ccdproc Documentation

Release 1.3.0

Steve Crawford and Matt Craig

Nov 01, 2017

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 propagation and bad pixel tracking throughout the reduction process.

Part I

Documentation

The documentation for this package is here:

CHAPTER 1

Installation

1.1 Requirements

Ccdproc has the following requirements:

- Astropy v1.0 or later
- NumPy
- SciPy
- scikit-image
- astroscrappy
- reproject

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

1.2 Installing ccdproc

1.2.1 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.

1.2.2 Using conda

To install ccdproc with anaconda, simply run:

```
conda install -c astropy ccdproc
```

1.3 Building from source

1.3.1 Obtaining the source packages

Source packages

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

Development repository

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

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

1.3.2 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
```

1.3.3 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
```

CHAPTER 2

CCD Data reduction (ccdproc)

2.1 Introduction

Note: ccdproc works only with astropy version 1.0 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 function for reprojecting an image onto another WCS, useful for stacking science images. The actual reprojection is done by the reproject package.
- A class for combining and/or clipping images, Combiner, and associated functions.
- A class, ImageFileCollection, for working with a directory of images.

2.2 Getting Started

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 = astropy.nddata.CCDData.read('my_image.fits', unit="electron")` (the `electron` unit is defined as part of `ccdproc`).

The metadata of a CCDData object may be any dictionary-like object, including a FITS header. When a CCDData object is initialized from FITS file its metadata is a FITS header.

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)
```

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

Every function returns a *copy* of the data with the operation performed.

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:

```
>>> dark_sub_gained = ccdproc.gain_correct(dark_sub, 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'}
>>> dark_sub_gained = ccdproc.gain_correct(dark_sub,
...                                        1.5 * u.photon/u.adu,
...                                        add_keyword=my_log)
```

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).

The function wcs_project allows you to reproject an image onto a different WCS.

To make applying the same operations to a set of files in a directory easier, use an ImageFileCollection. It constructs, given a directory, a Table containing the values of user-selected keywords in the directory. It also provides methods for iterating over the files. The example below was used to find an image in which the sky background was high for use in a talk:

```
>>> from __future__ import division, print_function
>>> from ccdproc import ImageFileCollection
>>> import numpy as np
>>> from glob import glob
>>> dirs = glob('/Users/mcraig/Documents/Data/feder-images/fixed_headers/20*-??-??')
```

```
>>> for d in dirs:
...     print(d)
...     ic = ImageFileCollection(d, keywords='*')
...     for data, fname in ic.data(imagetyp='LIGHT', return_fname=True):
...         if data.mean() > 4000.:
...             print(fname)
```

2.3 Using ccdproc

2.3.1 CCDData class

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.

In addition, the user can specify the extension in a FITS file to use:

```
>>> ccd = ccdproc.CCDData.read('my_file.fits', hdu=1, unit="adu")
```

If hdu is not specified, it will assume the data is in the primary extension. If there is no data in the primary extension, the first extension with data will be used.

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
```

Whether the metadata is case sensitive or not depends on how it is initialized. A FITS header, for example, is not case sensitive, but a python dictionary is.

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
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`.

WCS

The wcs attribute of CCDData object can be set two ways.

- If the CCDData object is created from a FITS file that has WCS keywords in the header, the wcs attribute is set to a astropy.wcs.WCS object using the information in the FITS header.
- The WCS can also be provided when the CCDData object is constructed with the wcs argument.

Either way, the wcs attribute is kept up to date if the CCDData image is trimmed.

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_deviation(ccd, readnoise=5 * u.electron)
```

See *Gain correct and create deviation image* 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. [...]
```

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(...)
```

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 if the operation is between two CCDData objects.

```
>>> 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")
```

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.

Note: If two images have different WCS values, the wcs on the first `CCDData` object will be used for the resultant object.

2.3.2 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.

Image masks/clipping

There are currently three methods of clipping. None affect 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.

Extrema clipping

For each pixel position in the input arrays, the algorithm will mask the highest `nhigh` and lowest `nlow` pixel values. The resulting image will be a combination of `Nimages-nlow-nhigh` pixel values instead of the combination of `Nimages` worth of pixel values.

You can mask the lowest pixel value and the highest two pixel values with:

```
>>> combiner.clip_extrema(nlow=1, nhigh=2)
```

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.

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
```

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`).

