIND) is based on the source symmetry and is the ratio of a measure of the object's bilateral (2-fold) to four-fold symmetry. The second roundness statistic (roundness2; called GROUND in DAOFIND) measures the ratio of the difference in the height of the best fitting Gaussian function in x minus the best fitting Gaussian function in y, divided by the average of the best fitting Gaussian functions in x and y. A circular source will have a zero roundness. An source extended in x or y will have a negative or positive roundness, respectively.

The sharpness statistic measures the ratio of the difference between the height of the central pixel and the mean of the surrounding non-bad pixels in the convolved image, to the height of the best fitting Gaussian function at that point.

Parameters

threshold : float

The absolute image value above which to select sources.

fwhm : float

The full-width half-maximum (FWHM) of the major axis of the Gaussian kernel in units of pixels.

ratio : float, optional

The ratio of the minor to major axis standard deviations of the Gaussian kernel. ratio must be strictly positive and less than or equal to 1.0. The default is 1.0 (i.e., a circular Gaussian kernel).

theta : float, optional

The position angle (in degrees) of the major axis of the Gaussian kernel measured counter-clockwise from the positive x axis.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [$1 \text{ sigma} = \text{FWHM} / (2.0 * \sqrt{2.0 * \log(2.0)})$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

sky : float, optional

The background sky level of the image. Setting sky affects only the output values of the object peak, flux, and mag values. The default is 0.0, which should be used to replicate the results from `DAOFIND`.

exclude_border : bool, optional

Set to `True` to exclude sources found within half the size of the convolution kernel from the image borders. The default is `False`, which is the mode used by `DAOFIND`.

See also:

[IRAFStarFinder](#)

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in `DAOFIND` are `boundary='constant'` and `constant=0.0`.

References

[R1], [R2]

Methods Summary

`find_stars(data)`

Find stars in an astronomical image.

Methods Documentation

`find_stars(data)`

Find stars in an astronomical image.

Parameters

data : array_like

The 2D image array.

Returns

table : `Table`

A table of found objects with the following parameters:

- **id**: unique object identification number.
- **xcentroid**, **ycentroid**: object centroid.
- **sharpness**: object sharpness.
- **roundness1**: object roundness based on symmetry.
- **roundness2**: object roundness based on marginal Gaussian fits.
- **npix**: number of pixels in the Gaussian kernel.

- `sky`: the input sky parameter.
- `peak`: the peak, sky-subtracted, pixel value of the object.
- `flux`: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of `DAOFIND` if `sky` is 0.0.
- `mag`: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of `DAOFIND` if `sky` is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF's `starfind` are `boundary='constant'` and `constant=0.0`.

IRAF's `starfind` uses `hwhmpsf`, `fradius`, and `sepmin` as input parameters. The equivalent input values for `IRAFStarFinder` are:

- `FWHM = hwhmpsf * 2`
- `Sigma_radius = fradius * sqrt(2.0*log(2.0))`
- `MinSep_FWHM = 0.5 * sepmin`

The main differences between `DAOStarFinder` and `IRAFStarFinder` are:

- `IRAFStarFinder` always uses a 2D circular Gaussian kernel, while `DAOStarFinder` can use an elliptical Gaussian kernel.
- `IRAFStarFinder` calculates the objects' centroid, roundness, and sharpness using image moments.

DBSCANGroup

```
class photutils.DBSCANGroup(crit_separation, min_samples=1, metric='euclidean', algorithm='auto',
                             leaf_size=30)
Bases: photutils.psf.GroupStarsBase
```

Class to create star groups according to a distance criteria using the Density-based Spatial Clustering of Applications with Noise (DBSCAN) from scikit-learn.

Parameters

crit_separation : float or int

Distance, in units of pixels, such that any two stars separated by less than this distance will be placed in the same group.

min_samples : int, optional (default=1)

Minimum number of stars necessary to form a group.

metric : string or callable (default='euclidean')

The metric to use when calculating distance between each pair of stars.

algorithm : {‘auto’, ‘ball_tree’, ‘kd_tree’, ‘brute’}, optional

The algorithm to be used to actually find nearest neighbors.

leaf_size : int, optional (default = 30)

Leaf size passed to BallTree or cKDTree.

Notes

- The attribute `crit_separation` corresponds to `eps` in `sklearn.cluster.DBSCAN`.
- This class provides more general algorithms than `photutils.psf.DAOGroup`. More precisely, `photutils.psf.DAOGroup` is a special case of `photutils.psf.DBSCANGroup` when `min_samples=1` and `metric=euclidean`. Additionally, `photutils.psf.DBSCANGroup` may be faster than `photutils.psf.DAOGroup`.

References

[1] Scikit Learn DBSCAN.

<http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN>

Methods Summary

<code>group_stars(starlist)</code>	Classify stars into groups.
------------------------------------	-----------------------------

Methods Documentation

group_stars(starlist)
Classify stars into groups.

Parameters

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : `Table`

`starlist` with an additional column named `group_id` whose unique values represent groups of mutually overlapping stars.

EllipticalAnnulus

`class photutils.EllipticalAnnulus`

Bases: `photutils.EllipticalMaskMixin`, `photutils.PixelAperture`

Elliptical annulus aperture(s), defined in pixel coordinates.

Parameters

`positions` : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (`x`, `y`) tuple
- list of (`x`, `y`) tuples
- `Nx2` or `2xN` `ndarray`
- `Nx2` or `2xN` `Quantity` in pixel units

Note that a 2×2 `ndarray` or `Quantity` is interpreted as $N \times 2$, i.e. two rows of (x, y) coordinates.

a_in : float

The inner semimajor axis.

a_out : float

The outer semimajor axis.

b_out : float

The outer semiminor axis. The inner semiminor axis is calculated as:

$$b_{in} = b_{out} \left(\frac{a_{in}}{a_{out}} \right)$$

theta : float

The rotation angle in radians of the semimajor axis from the positive x axis. The rotation angle increases counterclockwise.

Raises

ValueError : `ValueError`

If inner semimajor axis (`a_in`) is greater than outer semimajor axis (`a_out`).

ValueError : `ValueError`

If either the inner semimajor axis (`a_in`) or the outer semiminor axis (`b_out`) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

area : float

The aperture area.

plot(`origin=(0, 0)`, `indices=None`, `ax=None`, `fill=False`, `kwargs`)**

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

origin : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional
If `None`, then the current `Axes` instance is used.

fill : bool, optional
Set whether to fill the aperture patch. The default is `False`.

kwargs
Any keyword arguments accepted by `matplotlib.patches.Patch`.

EllipticalAperture

```
class photutils.EllipticalAperture
    Bases: photutils.EllipticalMaskMixin, photutils.PixelAperture

    Elliptical aperture(s), defined in pixel coordinates.

    Parameters
        positions : array_like or Quantity
            Pixel coordinates of the aperture center(s) in one of the following formats:
            •single (x, y) tuple
            •list of (x, y) tuples
            •Nx2 or 2xN ndarray
            •Nx2 or 2xN Quantity in pixel units
            Note that a 2x2 ndarray or Quantity is interpreted as Nx2, i.e. two rows of (x, y)
            coordinates.
        a : float
            The semimajor axis.
        b : float
            The semiminor axis.
        theta : float
            The rotation angle in radians of the semimajor axis from the positive x axis. The rotation
            angle increases counterclockwise.

    Raises
        ValueError : ValueError
            If either axis (a or b) is negative.
```

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

`area()`

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

`plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)`

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : matplotlib.axes.Axes instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

EllipticalMaskMixin

`class photutils.EllipticalMaskMixin`

Bases: `object`

Mixin class to create masks for elliptical and elliptical-annulus aperture objects.

Methods Summary

`to_mask([method, subpixels])`

Return a list of `ApertureMask` objects, one for each aperture position.

Methods Documentation

`to_mask(method=u'exact', subpixels=5)`

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- 'exact' (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- 'center': A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- 'subpixel': A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

mask : list of `ApertureMask`

A list of aperture mask objects.

FittableImageModel

```
class photutils.FittableImageModel  
    Bases: astropy.modeling.Fittable2DModel
```

A fittable 2D model of an image allowing for image intensity scaling and image translations.

This class takes 2D image data and computes the values of the model at arbitrary locations (including at intra-pixel, fractional positions) within this image using spline interpolation provided by `RectBivariateSpline`.

The fittable model provided by this class has three model parameters: an image intensity scaling factor (`flux`) which is applied to (normalized) image, and two positional parameters (`x_0` and `y_0`) indicating the location of a feature in the coordinate grid on which the model is to be evaluated.

If this class is initialized with `flux` (intensity scaling factor) set to `None`, then `flux` is going to be estimated as `sum(data)`.

Parameters

data : numpy.ndarray

Array containing 2D image.

origin : tuple, None, optional

A reference point in the input image data array. When origin is `None`, origin will be set at the middle of the image array.

If `origin` represents the location of a feature (e.g., the position of an intensity peak) in the input data, then model parameters `x_0` and `y_0` show the location of this peak in an another target image to which this model was fitted. Fundamentally, it is the coordinate in the model's image data that should map to coordinate (`x_0`, `y_0`) of the output coordinate system on which the model is evaluated.

Alternatively, when `origin` is set to `(0, 0)`, then model parameters `x_0` and `y_0` are shifts by which model's image should be translated in order to match a target image.

normalize : bool, optional

Indicates whether or not the model should be build on normalized input image data. If true, then the normalization constant (N) is computed so that

$$N \cdot C \cdot \sum_{i,j} D_{i,j} = 1,$$

where N is the normalization constant, C is correction factor given by the parameter `normalization_correction`, and $D_{i,j}$ are the elements of the input image data array.

normalization_correction : float, optional

A strictly positive number that represents correction that needs to be applied to model's data normalization (see C in the equation in the comments to `normalize` for more details).

A possible application for this parameter is to account for aperture correction. Assuming model's data represent a PSF to be fitted to some target star, we set `normalization_correction` to the aperture correction that needs to be applied to the model. That is, `normalization_correction` in this case should be set to the ratio between the total flux of the PSF (including flux outside model's data) to the flux of model's data. Then, best fitted value of the `flux` model parameter will represent an aperture-corrected flux of the target star.

fill_value : float, optional

The value to be returned by the `evaluate` or `astropy.modeling.Model.__call__` methods when evaluation is performed outside the definition domain of the model.

ikwargs : dict, optional

Additional optional keyword arguments to be passed directly to the `compute_interpolator` method. See `compute_interpolator` for more details.

Attributes Summary

<code>data</code>	Get original image data.
<code>fill_value</code>	Fill value to be returned for coordinates outside of the domain of definition of the interpolator.
<code>flux</code>	Intensity scaling factor for image data.
<code>interpolator_kwarg</code> s	Get current interpolator's arguments used when interpolator was created.
<code>normalization_constant</code>	Get normalization constant.
<code>normalization_correction</code>	Set/Get flux correction factor.
<code>normalization_status</code>	Get normalization status.
<code>normalized_data</code>	Get normalized and/or intensity-corrected image data.
<code>nx</code>	Number of columns in the data array.
<code>ny</code>	Number of rows in the data array.
<code>origin</code>	A tuple of x and y coordinates of the origin of the coordinate system in terms of pixels of model's image.
<code>oversampling</code>	The factor by which the stored image is oversampled.
<code>param_names</code>	
<code>shape</code>	A tuple of dimensions of the data array in numpy style (ny, nx).
<code>x_0</code>	X-position of a feature in the image in the output coordinate grid on which the model is evaluated.

Continued on next page

Table 17.25 – continued from previous page

<code>x_origin</code>	X-coordinate of the origin of the coordinate system.
<code>y_0</code>	Y-position of a feature in the image in the output coordinate grid on which the model is evaluated.
<code>y_origin</code>	Y-coordinate of the origin of the coordinate system.

Methods Summary

<code>compute_interpolator([ikwargs])</code>	Compute/define the interpolating spline.
<code>evaluate(x, y, flux, x_0, y_0)</code>	Evaluate the model on some input variables and provided model parameters.

Attributes Documentation

`data`

Get original image data.

`fill_value`

Fill value to be returned for coordinates outside of the domain of definition of the interpolator. If `fill_value` is `None`, then values outside of the domain of definition are the ones returned by the interpolator.

`flux`

Intensity scaling factor for image data.

`interpolator_kwarg`

Get current interpolator's arguments used when interpolator was created.

`normalization_constant`

Get normalization constant.

`normalization_correction`

Set/Get flux correction factor.

Note: When setting correction factor, model's flux will be adjusted accordingly such that if this model was a good fit to some target image before, then it will remain a good fit after correction factor change.

`normalization_status`

Get normalization status. Possible status values are:

- 0: **Performed**. Model has been successfully normalized at user's request.
- 1: **Failed**. Attempt to normalize has failed.
- 2: **NotRequested**. User did not request model to be normalized.

`normalized_data`

Get normalized and/or intensity-corrected image data.

`nx`

Number of columns in the data array.

`ny`

Number of rows in the data array.

`origin`

A tuple of x and y coordinates of the origin of the coordinate system in terms of pixels of model's image.

When setting the coordinate system origin, a tuple of two `int` or `float` may be used. If `origin` is set to `None`, the origin of the coordinate system will be set to the middle of the data array ($(\text{npix}-1)/2.0$).

Warning: Modifying `origin` will not adjust (modify) model's parameters `x_0` and `y_0`.

`oversampling`

The factor by which the stored image is oversampled. I.e., an input to this model is multiplied by this factor to yield the index into the stored image.

`param_names = ('flux', 'x_0', 'y_0')`

`shape`

A tuple of dimensions of the data array in numpy style (ny, nx).

`x_0`

X-position of a feature in the image in the output coordinate grid on which the model is evaluated.

`x_origin`

X-coordinate of the origin of the coordinate system.

`y_0`

Y-position of a feature in the image in the output coordinate grid on which the model is evaluated.

`y_origin`

Y-coordinate of the origin of the coordinate system.

Methods Documentation

`compute_interpolator(ikwargs={})`

Compute/define the interpolating spline. This function can be overriden in a subclass to define custom interpolators.

Parameters

`ikwargs` : dict, optional

Additional optional keyword arguments. Possible values are:

`•degree`

[int, tuple, optional] Degree of the interpolating spline. A tuple can be used to provide different degrees for the X- and Y-axes. Default value is degree=3.

`•s`

[float, optional] Non-negative smoothing factor. Default value s=0 corresponds to interpolation. See `RectBivariateSpline` for more details.

Notes

- When subclassing `FittableImageModel` for the purpose of overriding `compute_interpolator()`, the `evaluate()` may need to be overriden as well depending on the behavior of the new interpolator. In addition, for improved future compatibility, make sure that the overriding method stores keyword arguments `ikwargs` by calling `_store_interpolator_kwarg` method.
- Use caution when modifying interpolator's degree or smoothness in a computationally intensive part of the code as it may decrease code performance due to the need to recompute interpolator.

`evaluate(x, y, flux, x_0, y_0)`

Evaluate the model on some input variables and provided model parameters.

GaussianConst2D

class photutils.GaussianConst2D
Bases: astropy.modeling.Fittable2DModel

A model for a 2D Gaussian plus a constant.

Parameters

constant : float

Value of the constant.

amplitude : float

Amplitude of the Gaussian.

x_mean : float

Mean of the Gaussian in x.

y_mean : float

Mean of the Gaussian in y.

x_stddev : float

Standard deviation of the Gaussian in x. x_stddev and y_stddev must be specified unless a covariance matrix (cov_matrix) is input.

y_stddev : float

Standard deviation of the Gaussian in y. x_stddev and y_stddev must be specified unless a covariance matrix (cov_matrix) is input.

theta : float, optional

Rotation angle in radians. The rotation angle increases counterclockwise.

Other Parameters

fixed : a dict

A dictionary {parameter_name: boolean} of parameters to not be varied during fitting. True means the parameter is held fixed. Alternatively the `fixed` property of a parameter may be used.

tied : dict

A dictionary {parameter_name: callable} of parameters which are linked to some other parameter. The dictionary values are callables providing the linking relationship. Alternatively the `tied` property of a parameter may be used.

bounds : dict

A dictionary {parameter_name: boolean} of lower and upper bounds of parameters. Keys are parameter names. Values are a list of length 2 giving the desired range for the parameter. Alternatively the `min` and `max` properties of a parameter may be used.

eqcons : list

A list of functions of length n such that eqcons[j](x0,*args) == 0.0 in a successfully optimized problem.

ineqcons : list

A list of functions of length n such that ieqcons[j](x0,*args) >= 0.0 is a successfully optimized problem.

Attributes Summary

Methods Summary

<code>evaluate(x, y, constant, amplitude, x_mean, ...)</code>	Two dimensional Gaussian plus constant function.
---	--

Attributes Documentation

`amplitude`

`constant`

`param_names = ('constant', 'amplitude', 'x_mean', 'y_mean', 'x_stddev', 'y_stddev', 'theta')`

`theta`

`x_mean`

`x_stddev`

`y_mean`

`y_stddev`

Methods Documentation

`static evaluate(x, y, constant, amplitude, x_mean, y_mean, x_stddev, y_stddev, theta)`

Two dimensional Gaussian plus constant function.

GroupStarsBase

`class photutils.GroupStarsBase`

Bases: `object`

This base class provides the basic interface for subclasses that are capable of classifying stars in groups.

Methods Summary

<code>__call__(starlist)</code>	Classify stars into groups.
---------------------------------	-----------------------------

Methods Documentation

`__call__(starlist)`

Classify stars into groups.

