ccdproc Documentation

Release 0.2

Steve Crawford and Matt Craig

July 28, 2014

Contents

Welcome to the ccdproc documentation! Ccdproc is is an affiliated package for the AstroPy package for basic data reductions of CCD images. The ccdproc package provides many of the necessary tools for processing of ccd images built on a framework to provide error propogation and bad pixel tracking throughout the reduction process.

The documentation for this package is here:

Part I

Installation

CHAPTER 1

Requirements

Ccdproc has the following requirements:

- astropy v0.4 or later
- numpy
- scipy

One easy way to get these dependencies is to install a python distribution like anaconda.

CHAPTER 2

Installing ccdproc

2.1 Using pip

To install ccdproc with pip, simply run:

```
pip install --no-deps ccdproc
```

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

CHAPTER 3

Building from source

3.1 Obtaining the source packages

3.1.1 Source packages

The latest stable source package for ccdproc can be downloaded here.

3.1.2 Development repository

The latest development version of ccdproc can be cloned from github using this command:

```
git clone git://github.com/astropy/ccdproc.git
```

3.2 Building and Installing

To build ccdproc (from the root of the source tree):

```
python setup.py build
```

To install ccdproc (from the root of the source tree):

```
python setup.py install
```

3.3 Testing a source code build of ccdproc

The easiest way to test that your ccdproc built correctly (without installing ccdproc) is to run this from the root of the source tree:

```
python setup.py test
```

Part II

CCD Data reduction (ccdproc)

CHAPTER 4

Introduction

Note: ccdproc works only with astropy version 0.4.0 or later.

The ccdproc package provides:

- An image class, CCDData, that includes an uncertainty for the data, units and methods for performing arithmetic with images including the propagation of uncertainties.
- A set of functions performing common CCD data reduction steps (e.g. dark subtraction, flat field correction) with a flexible mechanism for logging reduction steps in the image metadata.
- A class for combining and/or clipping images, Combiner, and associated functions.

CHAPTER 5

Getting Started

Warning: ccdproc is still under active development. The API will almost certainly change.
In addition, testing of ccdproc on real data is currently very limited. Use with caution, and please report any errors you find at the GitHub repo for this project.

A CCDData object can be created from a numpy array (masked or not) or from a FITS file:

```
>>> import numpy as np
>>> from astropy import units as u
>>> import ccdproc
>>> image_1 = ccdproc.CCDData(np.ones((10, 10)), unit="adu")
```

An example of reading from a FITS file is `image_2 = ccdproc.CCDData.read('my_image.fits', unit="electron")` (the electron unit is defined as part of ccdproc).

The metadata of a CCDData object is a case-insensitive dictionary (though this may change in future versions).

The data is accessible either by indexing directly or through the data attribute:

```
>>> sub_image = image_1[:, 1:-3]  # a CCDData object
>>> sub_data =  image_1.data[:, 1:-3]  # a numpy array
```

See the documentation for CCDData for a complete list of attributes.

Most operations are performed by functions in ccdproc:

```
>>> dark = ccdproc.CCDData(np.random.normal(size=(10, 10)), unit="adu")
>>> dark_sub = ccdproc.subtract_dark(image_1, dark,
...                                  dark_exposure=30*u.second,
...                                  data_exposure=15*u.second,
...                                  scale=True)
```

Every function returns a *copy* of the data with the operation performed. If, for some reason, you wanted to modify the data in-place, do this:

```
>>> image_2 = ccdproc.subtract_dark(image_1, dark, dark_exposure=30*u.second, data_exposure=15*u.second, scale=True)
```

See the documentation for subtract_dark for more compact ways of providing exposure times.

Every function in ccdproc supports logging through the addition of information to the image metadata.

Logging can be simple – add a string to the metadata:

```
>>> image_2_gained = ccdproc.gain_correct(image_2, 1.5 * u.photon/u.adu, add_keyword='gain_corrected')
```

Logging can be more complicated – add several keyword/value pairs by passing a dictionary to `add_keyword`:

```
>>> my_log = {'gain_correct': 'Gain value was 1.5',
...           'calstat': 'G'}
>>> image_2_gained = ccdproc.gain_correct(image_2,
...                                       1.5 * u.photon/u.adu,
...                                       add_keyword=my_log)
```

The `Keyword` class provides a compromise between the simple and complicated cases for providing a single key/value pair:

```
>>> key = ccdproc.Keyword('gain_corrected', value='Yes')
>>> image_2_gained = ccdproc.gain_correct(image_2,
...                                       1.5 * u.photon/u.adu,
...                                       add_keyword=key)
```

`Keyword` also provides a convenient way to get a value from image metadata and specify its unit:

```
>>> image_2.header['gain']  = 1.5
>>> gain = ccdproc.Keyword('gain', unit=u.photon/u.adu)
>>> image_2_var = ccdproc.create_variance(image_2,
...                                       gain=gain.value_from(image_2.header),
...                                       readnoise=3.0 * u.photon)
```

You might wonder why there is a `gain_correct` at all, since the implemented gain correction simple multiplies by a constant. There are two things you get with `gain_correct` that you do not get with multiplication:

- Appropriate scaling of uncertainties.
- Units

The same advantages apply to operations that are more complex, like flat correction, in which one image is divided by another:

```
>>> flat = ccdproc.CCDData(np.random.normal(1.0, scale=0.1, size=(10, 10)),
...                        unit='adu')
>>> image_1_flat = ccdproc.flat_correct(image_1, flat)
```

In addition to doing the necessary division, `flat_correct` propagates uncertainties (if they are set).

CHAPTER 6

Using ccdproc

6.1 Image class

6.1.1 Getting started

Getting data in

The tools in ccdproc accept only CCDData objects, a subclass of NDData.

Creating a CCDData object from any array-like data is easy:

```
>>> import numpy as np
>>> import ccdproc
>>> ccd = ccdproc.CCDData(np.arange(10), unit="adu")
```

Note that behind the scenes, NDData creates references to (not copies of) your data when possible, so modifying the data in ccd will modify the underlying data.