With image transformation

Note: **Flux conservation** Whether flux is conserved in performing the reprojection depends on the method you use for reprojecting and the extent to which pixel area varies across the image. `wcs_project` rescales counts by the ratio of pixel area *of the pixel indicated by the keywords* `CRPIX` of the input and output images.

The reprojection methods available are described in detail in the documentation for the reproject project; consult those documents for details.

You should carefully check whether flux conservation provided in CCDPROC is adequate for your needs. Suggestions for improvement are welcome!

Align and then combine images based on World Coordinate System (WCS) information in the image headers in two steps.

First, reproject each image onto the same footprint using `wcs_project`. The example below assumes you have an image with WCS information and another image (or WCS) onto which you want to project your images:

```
>>> from ccdproc import wcs_project
>>> reprojected_image = wcs_project(input_image, target_wcs)
```

Repeat this for each of the images you want to combine, building up a list of reprojected images:

```
>>> reprojected = []
>>> for img in my_list_of_images:
...     new_image = wcs_project(img, target_wcs)
...     reprojected.append(new_image)
```

Then, combine the images as described above for any set of images:

```
>>> combiner = Combiner(reprojected)
>>> stacked_image = combiner.average_combine()
```

2.3.3 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.

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`.

Gain correct and create deviation image

Uncertainty

An uncertainty can be calculated from your data with `create_deviation`:

```
>>> 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_deviation = ccdproc.create_deviation(
...                           data, gain=1.5 * u.electron/u.adu,
...                           readnoise=5 * u.electron)
>>> data_with_deviation.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_deviation` has its uncertainty set.

Gain

To apply a gain to an image, do:

```
>>> gain_corrected = ccdproc.gain_correct(data_with_deviation, 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.

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] as implemented in [astroscrappy](https://github.com/astropy/astroscrappy)[2].

[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

[2] McCully, C., 2014, "Astro-SCRAPPY", https://github.com/astropy/astroscrappy

Use this technique with cosmicray_lacosmic:

```
>>> cr_cleaned = ccdproc.cosmicray_lacosmic(gain_corrected, sigclip=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, 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.

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 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. 200 through 231) 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=232 and NAXIS2=100, 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`.

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.

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)
```

Basic Processing

All of the basic processing steps can be accomplished in a single step using `ccd_process`. This step will call overscan correct, trim, gain correct, add a bad pixel mask, create an uncertainty frame, subtract the master bias, and flat-field the image. The unit of the master calibration frames must match that of the image *after* the gain, if any, is applied. In the example below, `img` has unit `adu`, but the master frames have unit `electron`. These can be run together as:

```
>>> ccd = ccdproc.CCDData(img, unit=u.adu)
>>> ccd.header['exposure'] = 30.0  # for dark subtraction
>>> nccd = ccdproc.ccd_process(ccd, oscan='[201:232,1:100]',
...                            trim='[1:200, 1:100]',
...                            error=True,
...                            gain=2.0*u.electron/u.adu,
...                            readnoise=5*u.electron,
...                            dark_frame=master_dark,
...                            exposure_key='exposure',
...                            exposure_unit=u.second,
...                            dark_scale=True,
...                            master_flat=master_flat)
```

Reprojecting onto a different image footprint

An image with coordinate information (WCS) can be reprojected onto a different image footprint. The underlying functionality is proved by the reproject project. Please see *With image transformation* for more details.

Data Quality Flags (Bitfields and bitmasks)

Some FITS files contain data quality flags or bitfield extension, while these are currently not supported as part of `CCDData` these can be loaded manually using `fits` and converted to regular (`numpy`-like) masks (with `bitfield_to_boolean_mask`) that are supported by many operations in `ccdproc`.

```
import numpy as np
from astropy.io import fits
from ccdproc import bitfield_to_boolean_mask, CCDData

fitsfilename = 'some_fits_file.fits'
bitfieldextension = extensionname_or_extensionnumber

# Read the data of the fits file as CCDData object
ccd = CCDData.read(fitsfilename)

# Open the file again (assuming the bitfield is saved in the same FITS file)
mask = bitfield_to_boolean_mask(fits.getdata(fitsfilename, bitfieldextension))

# Save the mask as "mask" attribute of the ccd
ccd.mask = mask
```

Filter and Convolution

There are several convolution and filter functions for `numpy.ndarray` across the scientific python packages:

- `scipy.ndimage.filters`, offers a variety of filters.
- `astropy.convolution`, offers some filters which also handle `NaN` values.
- `scikit-image.filters`, offers several filters which can also handle masks but are mostly limited to special data types (mostly unsigned integers).