Parameters

`starlist` : `Table`

List of star positions. Columns named as `x_0` and `y_0`, which corresponds to the centroid coordinates of the sources, must be provided.

Returns

`group_starlist` : `Table`

`starlist` with an additional column named `group_id` whose unique values represent groups of mutually overlapping stars.

HanningWindow

`class photutils.HanningWindow`

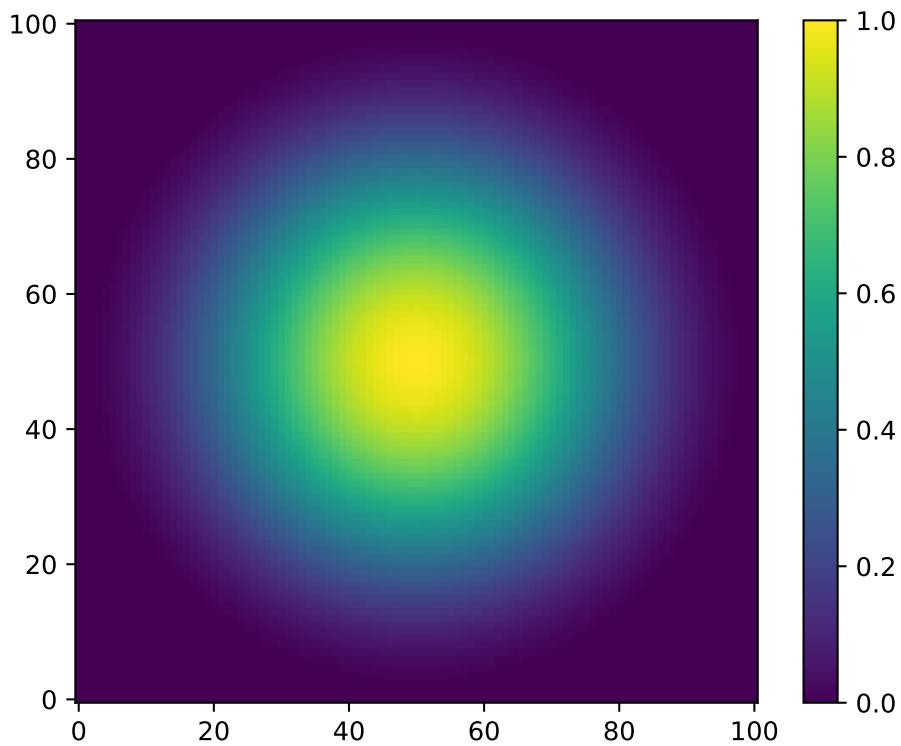
Bases: `photutils.psf.matching.SplitCosineBellWindow`

Class to define a 2D Hanning (or Hann) window function.

The Hann window is a taper formed by using a raised cosine with ends that touch zero.

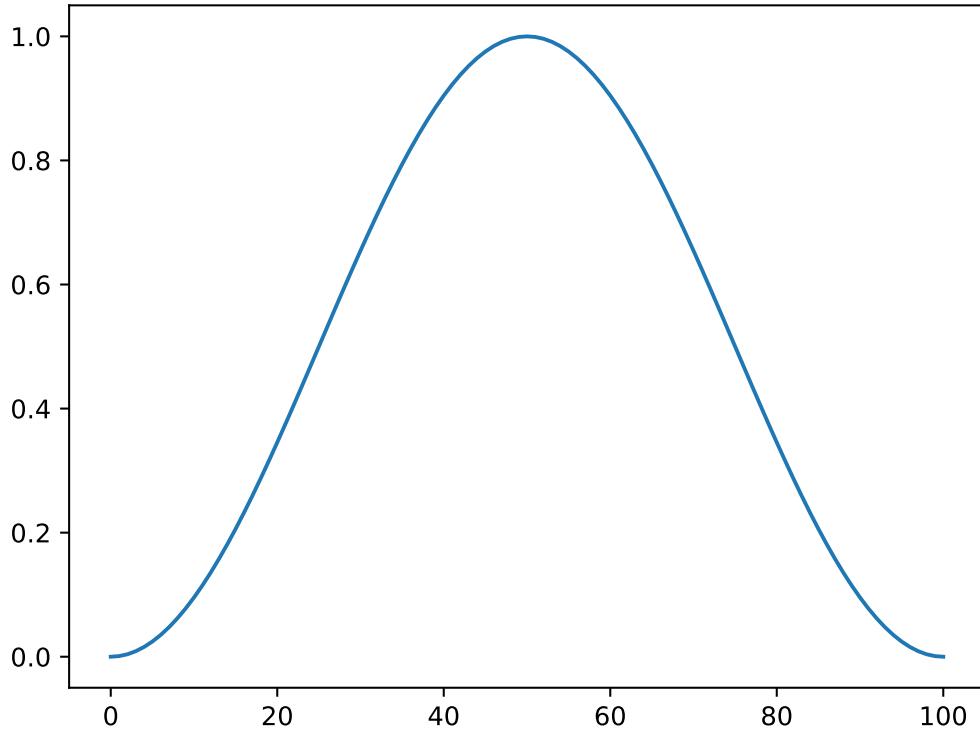
Examples

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import HanningWindow
taper = HanningWindow()
data = taper((101, 101))
plt.plot(data[50, :])
```



IRAFStarFinder

`class photutils.IRAFStarFinder`

Bases: `photutils.StarFinderBase`

Detect stars in an image using IRAF’s “starfind” algorithm.

`starfind` searches images for local density maxima that have a peak amplitude greater than `threshold` above the local background and have a PSF full-width half-maximum similar to the input `fwhm`. The objects’ centroid, roundness (ellipticity), and sharpness are calculated using image moments.

Parameters

threshold : float

The absolute image value above which to select sources.

fwhm : float

The full-width half-maximum (FWHM) of the 2D circular Gaussian kernel in units of pixels.

minsep_fwhm : float, optional

The minimum separation for detected objects in units of `fwhm`.

sigma_radius : float, optional

The truncation radius of the Gaussian kernel in units of sigma (standard deviation) [1
 $\text{sigma} = \text{FWHM} / 2.0 * \sqrt{2.0 * \log(2.0)}$].

sharplo : float, optional

The lower bound on sharpness for object detection.

sharphi : float, optional

The upper bound on sharpness for object detection.

roundlo : float, optional

The lower bound on roundness for object detection.

roundhi : float, optional

The upper bound on roundness for object detection.

sky : float, optional

The background sky level of the image. Inputting a sky value will override the background sky estimate. Setting sky affects only the output values of the object peak, flux, and mag values. The default is None, which means the sky value will be estimated using the `starfind` method.

exclude_border : bool, optional

Set to `True` to exclude sources found within half the size of the convolution kernel from the image borders. The default is `False`, which is the mode used by `starfind`.

See also:

[DAOStarFinder](#)

References

[R3]

Methods Summary

`find_stars(data)`

Find stars in an astronomical image.

Methods Documentation

`find_stars(data)`

Find stars in an astronomical image.

Parameters

`data` : array_like

The 2D image array.

Returns

`table` : `Table`

A table of found objects with the following parameters:

- `id`: unique object identification number.
- `xcentroid`, `ycentroid`: object centroid.
- `sharpness`: object sharpness.

- roundness1: object roundness based on symmetry.
- roundness2: object roundness based on marginal Gaussian fits.
- npix: number of pixels in the Gaussian kernel.
- sky: the input sky parameter.
- peak: the peak, sky-subtracted, pixel value of the object.
- flux: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of [DAOFLUX](#) if sky is 0.0.
- mag: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of [DAOFLUX](#) if sky is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF's [starfind](#) are boundary='constant' and constant=0.0.

IRAF's [starfind](#) uses hwhm, fradius, and sepmin as input parameters. The equivalent input values for [IRAFStarFinder](#) are:

- fwhm = hwhm * 2
- sigma_radius = fradius * sqrt(2.0*log(2.0))
- minsep_fwhm = 0.5 * sepmin

The main differences between [DAOStarFinder](#) and [IRAFStarFinder](#) are:

- [IRAFStarFinder](#) always uses a 2D circular Gaussian kernel, while [DAOStarFinder](#) can use an elliptical Gaussian kernel.
- [IRAFStarFinder](#) calculates the objects' centroid, roundness, and sharpness using image moments.

IntegratedGaussianPRF

```
class photutils.IntegratedGaussianPRF
    Bases: astropy.modeling.Fittable2DModel
```

Circular Gaussian model integrated over pixels. Because it is integrated, this model is considered a PRF, *not* a PSF (see [Terminology](#) for more about the terminology used here.)

This model is a Gaussian *integrated* over an area of 1 (in units of the model input coordinates, e.g. 1 pixel). This is in contrast to the apparently similar [astropy.modeling.functional_models.Gaussian2D](#), which is the value of a 2D Gaussian *at* the input coordinates, with no integration. So this model is equivalent to assuming the PSF is Gaussian at a *sub-pixel* level.

Parameters

sigma : float

Width of the Gaussian PSF.

flux : float (default 1)

Total integrated flux over the entire PSF

x_0 : float (default 0)

Position of the peak in x direction.

y_0 : float (default 0)

Position of the peak in y direction.

Notes

This model is evaluated according to the following formula:

$$f(x, y) = \frac{F}{4} \left[\operatorname{erf} \left(\frac{x - x_0 + 0.5}{\sqrt{2}\sigma} \right) - \operatorname{erf} \left(\frac{x - x_0 - 0.5}{\sqrt{2}\sigma} \right) \right] \left[\operatorname{erf} \left(\frac{y - y_0 + 0.5}{\sqrt{2}\sigma} \right) - \operatorname{erf} \left(\frac{y - y_0 - 0.5}{\sqrt{2}\sigma} \right) \right]$$

where erf denotes the error function and F the total integrated flux.

Attributes Summary

`fit_deriv`

`flux`

`param_names`

`sigma`

`x_0`

`y_0`

Methods Summary

`evaluate(x, y, flux, x_0, y_0, sigma)`

Model function Gaussian PSF model.

Attributes Documentation

fit_deriv = `None`

flux

param_names = (`'flux'`, `'x_0'`, `'y_0'`, `'sigma'`)

sigma

x_0

y_0

Methods Documentation

evaluate(*x, y, flux, x_0, y_0, sigma*)
 Model function Gaussian PSF model.

IterativelySubtractedPSFPhotometry

```
class photutils.IterativelySubtractedPSFPhotometry(group_maker, bkg_estimator,
                                                 psf_model, fitshape, finder, fitter=<astropy.modeling.fitting.LevMarLSQFitter
                                                 object>, nitors=3, aperture_radius=None)
```

Bases: `photutils.psf.BasicPSFPhotometry`

This class implements an iterative algorithm to perform point spread function photometry in crowded fields. This consists of applying a loop of find sources, make groups, fit groups, subtract groups, and then repeat until no more stars are detected or a given number of iterations is reached.

Parameters

group_maker : callable or `GroupStarsBase`

`group_maker` should be able to decide whether a given star overlaps with any other and label them as belonging to the same group. `group_maker` receives as input an `Table` object with columns named as `id`, `x_0`, `y_0`, in which `x_0` and `y_0` have the same meaning of `xcentroid` and `ycentroid`. This callable must return an `Table` with columns `id`, `x_0`, `y_0`, and `group_id`. The column `group_id` should contain integers starting from 1 that indicate which group a given source belongs to. See, e.g., `DAOGroup`.

bkg_estimator : callable, instance of any `BackgroundBase` subclass, or None

`bkg_estimator` should be able to compute either a scalar background or a 2D background of a given 2D image. See, e.g., `MedianBackground`. If None, no background subtraction is performed.

psf_model : `astropy.modeling.Fittable2DModel` instance

PSF or PRF model to fit the data. Could be one of the models in this package like `DiscretePRF`, `IntegratedGaussianPRF`, or any other suitable 2D model. This object needs to identify three parameters (position of center in x and y coordinates and the flux) in order to set them to suitable starting values for each fit. The names of these parameters should be given as `x_0`, `y_0` and `flux`. `prepare_psf_model` can be used to prepare any 2D model to match this assumption.

fitshape : int or length-2 array-like

Rectangular shape around the center of a star which will be used to collect the data to do the fitting. Can be an integer to be the same along both axes. E.g., 5 is the same as (5, 5), which means to fit only at the following relative pixel positions: [-2, -1, 0, 1, 2]. Each element of `fitshape` must be an odd number.

finder : callable or instance of any `StarFinderBase` subclasses

`finder` should be able to identify stars, i.e. compute a rough estimate of the centroids, in a given 2D image. `finder` receives as input a 2D image and returns an `Table` object which contains columns with names: `id`, `xcentroid`, `ycentroid`, and `flux`. In which `id` is an integer-valued column starting from 1, `xcentroid` and `ycentroid` are center position estimates of the sources and `flux` contains flux estimates of the sources. See, e.g., `DAOStarFinder` or `IRAFStarFinder`.

fitter : `Fitter` instance

Fitter object used to compute the optimized centroid positions and/or flux of the identified sources. See `fitting` for more details on fitters.

aperture_radius : float

The radius (in units of pixels) used to compute initial estimates for the fluxes of sources. If None, one FWHM will be used if it can be determined from the `'psf_model'`.

niters : int or None

Number of iterations to perform of the loop FIND, GROUP, SUBTRACT, NSTAR. If None, iterations will proceed until no more stars remain. Note that in this case it is *possible* that the loop will never end if the PSF has structure that causes subtraction to create new sources infinitely.

Notes

If there are problems with fitting large groups, change the parameters of the grouping algorithm to reduce the number of sources in each group or input a `star_groups` table that only includes the groups that are relevant (e.g. manually remove all entries that coincide with artifacts).

References

[1] Stetson, Astronomical Society of the Pacific, Publications,

(ISSN 0004-6280), vol. 99, March 1987, p. 191-222. Available at: <http://adsabs.harvard.edu/abs/1987PASP...99..191S>

Attributes Summary

`finder`

`niters`

Methods Summary

`do_photometry(image[, positions])`

Perform PSF photometry in `image`.

Attributes Documentation

finder

niters

Methods Documentation

`do_photometry(image, positions=None)`

Perform PSF photometry in `image`.

This method assumes that `psf_model` has centroids and flux parameters which will be fitted to the data provided in `image`. A compound model, in fact a sum of `psf_model`, will be fitted to groups of stars automatically identified by `group_maker`. Also, `image` is not assumed to be background subtracted. If `positions` are not `None` then this method uses `positions` as initial guesses for the centroids. If the centroid positions are set as fixed in the PSF model `psf_model`, then the optimizer will only consider the flux as a variable.

Parameters

`image` : 2D array-like, `ImageHDU`, `HDUList`

Image to perform photometry.

`positions`: ‘`~astropy.table.Table`‘

Positions (in pixel coordinates) at which to *start* the fit for each object. Columns ‘`x_0`’ and ‘`y_0`’ must be present. ‘`flux_0`’ can also be provided to set initial fluxes. If ‘`flux_0`’ is not provided, aperture photometry is used to estimate initial values for the fluxes.

Returns

`output_table` : `Table` or `None`

Table with the photometry results, i.e., centroids and fluxes estimations and the initial estimates used to start the fitting process. `None` is returned if no sources are found in `image`.

MADStdBackgroundRMS

`class photutils.MADStdBackgroundRMS`

Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as using the median absolute deviation (MAD).

The standard deviation estimator is given by:

$$\sigma \approx \frac{\text{MAD}}{\Phi^{-1}(3/4)} \approx 1.4826 \text{ MAD}$$

where $\Phi^{-1}(P)$ is the normal inverse cumulative distribution function evaluated at probability $P = 3/4$.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, MADStdBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = MADStdBackgroundRMS(sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
37.065055462640053
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
37.065055462640053
```

Methods Summary

<code>calc_background_rms(data[, axis])</code>	Calculate the background RMS value.
--	-------------------------------------

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

`axis` : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

MMMBBackground

`class photutils.MMMBackground`

Bases: `photutils.ModeEstimatorBackground`

Class to calculate the background in an array using the DAOPHOT MMM algorithm.

The background is calculated using a mode estimator of the form $(3 * \text{median}) - (2 * \text{mean})$.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MMMBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MMMBackground(sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

MeanBackground

`class photutils.MeanBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array as the (sigma-clipped) mean.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MeanBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MeanBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

`calc_background(data[, axis])`

Calculate the background value.

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

MedianBackground

`class photutils.MedianBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array as the (sigma-clipped) median.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, MedianBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = MedianBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

ModeEstimatorBackground

`class photutils.ModeEstimatorBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array using a mode estimator of the form (`median_factor * median`) - (`mean_factor * mean`).

Parameters

`median_factor` : float, optional

The multiplicative factor for the data median. Defaults to 3.

`mean_factor` : float, optional

The multiplicative factor for the data mean. Defaults to 2.

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, ModeEstimatorBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = ModeEstimatorBackground(median_factor=3., mean_factor=2.,
...                                 sigma_clip=sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

NonNormalizable

`exception photutils.NonNormalizable`

Used to indicate that a `FittableImageModel` model is non-normalizable.

PRFAdapter

`class photutils.PRFAdapter`

Bases: `astropy.modeling.Fittable2DModel`

A model that adapts a supplied PSF model to act as a PRF. It integrates the PSF model over pixel “boxes”. A critical built-in assumption is that the PSF model scale and location parameters are in *pixel* units.