You are **required** to provide a unit for your data. The most frequently used units for these objects are likely to be adu, photon and electron, which can be set either by providing the string name of the unit (as in the example above) or from unit objects:

```
>>> from astropy import units as u
>>> ccd_photon = ccdproc.CCDData([1, 2, 3], unit=u.photon)
>>> ccd_electron = ccdproc.CCDData([1, 2, 3], unit="electron")
```

If you prefer *not* to use the unit functionality then use the special unit u.dimensionless_unscaled when you create your CCDData images:

```
>>> ccd_unitless = ccdproc.CCDData(np.zeros((10, 10)),
...                                unit=u.dimensionless_unscaled)
```

A CCDData object can also be initialized from a FITS file:

```
>>> ccd = ccdproc.CCDData.read('my_file.fits', unit="adu")
```

If there is a unit in the FITS file (in the BUNIT keyword), that will be used, but a unit explicitly provided in read will override any unit in the FITS file.

There is no restriction at all on what the unit can be – any unit in astropy.units or that you create yourself will work.

Metadata

When initializing from a FITS file, the `header` property is initialized using the header of the FITS file. Metadata is optional, and can be provided by any dictionary or dict-like object:

```
>>> ccd_simple = ccdproc.CCDData(np.arange(10), unit="adu")
>>> my_meta = {'observer': 'Edwin Hubble', 'exposure': 30.0}
>>> ccd_simple.header = my_meta  # or use ccd_simple.meta = my_meta
```

Search of the metadata is case-insensitive:

```
>>> 'OBSERVER' in ccd_simple.header
True
>>> ccd_simple.header['ExPoSuRe']
30.0
```

Note, however, that internally all keywords are converted to lowercase.

Getting data out

A `CCDData` object behaves like a numpy array (masked if the `CCDData` mask is set) in expressions, and the underlying data (ignoring any mask) is accessed through `data` attribute:

```
>>> ccd_masked = ccdproc.CCDData([1, 2, 3], unit="adu", mask=[0, 0, 1])
>>> 2 * np.ones(3) * ccd_masked   # one return value will be masked
masked_array(data = [2.0 4.0 --],
             mask = [False False  True],
       fill_value = 1e+20)

>>> 2 * np.ones(3) * ccd_masked.data   # ignores the mask
array([ 2.,  4.,  6.])
```

You can force conversion to a numpy array with:

```
>>> np.asarray(ccd_masked)
array([1, 2, 3])
>>> np.ma.array(ccd_masked.data, mask=ccd_masked.mask)
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
```

A method for converting a `CCDData` object to a FITS HDU list is also available. It converts the metadata to a FITS header:

```
>>> hdulist = ccd_masked.to_hdu()
```

You can also write directly to a FITS file:

```
>>> ccd_masked.write('my_image.fits')
```

Masks and flags

Although not required when a `CCDData` image is created you can also specify a mask and/or flags.

A mask is a boolean array the same size as the data in which a value of `True` indicates that a particular pixel should be masked, *i.e.* not be included in arithmetic operations or aggregation.

Flags are one or more additional arrays (of any type) whose shape matches the shape of the data. For more details on setting flags see astropy.nddata.NDData.

6.1.2 Uncertainty

Pixel-by-pixel uncertainty can be calculated for you:

```
>>> data = np.random.normal(size=(10, 10), loc=1.0, scale=0.1)
>>> ccd = ccdproc.CCDData(data, unit="electron")
>>> ccd_new = ccdproc.create_variance(ccd, readnoise=5 * u.electron)
```

See *Gain correct and create variance* for more details.

You can also set the uncertainty directly, either by creating a StdDevUncertainty object first:

```
>>> from astropy.nddata.nduncertainty import StdDevUncertainty
>>> uncertainty = 0.1 * ccd.data  # can be any array whose shape matches the data
>>> my_uncertainty = StdDevUncertainty(uncertainty)
>>> ccd.uncertainty = my_uncertainty
```

or by providing a ndarray with the same shape as the data:

```
>>> ccd.uncertainty = 0.1 * ccd.data
INFO: Array provided for uncertainty; assuming it is a StdDevUncertainty. [ccdproc.ccddata]
```

In this case the uncertainty is assumed to be StdDevUncertainty. Using StdDevUncertainty is required to enable error propagation in CCDData

If you want access to the underlying uncertainty use its .array attribute:

```
>>> ccd.uncertainty.array
array(...)
```

6.1.3 Arithmetic with images

Methods are provided to perform arithmetic operations with a CCDData image and a number, an astropy Quantity (a number with units) or another CCDData image.

Using these methods propagates errors correctly (if the errors are uncorrelated), take care of any necessary unit conversions, and apply masks appropriately. Note that the metadata of the result is *not* set:

```
>>> result = ccd.multiply(0.2 * u.adu)
>>> uncertainty_ratio = result.uncertainty.array[0, 0]/ccd.uncertainty.array[0, 0]
>>> round(uncertainty_ratio, 5)
0.2
>>> result.unit
Unit("adu electron")
>>> result.header
CaseInsensitiveOrderedDict()
```

Note: In most cases you should use the functions described in *Reduction toolbox* to perform common operations like scaling by gain or doing dark or sky subtraction. Those functions try to construct a sensible header for the result and provide a mechanism for logging the action of the function in the header.

The arithmetic operators *, /, + and - are *not* overridden.

6.2 Combining images and generating masks from clipping

Note: No attempt has been made yet to optimize memory usage in `Combiner`. A copy is made, and a mask array constructed, for each input image.

The first step in combining a set of images is creating a `Combiner` instance:

```
>>> from astropy import units as u
>>> from ccdproc import CCDData, Combiner
>>> import numpy as np
>>> ccd1 = CCDData(np.random.normal(size=(10,10)),
...                unit=u.adu)
>>> ccd2 = ccd1.copy()
>>> ccd3 = ccd1.copy()
>>> combiner = Combiner([ccd1, ccd2, ccd3])
```

The combiner task really combines two things: generation of masks for individual images via several clipping techniques and combination of images.

6.2.1 Image masks/clipping

There are currently two methods of clipping. Neither affects the data directly; instead each constructs a mask that is applied when images are combined.

Masking done by clipping operations is combined with the image mask provided when the `Combiner` is created.

Min/max clipping

`minmax_clipping` masks all pixels above or below user-specified levels. For example, to mask all values above the value `0.1` and below the value `-0.3`:

```
>>> combiner.minmax_clipping(min_clip=-0.3, max_clip=0.1)
```

Either `min_clip` or `max_clip` can be omitted.

Sigma clipping

For each pixel of an image in the combiner, `sigma_clipping` masks the pixel if is more than a user-specified number of deviations from the central value of that pixel in the list of images.