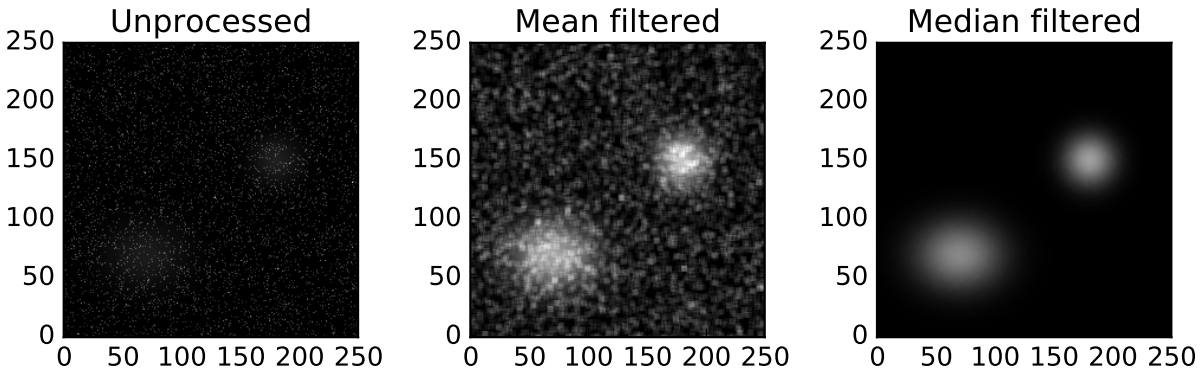
For convenience one of these is also accessible through the `ccdproc` package namespace which accepts `CCDData` objects and then also returns one:

- `median_filter`

Median Filter

The median filter is especially useful if the data contains sharp noise peaks which should be removed rather than propagated:

```
import ccdproc
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.functional_models import Gaussian2D
from astropy.utils.misc import NumpyRNGContext
from scipy.ndimage import uniform_filter

# Create some source signal
source = Gaussian2D(60, 70, 70, 20, 25)
data = source(*np.mgrid[0:250, 0:250])

# and another one
source = Gaussian2D(70, 150, 180, 15, 15)
data += source(*np.mgrid[0:250, 0:250])

# create some random signals
with NumpyRNGContext(1234):
    noise = np.random.exponential(40, (250, 250))
    # remove low signal
    noise[noise < 100] = 0
    data += noise

# create a CCD object based on the data
ccd = ccdproc.CCDData(data, unit='adu')

# Create some plots
fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
ax1.set_title('Unprocessed')
ax1.imshow(ccd, origin='lower', interpolation='none', cmap=plt.cm.gray)
ax2.set_title('Mean filtered')
ax2.imshow(uniform_filter(ccd.data, 5), origin='lower', interpolation='none', cmap=plt.cm.gray)
ax3.set_title('Median filtered')
ax3.imshow(ccdproc.median_filter(ccd, 5), origin='lower', interpolation='none', cmap=plt.cm.gray)
plt.tight_layout()
plt.show()
```



2.3.4 Image Management

Working with a directory of images

For the sake of argument all of the examples below assume you are working in a directory that contains FITS images.

The class `ImageFileCollection` is meant to make working with a directory of FITS images easier by allowing you select the files you act on based on the values of FITS keywords in their headers or based on Unix shell-style filename matching.