Parameters

`psfmodel` : a 2D model

The model to assume as representative of the PSF

`renormalize_psf` : bool

If True, the model will be integrated from -inf to inf and re-scaled so that the total integrates to 1. Note that this renormalization only occurs *once*, so if the total flux of psfmodel depends on position, this will *not* be correct.

xname : str or None

The name of the psfmodel parameter that corresponds to the x-axis center of the PSF. If None, the model will be assumed to be centered at x=0.

yname : str or None

The name of the psfmodel parameter that corresponds to the y-axis center of the PSF. If None, the model will be assumed to be centered at y=0.

fluxname : str or None

The name of the `psfmodel` parameter that corresponds to the total flux of the star. If `None`, a scaling factor will be applied by the `PRFAdapter` instead of modifying the `psfmodel`.

Notes

This current implementation of this class (using numerical integration for each pixel) is extremely slow, and only suited for experimentation over relatively few small regions.

Attributes Summary

```
flux  
param_names  
x_0  
y_0
```

Methods Summary

`evaluate(x, y, flux, x_0, y_0)` The evaluation function for PRFAdapter.

Attributes Documentation

flux

```
param_names = ('flux', 'x_0', 'y_0')
```

$$x_0$$

$y=0$

Methods Documentation

evaluate(*x, y, flux, x_0, y_0*)

The evaluation function for PRFAdapter.

PixelAperture

class photutils.PixelAperture

Bases: [photutils.Aperture](#)

Abstract base class for 2D apertures defined in pixel coordinates.

Derived classes must define a `_slices` property, `to_mask` and `plot` methods, and optionally an `area` method.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>do_photometry(data[, error, ...])</code>	Perform aperture photometry on the input data.
<code>mask_area([method, subpixels])</code>	Return the area of the aperture(s) mask.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.
<code>to_mask([method, subpixels])</code>	Return a list of <code>ApertureMask</code> objects, one for each aperture position.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

do_photometry(*data, error=None, pixelwise_error=True, mask=None, method='exact', subpixels=5, unit=None*)

Perform aperture photometry on the input data.

Parameters

`data` : array_like or [Quantity](#) instance

The 2D array on which to perform photometry. `data` should be background subtracted.

`error` : array_like or [Quantity](#), optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include all sources of error, including the Poisson error of the sources (see `calc_total_error`). `.error` must have the same shape as the input data.

`pixelwise_error` : bool, optional

If `True` (default), the photometric error is calculated using the `error` values from each pixel within the aperture. If `False`, the `error` value at the center of the aperture is used for the entire aperture.

`mask` : array_like (bool), optional

A boolean mask with the same shape as data where a `True` value indicates the corresponding element of data is masked. Masked data are excluded from all calculations.

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

unit : `UnitBase` object or str, optional

An object that represents the unit associated with the input data and error arrays. Must be a `UnitBase` object or a string parseable by the `units` package. If data or error already have a different unit, the input unit will not be used and a warning will be raised.

Returns

aperture_sums : `ndarray` or `Quantity`

The sums within each aperture.

aperture_sum_errs : `ndarray` or `Quantity`

The errors on the sums within each aperture.

mask_area(*method=u'exact'*, *subpixels=5*)

Return the area of the aperture(s) mask.

For `method` other than ‘exact’, this area will be less than the exact analytical area (e.g. the `area` method). Note that for these methods, the values can also differ because of fractional pixel positions.

Parameters

method : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether

its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

area : float

A list of the mask area of the aperture(s).

plot(*origin=(0, 0)*, *indices=None*, *ax=None*, *fill=False*, ***kwargs*)

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

origin : array_like, optional

The (x, y) position of the origin of the displayed image.

indices : int or array of int, optional

The indices of the aperture(s) to plot.

ax : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

fill : bool, optional

Set whether to fill the aperture patch. The default is `False`.

kwargs

Any keyword arguments accepted by `matplotlib.patches.Patch`.

to_mask(*method='exact'*, *subpixels=5*)

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

method : {'exact', 'center', 'subpixel'}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- 'exact' (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- 'center': A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- 'subpixel': A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to 'center'. The returned mask will contain values between 0 and 1.

subpixels : int, optional

For the 'subpixel' method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

mask : list of `ApertureMask`

A list of aperture mask objects.

RectangularAnnulus

`class photutils.RectangularAnnulus`

Bases: `photutils.RectangularMaskMixin, photutils.PixelAperture`

Rectangular annulus aperture(s), defined in pixel coordinates.

Parameters

`positions` : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

`w_in` : float

The inner full width of the aperture. For `theta=0` the width side is along the x axis.

`w_out` : float

The outer full width of the aperture. For `theta=0` the width side is along the x axis.

`h_out` : float

The outer full height of the aperture. The inner full height is calculated as:

$$h_{in} = h_{out} \left(\frac{w_{in}}{w_{out}} \right)$$

For `theta=0` the height side is along the y axis.

`theta` : float

The rotation angle in radians of the width side from the positive x axis. The rotation angle increases counterclockwise.

Raises

`ValueError` : `ValueError`

If inner width (`w_in`) is greater than outer width (`w_out`).

`ValueError` : `ValueError`

If either the inner width (`w_in`) or the outer height (`h_out`) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

`area()`

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

`plot(origin=(0, 0), indices=None, ax=None, fill=False, **kwargs)`

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : matplotlib.axes.Axes instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

RectangularAperture

`class photutils.RectangularAperture`

Bases: `photutils.RectangularMaskMixin, photutils.PixelAperture`

Rectangular aperture(s), defined in pixel coordinates.

Parameters

`positions` : array_like or `Quantity`

Pixel coordinates of the aperture center(s) in one of the following formats:

- single (x, y) tuple
- list of (x, y) tuples
- Nx2 or 2xN `ndarray`
- Nx2 or 2xN `Quantity` in pixel units

Note that a 2x2 `ndarray` or `Quantity` is interpreted as Nx2, i.e. two rows of (x, y) coordinates.

`w` : float

The full width of the aperture. For `theta=0` the width side is along the x axis.

`h` : float

The full height of the aperture. For `theta=0` the height side is along the y axis.

`theta` : float

The rotation angle in radians of the width (w) side from the positive x axis. The rotation angle increases counterclockwise.

Raises

ValueError : `ValueError`

If either width (w) or height (h) is negative.

Methods Summary

<code>area()</code>	Return the exact area of the aperture shape.
<code>plot([origin, indices, ax, fill])</code>	Plot the aperture(s) on a matplotlib <code>Axes</code> instance.

Methods Documentation

area()

Return the exact area of the aperture shape.

Returns

`area` : float

The aperture area.

plot(`origin=(0, 0)`, `indices=None`, `ax=None`, `fill=False`, `kwargs`)**

Plot the aperture(s) on a matplotlib `Axes` instance.

Parameters

`origin` : array_like, optional

The (x, y) position of the origin of the displayed image.

`indices` : int or array of int, optional

The indices of the aperture(s) to plot.

`ax` : `matplotlib.axes.Axes` instance, optional

If `None`, then the current `Axes` instance is used.

`fill` : bool, optional

Set whether to fill the aperture patch. The default is `False`.

`kwargs`

Any keyword arguments accepted by `matplotlib.patches.Patch`.

RectangularMaskMixin

`class photutils.RectangularMaskMixin`

Bases: `object`

Mixin class to create masks for rectangular or rectangular-annulus aperture objects.

Methods Summary

<code>to_mask([method, subpixels])</code>	Return a list of <code>ApertureMask</code> objects, one for each aperture position.
---	---

Methods Documentation

`to_mask(method=u'exact', subpixels=5)`

Return a list of `ApertureMask` objects, one for each aperture position.

Parameters

`method` : {‘exact’, ‘center’, ‘subpixel’}, optional

The method used to determine the overlap of the aperture on the pixel grid. Not all options are available for all aperture types. Note that the more precise methods are generally slower. The following methods are available:

- ‘exact’ (default): The the exact fractional overlap of the aperture and each pixel is calculated. The returned mask will contain values between 0 and 1.
- ‘center’: A pixel is considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. The returned mask will contain values only of 0 (out) and 1 (in).
- ‘subpixel’: A pixel is divided into subpixels (see the `subpixels` keyword), each of which are considered to be entirely in or out of the aperture depending on whether its center is in or out of the aperture. If `subpixels=1`, this method is equivalent to ‘center’. The returned mask will contain values between 0 and 1.

`subpixels` : int, optional

For the ‘subpixel’ method, resample pixels by this factor in each dimension. That is, each pixel is divided into `subpixels ** 2` subpixels.

Returns

`mask` : list of `ApertureMask`

A list of aperture mask objects.

SExtractorBackground

`class photutils.SExtractorBackground`

Bases: `photutils.BackgroundBase`

Class to calculate the background in an array using the SExtractor algorithm.

The background is calculated using a mode estimator of the form $(2.5 * \text{median}) - (1.5 * \text{mean})$.

If $(\text{mean} - \text{median}) / \text{std} > 0.3$ then the median is used instead. Despite what the SExtractor User’s Manual says, this is the method it *always* uses.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3`. and `iters=5`.

Examples

```
>>> from photutils import SigmaClip, SExtractorBackground
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkg = SExtractorBackground(sigma_clip)
```

The background value can be calculated by using the `calc_background` method, e.g.:

```
>>> bkg_value = bkg.calc_background(data)
>>> print(bkg_value)
49.5
```

Alternatively, the background value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkg_value = bkg(data)
>>> print(bkg_value)
49.5
```

Methods Summary

<code>calc_background(data[, axis])</code>	Calculate the background value.
--	---------------------------------

Methods Documentation

`calc_background(data, axis=None)`

Calculate the background value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background value.

`axis` : int or `None`, optional

The array axis along which the background is calculated. If `None`, then the entire array is used.

Returns

`result` : float or `MaskedArray`

The calculated background value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

SegmentationImage

`class photutils.SegmentationImage(data)`

Bases: `object`

Class for a segmentation image.

Parameters

`data` : array_like (int)

A 2D segmentation image where sources are labeled by different positive integer values.
A value of zero is reserved for the background.

Attributes Summary

<code>areas</code>	The areas (in pixel **2) of all labeled regions.
<code>array</code>	The 2D segmentation image.
<code>data</code>	The 2D segmentation image.
<code>data_masked</code>	A <code>MaskedArray</code> version of the segmentation image where the background (label = 0) has been masked.
<code>is_sequential</code>	Determine whether or not the non-zero labels in the segmentation image are sequential (with no missing values).
<code>labels</code>	The sorted non-zero labels in the segmentation image.
<code>max</code>	The maximum non-zero label in the segmentation image.
<code>nlabels</code>	The number of non-zero labels in the segmentation image.
<code>shape</code>	The shape of the 2D segmentation image.
<code>slices</code>	The minimal bounding box slices for each labeled region.

Methods Summary

<code>area(labels)</code>	The areas (in pixel **2) of the regions for the input labels.
<code>check_label(label[, allow_zero])</code>	Check for a valid label label number within the segmentation image.
<code>copy()</code>	Return a deep copy of this class instance.
<code>keep_labels(labels[, relabel])</code>	Keep only the specified label numbers.
<code>outline_segments([mask_background])</code>	Outline the labeled segments.
<code>relabel(labels, new_label)</code>	Relabel one or more label numbers.
<code>relabel_sequential([start_label])</code>	Relabel the label numbers sequentially, such that there are no missing label numbers (up to the maximum label number).
<code>remove_border_labels(border_width[, ...])</code>	Remove labeled segments near the image border.
<code>remove_labels(labels[, relabel])</code>	Remove one or more label numbers.
<code>remove_masked_labels(mask[, ...])</code>	Remove labeled segments located within a masked region.

Attributes Documentation

`areas`

The areas (in pixel **2) of all labeled regions.

`array`

The 2D segmentation image.

`data`

The 2D segmentation image.

data_masked

A `MaskedArray` version of the segmentation image where the background (`label = 0`) has been masked.

is_sequential

Determine whether or not the non-zero labels in the segmentation image are sequential (with no missing values).

labels

The sorted non-zero labels in the segmentation image.

max

The maximum non-zero label in the segmentation image.

nlabels

The number of non-zero labels in the segmentation image.

shape

The shape of the 2D segmentation image.

slices

The minimal bounding box slices for each labeled region.

Methods Documentation

area(*labels*)

The areas (in pixel $^{**}2$) of the regions for the input labels.

Parameters

labels : int, array-like (1D, int)

The label(s) for which to return areas.

Returns

areas : `ndarray`

The areas of the labeled regions.

check_label(*label*, *allow_zero=False*)

Check for a valid label *label* number within the segmentation image.

Parameters

label : int

The label number to check.

allow_zero : bool

If `True` then a label of 0 is valid, otherwise 0 is invalid.

Raises

ValueError

If the input label is invalid.

copy()

Return a deep copy of this class instance.

Deep copy is used so that all attributes and values are copied.

keep_labels(*labels*, *relabel=False*)

Keep only the specified label numbers.

Parameters

labels : int, array-like (1D, int)

The label number(s) to keep. Labels of zero and those not in the segmentation image will be ignored.

relabel : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.keep_labels(labels=3)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.keep_labels(labels=[5, 3])
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [0, 0, 0, 0, 0, 5],
       [0, 0, 0, 5, 5, 5],
       [0, 0, 0, 0, 5, 5]])
```

outline_segments(*mask_background=False*)

Outline the labeled segments.

The “outlines” represent the pixels *just inside* the segments, leaving the background pixels unmodified. This corresponds to the `mode='inner'` in `skimage.segmentation.find_boundaries`.

Parameters

mask_background : bool, optional

Set to `True` to mask the background pixels (`labels = 0`) in the returned image. This is useful for overplotting the segment outlines on an image. The default is `False`.

Returns

boundaries : 2D `ndarray` or `MaskedArray`

An image with the same shape of the segmentation image containing only the outlines of the labeled segments. The pixel values in the outlines correspond to the labels in the segmentation image. If `mask_background` is `True`, then a `MaskedArray` is returned.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[0, 0, 0, 0, 0, 0],
...                             [0, 2, 2, 2, 2, 0],
...                             [0, 2, 2, 2, 2, 0],
...                             [0, 2, 2, 2, 2, 0],
...                             [0, 2, 2, 2, 2, 0],
...                             [0, 0, 0, 0, 0, 0]])
>>> segm.outline_segments()
array([[0, 0, 0, 0, 0, 0],
       [0, 2, 2, 2, 2, 0],
       [0, 2, 0, 0, 2, 0],
       [0, 2, 0, 0, 2, 0],
       [0, 2, 2, 2, 2, 0],
       [0, 0, 0, 0, 0, 0]])
```

`relabel(labels, new_label)`

Relabel one or more label numbers.

The input `labels` will all be relabeled to `new_label`.

Parameters

`labels` : int, array-like (1D, int)

The label numbers(s) to relabel.

`new_label` : int

The relabeled label number.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                             [0, 0, 0, 0, 0, 4],
...                             [0, 0, 3, 3, 0, 0],
...                             [7, 0, 0, 0, 0, 5],
...                             [7, 7, 0, 5, 5, 5],
...                             [7, 7, 0, 0, 5, 5]])
>>> segm.relabel(labels=[1, 7], new_label=2)
>>> segm.data
array([[2, 2, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 3, 3, 0, 0],
       [2, 0, 0, 0, 0, 5],
       [2, 2, 0, 5, 5, 5],
       [2, 2, 0, 0, 5, 5]])
```

`relabel_sequential(start_label=1)`

Relabel the label numbers sequentially, such that there are no missing label numbers (up to the maximum label number).

Parameters**start_label** : int, optional

The starting label number, which should be a positive integer. The default is 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.relabel_sequential()
>>> segm.data
array([[1, 1, 0, 0, 3, 3],
       [0, 0, 0, 0, 0, 3],
       [0, 0, 2, 2, 0, 0],
       [5, 0, 0, 0, 0, 4],
       [5, 5, 0, 4, 4, 4],
       [5, 5, 0, 0, 4, 4]])
```

remove_border_labels(*border_width*, *partial_overlap=True*, *relabel=False*)

Remove labeled segments near the image border.

Labels within the defined border region will be removed.

Parameters**border_width** : int

The width of the border region in pixels.

partial_overlap : bool, optional

If this is set to `True` (the default), a segment that partially extends into the border region will be removed. Segments that are completely within the border region are always removed.

relabel : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_border_labels(border_width=1)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
```

```
[0, 0, 3, 3, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_border_labels(border_width=1,
...                             partial_overlap=False)
>>> segm.data
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 3, 0, 0],
       [7, 0, 0, 0, 0, 5],
       [7, 7, 0, 5, 5, 5],
       [7, 7, 0, 0, 5, 5]])
```

remove_labels(labels, relabel=False)

Remove one or more label numbers.