The `sigma_clipping` method is very flexible: you can specify both the function for calculating the central value and the function for calculating the deviation. The default is to use the mean (ignoring any masked pixels) for the central value and the standard deviation (again ignoring any masked values) for the deviation.

You can mask pixels more than 5 standard deviations above or 2 standard deviations below the median with

```
>>> combiner.sigma_clipping(low_thresh=2, high_thresh=5, func=np.ma.median)
```

Note: Numpy masked median can be very slow in exactly the situation typically encountered in reducing ccd data: a cube of data in which one dimension (in the case the number of frames in the combiner) is much smaller than the number of pixels.

A much faster library for this case is bottleneck; a detailed example which uses bottleneck is at *bottleneck_example*.

Iterative clipping

To clip iteratively, continuing the clipping process until no more pixels are rejected, loop in the code calling the clipping method:

```
>>> old_n_masked = 0  # dummy value to make loop execute at least once
>>> new_n_masked = combiner.data_arr.mask.sum()
>>> while (new_n_masked > old_n_masked):
...     combiner.sigma_clipping(func=np.ma.median)
...     old_n_masked = new_n_masked
...     new_n_masked = combiner.data_arr.mask.sum()
```

Note that the default values for the high and low thresholds for rejection are 3 standard deviations.

6.2.2 Image combination

Image combination is straightforward; to combine by taking the average, excluding any pixels mapped by clipping:

```
>>> combined_average = combiner.average_combine()
```

Performing a median combination is also straightforward,

```
>>> combined_median = combiner.median_combine()  # can be slow, see below
```

The masked median function provided by numpy (and used by default in `median_combine`) can be very slow, so `median_combine` accepts an argument `median_func` for calculating the median instead. One fast alternative is provided by the bottleneck package; an example using it is at *bottleneck_example*.

6.2.3 With image scaling

In some circumstances it may be convenient to scale all images to some value before combining them. Do so by setting `scaling`:

```
>>> scaling_func = lambda arr: 1/np.ma.average(arr)
>>> combiner.scaling = scaling_func
>>> combined_average_scaled = combiner.average_combine()
```

This will normalize each image by its mean before combining (note that the underlying images are *not* scaled; scaling is only done as part of combining using `average_combine` or `median_combine`).

6.2.4 With image transformation

TBD

6.3 Reduction toolbox

Note: This is not intended to be an introduction to image reduction. While performing the steps presented here may be the correct way to reduce data in some cases, it is not correct in all cases.

6.3.1 Logging in ccdproc

All logging in ccdproc is done in the sense of recording the steps performed in image metadata. if you want to do logging in the python sense of the word please see those docs.

There are basically three logging options:

1. Implicit logging: No setup or keywords needed, each of the functions below adds a note to the metadata when it is performed.
2. Explicit logging: You can specify what information is added to the metadata using the add_keyword argument for any of the functions below.
3. No logging: If you prefer no logging be done you can "opt-out" by calling each function with add_keyword=None.

6.3.2 Gain correct and create variance

Uncertainty

An uncertainty can be calculated from your data with create_variance:

```
>>> from astropy import units as u
>>> import numpy as np
>>> import ccdproc
>>> img = np.random.normal(loc=10, scale=0.5, size=(100, 232))
>>> data = ccdproc.CCDData(img, unit=u.adu)
>>> data_with_variance = ccdproc.create_variance(data,
...                                              gain=1.5 * u.electron/u.adu,
...                                              readnoise=5 * u.electron)
>>> data_with_variance.header['exposure'] = 30.0  # for dark subtraction
```

The uncertainty, u_{ij}, at pixel $(i,\ j)$ with value p_{ij} is calculated as

$$u_{ij} = \left(g * p_{ij} + \sigma_{rn}^2\right)^{\frac{1}{2}},$$

where σ_{rn} is the read noise. Gain is only necessary when the image units are different than the units of the read noise, and is used only to calculate the uncertainty. The data itself is not scaled by this function.

As with all of the functions in ccdproc, *the input image is not modified*.

In the example above the new image data_with_variance has its uncertainty set.

Gain

To apply a gain to an image, do:

```
>>> gain_corrected = ccdproc.gain_correct(data_with_variance, 1.5*u.electron/u.adu)
```

The result gain_corrected has its data *and uncertainty* scaled by the gain and its unit updated.

There are several ways to provide the gain, among them as an astropy.units.Quantity, as in the example above, as a ccdproc.Keyword. See to documentation for gain_correct for details.

6.3.3 Clean image

There are two ways to clean an image of cosmic rays. One is to use clipping to create a mask for a stack of images, as described in *Image masks/clipping*.

The other is to replace, in a single image, each pixel that is several standard deviations from a central value in a region surrounding that pixel. The methods below describe how to do that.

LACosmic

The lacosmic technique identifies cosmic rays by identifying pixels based on a variation of the Laplacian edge detection. The algorithm is an implementation of the code describe in van Dokkum (2001) [1].

Use this technique with `cosmicray_lacosmic`:

```
>>> cr_cleaned = ccdproc.cosmicray_lacosmic(gain_corrected, threshold,
...                                         thresh=5, mbox=11, rbox=11,
...                                         gbox=5)
```

median

Another cosmic ray cleaning algorithm available in ccdproc is `cosmicray_median` that is analogous to iraf.imred.crutil.crmedian. This technique can be used with `ccdproc.cosmicray_median`:

```
>>> cr_cleaned = ccdproc.cosmicray_median(gain_corrected, threshold,
...                                       mbox=11, rbox=11, gbox=5)
```

Although `ccdproc` provides functions for identifying outlying pixels and for calculating the deviation of the background you are free to provide your own error image instead.

There is one additional argument, `gbox`, that specifies the size of the box, centered on a outlying pixel, in which pixel should be grown. The argument `rbox` specifies the size of the box used to calculate a median value if values for bad pixels should be replaced.

6.3.4 Subtract overscan and trim images

Note:

- Images reduced with `ccdproc` do **NOT** have to come from FITS files. The discussion below is intended to ease the transition from the indexing conventions used in FITS and IRAF to python indexing.
- No bounds checking is done when trimming arrays, so indexes that are too large are silently set to the upper bound of the array. This is because `numpy`, which provides the infrastructure for the arrays in `ccdproc` has this behavior.