It is initialized with the name of a directory containing FITS images and a list of FITS keywords you want the `ImageFileCollection` to be aware of. An example initialization looks like:

```
>>> from ccdproc import ImageFileCollection
>>> keys = ['imagetyp', 'object', 'filter', 'exposure']
>>> ic1 = ImageFileCollection('.', keywords=keys) # only keep track of keys
```

You can use the wildcard `*` in place of a list to indicate you want the collection to use all keywords in the headers:

```
>>> ic_all = ImageFileCollection('.', keywords='*')
```

You can indicate filename patterns to include or exclude using Unix shell-style expressions. For example, to include all filenames that begin with `1d_` but not ones that include the word `bad`, you could do:

```
>>> ic_all = ImageFileCollection('.', glob_include='1d_*',
...                              glob_exclude='*bad*')
```

Most of the useful interaction with the image collection is via its `.summary` property, a `Table` of the value of each keyword for each file in the collection:

```
>>> ic1.summary.colnames
['file', 'imagetyp', 'object', 'filter', 'exposure']
>>> ic_all.summary.colnames
# long list of keyword names omitted
```

Note that the name of the file is automatically added to the table as a column named `file`.

Selecting files

Selecting the files that match a set of criteria, for example all images in the I band with exposure time less than 60 seconds you could do:

```
>>> matches = (ic1.summary['filter'] == 'I') & (ic1.summary['exposure'] < 60)
>>> my_files = ic1.summary['file'][matches]
```

The column `file` is added automatically when the image collection is created.

For more simple selection, when you just want files whose keywords exactly match particular values, say all I band images with exposure time of 30 seconds, there is a convenience method `.files_filtered`:

```
>>> my_files = ic1.files_filtered(filter='I', exposure=30)
```

The optional arguments to `files_filtered` are used to filter the list of files.

Sorting files

Sometimes it is useful to bring the files into a specific order, e.g. if you make a plot for each object you probably want all images of the same object next to each other. To do this, the images in a collection can be sorted with the `sort` method using the fits header keys in the same way you would sort a `Table`:

```
>>> ic1.sort(['object', 'filter'])
```

Iterating over hdus, headers, data, or ccds

Four methods are provided for iterating over the images in the collection, optionally filtered by keyword values.

For example, to iterate over all of the I band images with exposure of 30 seconds, performing some basic operation on the data (very contrived example):

```
>>> for hdu in ic1.hdus(imagetyp='LiGhT', filter='I', exposure=30):
...     hdu.header['exposure']
...     new_data = hdu.data - hdu.data.mean()
```

Note that the names of the arguments to `hdus` here are the names of FITS keywords in the collection and the values are the values of those keywords you want to select. Note also that string comparisons are not case sensitive.

The other iterators are `headers`, `data`, and `ccds`.

All of them have the option to also provide the file name in addition to the hdu (or header or data):

```
>>> for hdu, fname in ic1.hdus(return_fname=True,
...                            imagetyp='LiGhT', filter='I', exposure=30):
...     hdu.header['meansub'] = True
...     hdu.data = hdu.data - hdu.data.mean()
...     hdu.writeto(fname + '.new')
```

That last use case, doing something to several files and saving them somewhere afterwards, is common enough that the iterators provide arguments to automate it.

Automatic saving from the iterators

There are three ways of triggering automatic saving.

1. One is with the argument `save_with_name`; it adds the value of the argument to the file name between the original base name and extension. The example below has (almost) the same effect of the example above, subtracting the mean from each image and saving to a new file:

```
>>> for hdu in ic1.hdus(save_with_name='_new',
...                     imagetyp='LiGhT', filter='I', exposure=30):
...     hdu.header['meansub'] = True
...     hdu.data = hdu.data - hdu.data.mean()
```

It saves, in the `location` of the image collection, a new FITS file with the mean subtracted from the data, with `_new` added to the name; as an example, if one of the files iterated over was `intput001.fit` then a new file, in the same directory, called `input001_new.fit` would be created.

2. You can also provide the directory to which you want to save the files with `save_location`; note that you do not need to actually do anything to the hdu (or header or data) to cause the copy to be made. The example below copies all of the I band images with 30 second exposure from the original location to `other_dir`:

```
>>> for hdu in ic1.hdus(save_location='other_dir',
...                     imagetyp='LiGhT', filter='I', exposure=30):
...     pass
```

This option can be combined with the previous one to also give the files a new name.