Parameters

labels : int, array-like (1D, int)

The label number(s) to remove. Labels of zero and those not in the segmentation image will be ignored.

relabel : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
...                           [7, 0, 0, 0, 0, 5],
...                           [7, 7, 0, 5, 5, 5],
...                           [7, 7, 0, 0, 5, 5]])
>>> segm.remove_labels(labels=5)
>>> segm.data
array([[1, 1, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 3, 3, 0, 0],
       [7, 0, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0],
       [7, 7, 0, 0, 0, 0]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],
...                           [0, 0, 0, 0, 0, 4],
...                           [0, 0, 3, 3, 0, 0],
```

```
... [7, 0, 0, 0, 0, 5],  
... [7, 7, 0, 5, 5, 5],  
... [7, 7, 0, 0, 5, 5])  
>>> segm.remove_labels(labels=[5, 3])  
>>> segm.data  
array([[1, 1, 0, 0, 4, 4],  
       [0, 0, 0, 0, 0, 4],  
       [0, 0, 0, 0, 0, 0],  
       [7, 0, 0, 0, 0, 0],  
       [7, 7, 0, 0, 0, 0],  
       [7, 7, 0, 0, 0, 0]])
```

`remove_masked_labels(mask, partial_overlap=True, relabel=False)`

Remove labeled segments located within a masked region.

Parameters

mask : array_like (bool)

A boolean mask, with the same shape as the segmentation image (.data), where `True` values indicate masked pixels.

partial_overlap : bool, optional

If this is set to `True` (the default), a segment that partially extends into a masked region will also be removed. Segments that are completely within a masked region are always removed.

relabel : bool, optional

If `True`, then the segmentation image will be relabeled such that the labels are in sequential order starting from 1.

Examples

```
>>> from photutils import SegmentationImage  
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],  
... [0, 0, 0, 0, 0, 4],  
... [0, 0, 3, 3, 0, 0],  
... [7, 0, 0, 0, 0, 5],  
... [7, 7, 0, 5, 5, 5],  
... [7, 7, 0, 0, 5, 5]])  
>>> mask = np.zeros_like(segm.data, dtype=np.bool)  
>>> mask[0, :] = True # mask the first row  
>>> segm.remove_masked_labels(mask)  
>>> segm.data  
array([[0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0],  
       [0, 0, 3, 3, 0, 0],  
       [7, 0, 0, 0, 0, 5],  
       [7, 7, 0, 5, 5, 5],  
       [7, 7, 0, 0, 5, 5]])
```

```
>>> segm = SegmentationImage([[1, 1, 0, 0, 4, 4],  
... [0, 0, 0, 0, 0, 4],  
... [0, 0, 3, 3, 0, 0],  
... [7, 0, 0, 0, 0, 5],  
... [7, 7, 0, 5, 5, 5],  
... [7, 7, 0, 0, 5, 5],
```

```

...
[7, 7, 0, 0, 5, 5]])
>>> segm.remove_masked_labels(mask, partial_overlap=False)
>>> segm.data
array([[0, 0, 0, 0, 4, 4],
       [0, 0, 0, 0, 0, 4],
       [0, 0, 3, 3, 0, 0],
       [7, 0, 0, 0, 0, 5],
       [7, 7, 0, 5, 5, 5],
       [7, 7, 0, 0, 5, 5]])

```

SigmaClip

```

class photutils.SigmaClip(sigma=3.0, sigma_lower=None, sigma_upper=None, iters=5, cen-
                           func=<function median>, stdfunc=<function std>)
Bases: object

```

Class to perform sigma clipping.

Parameters

sigma : float, optional

The number of standard deviations to use for both the lower and upper clipping limit. These limits are overridden by `sigma_lower` and `sigma_upper`, if input. Defaults to 3.

sigma_lower : float or `None`, optional

The number of standard deviations to use as the lower bound for the clipping limit. If `None` then the value of `sigma` is used. Defaults to `None`.

sigma_upper : float or `None`, optional

The number of standard deviations to use as the upper bound for the clipping limit. If `None` then the value of `sigma` is used. Defaults to `None`.

iters : int or `None`, optional

The number of iterations to perform sigma clipping, or `None` to clip until convergence is achieved (i.e., continue until the last iteration clips nothing). Defaults to 5.

cenfunc : callable, optional

The function used to compute the center for the clipping. Must be a callable that takes in a masked array and outputs the central value. Defaults to the median (`numpy.ma.median`).

stdfunc : callable, optional

The function used to compute the standard deviation about the center. Must be a callable that takes in a masked array and outputs a width estimator. Masked (rejected) pixels are those where:

```

deviation < (-sigma_lower * stdfunc(deviation))
deviation > (sigma_upper * stdfunc(deviation))

```

where:

```

deviation = data - cenfunc(data [,axis=int])

```

Defaults to the standard deviation (`numpy.std`).

Methods Summary

<code>__call__(data[, axis, copy])</code>	Perform sigma clipping on the provided data.
---	--

Methods Documentation

`__call__(data, axis=None, copy=True)`

Perform sigma clipping on the provided data.

Parameters

`data` : array-like

The data to be sigma clipped.

`axis` : int or `None`, optional

If not `None`, clip along the given axis. For this case, `axis` will be passed on to `cenfunc` and `stdfunc`, which are expected to return an array with the axis dimension removed (like the numpy functions). If `None`, clip over all axes. Defaults to `None`.

`copy` : bool, optional

If `True`, the data array will be copied. If `False`, the returned masked array data will contain the same array as `data`. Defaults to `True`.

Returns

`filtered_data` : `numpy.ma.MaskedArray`

A masked array with the same shape as `data` input, where the points rejected by the algorithm have been masked.

SkyAperture

`class photutils.SkyAperture`

Bases: `photutils.Aperture`

Abstract base class for 2D apertures defined in celestial coordinates.

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>PixelAperture</code> object in pixel coordinates.
------------------------------------	---

Methods Documentation

`to_pixel(wcs, mode=u'all')`

Convert the aperture to a `PixelAperture` object in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

aperture : [PixelAperture](#) object

A [PixelAperture](#) object.

SkyCircularAnnulus

class photutils.SkyCircularAnnulus

Bases: [photutils.SkyAperture](#)

Circular annulus aperture(s), defined in sky coordinates.

Parameters

positions : [SkyCoord](#)

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

r_in : [Quantity](#)

The inner radius of the annulus, either in angular or pixel units.

r_out : [Quantity](#)

The outer radius of the annulus, either in angular or pixel units.

Methods Summary

to_pixel(wcs[, mode])

Convert the aperture to a [CircularAnnulus](#) instance in pixel coordinates.

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a [CircularAnnulus](#) instance in pixel coordinates.

Parameters

wcs : [WCS](#)

The WCS transformation to use.

mode : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

aperture : [CircularAnnulus](#) object

A [CircularAnnulus](#) object.

SkyCircularAperture

class photutils.SkyCircularAperture

Bases: photutils.SkyAperture

Circular aperture(s), defined in sky coordinates.

Parameters

positions : SkyCoord

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

r : Quantity

The radius of the aperture(s), either in angular or pixel units.

Methods Summary

to_pixel(wcs[, mode])

Convert the aperture to a CircularAperture instance in pixel coordinates.

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a CircularAperture instance in pixel coordinates.

Parameters

wcs : WCS

The WCS transformation to use.

mode : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions (‘all’; default) or only including only the core WCS transformation (‘wcs’).

Returns

aperture : CircularAperture object

A CircularAperture object.

SkyEllipticalAnnulus

class photutils.SkyEllipticalAnnulus

Bases: photutils.SkyAperture

Elliptical annulus aperture(s), defined in sky coordinates.

Parameters

positions : SkyCoord

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

a_in : Quantity

The inner semimajor axis, either in angular or pixel units.

a_out : `Quantity`

The outer semimajor axis, either in angular or pixel units.

b_out : `float`

The outer semiminor axis, either in angular or pixel units. The inner semiminor axis is calculated as:

$$b_{in} = b_{out} \left(\frac{a_{in}}{a_{out}} \right)$$

theta : `Quantity`

The position angle (in angular units) of the semimajor axis. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to an <code>EllipticalAnnulus</code> instance in pixel coordinates.
------------------------------------	--

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to an `EllipticalAnnulus` instance in pixel coordinates.

Parameters`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns`aperture` : `EllipticalAnnulus` object

An `EllipticalAnnulus` object.

SkyEllipticalAperture

class photutils.SkyEllipticalAperture

Bases: `photutils.SkyAperture`

Elliptical aperture(s), defined in sky coordinates.

Parameters`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`a` : `Quantity`

The semimajor axis, either in angular or pixel units.

b : `Quantity`

The semiminor axis, either in angular or pixel units.

theta : `Quantity`

The position angle (in angular units) of the semimajor axis. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to an <code>EllipticalAperture</code> instance in pixel coordinates.
------------------------------------	---

Methods Documentation

`to_pixel(wcs, mode=u'all')`

Convert the aperture to an `EllipticalAperture` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : { ‘all’, ‘wcs’ }, optional

Whether to do the transformation including distortions ('all'; default) or only including only the core WCS transformation ('wcs').

Returns

`aperture` : `EllipticalAperture` object

An `EllipticalAperture` object.

SkyRectangularAnnulus

`class photutils.SkyRectangularAnnulus`

Bases: `photutils.SkyAperture`

Rectangular annulus aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`w_in` : `Quantity`

The inner full width of the aperture, either in angular or pixel units. For theta=0 the width side is along the North-South axis.

`w_out` : `Quantity`

The outer full width of the aperture, either in angular or pixel units. For theta=0 the width side is along the North-South axis.

h_out : `Quantity`

The outer full height of the aperture, either in angular or pixel units. The inner full height is calculated as:

$$h_{in} = h_{out} \left(\frac{w_{in}}{w_{out}} \right)$$

For theta=0 the height side is along the East-West axis.

theta : `Quantity`

The position angle (in angular units) of the width side. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>RectangularAnnulus</code> instance in pixel coordinates.
------------------------------------	--

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a `RectangularAnnulus` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions (‘all’; default) or only including only the core WCS transformation (‘wcs’).

Returns

`aperture` : `RectangularAnnulus` object

A `RectangularAnnulus` object.

SkyRectangularAperture

`class photutils.SkyRectangularAperture`

Bases: `photutils.SkyAperture`

Rectangular aperture(s), defined in sky coordinates.

Parameters

`positions` : `SkyCoord`

Celestial coordinates of the aperture center(s). This can be either scalar coordinates or an array of coordinates.

`w` : `Quantity`

The full width of the aperture, either in angular or pixel units. For theta=0 the width side is along the North-South axis.

h : `Quantity`

The full height of the aperture, either in angular or pixel units. For theta=0 the height side is along the East-West axis.

theta : `Quantity`

The position angle (in angular units) of the width side. For a right-handed world coordinate system, the position angle increases counterclockwise from North (PA=0).

Methods Summary

<code>to_pixel(wcs[, mode])</code>	Convert the aperture to a <code>RectangularAperture</code> instance in pixel coordinates.
------------------------------------	---

Methods Documentation

to_pixel(wcs, mode=u'all')

Convert the aperture to a `RectangularAperture` instance in pixel coordinates.

Parameters

`wcs` : `WCS`

The WCS transformation to use.

`mode` : {‘all’, ‘wcs’}, optional

Whether to do the transformation including distortions (‘all’; default) or only including only the core WCS transformation (‘wcs’).

Returns

`aperture` : `RectangularAperture` object

A `RectangularAperture` object.

SourceProperties