Indexing: python and FITS

Overscan subtraction and image trimming are done with two separate functions. Both are straightforward to use once you are familiar with python's rules for array indexing; both have arguments that allow you to specify the part of the image you want in the FITS standard way. The difference between python and FITS indexing is that python starts

[1] van Dokkum, P; 2001, "Cosmic-Ray Rejection by Laplacian Edge Detection". The Publications of the Astronomical Society of the Pacific, Volume 113, Issue 789, pp. 1420-1427. doi: 10.1086/323894

indexes at 0, FITS starts at 1, and the order of the indexes is switched (FITS follows the FORTRAN convention for array ordering, python follows the C convention).

The examples below include both python-centric versions and FITS-centric versions to help illustrate the differences between the two.

Consider an image from a FITS file in which `NAXIS1=232` and `NAXIS2=100`, in which the last 32 columns along `NAXIS1` are overscan.

In FITS parlance, the overscan is described by the region `[201:232, 1:100]`.

If that image has been read into a python array `img` by `astropy.io.fits` then the overscan is `img[0:100, 200:232]` (or, more compactly `img[:, 200:]`), the starting value of the first index implicitly being zero, and the ending value for both indices implicitly the last index).

One aspect of python indexing may particularly surprising to newcomers: indexing goes up to *but not including* the end value. In `img[0:100, 200:232]` the end value of the first index is 99 and the second index is 231, both what you would expect given that python indexing starts at zero, not one.

Those transitioning from IRAF to ccdproc do not need to worry about this too much because the functions for overscan subtraction and image trimming both allow you to use the familiar `BIASSEC` and `TRIMSEC` conventions for specifying the overscan and region to be retained in a trim.

Overscan subtraction

To subtract the overscan in our image from a FITS file in which `NAXIS1=232` and `NAXIS2=100`, in which the last 32 columns along `NAXIS1` are overscan, use `subtract_overscan`:

```
>>> # python-style indexing first
>>> oscan_subtracted = ccdproc.subtract_overscan(cr_cleaned,
...                                              overscan=cr_cleaned[:, 200:],
...                                              overscan_axis=1)
>>> # FITS/IRAF-style indexing to accomplish the same thing
>>> oscan_subtracted = ccdproc.subtract_overscan(cr_cleaned,
...                                              fits_section='[201:232,1:100]',
...                                              overscan_axis=1)
```

Note well that the argument `overscan_axis` *always* follows the python convention for axis ordering. Since the order of the indexes in the `fits_section` get switched in the (internal) conversion to a python index, the overscan axis ends up being the *second* axis, which is numbered 1 in python zero-based numbering.

With the arguments in this example the overscan is averaged over the overscan columns (i.e. 2000 through 2031) and then subtracted row-by-row from the image. The `median` argument can be used to median combine instead.

This example is not very realistic: typically one wants to fit a low-order polynomial to the overscan region and subtract that fit:

```
>>> from astropy.modeling import models
>>> poly_model = models.Polynomial1D(1)  # one-term, i.e. constant
>>> oscan_subtracted = ccdproc.subtract_overscan(cr_cleaned,
...                                              overscan=cr_cleaned[:, 200:],
...                                              overscan_axis=1,
...                                              model=poly_model)
```

See the documentation for `astropy.modeling.polynomial` for more examples of the available models and for a description of creating your own model.

Trim an image

The overscan-subtracted image constructed above still contains the overscan portion. We are assuming came from a FITS file in which `NAXIS1=2032` and `NAXIS2=1000`, in which the last 32 columns along `NAXIS1` are overscan.

Trim it using `trim_image`,shown below in both python- style and FITS-style indexing:

```
>>> # FITS-style:
>>> trimmed = ccdproc.trim_image(oscan_subtracted,
...                              fits_section='[1:200, 1:100]')
>>> # python-style:
>>> trimmed = ccdproc.trim_image(oscan_subtracted[:, :200])
```

Note again that in python the order of indices is opposite that assumed in FITS format, that the last value in an index means "up to, but not including", and that a missing value implies either first or last value.

Those familiar with python may wonder what the point of `trim_image` is; it looks like simply indexing `oscan_subtracted` would accomplish the same thing. The only additional thing `trim_image` does is to make a copy of the image before trimming it.

Note: By default, python automatically reduces array indices that extend beyond the actual length of the array to the actual length. In practice, this means you can supply an invalid shape for, e.g. trimming, and an error will not be raised. To make this concrete, `ccdproc.trim_image(oscan_subtracted[:, :200000000])` will be treated as if you had put in the correct upper bound, `200`.

6.3.5 Subtract bias and dark

Both of the functions below propagate the uncertainties in the science and calibration images if either or both is defined.

Assume in this section that you have created a master bias image called `master_bias` and a master dark image called `master_dark` that *has been bias-subtracted* so that it can be scaled by exposure time if necessary.

Subtract the bias with `subtract_bias`:

```
>>> fake_bias_data = np.random.normal(size=trimmed.shape)  # just for illustration
>>> master_bias = ccdproc.CCDData(fake_bias_data,
...                               unit=u.electron,
...                               mask=np.zeros(trimmed.shape))
>>> bias_subtracted = ccdproc.subtract_bias(trimmed, master_bias)
```

There are several ways you can specify the exposure times of the dark and science images; see `subtract_dark` for a full description.

In the example below we assume there is a keyword `exposure` in the metadata of the trimmed image and the master dark and that the units of the exposure are seconds (note that you can instead explicitly provide these times).

To perform the dark subtraction use `subtract_dark`:

```
>>> master_dark = master_bias.multiply(0.1)  # just for illustration
>>> master_dark.header['exposure'] = 15.0
>>> dark_subtracted = ccdproc.subtract_dark(bias_subtracted, master_dark,
...                                         exposure_time='exposure',
...                                         exposure_unit=u.second,
...                                         scale=True)
```

Note that scaling of the dark is not done by default; use `scale=True` to scale.

6.3.6 Correct flat

Given a flat frame called `master_flat`, use `flat_correct` to perform this calibration:

```
>>> fake_flat_data = np.random.normal(loc=1.0, scale=0.05, size=trimmed.shape)
>>> master_flat = ccdproc.CCDData(fake_flat_data, unit=u.electron)
>>> reduced_image = ccdproc.flat_correct(dark_subtracted, master_flat)
```

As with the additive calibrations, uncertainty is propagated in the division.

The flat is scaled by the mean of `master_flat` before dividing.