3. Finally, if you want to live dangerously, you can overwrite the files in the same location with the `overwrite` argument; use it carefully because it preserves no backup. The example below replaces each of the I band images with 30 second exposure with a file that has had the mean subtracted:

```
>>> for hdu in ic1.hdus(overwrite=True,
...                     imagetyp='LiGhT', filter='I', exposure=30):
...     hdu.header['meansub'] = True
...     hdu.data = hdu.data - hdu.data.mean()
```

Note: This functionality is not currently available on Windows.

2.3.5 Reduction examples

Here are some examples and different repositories using `ccdproc`.

- ipython notebook
- WHT basic reductions
- pyhrs

- reduceccd
- astrolib

2.4 ccdproc Package

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.

2.4.1 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 bbox.
bitfield_to_boolean_mask(bitfield[, ...])	Convert an integer bit field to a boolean mask.
block_average(ccd, block_size)	Like block_reduce but with predefined func=np.mean.
block_reduce(ccd, block_size[, func])	Thin wrapper around astropy.nddata.block_reduce.
block_replicate(ccd, block_size[, conserve_sum])	Thin wrapper around astropy.nddata.block_replicate.
ccd_process(ccd[, oscan, trim, error, ...])	Perform basic processing on ccd data.
ccdmask(ratio[, findbadcolumns, byblocks, ...])	Uses method based on the IRAF ccdmask task to generate a mask based on the given input.
combine(img_list[, output_file, method, ...])	Convenience function for combining multiple images.
cosmicray_lacosmic(ccd[, sigclip, sigfrac, ...])	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, ...])	Correct the image for flat fielding.
gain_correct(ccd, gain[, gain_unit, add_keyword])	Correct the gain in the image.
median_filter(data, *args, **kwargs)	See scipy.ndimage.median_filter for arguments.
rebin(*args, **kwargs)	Deprecated since version 1.1.
sigma_func(arr[, axis])	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.
trim_image(ccd[, fits_section, add_keyword])	Trim the image to the dimensions indicated.
wcs_project(ccd, target_wcs[, target_shape, ...])	Given a CCDData image with WCS, project it onto a target WCS and return the reprojected data as a new CCDData image.

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 : numpy.ndarray or numpy.ma.MaskedArray

Data to measure background deviation.

bbox : int

Box size for calculating background deviation.

Returns

background : numpy.ndarray or numpy.ma.MaskedArray

An array with the measured background deviation in each pixel.

Raises

ValueError

A value error is raised if bbox is less than 1.

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 : numpy.ndarray

Data to measure background deviation.

bbox : int

Box size for calculating background deviation.

Returns

background : numpy.ndarray or numpy.ma.MaskedArray

An array with the measured background deviation in each pixel.

Raises

ValueError

A value error is raised if bbox is less than 1.

bitfield_to_boolean_mask

ccdproc.**bitfield_to_boolean_mask**(*bitfield*, *ignore_bits=0*, *flip_bits=None*)

Convert an integer bit field to a boolean mask.

Parameters

bitfield : numpy.ndarray of integer dtype

The array of bit flags.

ignore_bits : int, None or str, optional

The bits to ignore when converting the bitfield.

- If it's an integer it's binary representation is interpreted as the bits to ignore. `0` means that all bit flags are taken into account while a binary representation of all `1` means that all flags would be ignored.
- If it's `None` then all flags are ignored

- If it's a string then it must be a `,` or `+` separated string of integers that bits to ignore. If the string starts with an `~` the integers are interpreted as **the only flags** to take into account.

Default is `0`.

Returns

mask : `numpy.ndarray` of boolean dtype

The bitfield converted to a boolean mask that can be used for `numpy.ma.MaskedArray` or `CCDData`.

Examples

Bitfields (or data quality arrays) are integer arrays where the binary representation of the values indicates whether a specific flag is set or not. The convention is that a value of `0` represents a **good value** and a value that is `!= 0` represents a value that is in some (or multiple) ways considered a **bad value**. The `bitfield_to_boolean_mask` function can be used by default to create a boolean mask wherever any bit flag is set:

```
>>> import ccdproc
>>> import numpy as np
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8))
array([False,  True,  True,  True,  True,  True,  True,  True], dtype=bool)
```

To ignore all bit flags `ignore_bits=None` can be used:

```
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits=None)
array([False, False, False, False, False, False, False, False], dtype=bool)
```