```
class photutils.SourceProperties(data, segment_img, label, filtered_data=None, error=None,
                                 mask=None, background=None, wcs=None)
```

Bases: `object`

Class to calculate photometry and morphological properties of a single labeled source.

Parameters

`data` : array_like or `Quantity`

The 2D array from which to calculate the source photometry and properties. If `filtered_data` is input, then it will be used instead of `data` to calculate the source centroid and morphological properties. Source photometry is always measured from `data`. For accurate source properties and photometry, `data` should be background-subtracted.

`segment_img` : `SegmentationImage` or array_like (int)

A 2D segmentation image, either as a `SegmentationImage` object or an `ndarray`, with the same shape as data where sources are labeled by different positive integer values. A value of zero is reserved for the background.

label : int

The label number of the source whose properties to calculate.

filtered_data : array-like or `Quantity`, optional

The filtered version of the background-subtracted data from which to calculate the source centroid and morphological properties. The kernel used to perform the filtering should be the same one used in defining the source segments (e.g., see `detect_sources()`). If `None`, then the unfiltered data will be used instead. Note that SExtractor’s centroid and morphological parameters are calculated from the filtered “detection” image.

error : array_like or `Quantity`, optional

The pixel-wise Gaussian 1-sigma errors of the input data. `error` is assumed to include *all* sources of error, including the Poisson error of the sources (see `calc_total_error`) . `error` must have the same shape as the input data. See the Notes section below for details on the error propagation.

mask : array_like (bool), optional

A boolean mask with the same shape as data where a `True` value indicates the corresponding element of data is masked. Masked data are excluded from all calculations.

background : float, array_like, or `Quantity`, optional

The background level that was *previously* present in the input data. `background` may either be a scalar value or a 2D image with the same shape as the input data. Inputting the background merely allows for its properties to be measured within each source segment. The input background does *not* get subtracted from the input data, which should already be background-subtracted.

wcs : `WCS`

The WCS transformation to use. If `None`, then `icrs_centroid`, `ra_icrs_centroid`, and `dec_icrs_centroid` will be `None`.

Notes

SExtractor’s centroid and morphological parameters are always calculated from the filtered “detection” image. The usual downside of the filtering is the sources will be made more circular than they actually are. If you wish to reproduce SExtractor results, then use the `filtered_data` input. If `filtered_data` is `None`, then the unfiltered data will be used for the source centroid and morphological parameters.

Negative (background-subtracted) data values within the source segment are set to zero when measuring morphological properties based on image moments. This could occur, for example, if the segmentation image was defined from a different image (e.g., different bandpass) or if the background was oversubtracted. Note that `source_sum` includes the contribution of negative (background-subtracted) data values.

The input `error` is assumed to include *all* sources of error, including the Poisson error of the sources. `source_sum_err` is simply the quadrature sum of the pixel-wise total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, S are the non-masked pixels in the source segment, and $\sigma_{\text{tot},i}$ is the input error array.

Custom errors for source segments can be calculated using the `error_cutout_ma` and `background_cutout_ma` properties, which are 2D `MaskedArray` cutout versions of the input error and background. The mask is `True` for both pixels outside of the source segment and masked pixels from the `mask` input.

Attributes Summary

<code>area</code>	The area of the source segment in units of pixels**2.
<code>background_at_centroid</code>	The value of the background at the position of the source centroid.
<code>background_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the input background, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.
<code>background_mean</code>	The mean of background values within the source segment.
<code>background_sum</code>	The sum of background values within the source segment.
<code>bbox</code>	The bounding box (<code>ymin</code> , <code>xmin</code> , <code>ymax</code> , <code>xmax</code>) of the minimal rectangular region containing the source segment.
<code>centroid</code>	The (<code>y</code> , <code>x</code>) coordinate of the centroid within the source segment.
<code>coords</code>	A tuple of <code>ndarrays</code> containing the <code>y</code> and <code>x</code> pixel coordinates of the source segment.
<code>covar_sigx2</code>	The (<code>0</code> , <code>0</code>) element of the <code>covariance</code> matrix, representing σ_x^2 , in units of pixel**2.
<code>covar_sigxy</code>	The (<code>0</code> , <code>1</code>) and (<code>1</code> , <code>0</code>) elements of the <code>covariance</code> matrix, representing $\sigma_x \sigma_y$, in units of pixel**2.
<code>covar_sigy2</code>	The (<code>1</code> , <code>1</code>) element of the <code>covariance</code> matrix, representing σ_y^2 , in units of pixel**2.
<code>covariance</code>	The covariance matrix of the 2D Gaussian function that has the same second-order moments as the source.
<code>covariance_eigvals</code>	The two eigenvalues of the <code>covariance</code> matrix in decreasing order.
<code>cutout_centroid</code>	The (<code>y</code> , <code>x</code>) coordinate, relative to the <code>data_cutout</code> , of the centroid within the source segment.
<code>cxx</code>	<code>SExtractor</code> 's CXX ellipse parameter in units of pixel**(-2).
<code>cxy</code>	<code>SExtractor</code> 's CXYY ellipse parameter in units of pixel**(-2).
<code>cyy</code>	<code>SExtractor</code> 's CYYY ellipse parameter in units of pixel**(-2).
<code>data_cutout</code>	A 2D cutout from the (background-subtracted) data of the source segment.
<code>data_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the (background-subtracted) data, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.
<code>dec_icrs_centroid</code>	The ICRS Declination coordinate (in degrees) of the centroid within the source segment.

Continued on next page

Table 17.56 – continued from previous page

<code>eccentricity</code>	The eccentricity of the 2D Gaussian function that has the same second-order moments as the source.
<code>ellipticity</code>	1 minus the ratio of the lengths of the semimajor and
<code>elongation</code>	The ratio of the lengths of the semimajor and semiminor axes:
<code>equivalent_radius</code>	The radius of a circle with the same <code>area</code> as the source segment.
<code>error_cutout_ma</code>	A 2D <code>MaskedArray</code> cutout from the input <code>error</code> image, where the mask is <code>True</code> for both pixels outside of the source segment and masked pixels.
<code>icrs_centroid</code>	The International Celestial Reference System (ICRS) coordinates of the centroid within the source segment, returned as a <code>SkyCoord</code> object.
<code>id</code>	The source identification number corresponding to the object label in the segmentation image.
<code>inertia_tensor</code>	The inertia tensor of the source for the rotation around its center of mass.
<code>max_value</code>	The maximum pixel value of the (background-subtracted) data within the source segment.
<code>maxval_cutout_pos</code>	The (y, x) coordinate, relative to the <code>data_cutout</code> , of the maximum pixel value of the (background-subtracted) data.
<code>maxval_pos</code>	The (y, x) coordinate of the maximum pixel value of the (background-subtracted) data.
<code>maxval_xpos</code>	The x coordinate of the maximum pixel value of the (background-subtracted) data.
<code>maxval_ypos</code>	The y coordinate of the maximum pixel value of the (background-subtracted) data.
<code>min_value</code>	The minimum pixel value of the (background-subtracted) data within the source segment.
<code>minval_cutout_pos</code>	The (y, x) coordinate, relative to the <code>data_cutout</code> , of the minimum pixel value of the (background-subtracted) data.
<code>minval_pos</code>	The (y, x) coordinate of the minimum pixel value of the (background-subtracted) data.
<code>minval_xpos</code>	The x coordinate of the minimum pixel value of the (background-subtracted) data.
<code>minval_ypos</code>	The y coordinate of the minimum pixel value of the (background-subtracted) data.
<code>moments</code>	Spatial moments up to 3rd order of the source.
<code>moments_central</code>	Central moments (translation invariant) of the source up to 3rd order.
<code>orientation</code>	The angle in radians between the x axis and the major axis of the 2D Gaussian function that has the same second-order moments as the source.
<code>perimeter</code>	The perimeter of the source segment, approximated lines through the centers of the border pixels using a 4-connectivity.
<code>ra_icrs_centroid</code>	The ICRS Right Ascension coordinate (in degrees) of the centroid within the source segment.

Continued on next page

Table 17.56 – continued from previous page

semimajor_axis_sigma	The 1-sigma standard deviation along the semimajor axis of the 2D Gaussian function that has the same second-order central moments as the source.
semiminor_axis_sigma	The 1-sigma standard deviation along the semiminor axis of the 2D Gaussian function that has the same second-order central moments as the source.
source_sum	The sum of the non-masked (background-subtracted) data values within the source segment.
source_sum_err	The uncertainty of <code>source_sum</code> , propagated from the input error array.
values	A <code>ndarray</code> of the (background-subtracted) pixel values within the source segment.
xcentroid	The x coordinate of the centroid within the source segment.
xmax	The maximum x pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
xmin	The minimum x pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
ycentroid	The y coordinate of the centroid within the source segment.
ymax	The maximum y pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.
ymin	The minimum y pixel location of the minimal bounding box (<code>bbox</code>) of the source segment.

Methods Summary

<code>make_cutout(data[, masked_array])</code>	Create a (masked) cutout array from the input data using the minimal bounding box of the source segment.
<code>to_table([columns, exclude_columns])</code>	Create a <code>Table</code> of properties.

Attributes Documentation

`area`

The area of the source segment in units of pixels**2.

`background_at_centroid`

The value of the background at the position of the source centroid. Fractional position values are determined using bilinear interpolation.

`background_cutout_ma`

A 2D `MaskedArray` cutout from the input background, where the mask is `True` for both pixels outside of the source segment and masked pixels. If `background` is `None`, then `background_cutout_ma` is also `None`.

`background_mean`

The mean of background values within the source segment.

`background_sum`

The sum of background values within the source segment.

`bbox`

The bounding box (ymin, xmin, ymax, xmax) of the minimal rectangular region containing the source segment.

centroid

The (y, x) coordinate of the centroid within the source segment.

coords

A tuple of `ndarrays` containing the y and x pixel coordinates of the source segment. Masked pixels are not included.

covar_sigx2

The (0, 0) element of the `covariance` matrix, representing σ_x^2 , in units of pixel**2.

Note that this is the same as `SExtractor`'s X2 parameter.

covar_sigxy

The (0, 1) and (1, 0) elements of the `covariance` matrix, representing $\sigma_x\sigma_y$, in units of pixel**2.

Note that this is the same as `SExtractor`'s XY parameter.

covar_sigy2

The (1, 1) element of the `covariance` matrix, representing σ_y^2 , in units of pixel**2.

Note that this is the same as `SExtractor`'s Y2 parameter.

covariance

The covariance matrix of the 2D Gaussian function that has the same second-order moments as the source.

covariance_eigvals

The two eigenvalues of the `covariance` matrix in decreasing order.

cutout_centroid

The (y, x) coordinate, relative to the `data_cutout`, of the centroid within the source segment.

cxx

`SExtractor`'s CXX ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$cxx(x - \bar{x})^2 + cxy(x - \bar{x})(y - \bar{y}) + cyy(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). `SExtractor` reports that the isophotal limit of a source is well represented by $R \approx 3$.

cxy

`SExtractor`'s CXY ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$cxx(x - \bar{x})^2 + cxy(x - \bar{x})(y - \bar{y}) + cyy(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). `SExtractor` reports that the isophotal limit of a source is well represented by $R \approx 3$.

cyy

`SExtractor`'s CYY ellipse parameter in units of pixel**(-2).

The ellipse is defined as

$$c_{xx}(x - \bar{x})^2 + c_{xy}(x - \bar{x})(y - \bar{y}) + c_{yy}(y - \bar{y})^2 = R^2$$

where R is a parameter which scales the ellipse (in units of the axes lengths). SExtractor reports that the isophotal limit of a source is well represented by $R \approx 3$.

data_cutout

A 2D cutout from the (background-subtracted) data of the source segment.

data_cutout_ma

A 2D `MaskedArray` cutout from the (background-subtracted) data, where the mask is `True` for both pixels outside of the source segment and masked pixels.

dec_icrs_centroid

The ICRS Declination coordinate (in degrees) of the centroid within the source segment.

eccentricity

The eccentricity of the 2D Gaussian function that has the same second-order moments as the source.

The eccentricity is the fraction of the distance along the semimajor axis at which the focus lies.

$$e = \sqrt{1 - \frac{b^2}{a^2}}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

ellipticity

1 minus the ratio of the lengths of the semimajor and semiminor axes (or 1 minus the `elongation`):

$$\text{ellipticity} = 1 - \frac{b}{a}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

Note that this is the same as SExtractor’s ellipticity parameter.

elongation

The ratio of the lengths of the semimajor and semiminor axes:

$$\text{elongation} = \frac{a}{b}$$

where a and b are the lengths of the semimajor and semiminor axes, respectively.

Note that this is the same as SExtractor’s elongation parameter.

equivalent_radius

The radius of a circle with the same `area` as the source segment.

error_cutout_ma

A 2D `MaskedArray` cutout from the input error image, where the mask is `True` for both pixels outside of the source segment and masked pixels. If `error` is `None`, then `error_cutout_ma` is also `None`.

icrs_centroid

The International Celestial Reference System (ICRS) coordinates of the centroid within the source segment, returned as a `SkyCoord` object.

id

The source identification number corresponding to the object label in the segmentation image.

inertia_tensor

The inertia tensor of the source for the rotation around its center of mass.

max_value

The maximum pixel value of the (background-subtracted) data within the source segment.

maxval_cutout_pos

The (y, x) coordinate, relative to the `data_cutout`, of the maximum pixel value of the (background-subtracted) data.

maxval_pos

The (y, x) coordinate of the maximum pixel value of the (background-subtracted) data.

maxval_xpos

The x coordinate of the maximum pixel value of the (background-subtracted) data.

maxval_ypos

The y coordinate of the maximum pixel value of the (background-subtracted) data.

min_value

The minimum pixel value of the (background-subtracted) data within the source segment.

minval_cutout_pos

The (y, x) coordinate, relative to the `data_cutout`, of the minimum pixel value of the (background-subtracted) data.

minval_pos

The (y, x) coordinate of the minimum pixel value of the (background-subtracted) data.

minval_xpos

The x coordinate of the minimum pixel value of the (background-subtracted) data.

minval_ypos

The y coordinate of the minimum pixel value of the (background-subtracted) data.

moments

Spatial moments up to 3rd order of the source.

moments_central

Central moments (translation invariant) of the source up to 3rd order.

orientation

The angle in radians between the x axis and the major axis of the 2D Gaussian function that has the same second-order moments as the source. The angle increases in the counter-clockwise direction.

perimeter

The perimeter of the source segment, approximated lines through the centers of the border pixels using a 4-connectivity.

ra_icrs_centroid

The ICRS Right Ascension coordinate (in degrees) of the centroid within the source segment.

semimajor_axis_sigma

The 1-sigma standard deviation along the semimajor axis of the 2D Gaussian function that has the same second-order central moments as the source.

semiminor_axis_sigma

The 1-sigma standard deviation along the semiminor axis of the 2D Gaussian function that has the same second-order central moments as the source.

source_sum

The sum of the non-masked (background-subtracted) data values within the source segment.

$$F = \sum_{i \in S} (I_i - B_i)$$

where F is `source_sum`, $(I_i - B_i)$ is the background-subtracted input data, and S are the non-masked pixels in the source segment.

source_sum_err

The uncertainty of `source_sum`, propagated from the input error array.

`source_sum_err` is the quadrature sum of the total errors over the non-masked pixels within the source segment:

$$\Delta F = \sqrt{\sum_{i \in S} \sigma_{\text{tot},i}^2}$$

where ΔF is `source_sum_err`, $\sigma_{\text{tot},i}$ are the pixel-wise total errors, and S are the non-masked pixels in the source segment.

values

A `ndarray` of the (background-subtracted) pixel values within the source segment. Masked pixels are not included.

xcentroid

The x coordinate of the centroid within the source segment.

xmax

The maximum x pixel location of the minimal bounding box (`bbox`) of the source segment.

xmin

The minimum x pixel location of the minimal bounding box (`bbox`) of the source segment.

ycentroid

The y coordinate of the centroid within the source segment.

ymax

The maximum y pixel location of the minimal bounding box (`bbox`) of the source segment.

ymin

The minimum y pixel location of the minimal bounding box (`bbox`) of the source segment.

Methods Documentation

make_cutout(data, masked_array=False)

Create a (masked) cutout array from the input data using the minimal bounding box of the source segment.

Parameters

data : array-like (2D)

The data array from which to create the masked cutout array. `data` must have the same shape as the segmentation image input into `SourceProperties`.

masked_array : bool, optional

If `True` then a `MaskedArray` will be created where the mask is `True` for both pixels outside of the source segment and any masked pixels. If `False`, then a `ndarray` will be generated.

Returns

result : `ndarray` or `MaskedArray` (2D)

The 2D cutout array or masked array.

to_table(*columns=None*, *exclude_columns=None*)

Create a [Table](#) of properties.

If *columns* or *exclude_columns* are not input, then the [Table](#) will include all scalar-valued properties. Multi-dimensional properties, e.g. `data_cutout`, can be included in the *columns* input.

Parameters

columns : str or list of str, optional

Names of columns, in order, to include in the output [Table](#). The allowed column names are any of the attributes of [SourceProperties](#).

exclude_columns : str or list of str, optional

Names of columns to exclude from the default properties list in the output [Table](#). The default properties are those with scalar values.

Returns

table : [Table](#)

A single-row table of properties of the source.

SplitCosineBellWindow

class photutils.SplitCosineBellWindow(*alpha*, *beta*)

Bases: `object`

Class to define a 2D split cosine bell taper function.

Parameters

alpha : float, optional

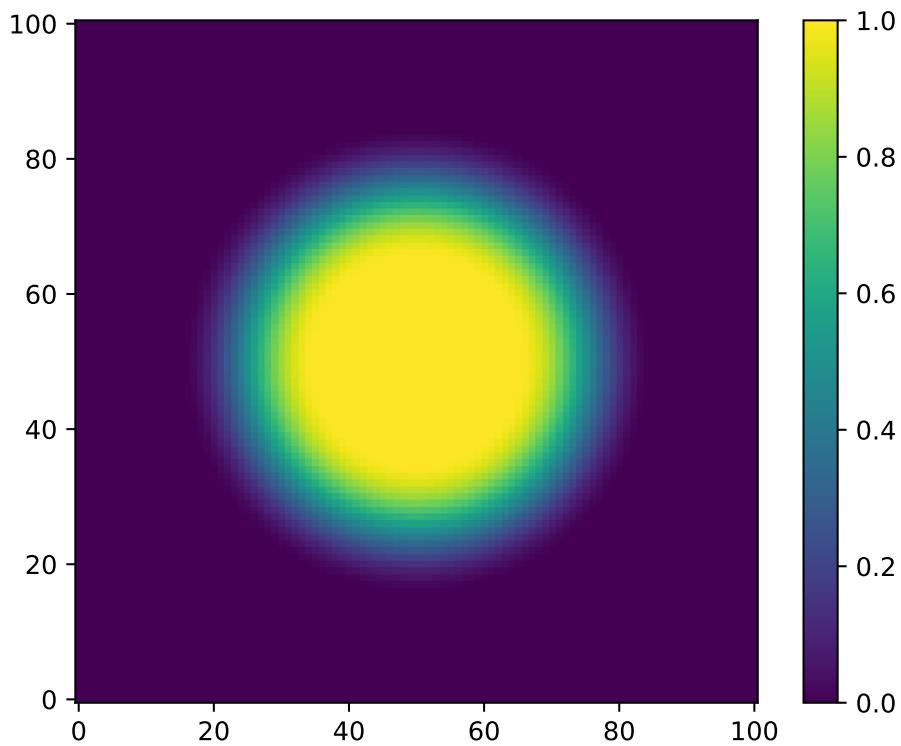
The percentage of array values that are tapered.

beta : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. *beta* must be less or equal to 1.0.

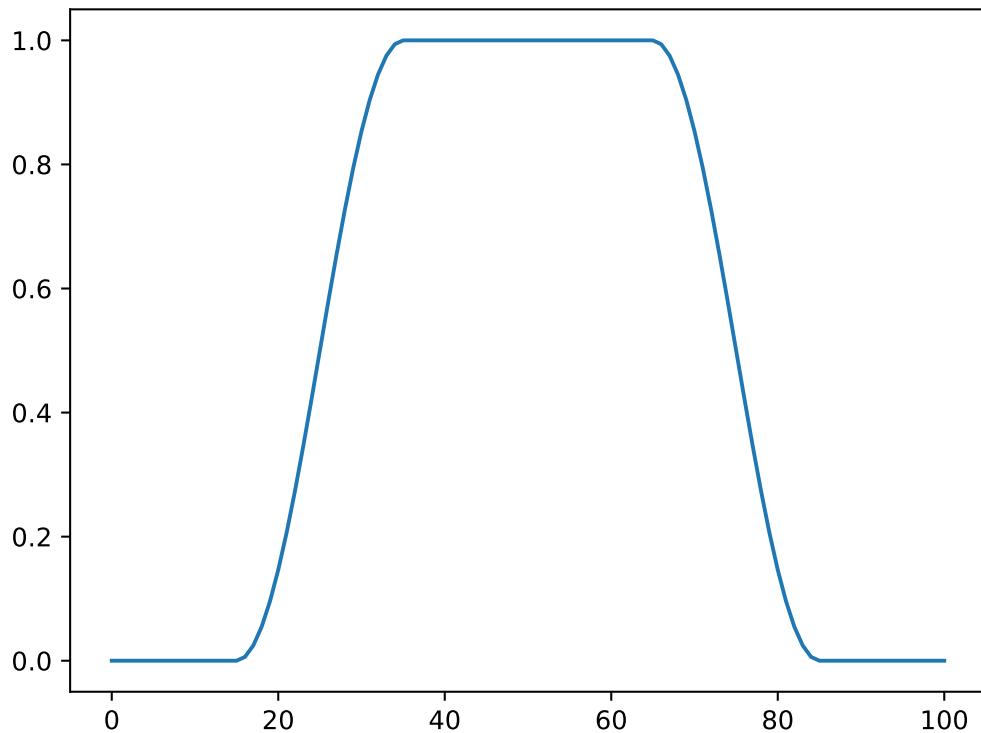
Examples

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import SplitCosineBellWindow
taper = SplitCosineBellWindow(alpha=0.4, beta=0.3)
data = taper((101, 101))
plt.plot(data[50, :])
```



Methods Summary

<code>__call__(shape)</code>	Return a 2D split cosine bell.
------------------------------	--------------------------------

Methods Documentation

`__call__(shape)`
Return a 2D split cosine bell.

Parameters
`shape` : tuple of int

The size of the output array along each axis.

Returns
`result` : ndarray
A 2D array containing the cosine bell values.

StarFinderBase

`class photutils.StarFinderBase`
Bases: `object`

Abstract base class for Star Finders.

Methods Summary

`__call__(...) <==> x(...)`

`find_stars(data)`

Find stars in an astronomical image.

Methods Documentation

`__call__(...) <==> x(...)`

`find_stars(data)`

Find stars in an astronomical image.

Parameters

`data : array_like`

The 2D image array.

Returns

`table : Table`

A table of found objects with the following parameters:

- `id`: unique object identification number.
- `xcentroid`, `ycentroid`: object centroid.
- `sharpness`: object sharpness.
- `roundness1`: object roundness based on symmetry.
- `roundness2`: object roundness based on marginal Gaussian fits.
- `npix`: number of pixels in the Gaussian kernel.
- `sky`: the input sky parameter.
- `peak`: the peak, sky-subtracted, pixel value of the object.
- `flux`: the object flux calculated as the peak density in the convolved image divided by the detection threshold. This derivation matches that of `DAOFIND` if `sky` is 0.0.
- `mag`: the object instrumental magnitude calculated as $-2.5 * \log_{10}(\text{flux})$. The derivation matches that of `DAOFIND` if `sky` is 0.0.

Notes

For the convolution step, this routine sets pixels beyond the image borders to 0.0. The equivalent parameters in IRAF's `starfind` are `boundary='constant'` and `constant=0.0`.