If desired, you can specify a minimum value the flat can have (e.g. to prevent division by zero). Any pixels in the flat whose value is less than `min_value` are replaced with `min_value`):

```
>>> reduced_image = ccdproc.flat_correct(dark_subtracted, master_flat,
...                                      min_value=0.9)
```

6.4 Reduction examples

Mostly still TBD, hopefully filled in with examples from users. There is one example ipython notebook.

CHAPTER 7

ccdproc Module

The ccdproc package is a collection of code that will be helpful in basic CCD processing. These steps will allow reduction of basic CCD data as either a stand-alone processing or as part of a pipeline.

7.1 Functions

background_deviation_box(data, bbox)	Determine the background deviation with a box size of bbox.
background_deviation_filter(data, bbox)	Determine the background deviation for each pixel from a box with size of bb
cosmicray_lacosmic(ccd[, error_image, ...])	Identify cosmic rays through the lacosmic technique.
cosmicray_median(ccd[, error_image, thresh, ...])	Identify cosmic rays through median technique.
create_deviation(ccd_data[, gain, ...])	Create a uncertainty frame.
flat_correct(ccd, flat[, min_value, add_keyword])	Correct the image for flat fielding.
gain_correct(ccd, gain[, gain_unit, add_keyword])	Correct the gain in the image.
rebin(ccd, newshape)	Rebin an array to have a new shape.
sigma_func(arr)	Robust method for calculating the deviation of an array.
subtract_bias(ccd, master[, add_keyword])	Subtract master bias from image.
subtract_dark(ccd, master[, dark_exposure, ...])	Subtract dark current from an image.
subtract_overscan(ccd[, overscan, ...])	Subtract the overscan region from an image.
test([package, test_path, args, plugins, ...])	Run the tests using py.test.
transform_image(ccd, transform_func[, ...])	Transform the image Using the function specified by transform_func, the trans
trim_image(ccd[, fits_section, add_keyword])	Trim the image to the dimensions indicated.

7.1.1 background_deviation_box

ccdproc.**background_deviation_box**(*data*, *bbox*)

Determine the background deviation with a box size of bbox. The algorithm steps through the image and calculates the deviation within each box. It returns an array with the pixels in each box filled with the deviation value.

Parameters

data : ndarray or MaskedArray

Data to measure background deviation

bbox : int

Box size for calculating background deviation

Returns

background : ndarray or MaskedArray

An array with the measured background deviation in each pixel

Raises

ValueError

A value error is raised if bbox is less than 1

7.1.2 background_deviation_filter

ccdproc.**background_deviation_filter**(*data*, *bbox*)

Determine the background deviation for each pixel from a box with size of bbox.

Parameters

data : ndarray

Data to measure background deviation

bbox : int

Box size for calculating background deviation

Returns

background : ndarray or MaskedArray

An array with the measured background deviation in each pixel

Raises

ValueError

A value error is raised if bbox is less than 1

7.1.3 cosmicray_lacosmic

ccdproc.**cosmicray_lacosmic**(*ccd*, *error_image=None*, *thresh=5*, *fthresh=5*, *gthresh=1.5*, *b_factor=2*, *mbox=5*, *min_limit=0.01*, *gbox=0*, *rbox=0*, *f_conv=None*)

Identify cosmic rays through the lacosmic technique. The lacosmic technique identifies cosmic rays by identifying pixels based on a variation of the Laplacian edge detection. The algorithm is an implementation of the code describe in van Dokkum (2001) [R1].

Parameters

ccd: ‘~ccdproc.CCDData‘ or ‘numpy.ndarray‘

Data to have cosmic ray cleaned

error_image : numpy.ndarray

Error level in the image. It should have the same shape as data as data. This is the same as the noise array in lacosmic.cl

thresh : float

Threshold for detecting cosmic rays. This is the same as sigmaclip in lacosmic.cl

fthresh : float

Threshold for differentiating compact sources from cosmic rays. This is the same as objlim in lacosmic.cl

gthresh : float

Threshold for checking for surrounding cosmic rays from source. This is the same as sigclip*sigfrac from lacosmic.cl

b_factor : int

Factor for block replication

mbox : int

Median box for detecting cosmic rays

min_limit: float

Minimum value for all pixels so as to avoid division by zero errors

gbox : int

Box size to grow cosmic rays. If zero, no growing will be done.

rbox : int

Median box for calculating replacement values. If zero, no pixels will be replaced.

f_conv: 'numpy.ndarray', optional

Convolution kernal for detecting edges. The default kernel is `np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])`.

{log}

Returns

nccd : `CCDData` or `ndarray`

An object of the same type as ccd is returned. If it is a `CCDData`, the mask attribute will also be updated with areas identified with cosmic rays masked.

nccd : `ndarray`

If an `ndarray` is provided as ccd, a boolean ndarray with the cosmic rays identified will also be returned.

Notes

Implementation of the cosmic ray identification L.A.Cosmic: http://www.astro.yale.edu/dokkum/lacosmic/

References

[R1]

Examples

1. Given an numpy.ndarray object, the syntax for running cosmicrar_lacosmic would be:

```
>>> newdata, mask = cosmicray_lacosmic(data, error_image=error_image,
                                       thresh=5, mbox=11, rbox=11,
                                       gbox=5)
```

where the error is an array that is the same shape as data but includes the pixel error. This would return a data array, newdata, with the bad pixels replaced by the local median from a box of 11 pixels; and it would return a mask indicating the bad pixels.

2. Given an `CCDData` object with an uncertainty frame, the syntax for running cosmicrar_lacosmic would be:

```
>>> newccd = cosmicray_lacosmic(ccd, thresh=5, mbox=11, rbox=11, gbox=5)
```

The newccd object will have bad pixels in its data array replace and the mask of the object will be created if it did not previously exist or be updated with the detected cosmic rays.

7.1.4 cosmicray_median

ccdproc.**cosmicray_median**(*ccd*, *error_image=None*, *thresh=5*, *mbox=11*, *gbox=0*, *rbox=0*)

Identify cosmic rays through median technique. The median technique identifies cosmic rays by identifying pixels by subtracting a median image from the initial data array.