To ignore only specific bit flags one can use a `list` of bits flags to ignore:

```
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits=[1, 4])
array([False, False,  True,  True, False, False,  True,  True], dtype=bool)
```

There are some equivalent ways:

```
>>> # pass in the sum of the "ignore_bits" directly
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits=5)  # 1 + 4
array([False, False,  True,  True, False, False,  True,  True], dtype=bool)
>>> # use a comma seperated string of integers
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits='1, 4')
array([False, False,  True,  True, False, False,  True,  True], dtype=bool)
>>> # use a + seperated string of integers
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits='1+4')
array([False, False,  True,  True, False, False,  True,  True], dtype=bool)
```

Instead of directly specifying the **bits flags to ignore** one can also pass in the **only bits that shouldn't be ignored** by prepending a `~` to the string of `ignore_bits` (or if it's not a string in `ignore_bits` one can set `flip_bits=True`):

```
>>> # ignore all bit flags except the one for 2.
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits='~(2)')
array([False, False,  True,  True, False, False,  True,  True], dtype=bool)
>>> # ignore all bit flags except the one for 1, 8 and 32.
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits='~(1, 8, 32)')
array([False,  True, False,  True, False,  True, False,  True], dtype=bool)
```

```
>>> # Equivalent for a list using flip_bits.
>>> ccdproc.bitfield_to_boolean_mask(np.arange(8), ignore_bits=[1, 8, 32], flip_bits=True)
array([False,  True, False,  True, False,  True, False,  True], dtype=bool)
```

block_average

ccdproc.**block_average**(*ccd*, *block_size*)

Like block_reduce but with predefined func=np.mean.

block_reduce

ccdproc.**block_reduce**(*ccd*, *block_size*, *func=<function sum>*)

Thin wrapper around astropy.nddata.block_reduce. Downsample a data array by applying a function to local blocks.

If data is not perfectly divisible by block_size along a given axis then the data will be trimmed (from the end) along that axis.

Parameters

data : array_like

The data to be resampled.

block_size : int or array_like (int)

The integer block size along each axis. If block_size is a scalar and data has more than one dimension, then block_size will be used for for every axis.

func : callable, optional

The method to use to downsample the data. Must be a callable that takes in a ndarray along with an axis keyword, which defines the axis along which the function is applied. The default is sum, which provides block summation (and conserves the data sum).

Returns

output : array-like

The resampled data.

Examples

```
>>> import numpy as np
>>> from astropy.nddata.utils import block_reduce
>>> data = np.arange(16).reshape(4, 4)
>>> block_reduce(data, 2)
array([[10, 18],
       [42, 50]])
```

```
>>> block_reduce(data, 2, func=np.mean)
array([[  2.5,   4.5],
       [ 10.5,  12.5]])
```

block_replicate

ccdproc.**block_replicate**(*ccd*, *block_size*, *conserve_sum=True*)

Thin wrapper around `astropy.nddata.block_replicate`. Upsample a data array by block replication.

Parameters

data : array_like

The data to be block replicated.

block_size : int or array_like (int)

The integer block size along each axis. If `block_size` is a scalar and `data` has more than one dimension, then `block_size` will be used for for every axis.

conserve_sum : bool, optional

If `True` (the default) then the sum of the output block-replicated data will equal the sum of the input `data`.

Returns

output : array_like

The block-replicated data.

Examples

```
>>> import numpy as np
>>> from astropy.nddata.utils import block_replicate
>>> data = np.array([[0., 1.], [2., 3.]])
>>> block_replicate(data, 2)
array([[ 0.  ,  0.  ,  0.25,  0.25],
       [ 0.  ,  0.  ,  0.25,  0.25],
       [ 0.5 ,  0.5 ,  0.75,  0.75],
       [ 0.5 ,  0.5 ,  0.75,  0.75]])
```

```
>>> block_replicate(data, 2, conserve_sum=False)
array([[ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  1.,  1.],
       [ 2.,  2.,  3.,  3.],
       [ 2.,  2.,  3.,  3.]])
```

ccd_process

ccdproc.**ccd_process**(*ccd*, *oscan=None*, *trim=None*, *error=False*, *master_bias=None*, *dark_frame=None*, *master_flat=None*, *bad_pixel_mask=None*, *gain=None*, *readnoise=None*, *oscan_median=True*, *oscan_model=None*, *min_value=None*, *dark_exposure=None*, *data_exposure=None*, *exposure_key=None*, *exposure_unit=None*, *dark_scale=False*, *gain_corrected=True*, *add_keyword=True*)

Perform basic processing on ccd data.