IRAF's `starfind` uses `hwhm`, `fradius`, and `sepmin` as input parameters. The equivalent input values for `IRAFStarFinder` are:

- `FWHM = hwhm * 2`
- `Sigma_Radius = fradius * sqrt(2.0*log(2.0))`
- `MinSep_FWHM = 0.5 * sepmin`

The main differences between `DAOStarFinder` and `IRAFStarFinder` are:

- `IRAFStarFinder` always uses a 2D circular Gaussian kernel, while `DAOStarFinder` can use an elliptical Gaussian kernel.
- `IRAFStarFinder` calculates the objects' centroid, roundness, and sharpness using image moments.

StdBackgroundRMS

`class photutils.StdBackgroundRMS`

Bases: `photutils.BackgroundRMSBase`

Class to calculate the background RMS in an array as the (sigma-clipped) standard deviation.

Parameters

`sigma_clip` : `SigmaClip` object, optional

A `SigmaClip` object that defines the sigma clipping parameters. If `None` then no sigma clipping will be performed. The default is to perform sigma clipping with `sigma=3.` and `iters=5.`

Examples

```
>>> from photutils import SigmaClip, StdBackgroundRMS
>>> data = np.arange(100)
>>> sigma_clip = SigmaClip(sigma=3.)
>>> bkgrms = StdBackgroundRMS(sigma_clip)
```

The background RMS value can be calculated by using the `calc_background_rms` method, e.g.:

```
>>> bkgrms_value = bkgrms.calc_background_rms(data)
>>> print(bkgrms_value)
28.866070047722118
```

Alternatively, the background RMS value can be calculated by calling the class instance as a function, e.g.:

```
>>> bkgrms_value = bkgrms(data)
>>> print(bkgrms_value)
28.866070047722118
```

Methods Summary

`calc_background_rms(data[, axis])`

Calculate the background RMS value.

Methods Documentation

`calc_background_rms(data, axis=None)`

Calculate the background RMS value.

Parameters

`data` : array_like or `MaskedArray`

The array for which to calculate the background RMS value.

axis : int or `None`, optional

The array axis along which the background RMS is calculated. If `None`, then the entire array is used.

Returns

result : float or `MaskedArray`

The calculated background RMS value. If `axis` is `None` then a scalar will be returned, otherwise a `MaskedArray` will be returned.

TopHatWindow

`class photutils.TopHatWindow(beta)`
Bases: `photutils.psf.matching.SplitCosineBellWindow`

Class to define a 2D top hat window function.

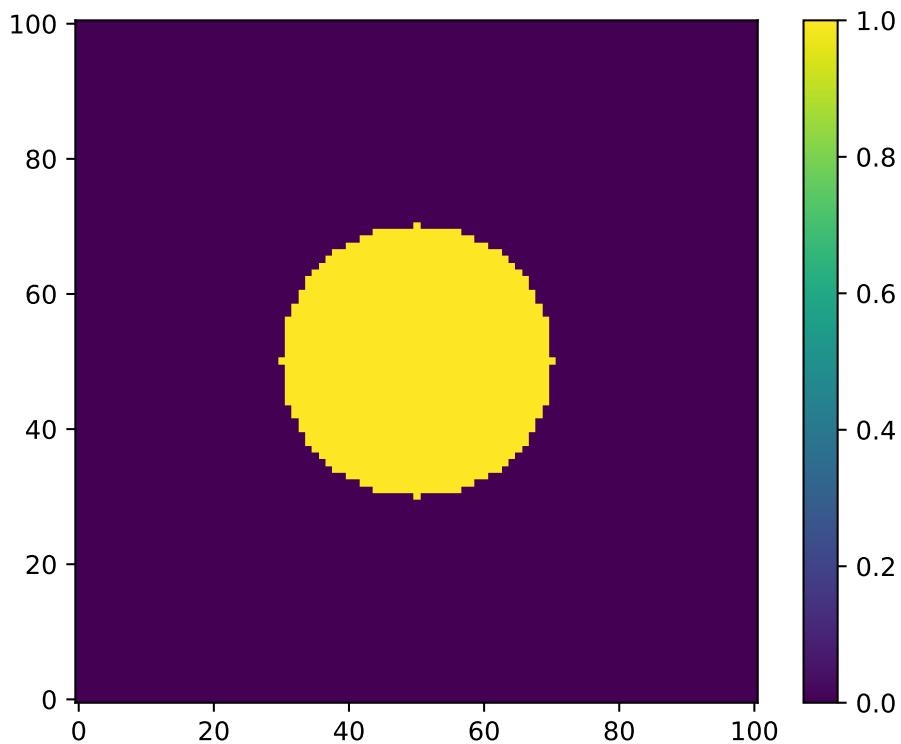
Parameters

beta : float, optional

The inner diameter as a fraction of the array size beyond which the taper begins. `beta` must be less or equal to 1.0.

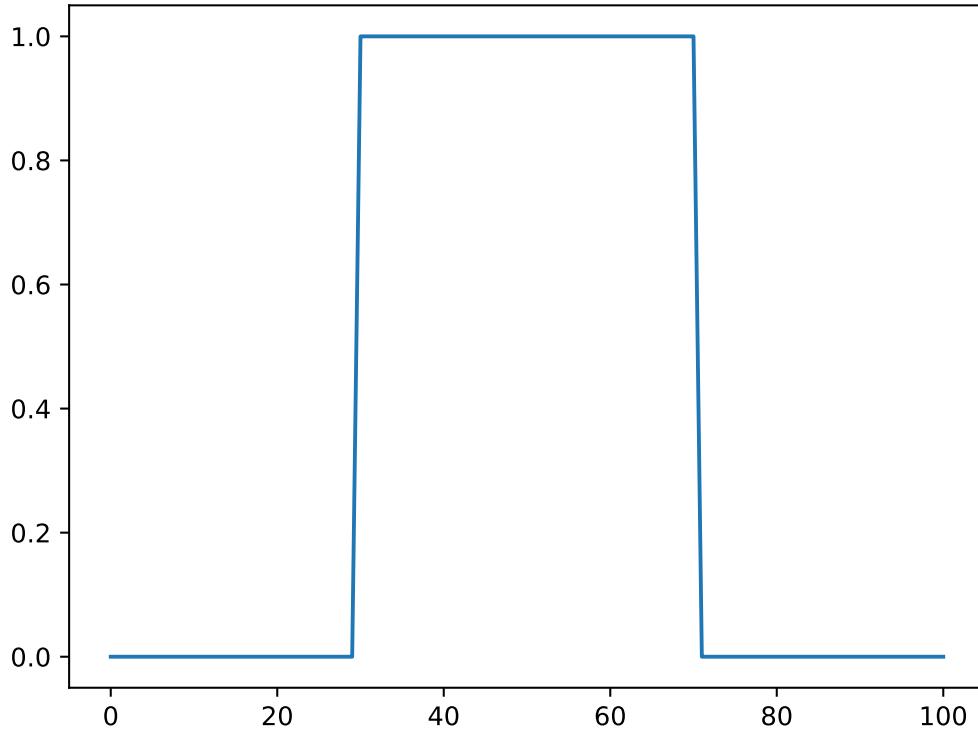
Examples

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower',
           interpolation='nearest')
plt.colorbar()
```



A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TopHatWindow
taper = TopHatWindow(beta=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



TukeyWindow

```
class photutils.TukeyWindow(alpha)
Bases: photutils.psf.matching.SplitCosineBellWindow
```

Class to define a 2D Tukey window function.

The Tukey window is a taper formed by using a split cosine bell function with ends that touch zero.

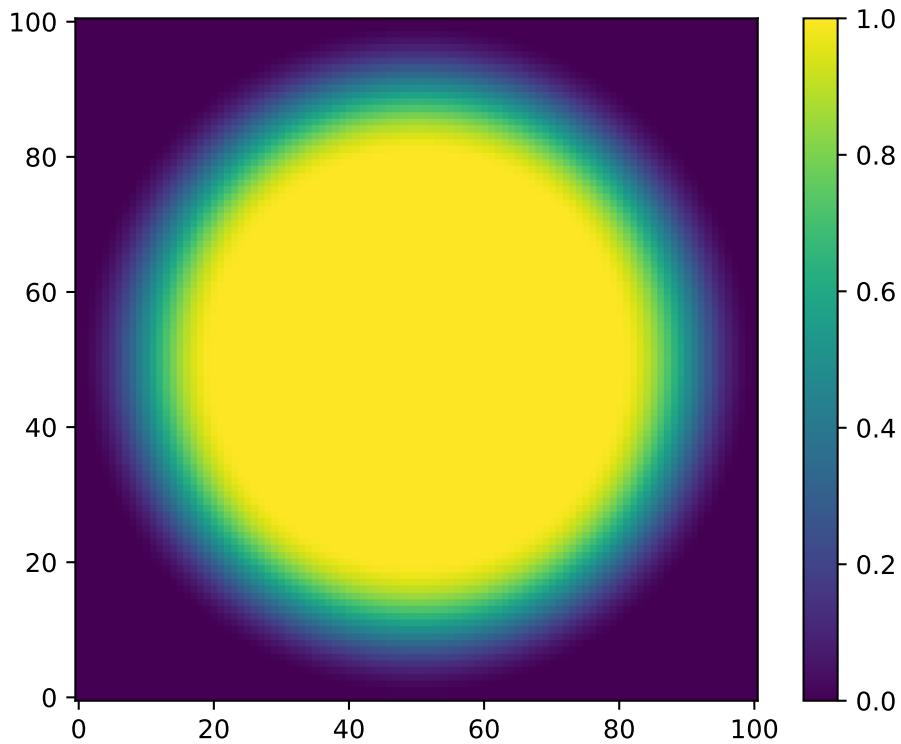
Parameters

alpha : float, optional

The percentage of array values that are tapered.

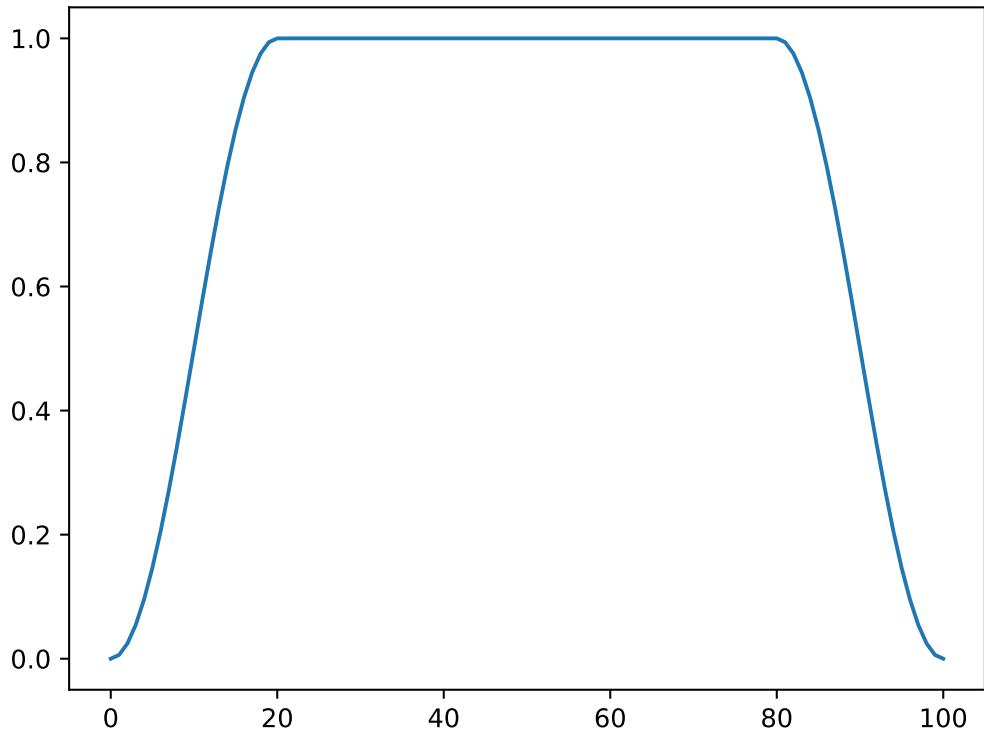
Examples

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.imshow(data, cmap='viridis', origin='lower')
plt.colorbar()
```

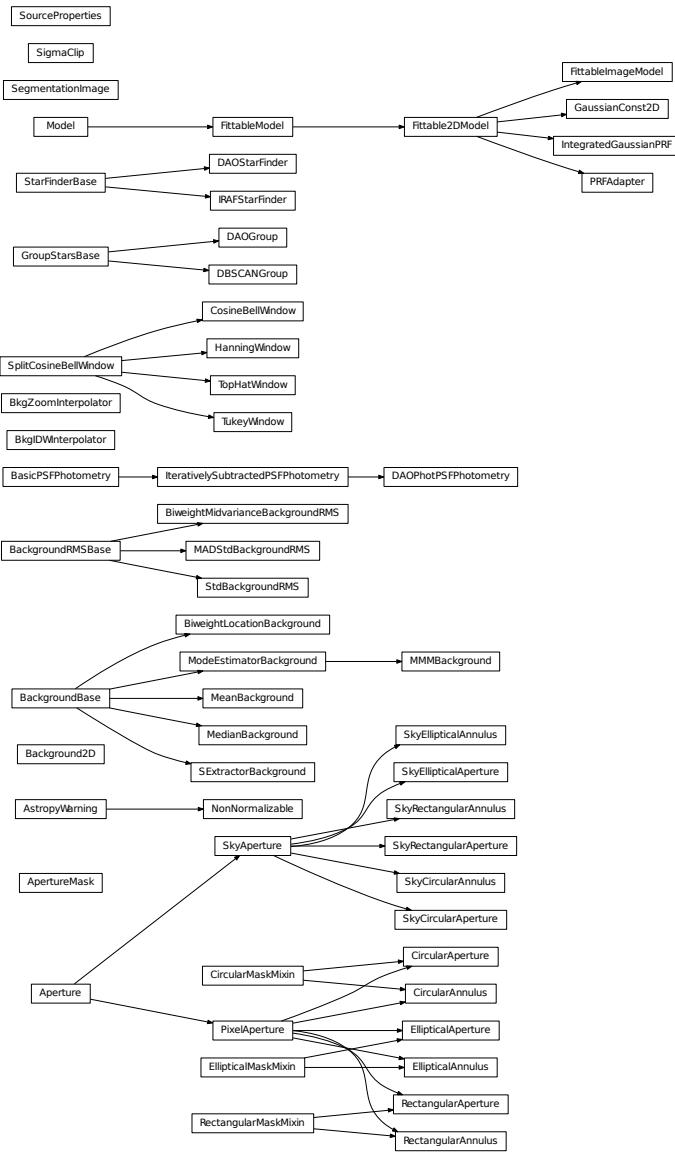


A 1D cut across the image center:

```
import matplotlib.pyplot as plt
from photutils import TukeyWindow
taper = TukeyWindow(alpha=0.4)
data = taper((101, 101))
plt.plot(data[50, :])
```



Class Inheritance Diagram



Note: Like much astronomy software, Photutils is an evolving package. The developers make an effort to maintain backwards compatibility, but at times the API may change if there is a benefit to doing so. If there are specific areas you think API stability is important, please let us know as part of the development process!

Part III

Reporting Issues

If you have found a bug in Photutils please report it by creating a new issue on the [Photutils GitHub issue tracker](#).

Please include an example that demonstrates the issue that will allow the developers to reproduce and fix the problem. You may be asked to also provide information about your operating system and a full Python stack trace. The developers will walk you through obtaining a stack trace if it is necessary.

Photutils uses a package of utilities called [astropy-helpers](#) during building and installation. If you have any build or installation issue mentioning the `astropy_helpers` or `ah_bootstrap` modules please send a report to the [astropy-helpers issue tracker](#). If you are unsure, then it's fine to report to the main Photutils issue tracker.

Part IV

Contributing

Like the [Astropy](#) project, Photutils is made both by and for its users. We accept contributions at all levels, spanning the gamut from fixing a typo in the documentation to developing a major new feature. We welcome contributors who will abide by the [Python Software Foundation Code of Conduct](#).

Photutils follows the same workflow and coding guidelines as [Astropy](#). The following pages will help you get started with contributing fixes, code, or documentation (no git or GitHub experience necessary):

- [How to make a code contribution](#)
- [Coding Guidelines](#)
- [Try the development version](#)
- [Developer Documentation](#)

Part V

Citing Photutils

If you use Photutils, please consider citing the package via its Zenodo record. If you just want the latest release, cite this (follow the link on the badge and then use one of the citation methods on the right): If you want to cite an earlier version, you can [search for photutils on Zenodo](#). Then cite the Zenodo DOI for whatever version(s) of Photutils you are using.