Parameters

ccd : CCDData or numpy.ndarray or numpy.MaskedArary

Data to have cosmic ray cleaned

thresh : float

Threshold for detecting cosmic rays

error_image : None, float, or ndarray

Error level. If None, the task will use the standard deviation of the data. If an ndarray, it should have the same shape as data.

mbox : int

Median box for detecting cosmic rays

gbox : int

Box size to grow cosmic rays. If zero, no growing will be done.

rbox : int

Median box for calculating replacement values. If zero, no pixels will be replaced.

{log}

Returns

nccd : CCDData or ndarray

An object of the same type as ccd is returned. If it is a CCDData, the mask attribute will also be updated with areas identified with cosmic rays masked.

nccd : ndarray

If an ndarray is provided as ccd, a boolean ndarray with the cosmic rays identified will also be returned.

Notes

Similar implementation to crmedian in iraf.imred.crutil.crmedian

Examples

1.Given an numpy.ndarray object, the syntax for running cosmicray_median would be:

```
>>> newdata, mask = cosmicray_median(data, error_image=error,
                                     thresh=5, mbox=11, rbox=11, gbox=5)
```

where error is an array that is the same shape as data but includes the pixel error. This would return a data array, newdata, with the bad pixels replaced by the local median from a box of 11 pixels; and it would return a mask indicating the bad pixels.

2. Given an `CCDData` object with an uncertainty frame, the syntax for running cosmicray_median would be:

```
>>> newccd = cosmicray_median(ccd, thresh=5, mbox=11, rbox=11, gbox=5)
```

The newccd object will have bad pixels in its data array replace and the mask of the object will be created if it did not previously exist or be updated with the detected cosmic rays.

7.1.5 create_deviation

ccdproc.**create_deviation**(*ccd_data*, *gain=None*, *readnoise=None*, *add_keyword=None*)

Create a uncertainty frame. The function will update the uncertainty plane which gives the standard deviation for the data. Gain is used in this function only to scale the data in constructing the deviation; the data is not scaled.

Parameters

ccd_data : `CCDData`

Data whose deviation will be calculated.

gain : `Quantity`, optional

Gain of the CCD; necessary only if `ccd_data` and `readnoise` are not in the same units. In that case, the units of `gain` should be those that convert `ccd_data.data` to the same units as `readnoise`.

readnoise : `Quantity`

Read noise per pixel.

add_keyword : str, `Keyword` or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

ccd : `CCDData`

CCDData object with uncertainty created; uncertainty is in the same units as the data in the parameter `ccd_data`.

Raises

UnitsError

Raised if `readnoise` units are not equal to product of `gain` and `ccd_data` units.

7.1.6 flat_correct

ccdproc.**flat_correct**(*ccd*, *flat*, *min_value=None*, *add_keyword=None*)

Correct the image for flat fielding.

The flat field image is normalized by its mean before flat correcting.

Parameters

ccd : `CCDData`

Data to be flatfield corrected

flat : CCDData

Flatfield to apply to the data

min_value : None or float

Minimum value for flat field. The value can either be None and no minimum value is applied to the flat or specified by a float which will replace all values in the flat by the min_value.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

ccd : CCDData

CCDData object with flat corrected

7.1.7 gain_correct

ccdproc.**gain_correct**(*ccd*, *gain*, *gain_unit=None*, *add_keyword=None*)

Correct the gain in the image.

Parameters

ccd : CCDData

Data to have gain corrected

gain : Quantity or Keyword

gain value for the image expressed in electrons per adu

gain_unit : Unit, optional

Unit for the gain; used only if gain itself does not provide units.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

result : CCDData

CCDData object with gain corrected

7.1.8 rebin

ccdproc.**rebin**(*ccd*, *newshape*)

Rebin an array to have a new shape.

Parameters

data : CCDData or ndarray

Data to rebin

newshape : tuple

Tuple containing the new shape for the array

Returns

output : CCDData or ndarray

An array with the new shape. It will have the same type as the input object.

Raises

TypeError

A type error is raised if data is not an numpy.ndarray or CCDData

ValueError

A value error is raised if the dimenisions of new shape is not equal to data

Notes

This is based on the scipy cookbook for rebinning: http://wiki.scipy.org/Cookbook/Rebinning

If rebinning a CCDData object to a smaller shape, the masking and uncertainty are not handled correctly.

Examples

Given an array that is 100x100,

```
>>> import numpy as np
>>> from astropy import units as u
>>> arr1 = CCDData(np.ones([10, 10]), unit=u.adu)
```

the syntax for rebinning an array to a shape of (20,20) is

```
>>> rebinned = rebin(arr1, (20,20))
```

7.1.9 sigma_func

ccdproc.**sigma_func**(*arr*)

Robust method for calculating the deviation of an array. sigma_func uses the median absolute deviation to determine the standard deviation.

Parameters

arr : CCDData or ndarray

Array whose deviation is to be calculated.

Returns

float

standard deviation of array

7.1.10 subtract_bias

ccdproc.**subtract_bias**(*ccd*, *master*, *add_keyword=None*)

Subtract master bias from image.

Parameters

ccd : CCDData

Image from which bias will be subtracted

master : CCDData

Master image to be subtracted from ccd

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

result : CCDData

CCDData object with bias subtracted

7.1.11 subtract_dark

ccdproc.**subtract_dark**(*ccd*, *master*, *dark_exposure=None*, *data_exposure=None*, *exposure_time=None*, *exposure_unit=None*, *scale=False*, *add_keyword=None*)

Subtract dark current from an image.

Parameters

ccd : CCDData

Image from which dark will be subtracted

master : CCDData

Dark image

dark_exposure : Quantity

Exposure time of the dark image; if specified, must also provided data_exposure.

data_exposure : Quantity

Exposure time of the science image; if specified, must also provided dark_exposure.

exposure_time : str or Keyword

Name of key in image metadata that contains exposure time.

exposure_unit : Unit

Unit of the exposure time if the value in the meta data does not include a unit.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

result : CCDData

Dark-subtracted image

7.1.12 subtract_overscan

ccdproc.**subtract_overscan**(*ccd*, *overscan=None*, *overscan_axis=1*, *fits_section=None*, *median=False*, *model=None*, *add_keyword=None*)

Subtract the overscan region from an image.