The following steps can be included:

- overscan correction (`subtract_overscan()`)
- trimming of the image (`trim_image()`)
- create deviation frame (`create_deviation()`)

- gain correction (gain_correct())
- add a mask to the data
- subtraction of master bias (subtract_bias())
- subtraction of a dark frame (subtract_dark())
- correction of flat field (flat_correct())

The task returns a processed CCDData object.

Parameters

ccd : CCDData

Frame to be reduced.

oscan : CCDData, str or None, optional

For no overscan correction, set to None. Otherwise provide a region of ccd from which the overscan is extracted, using the FITS conventions for index order and index start, or a slice from ccd that contains the overscan. Default is None.

trim : str or None, optional

For no trim correction, set to None. Otherwise provide a region of ccd from which the image should be trimmed, using the FITS conventions for index order and index start. Default is None.

error : bool, optional

If True, create an uncertainty array for ccd. Default is False.

master_bias : CCDData or None, optional

A master bias frame to be subtracted from ccd. The unit of the master bias frame should match the unit of the image **after gain correction** if gain_corrected is True. Default is None.

dark_frame : CCDData or None, optional

A dark frame to be subtracted from the ccd. The unit of the master dark frame should match the unit of the image **after gain correction** if gain_corrected is True. Default is None.

master_flat : CCDData or None, optional

A master flat frame to be divided into ccd. The unit of the master flat frame should match the unit of the image **after gain correction** if gain_corrected is True. Default is None.

bad_pixel_mask : numpy.ndarray or None, optional

A bad pixel mask for the data. The bad pixel mask should be in given such that bad pixels have a value of 1 and good pixels a value of 0. Default is None.

gain : Quantity or None, optional

Gain value to multiple the image by to convert to electrons. Default is None.

readnoise : Quantity or None, optional

Read noise for the observations. The read noise should be in electrons. Default is None.

oscan_median : bool, optional

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

oscan_model : Model or None, optional

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

min_value : float or None, optional

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. Default is None.

dark_exposure : Quantity or None, optional

Exposure time of the dark image; if specified, must also provided data_exposure. Default is None.

data_exposure : Quantity or None, optional

Exposure time of the science image; if specified, must also provided dark_exposure. Default is None.

exposure_key : Keyword, str or None, optional

Name of key in image metadata that contains exposure time. Default is None.

exposure_unit : Unit or None, optional

Unit of the exposure time if the value in the meta data does not include a unit. Default is None.

dark_scale : bool, optional

If True, scale the dark frame by the exposure times. Default is False.

gain_corrected : bool, optional

If True, the master_bias, master_flat, and dark_frame have already been gain corrected. Default is True.

Returns

occd : CCDData

Reduded ccd.