Bibliography

- [R4] Stetson, P. 1987; PASP 99, 191 (<http://adsabs.harvard.edu/abs/1987PASP...99..191S>)
- [R5] <http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?daofind>
- [R6] <http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?starfind>
- [R1] Stetson, P. 1987; PASP 99, 191 (<http://adsabs.harvard.edu/abs/1987PASP...99..191S>)
- [R2] <http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?daofind>
- [R3] <http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?starfind>

Python Module Index

p

`photutils`, [253](#)
`photutils.aperture`, [83](#)
`photutils.background`, [33](#)
`photutils.centroids`, [209](#)
`photutils.datasets`, [224](#)
`photutils.detection`, [59](#)
`photutils.geometry`, [219](#)
`photutils.morphology`, [216](#)
`photutils.psf`, [118](#)
`photutils.psf.matching`, [160](#)
`photutils.psf.sandbox`, [149](#)
`photutils.segmentation`, [180](#)
`photutils.utils`, [243](#)

Symbols

__call__(photutils.BackgroundBase method), 281
__call__(photutils.BackgroundRMSBase method), 282
__call__(photutils.BasicPSFPhotometry method), 284
__call__(photutils.BkgIDWInterpolator method), 288
__call__(photutils.BkgZoomInterpolator method), 289
__call__(photutils.GroupStarsBase method), 311
__call__(photutils.SigmaClip method), 342
__call__(photutils.SplitCosineBellWindow method), 359
__call__(photutils.StarFinderBase method), 360
__call__(photutils.background.BackgroundBase method), 38
__call__(photutils.background.BackgroundRMSBase method), 39
__call__(photutils.background.BkgIDWInterpolator method), 42
__call__(photutils.background.BkgZoomInterpolator method), 43
__call__(photutils.background.SigmaClip method), 50
__call__(photutils.detection.StarFinderBase method), 67
__call__(photutils.psf.BasicPSFPhotometry method), 123
__call__(photutils.psf.GroupStarsBase method), 135
__call__(photutils.psf.SplitCosineBellWindow method), 144
__call__(photutils.psf.matching.SplitCosineBellWindow method), 168
__call__(photutils.utils.ShepardIDWInterpolator method), 250

A

amplitude (photutils.centroids.GaussianConst2D attribute), 213
amplitude (photutils.GaussianConst2D attribute), 310
Aperture (class in photutils), 275
Aperture (class in photutils.aperture), 86
aperture_photometry() (in module photutils), 254
aperture_photometry() (in module photutils.aperture), 83
aperture_radius (photutils.BasicPSFPhotometry attribute), 284
aperture_radius (photutils.psf.BasicPSFPhotometry attribute), 123
ApertureMask (class in photutils), 276
ApertureMask (class in photutils.aperture), 86
apply() (photutils.aperture.ApertureMask method), 87
apply() (photutils.ApertureMask method), 276
area (photutils.segmentation.SourceProperties attribute), 200
area (photutils.SourceProperties attribute), 352
area() (photutils.aperture.CircularAnnulus method), 88
area() (photutils.aperture.CircularAperture method), 89
area() (photutils.aperture.EllipticalAnnulus method), 92
area() (photutils.aperture.EllipticalAperture method), 93
area() (photutils.aperture.PixelAperture method), 95
area() (photutils.aperture.RectangularAnnulus method), 98
area() (photutils.aperture.RectangularAperture method), 100
area() (photutils.CircularAnnulus method), 290
area() (photutils.CircularAperture method), 291
area() (photutils.EllipticalAnnulus method), 302
area() (photutils.EllipticalAperture method), 304
area() (photutils.PixelAperture method), 326
area() (photutils.RectangularAnnulus method), 330
area() (photutils.RectangularAperture method), 331
area() (photutils.segmentation.SegmentationImage method), 190
area() (photutils.SegmentationImage method), 335
areas (photutils.segmentation.SegmentationImage attribute), 190
areas (photutils.SegmentationImage attribute), 334
array (photutils.aperture.ApertureMask attribute), 86
array (photutils.ApertureMask attribute), 276
array (photutils.segmentation.SegmentationImage attribute), 190
array (photutils.SegmentationImage attribute), 334

B

background (photutils.background.Background2D attribute), 36
 background (photutils.Background2D attribute), 280
 Background2D (class in photutils), 277
 Background2D (class in photutils.background), 34
 background_at_centroid (photutils.segmentation.SourceProperties attribute), 200
 background_at_centroid (photutils.SourceProperties attribute), 352
 background_cutout_ma (photutils.segmentation.SourceProperties attribute), 200
 background_cutout_ma (photutils.SourceProperties attribute), 352
 background_mean (photutils.segmentation.SourceProperties attribute), 200
 background_mean (photutils.SourceProperties attribute), 352
 background_median (photutils.background.Background2D attribute), 36
 background_median (photutils.Background2D attribute), 280
 background_mesh_ma (photutils.background.Background2D attribute), 37
 background_mesh_ma (photutils.Background2D attribute), 280
 background_rms (photutils.background.Background2D attribute), 37
 background_rms (photutils.Background2D attribute), 280
 background_rms_median (photutils.background.Background2D attribute), 37
 background_rms_median (photutils.Background2D attribute), 280
 background_rms_mesh_ma (photutils.background.Background2D attribute), 37
 background_rms_mesh_ma (photutils.Background2D attribute), 280
 background_sum (photutils.segmentation.SourceProperties attribute), 200
 background_sum (photutils.SourceProperties attribute), 352
 BackgroundBase (class in photutils), 281
 BackgroundBase (class in photutils.background), 37
 BackgroundRMSBase (class in photutils), 281
 BackgroundRMSBase (class in photutils.background), 38
 BasicPSFPhotometry (class in photutils), 282

BasicPSFPhotometry (class in photutils.psf), 121
 bbox (photutils.segmentation.SourceProperties attribute), 200
 bbox (photutils.SourceProperties attribute), 352
 BiweightLocationBackground (class in photutils), 285
 BiweightLocationBackground (class in photutils.background), 39
 BiweightMidvarianceBackgroundRMS (class in photutils), 286
 BiweightMidvarianceBackgroundRMS (class in photutils.background), 40
 BkgIDWInterpolator (class in photutils), 287
 BkgIDWInterpolator (class in photutils.background), 41
 BkgZoomInterpolator (class in photutils), 288
 BkgZoomInterpolator (class in photutils.background), 42

C

calc_background() (photutils.background.BackgroundBase method), 38
 calc_background() (photutils.background.BiweightLocationBackground method), 40
 calc_background() (photutils.background.MeanBackground method), 46
 calc_background() (photutils.background.MedianBackground method), 47
 calc_background() (photutils.background.ModeEstimatorBackground method), 48
 calc_background() (photutils.background.SExtractorBackground method), 49
 calc_background() (photutils.BackgroundBase method), 281
 calc_background() (photutils.BiweightLocationBackground method), 286
 calc_background() (photutils.MeanBackground method), 322
 calc_background() (photutils.MedianBackground method), 323
 calc_background() (photutils.ModeEstimatorBackground method), 324
 calc_background() (photutils.SExtractorBackground method), 333
 calc_background_rms() (photutils.background.BackgroundRMSBase method), 39
 calc_background_rms() (photutils.background.BiweightMidvarianceBackgroundRMS method), 41

calc_background_rms() (photutils.background.MADStdBackgroundRMS method), 44

calc_background_rms() (photutils.background.StdBackgroundRMS method), 52

calc_background_rms() (photutils.BackgroundRMSBase method), 282

calc_background_rms() (photutils.BiweightMidvarianceBackgroundRMS method), 287

calc_background_rms() (photutils.MADStdBackgroundRMS method), 320

calc_background_rms() (photutils.StdBackgroundRMS method), 361

calc_total_error() (in module photutils.utils), 244

centroid (photutils.segmentation.SourceProperties attribute), 200

centroid (photutils.SourceProperties attribute), 353

centroid_1dg() (in module photutils), 256

centroid_1dg() (in module photutils.centroids), 209

centroid_2dg() (in module photutils), 256

centroid_2dg() (in module photutils.centroids), 210

centroid_com() (in module photutils), 257

centroid_com() (in module photutils.centroids), 210

check_label() (photutils.segmentation.SegmentationImage method), 190

check_label() (photutils.SegmentationImage method), 335

check_random_state() (in module photutils.utils), 244

circular_overlap_grid() (in module photutils.geometry), 219

CircularAnnulus (class in photutils), 289

CircularAnnulus (class in photutils.aperture), 88

CircularAperture (class in photutils), 290

CircularAperture (class in photutils.aperture), 89

CircularMaskMixin (class in photutils), 292

CircularMaskMixin (class in photutils.aperture), 90

compute_interpolator() (photutils.FittableImageModel method), 308

compute_interpolator() (photutils.psf.FittableImageModel method), 134

constant (photutils.centroids.GaussianConst2D attribute), 213

constant (photutils.GaussianConst2D attribute), 310

coords (photutils.segmentation.SourceProperties attribute), 200

coords (photutils.SourceProperties attribute), 353

copy() (photutils.segmentation.SegmentationImage method), 191

copy() (photutils.SegmentationImage method), 335

CosineBellWindow (class in photutils), 292

CosineBellWindow (class in photutils.psf), 124

CosineBellWindow (class in photutils.psf.matching), 162

covar_sigx2 (photutils.segmentation.SourceProperties attribute), 200

covar_sigx2 (photutils.SourceProperties attribute), 353

covar_sigxy (photutils.segmentation.SourceProperties attribute), 201

covar_sigxy (photutils.SourceProperties attribute), 353

covar_sigy2 (photutils.segmentation.SourceProperties attribute), 201

covar_sigy2 (photutils.SourceProperties attribute), 353

covariance (photutils.segmentation.SourceProperties attribute), 201

covariance (photutils.SourceProperties attribute), 353

covariance_eigvals (photutils.segmentation.SourceProperties attribute), 201

covariance_eigvals (photutils.SourceProperties attribute), 353

create_from_image() (photutils.psf.sandbox.DiscretePRF class method), 151

create_matching_kernel() (in module photutils), 257

create_matching_kernel() (in module photutils.psf), 118

create_matching_kernel() (in module photutils.psf.matching), 161

crit_separation (photutils.DAOGroup attribute), 295

crit_separation (photutils.psf.DAOGroup attribute), 127

cutout() (photutils.aperture.ApertureMask method), 87

cutout() (photutils.ApertureMask method), 277

cutout_centroid (photutils.segmentation.SourceProperties attribute), 201

cutout_centroid (photutils.SourceProperties attribute), 353

cutout_footprint() (in module photutils.utils), 245

cxx (photutils.segmentation.SourceProperties attribute), 201

cxx (photutils.SourceProperties attribute), 353

cxy (photutils.segmentation.SourceProperties attribute), 201

cxy (photutils.SourceProperties attribute), 353

cyy (photutils.segmentation.SourceProperties attribute), 201

cyy (photutils.SourceProperties attribute), 353

D

daofind() (in module photutils), 258

daofind() (in module photutils.detection), 59

DAOGroup (class in photutils), 294

DAOGroup (class in photutils.psf), 126

DAOPhotPSFPhotometry (class in photutils), 296

DAOPhotPSFPhotometry (class in photutils.psf), 128

DAOStarFinder (class in photutils), 298

DAOStarFinder (class in photutils.detection), 62

data (photutils.FittableImageModel attribute), 307

data (photutils.psf.FittableImageModel attribute), 133

data (photutils.segmentation.SegmentationImage attribute), 190

data (photutils.SegmentationImage attribute), 334

data_cutout (photutils.segmentation.SourceProperties attribute), 201

data_cutout (photutils.SourceProperties attribute), 354

data_cutout_ma (photutils.segmentation.SourceProperties attribute), 201

data_cutout_ma (photutils.SourceProperties attribute), 354

data_masked (photutils.segmentation.SegmentationImage attribute), 190

data_masked (photutils.SegmentationImage attribute), 334

data_properties() (in module photutils), 258

data_properties() (in module photutils.morphology), 217

DBSCANGroup (class in photutils), 300

DBSCANGroup (class in photutils.psf), 130

deblend_sources() (in module photutils), 258

deblend_sources() (in module photutils.segmentation), 181

dec_icrs_centroid (photutils.segmentation.SourceProperties attribute), 202

dec_icrs_centroid (photutils.SourceProperties attribute), 354

detect_sources() (in module photutils), 259

detect_sources() (in module photutils.segmentation), 182

detect_threshold() (in module photutils), 262

detect_threshold() (in module photutils.detection), 59

DiscretePRF (class in photutils.psf.sandbox), 149

do_photometry() (photutils.aperture.PixelAperture method), 95

do_photometry() (photutils.BasicPSFPhotometry method), 284

do_photometry() (photutils.IterativelySubtractedPSFPhotometry method), 318

do_photometry() (photutils.PixelAperture method), 326

do_photometry() (photutils.psf.BasicPSFPhotometry method), 123

do_photometry() (photutils.psf.IterativelySubtractedPSFPhotometry method), 140

E

eccentricity (photutils.segmentation.SourceProperties attribute), 202

eccentricity (photutils.SourceProperties attribute), 354

elliptical_overlap_grid() (in module photutils.geometry), 220

EllipticalAnnulus (class in photutils), 301

EllipticalAnnulus (class in photutils.aperture), 91

EllipticalAperture (class in photutils), 303

EllipticalAperture (class in photutils.aperture), 92

EllipticalMaskMixin (class in photutils), 304

EllipticalMaskMixin (class in photutils.aperture), 93

ellipticity (photutils.segmentation.SourceProperties attribute), 202

ellipticity (photutils.SourceProperties attribute), 354

elongation (photutils.segmentation.SourceProperties attribute), 202

elongation (photutils.SourceProperties attribute), 354

equivalent_radius (photutils.segmentation.SourceProperties attribute), 202

equivalent_radius (photutils.SourceProperties attribute), 354

error_cutout_ma (photutils.segmentation.SourceProperties attribute), 202

error_cutout_ma (photutils.SourceProperties attribute), 354

evaluate() (photutils.centroids.GaussianConst2D static method), 213

evaluate() (photutils.FittableImageModel method), 308

evaluate() (photutils.GaussianConst2D static method), 310

evaluate() (photutils.IntegratedGaussianPRF method), 317

evaluate() (photutils.PRFAdapter method), 326

evaluate() (photutils.psf.FittableImageModel method), 134

evaluate() (photutils.psf.IntegratedGaussianPRF method), 138

evaluate() (photutils.psf.PRFAdapter method), 142

evaluate() (photutils.psf.sandbox.DiscretePRF method), 151

F

fill_value (photutils.FittableImageModel attribute), 307

fill_value (photutils.psf.FittableImageModel attribute), 133

filter_data() (in module photutils.utils), 246

find_group() (photutils.DAOGroup method), 295

find_group() (photutils.psf.DAOGroup method), 127

find_peaks() (in module photutils), 263

find_peaks() (in module photutils.detection), 60

find_stars() (photutils.DAOStarFinder method), 299

find_stars() (photutils.detection.DAOStarFinder method), 63

find_stars() (photutils.detection.IRAFStarFinder method), 65

find_stars() (photutils.detection.StarFinderBase method), 67

find_stars() (photutils.IRAFStarFinder method), 314

find_stars() (photutils.StarFinderBase method), 360

finder (photutils.IterativelySubtractedPSFPhotometry attribute), 318
 finder (photutils.psf.IterativelySubtractedPSFPhotometry attribute), 140
 fit_2dgaussian() (in module photutils), 265
 fit_2dgaussian() (in module photutils.centroids), 211
 fit_deriv (photutils.IntegratedGaussianPRF attribute), 316
 fit_deriv (photutils.psf.IntegratedGaussianPRF attribute), 138
 fitshape (photutils.BasicPSFPhotometry attribute), 284
 fitshape (photutils.psf.BasicPSFPhotometry attribute), 123
 FittableImageModel (class in photutils), 305
 FittableImageModel (class in photutils.psf), 131
 flux (photutils.FittableImageModel attribute), 307
 flux (photutils.IntegratedGaussianPRF attribute), 316
 flux (photutils.PRFAdapter attribute), 325
 flux (photutils.psf.FittableImageModel attribute), 133
 flux (photutils.psf.IntegratedGaussianPRF attribute), 138
 flux (photutils.psf.PRFAdapter attribute), 142
 flux (photutils.psf.sandbox.DiscretePRF attribute), 150

G

gaussian1d_moments() (in module photutils), 265
 gaussian1d_moments() (in module photutils.centroids), 211
 GaussianConst2D (class in photutils), 309
 GaussianConst2D (class in photutils.centroids), 212
 get_grouped_psf_model() (in module photutils), 265
 get_grouped_psf_model() (in module photutils.psf), 119
 get_path() (in module photutils.datasets), 225
 get_residual_image() (photutils.BasicPSFPhotometry method), 284
 get_residual_image() (photutils.psf.BasicPSFPhotometry method), 123
 gini() (in module photutils), 266
 gini() (in module photutils.morphology), 217
 group_stars() (photutils.DAOGroup method), 295
 group_stars() (photutils.DBSCANGroup method), 301
 group_stars() (photutils.psf.DAOGroup method), 127
 group_stars() (photutils.psf.DBSCANGroup method), 130
 GroupStarsBase (class in photutils), 310
 GroupStarsBase (class in photutils.psf), 134

H

HanningWindow (class in photutils), 311
 HanningWindow (class in photutils.psf), 135
 HanningWindow (class in photutils.psf.matching), 164