Parameters

ccd : CCDData

Data to have overscan frame corrected

overscan : CCDData

Slice from `ccd` that contains the overscan. Must provide either this argument or `fits_section`, but not both.

overscan_axis : 0 or 1, optional

Axis along which overscan should combined with mean or median. Axis numbering follows the *python* convention for ordering, so 0 is the first axis and 1 is the second axis.

fits_section : str

Region of `ccd` from which the overscan is extracted, using the FITS conventions for index order and index start. See Notes and Examples below. Must provide either this argument or `overscan`, but not both.

median : bool, optional

If true, takes the median of each line. Otherwise, uses the mean

model : Model, optional

Model to fit to the data. If None, returns the values calculated by the median or the mean.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

ccd : CCDData

CCDData object with overscan subtracted

Raises

TypeError

A TypeError is raised if either `ccd` or `overscan` are not the correct objects.

Notes

The format of the `fits_section` string follow the rules for slices that are consistent with the FITS standard (v3) and IRAF usage of keywords like TRIMSEC and BIASSEC. Its indexes are one-based, instead of the python-standard zero-based, and the first index is the one that increases most rapidly as you move through the array in memory order, opposite the python ordering.

The 'fits_section' argument is provided as a convenience for those who are processing files that contain TRIMSEC and BIASSEC. The preferred, more pythonic, way of specifying the overscan is to do it by indexing the data array directly with the `overscan` argument.

Examples

```
>>> import numpy as np
>>> from astropy import units as u
>>> arr1 = CCDData(np.ones([100, 100]), unit=u.adu)
```

The statement below uses all rows of columns 90 through 99 as the overscan.

```
>>> no_scan = subtract_overscan(arr1, overscan=arr1[:, 90:100])
>>> assert (no_scan.data == 0).all()
```

This statement does the same as the above, but with a FITS-style section.

```
>>> no_scan = subtract_overscan(arr1, fits_section='[91:100, :]')
>>> assert (no_scan.data == 0).all()
```

Spaces are stripped out of the `fits_section` string.

7.1.13 test

ccdproc.**test**(*package=None*, *test_path=None*, *args=None*, *plugins=None*, *verbose=False*, *pastebin=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. Works only on platforms with a working `lsof` command.

parallel : int, optional

When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use the 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.

7.1.14 transform_image

ccdproc.**transform_image**(*ccd*, *transform_func*, *add_keyword=None*, ***kwargs*)

Transform the image

Using the function specified by transform_func, the transform will be applied to data, uncertainty, and mask in ccd.

Parameters

ccd : `CCDData`

Data to be flatfield corrected

transform_func : function

Function to be used to transform the data

kwargs: dict

Dictionary of arguments to be used by the transform_func.

add_keyword : str, `Keyword` or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

ccd : CCDData

A transformed CCDData object

Notes

At this time, transform will be applied to the uncertainy data but it will only transform the data. This will not properly handle uncertainties that arise due to correlation between the pixels.

These should only be geometric transformations of the images. Other methods should be used if the units of ccd need to be changed.

Examples

Given an array that is 100x100,

```
>>> import numpy as np
>>> from astropy import units as u
>>> arr1 = CCDData(np.ones([100, 100]), unit=u.adu)
```

the syntax for transforming the array using scipy.ndimage.interpolation.shift

```
>>> from scipy.ndimage.interpolation import shift
>>> transformed = transform(arr1, shift, shift=(5.5, 8.1))
```

7.1.15 trim_image

ccdproc.**trim_image**(*ccd*, *fits_section=None*, *add_keyword=None*)

Trim the image to the dimensions indicated.

Parameters

ccd : CCDData

CCD image to be trimmed, sliced if desired.

fits_section : str

Region of ccd from which the overscan is extracted; see subtract_overscan() for details.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to None to completely disable logging. Default is to add a dictionary with a single item: the key is the name of this function and the value is a string containing the arguments the function was called with, except the value of this argument.

Returns

trimmed_ccd : CCDData

Trimmed image.

Examples

Given an array that is 100x100,

```
>>> import numpy as np
>>> from astropy import units as u
>>> arr1 = CCDData(np.ones([100, 100]), unit=u.adu)
```

the syntax for trimming this to keep all of the first index but only the first 90 rows of the second index is

```
>>> trimmed = trim_image(arr1[:, :90])
>>> trimmed.shape
(100, 90)
>>> trimmed.data[0, 0] = 2
>>> arr1.data[0, 0]
1.0
```

This both trims *and makes a copy* of the image.

Indexing the image directly does *not* do the same thing, quite:

```
>>> not_really_trimmed = arr1[:, :90]
>>> not_really_trimmed.data[0, 0] = 2
>>> arr1.data[0, 0]
2.0
```

In this case, `not_really_trimmed` is a view of the underlying array `arr1`, not a copy.

7.2 Classes

CCDData(*args, **kwd)	A class describing basic CCD data The CCDData class is based on the NDData object and includes a
Combiner(ccd_list)	A class for combining CCDData objects.
Keyword(name[, unit, value])	

7.2.1 CCDData

class ccdproc.**CCDData**(**args*, ***kwd*)

Bases: astropy.nddata.nddata.NDData

A class describing basic CCD data

The CCDData class is based on the NDData object and includes a data array, uncertainty frame, mask frame, meta data, units, and WCS information for a single CCD image.

Parameters

data : ndarray or NDData

The actual data contained in this NDData object. Note that this will always be copies by *reference* , so you should make copy the data before passing it in if that's the desired behavior.

uncertainty : StdDevUncertainty or ndarray, optional

Uncertainties on the data.

mask : ndarray, optional

Mask for the data, given as a boolean Numpy array with a shape matching that of the data. The values must be `False` where the data is *valid* and `True` when it is not (like Numpy masked arrays). If `data` is a numpy masked array, providing `mask` here will causes the mask from the masked array to be ignored.

flags : `ndarray` or `FlagCollection`, optional

Flags giving information about each pixel. These can be specified either as a Numpy array of any type with a shape matching that of the data, or as a `FlagCollection` instance which has a shape matching that of the data.

wcs : undefined, optional

WCS-object containing the world coordinate system for the data.

> **Warning:** This is not yet defind because the discussion of how best to represent this class's WCS system generically is still under consideration. For now just leave it as None

meta : `dict`-like object, optional

Metadata for this object. "Metadata" here means all information that is included with this object but not part of any other attribute of this particular object. e.g., creation date, unique identifier, simulation parameters, exposure time, telescope name, etc.

unit : `Unit` instance or str, optional

The units of the data.

Raises

ValueError

If the `uncertainty` or `mask` inputs cannot be broadcast (e.g., match shape) onto `data`.

Notes

`NDData` objects can be easily converted to a regular Numpy array using `numpy.asarray`

For example:

```
>>> from astropy.nddata import NDData
>>> import numpy as np
>>> x = NDData([1,2,3])
>>> np.asarray(x)
array([1, 2, 3])
```

If the `NDData` object has a `mask`, `numpy.asarray` will return a Numpy masked array.

This is useful, for example, when plotting a 2D image using matplotlib:

```
>>> from astropy.nddata import NDData
>>> from matplotlib import pyplot as plt
>>> x = NDData([[1,2,3], [4,5,6]])
>>> plt.imshow(x)
```

Attributes Summary

`header`	

Continued on next page

Table 7.3 – continued from previous page

meta
uncertainty

Methods Summary

add(other)	
copy()	Return a copy of the CCDData object.
divide(other)	
multiply(other)	
subtract(other)	
to_hdu()	Creates an HDUList object from a CCDData object.

Attributes Documentation

header

meta

uncertainty

Methods Documentation

add(*other*)

copy()
Return a copy of the CCDData object.

divide(*other*)

multiply(*other*)

subtract(*other*)

to_hdu()
Creates an HDUList object from a CCDData object.

Returns
hdulist : astropy.io.fits.HDUList object

Raises
ValueError

Multi-Exenstion FITS files are not supported

7.2.2 Combiner

class ccdproc.**Combiner**(*ccd_list*)

Bases: object

A class for combining CCDData objects.

The Combiner class is used to combine together CCDData objects including the method for combining the data, rejecting outlying data, and weighting used for combining frames

Parameters

ccd_list : list

A list of CCDData objects that will be combined together.

Raises

TypeError

If the ccd_list are not CCDData objects, have different units, or are different shapes

Notes

The following is an example of combining together different CCDData objects:

```
>>> from combiner import combiner
>>> c = combiner([ccddata1, cccdata2, ccddata3])
>>> ccdall = c.median_combine()
```

Attributes Summary

scaling	Scaling factor used in combining images.
weights	

Methods Summary

average_combine([scale_func, scale_to])	Average combine together a set of arrays.
median_combine([median_func])	Median combine a set of arrays.
minmax_clipping([min_clip, max_clip])	Mask all pixels that are below min_clip or above max_clip.
sigma_clipping([low_thresh, high_thresh, ...])	Pixels will be rejected if they have deviations greater than those set by the threshol

Attributes Documentation

scaling

Scaling factor used in combining images.

Parameters

scale : function or array-like or None, optional

Images are multiplied by scaling prior to combining them. Scaling may be either a function, which will be applied to each image to determine the scaling factor, or a list or array whose length is the number of images in the Combiner. Default is None.

weights

Methods Documentation

average_combine(*scale_func=None*, *scale_to=1.0*)

Average combine together a set of arrays. A CCDData object is returned with the data property set to the average of the arrays. If the data was masked or any data have been rejected, those pixels will not be included in the median. A mask will be returned, and if a pixel has been rejected in all images, it will be masked. The uncertainty of the combined image is set by the standard deviation of the input images.

Returns

combined_image: CCDData

CCDData object based on the combined input of CCDData objects.

median_combine(*median_func=<function median at 0x7f32f42bd6e0>*)

Median combine a set of arrays.

A CCDData object is returned with the data property set to the median of the arrays. If the data was masked or any data have been rejected, those pixels will not be included in the median. A mask will be returned, and if a pixel has been rejected in all images, it will be masked. The uncertainty of the combined image is set by 1.4826 times the median absolute deviation of all input images.

Parameters

median_func : function, optional

Function that calculates median of a `numpy.ma.masked_array`. Default is to use `np.ma.median` to calculate median.

Returns

combined_image: CCDData

CCDData object based on the combined input of CCDData objects.

Warning: The uncertainty currently calculated using the median absolute deviation does not account for rejected pixels

minmax_clipping(*min_clip=None*, *max_clip=None*)

Mask all pixels that are below min_clip or above max_clip.

Parameters

min_clip : None or float

If specified, all pixels with values below min_clip will be masked

max_clip : None or float

If specified, all pixels with values above min_clip will be masked

sigma_clipping(*low_thresh=3*, *high_thresh=3*, *func=<numpy.ma.core._frommethod instance at 0x7f32f42bc170>*, *dev_func=<numpy.ma.core._frommethod instance at 0x7f32f42bc5a8>*)

Pixels will be rejected if they have deviations greater than those

set by the threshold values. The algorithm will first calculated a baseline value using the function specified in func and deviation based on dev_func and the input data array. Any pixel with a deviation from the baseline value greater than that set by high_thresh or lower than that set by low_thresh will be rejected.

Parameters

low_thresh : positive float or None

Threshold for rejecting pixels that deviate below the baseline value. If negative value, then will be convert to a positive value. If None, no rejection will be done based on low_thresh.

high_thresh : positive float or None

Threshold for rejecting pixels that deviate above the baseline value. If None, no rejection will be done based on high_thresh.

func : function

Function for calculating the baseline values (i.e. mean or median). This should be a function that can handle numpy.ma.core.MaskedArray objects.

dev_func : function

Function for calculating the deviation from the baseline value (i.e. std). This should be a function that can handle numpy.ma.core.MaskedArray objects.

7.2.3 Keyword

class ccdproc.**Keyword**(*name*, *unit=None*, *value=None*)
Bases: object

Attributes Summary

name	
unit	
value	

Methods Summary

value_from(header)	Set value of keyword from FITS header

Attributes Documentation

name

unit

value

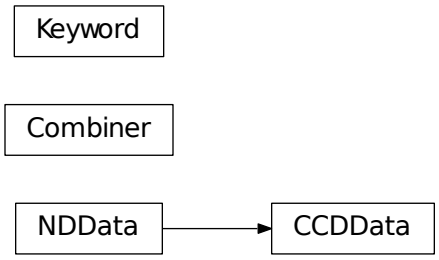
Methods Documentation

value_from(*header*)
Set value of keyword from FITS header

Parameters
header : astropy.io.fits.Header

FITS header containing a value for this keyword

7.3 Class Inheritance Diagram



[R1] van Dokkum, P; 2001, “Cosmic-Ray Rejection by Laplacian Edge Detection”. The Publications of the Astronomical Society of the Pacific, Volume 113, Issue 789, pp. 1420-1427. doi: 10.1086/323894

C