I

icrs_centroid (photutils.segmentation.SourceProperties attribute), 202
 icrs_centroid (photutils.SourceProperties attribute), 354
 id (photutils.segmentation.SourceProperties attribute), 202
 id (photutils.SourceProperties attribute), 354
 inertia_tensor (photutils.segmentation.SourceProperties attribute), 202
 inertia_tensor (photutils.SourceProperties attribute), 354
 IntegratedGaussianPRF (class in photutils), 315
 IntegratedGaussianPRF (class in photutils.psf), 137
 interpolate_masked_data() (in module photutils.utils), 246
 interpolator_kwarg (photutils.FittableImageModel attribute), 307
 interpolator_kwarg (photutils.psf.FittableImageModel attribute), 133
 irafstarfind() (in module photutils), 266
 irafstarfind() (in module photutils.detection), 61
 IRAFStarFinder (class in photutils), 313
 IRAFStarFinder (class in photutils.detection), 64
 is_sequential (photutils.segmentation.SegmentationImage attribute), 190
 is_sequential (photutils.SegmentationImage attribute), 335
 IterativelySubtractedPSFPhotometry (class in photutils), 317
 IterativelySubtractedPSFPhotometry (class in photutils.psf), 139

K

keep_labels() (photutils.segmentation.SegmentationImage method), 191
 keep_labels() (photutils.SegmentationImage method), 335

L

labels (photutils.segmentation.SegmentationImage attribute), 190
 labels (photutils.SegmentationImage attribute), 335
 load_fermi_image() (in module photutils.datasets), 225
 load_irac_psf() (in module photutils.datasets), 226
 load_spitzer_catalog() (in module photutils.datasets), 227
 load_spitzer_image() (in module photutils.datasets), 228
 load_star_image() (in module photutils.datasets), 229

M

MADStdBackgroundRMS (class in photutils), 319
 MADStdBackgroundRMS (class in photutils.background), 43
 make_100gaussians_image() (in module photutils.datasets), 230
 make_4gaussians_image() (in module photutils.datasets), 231
 make_cutout() (photutils.segmentation.SourceProperties method), 204
 make_cutout() (photutils.SourceProperties method), 356

make_gaussian_sources() (in module photutils.datasets), 232
 make_noise_image() (in module photutils.datasets), 236
 make_poisson_noise() (in module photutils.datasets), 237
 make_random_gaussians() (in module photutils.datasets), 239
 make_source_mask() (in module photutils), 267
 make_source_mask() (in module photutils.segmentation), 184
 mask_area() (photutils.aperture.PixelAperture method), 96
 mask_area() (photutils.PixelAperture method), 327
 mask_to_mirrored_num() (in module photutils.utils), 247
 max (photutils.segmentation.SegmentationImage attribute), 190
 max (photutils.SegmentationImage attribute), 335
 max_value (photutils.segmentation.SourceProperties attribute), 202
 max_value (photutils.SourceProperties attribute), 354
 maxval_cutout_pos (photutils.segmentation.SourceProperties attribute), 202
 maxval_cutout_pos (photutils.SourceProperties attribute), 355
 maxval_pos (photutils.segmentation.SourceProperties attribute), 202
 maxval_pos (photutils.SourceProperties attribute), 355
 maxval_xpos (photutils.segmentation.SourceProperties attribute), 202
 maxval_xpos (photutils.SourceProperties attribute), 355
 maxval_ypos (photutils.segmentation.SourceProperties attribute), 202
 maxval_ypos (photutils.SourceProperties attribute), 355
 MeanBackground (class in photutils), 321
 MeanBackground (class in photutils.background), 45
 MedianBackground (class in photutils), 322
 MedianBackground (class in photutils.background), 46
 mesh_nmasks (photutils.background.Background2D attribute), 37
 mesh_nmasks (photutils.Background2D attribute), 280
 min_value (photutils.segmentation.SourceProperties attribute), 203
 min_value (photutils.SourceProperties attribute), 355
 minval_cutout_pos (photutils.segmentation.SourceProperties attribute), 203
 minval_cutout_pos (photutils.SourceProperties attribute), 355
 minval_pos (photutils.segmentation.SourceProperties attribute), 203
 minval_pos (photutils.SourceProperties attribute), 355
 minval_xpos (photutils.segmentation.SourceProperties attribute), 203
 minval_xpos (photutils.SourceProperties attribute), 355
 minval_ypos (photutils.segmentation.SourceProperties attribute), 203
 minval_ypos (photutils.SourceProperties attribute), 355
 MMMBackground (class in photutils), 320
 MMMBackground (class in photutils.background), 44
 ModeEstimatorBackground (class in photutils), 323
 ModeEstimatorBackground (class in photutils.background), 47
 moments (photutils.segmentation.SourceProperties attribute), 203
 moments (photutils.SourceProperties attribute), 355
 moments_central (photutils.segmentation.SourceProperties attribute), 203
 moments_central (photutils.SourceProperties attribute), 355

N

niters (photutils.IterativelySubtractedPSFPhotometry attribute), 318
 niters (photutils.psf.IterativelySubtractedPSFPhotometry attribute), 140
 nlabs (photutils.segmentation.SegmentationImage attribute), 190
 nlabs (photutils.SegmentationImage attribute), 335
 NonNormalizable, 141, 324
 normalization_constant (photutils.FittableImageModel attribute), 307
 normalization_constant (photutils.psf.FittableImageModel attribute), 133
 normalization_correction (photutils.FittableImageModel attribute), 307
 normalization_correction (photutils.psf.FittableImageModel attribute), 133
 normalization_status (photutils.FittableImageModel attribute), 307
 normalization_status (photutils.psf.FittableImageModel attribute), 133
 normalized_data (photutils.FittableImageModel attribute), 307
 normalized_data (photutils.psf.FittableImageModel attribute), 133
 nstar() (photutils.BasicPSFPhotometry method), 285
 nstar() (photutils.psf.BasicPSFPhotometry method), 124
 nx (photutils.FittableImageModel attribute), 307
 nx (photutils.psf.FittableImageModel attribute), 133
 ny (photutils.FittableImageModel attribute), 307
 ny (photutils.psf.FittableImageModel attribute), 133

O

orientation (photutils.segmentation.SourceProperties attribute), 203
 orientation (photutils.SourceProperties attribute), 355
 origin (photutils.FittableImageModel attribute), 307

origin (photutils.psf.FittableImageModel attribute), 133
outline_segments() (photutils.segmentation.SegmentationImage method), 192
outline_segments() (photutils.SegmentationImage method), 336
oversampling (photutils.FittableImageModel attribute), 308
oversampling (photutils.psf.FittableImageModel attribute), 133

P

param_names (photutils.centroids.GaussianConst2D attribute), 213
param_names (photutils.FittableImageModel attribute), 308
param_names (photutils.GaussianConst2D attribute), 310
param_names (photutils.IntegratedGaussianPRF attribute), 316
param_names (photutils.PRFAdapter attribute), 325
param_names (photutils.psf.FittableImageModel attribute), 133
param_names (photutils.psf.IntegratedGaussianPRF attribute), 138
param_names (photutils.psf.PRFAdapter attribute), 142
param_names (photutils.psf.sandbox.DiscretePRF attribute), 150
perimeter (photutils.segmentation.SourceProperties attribute), 203
perimeter (photutils.SourceProperties attribute), 355
photutils (module), 253
photutils.aperture (module), 83
photutils.background (module), 33
photutils.centroids (module), 209
photutils.datasets (module), 224
photutils.detection (module), 59
photutils.geometry (module), 219
photutils.morphology (module), 216
photutils.psf (module), 118
photutils.psf.matching (module), 160
photutils.psf.sandbox (module), 149
photutils.segmentation (module), 180
photutils.utils (module), 243
PixelAperture (class in photutils), 326
PixelAperture (class in photutils.aperture), 94
plot() (photutils.aperture.CircularAnnulus method), 88
plot() (photutils.aperture.CircularAperture method), 90
plot() (photutils.aperture.EllipticalAnnulus method), 92
plot() (photutils.aperture.EllipticalAperture method), 93
plot() (photutils.aperture.PixelAperture method), 96
plot() (photutils.aperture.RectangularAnnulus method), 98
plot() (photutils.aperture.RectangularAperture method), 100

plot() (photutils.CircularAnnulus method), 290
plot() (photutils.CircularAperture method), 291
plot() (photutils.EllipticalAnnulus method), 302
plot() (photutils.EllipticalAperture method), 304
plot() (photutils.PixelAperture method), 328
plot() (photutils.RectangularAnnulus method), 330
plot() (photutils.RectangularAperture method), 331
plot_meshes() (photutils.background.Background2D method), 37
plot_meshes() (photutils.Background2D method), 280
prepare_psf_model() (in module photutils), 268
prepare_psf_model() (in module photutils.psf), 119
prf_shape (photutils.psf.sandbox.DiscretePRF attribute), 150
PRFAdapter (class in photutils), 324
PRFAdapter (class in photutils.psf), 141
properties_table() (in module photutils), 268
properties_table() (in module photutils.segmentation), 185

R

ra_icrs_centroid (photutils.segmentation.SourceProperties attribute), 203
ra_icrs_centroid (photutils.SourceProperties attribute), 355
random_cmap() (in module photutils.utils), 248
rectangular_overlap_grid() (in module photutils.geometry), 220
RectangularAnnulus (class in photutils), 329
RectangularAnnulus (class in photutils.aperture), 97
RectangularAperture (class in photutils), 330
RectangularAperture (class in photutils.aperture), 99
RectangularMaskMixin (class in photutils), 331
RectangularMaskMixin (class in photutils.aperture), 100
relabel() (photutils.segmentation.SegmentationImage method), 192
relabel() (photutils.SegmentationImage method), 337
relabel_sequential() (photutils.segmentation.SegmentationImage method), 193
relabel_sequential() (photutils.SegmentationImage method), 337
remove_border_labels() (photutils.segmentation.SegmentationImage method), 193
remove_border_labels() (photutils.SegmentationImage method), 338
remove_labels() (photutils.segmentation.SegmentationImage method), 194
remove_labels() (photutils.SegmentationImage method), 339

S
 remove_masked_labels() (photutils.segmentation.SegmentationImage method), 195
 remove_masked_labels() (photutils.SegmentationImage method), 340
 resize_psf() (in module photutils), 269
 resize_psf() (in module photutils.psf), 120
 resize_psf() (in module photutils.psf.matching), 161

S
 SegmentationImage (class in photutils), 333
 SegmentationImage (class in photutils.segmentation), 189
 semimajor_axis_sigma (photutils.segmentation.SourceProperties attribute), 203
 semimajor_axis_sigma (photutils.SourceProperties attribute), 355
 semiminor_axis_sigma (photutils.segmentation.SourceProperties attribute), 203
 semiminor_axis_sigma (photutils.SourceProperties attribute), 355
 SExtractorBackground (class in photutils), 332
 SExtractorBackground (class in photutils.background), 48
 shape (photutils.FittableImageModel attribute), 308
 shape (photutils.psf.FittableImageModel attribute), 133
 shape (photutils.segmentation.SegmentationImage attribute), 190
 shape (photutils.SegmentationImage attribute), 335
 ShepardIDWInterpolator (class in photutils.utils), 249
 sigma (photutils.IntegratedGaussianPRF attribute), 316
 sigma (photutils.psf.IntegratedGaussianPRF attribute), 138
 SigmaClip (class in photutils), 341
 SigmaClip (class in photutils.background), 50
 SkyAperture (class in photutils), 342
 SkyAperture (class in photutils.aperture), 101
 SkyCircularAnnulus (class in photutils), 343
 SkyCircularAnnulus (class in photutils.aperture), 102
 SkyCircularAperture (class in photutils), 344
 SkyCircularAperture (class in photutils.aperture), 102
 SkyEllipticalAnnulus (class in photutils), 344
 SkyEllipticalAnnulus (class in photutils.aperture), 103
 SkyEllipticalAperture (class in photutils), 345
 SkyEllipticalAperture (class in photutils.aperture), 104
 SkyRectangularAnnulus (class in photutils), 346
 SkyRectangularAnnulus (class in photutils.aperture), 106
 SkyRectangularAperture (class in photutils), 347
 SkyRectangularAperture (class in photutils.aperture), 107
 slices (photutils.segmentation.SegmentationImage attribute), 190
 slices (photutils.SegmentationImage attribute), 335
 source_properties() (in module photutils), 270

source_properties() (in module photutils.segmentation), 186
 source_sum (photutils.segmentation.SourceProperties attribute), 203
 source_sum (photutils.SourceProperties attribute), 355
 source_sum_err (photutils.segmentation.SourceProperties attribute), 203
 source_sum_err (photutils.SourceProperties attribute), 356
 SourceProperties (class in photutils), 348
 SourceProperties (class in photutils.segmentation), 196
 SplitCosineBellWindow (class in photutils), 357
 SplitCosineBellWindow (class in photutils.psf), 142
 SplitCosineBellWindow (class in photutils.psf.matching), 166
 StarFinderBase (class in photutils), 359
 StarFinderBase (class in photutils.detection), 66
 std_blocksum() (in module photutils.utils), 248
 StdBackgroundRMS (class in photutils), 361
 StdBackgroundRMS (class in photutils.background), 51
 subtract_psf() (in module photutils), 272
 subtract_psf() (in module photutils.psf), 120

T
 test() (in module photutils), 272
 theta (photutils.centroids.GaussianConst2D attribute), 213
 theta (photutils.GaussianConst2D attribute), 310
 to_image() (photutils.aperture.ApertureMask method), 87
 to_image() (photutils.ApertureMask method), 277
 to_mask() (photutils.aperture.CircularMaskMixin method), 90
 to_mask() (photutils.aperture.EllipticalMaskMixin method), 94
 to_mask() (photutils.aperture.PixelAperture method), 97
 to_mask() (photutils.aperture.RectangularMaskMixin method), 100
 to_mask() (photutils.CircularMaskMixin method), 292
 to_mask() (photutils.EllipticalMaskMixin method), 304
 to_mask() (photutils.PixelAperture method), 328
 to_mask() (photutils.RectangularMaskMixin method), 332
 to_pixel() (photutils.aperture.SkyAperture method), 101
 to_pixel() (photutils.aperture.SkyCircularAnnulus method), 102
 to_pixel() (photutils.aperture.SkyCircularAperture method), 103
 to_pixel() (photutils.aperture.SkyEllipticalAnnulus method), 104
 to_pixel() (photutils.aperture.SkyEllipticalAperture method), 106
 to_pixel() (photutils.aperture.SkyRectangularAnnulus method), 107

to_pixel() (photutils.aperture.SkyRectangularAperture method), 109
to_pixel() (photutils.SkyAperture method), 342
to_pixel() (photutils.SkyCircularAnnulus method), 343
to_pixel() (photutils.SkyCircularAperture method), 344
to_pixel() (photutils.SkyEllipticalAnnulus method), 345
to_pixel() (photutils.SkyEllipticalAperture method), 346
to_pixel() (photutils.SkyRectangularAnnulus method), 347
to_pixel() (photutils.SkyRectangularAperture method), 348
to_table() (photutils.segmentation.SourceProperties method), 204
to_table() (photutils.SourceProperties method), 356
TopHatWindow (class in photutils), 362
TopHatWindow (class in photutils.psf), 144
TopHatWindow (class in photutils.psf.matching), 168
TukeyWindow (class in photutils), 364
TukeyWindow (class in photutils.psf), 146
TukeyWindow (class in photutils.psf.matching), 170

V

values (photutils.segmentation.SourceProperties attribute), 203
values (photutils.SourceProperties attribute), 356

X

x_0 (photutils.FittableImageModel attribute), 308
x_0 (photutils.IntegratedGaussianPRF attribute), 316
x_0 (photutils.PRFAdapter attribute), 325
x_0 (photutils.psf.FittableImageModel attribute), 133
x_0 (photutils.psf.IntegratedGaussianPRF attribute), 138
x_0 (photutils.psf.PRFAdapter attribute), 142
x_0 (photutils.psf.sandbox.DiscretePRF attribute), 150
x_mean (photutils.centroids.GaussianConst2D attribute), 213
x_mean (photutils.GaussianConst2D attribute), 310
x_origin (photutils.FittableImageModel attribute), 308
x_origin (photutils.psf.FittableImageModel attribute), 133
x_stddev (photutils.centroids.GaussianConst2D attribute), 213
x_stddev (photutils.GaussianConst2D attribute), 310
xcentroid (photutils.segmentation.SourceProperties attribute), 204
xcentroid (photutils.SourceProperties attribute), 356
xmax (photutils.segmentation.SourceProperties attribute), 204
xmax (photutils.SourceProperties attribute), 356
xmin (photutils.segmentation.SourceProperties attribute), 204
xmin (photutils.SourceProperties attribute), 356

Y

y_0 (photutils.FittableImageModel attribute), 308
y_0 (photutils.IntegratedGaussianPRF attribute), 316
y_0 (photutils.PRFAdapter attribute), 325
y_0 (photutils.psf.FittableImageModel attribute), 134
y_0 (photutils.psf.IntegratedGaussianPRF attribute), 138
y_0 (photutils.psf.PRFAdapter attribute), 142
y_0 (photutils.psf.sandbox.DiscretePRF attribute), 150
y_mean (photutils.centroids.GaussianConst2D attribute), 213
y_mean (photutils.GaussianConst2D attribute), 310
y_origin (photutils.FittableImageModel attribute), 308
y_origin (photutils.psf.FittableImageModel attribute), 134
y_stddev (photutils.centroids.GaussianConst2D attribute), 213
y_stddev (photutils.GaussianConst2D attribute), 310
ycentroid (photutils.segmentation.SourceProperties attribute), 204
ycentroid (photutils.SourceProperties attribute), 356
ymax (photutils.segmentation.SourceProperties attribute), 204
ymax (photutils.SourceProperties attribute), 356
ymin (photutils.segmentation.SourceProperties attribute), 204
ymin (photutils.SourceProperties attribute), 356