Examples

1. To overscan, trim and gain correct a data set:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from ccdproc import CCDData
>>> from ccdproc import ccd_process
>>> ccd = CCDData(np.ones([100, 100]), unit=u.adu)
>>> nccd = ccd_process(ccd, oscan='[1:10,1:100]',
...                    trim='[10:100, 1:100]', error=False,
...                    gain=2.0*u.electron/u.adu)
```

ccdmask

ccdproc.**ccdmask**(*ratio*, *findbadcolumns=False*, *byblocks=False*, *ncmed=7*, *nlmed=7*, *ncsig=15*, *nlsig=15*, *lsigma=9*, *hsigma=9*, *ngood=5*)

Uses method based on the IRAF ccdmask task to generate a mask based on the given input.

Note: This function uses `lines` as synonym for the first axis and `columns` the second axis. Only two-dimensional `ratio` is currently supported.

Parameters

ratio : `CCDData`

Data to used to form mask. Typically this is the ratio of two flat field images.

findbadcolumns : `bool`, optional

If set to True, the code will search for bad column sections. Note that this treats columns as special and breaks symmetry between lines and columns and so is likely only appropriate for detectors which have readout directions. Default is `False`.

byblocks : `bool`, optional

If set to true, the code will divide the image up in to blocks of size nlsig by ncsig and determine the standard deviation estimate in each block (as described in the original IRAF task, see Notes below). If set to False, then the code will use `scipy.ndimage.percentile_filter` to generate a running box version of the standard deviation estimate and use that value for the standard deviation at each pixel. Default is `False`.

ncmed, nlmed : `int`, optional

The column and line size of the moving median rectangle used to estimate the uncontaminated local signal. The column median size should be at least 3 pixels to span single bad columns. Default is `7`.

ncsig, nlsig : `int`, optional

The column and line size of regions used to estimate the uncontaminated local sigma using a percentile. The size of the box should contain of order 100 pixels or more. Default is `15`.

lsigma, hsigma : `float`, optional

Positive sigma factors to use for selecting pixels below and above the median level based on the local percentile sigma. Default is `9`.

ngood : `int`, optional

Gaps of undetected pixels along the column direction of length less than this amount are also flagged as bad pixels, if they are between pixels masked in that column. Default is `5`.

Returns

mask : `numpy.ndarray`

A boolean ndarray where the bad pixels have a value of 1 (True) and valid pixels 0 (False), following the numpy.ma conventions.

Notes

Similar implementation to IRAF's ccdmask task. The Following documentation is copied directly from: http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?ccdmask

The input image is first subtracted by a moving box median. The median is unaffected by bad pixels provided the median size is larger that twice the size of a bad region. Thus, if 3 pixel wide bad columns are present then the column median box size should be at least 7 pixels. The median box can be a single pixel wide along one dimension if needed. This may be appropriate for spectroscopic long slit data.

The median subtracted image is then divided into blocks of size nclsig by nlsig. In each block the pixel values are sorted and the pixels nearest the 30.9 and 69.1 percentile points are found; this would be the one sigma points in a Gaussian noise distribution. The difference between the two count levels divided by two is then the local sigma estimate. This algorithm is used to avoid contamination by the bad pixel values. The block size must be at least 10 pixels in each dimension to provide sufficient pixels for a good estimate of the percentile sigma. The sigma uncertainty estimate of each pixel in the image is then the sigma from the nearest block.

The deviant pixels are found by comparing the median subtracted residual to a specified sigma threshold factor times the local sigma above and below zero (the lsigma and hsigma parameters). This is done for individual pixels and then for column sums of pixels (excluding previously flagged bad pixels) from two to the number of lines in the image. The sigma of the sums is scaled by the square root of the number of pixels summed so that statistically low or high column regions may be detected even though individual pixels may not be statistically deviant. For the purpose of this task one would normally select large sigma threshold factors such as six or greater to detect only true bad pixels and not the extremes of the noise distribution.

As a final step each column is examined to see if there are small segments of unflagged pixels between bad pixels. If the length of a segment is less than that given by the ngood parameter all the pixels in the segment are also marked as bad.

combine

ccdproc.**combine**(*img_list*, *output_file=None*, *method=u'average'*, *weights=None*, *scale=None*, *mem_limit=16000000000.0*, *clip_extrema=False*, *nlow=1*, *nhigh=1*, *minmax_clip=False*, *minmax_clip_min=None*, *minmax_clip_max=None*, *sigma_clip=False*, *sigma_clip_low_thresh=3*, *sigma_clip_high_thresh=3*, *sigma_clip_func=<numpy.ma.core._frommethod instance>*, *sigma_clip_dev_func=<numpy.ma.core._frommethod instance>*, *dtype=None*, *combine_uncertainty_function=None*, ***ccdkwargs*)

Convenience function for combining multiple images.

Parameters

img_list : `numpy.ndarray`, list or str

A list of fits filenames or `CCDData` objects that will be combined together. Or a string of fits filenames separated by comma ",".

output_file : str or None, optional

Optional output fits file-name to which the final output can be directly written. Default is `None`.

method : str, optional

Method to combine images:

- `'average'` : To combine by calculating the average.
- `'median'` : To combine by calculating the median.
- `'sum'` : To combine by calculating the sum.

Default is `'average'`.

weights : `numpy.ndarray` or None, optional

Weights to be used when combining images. An array with the weight values. The dimensions should match the the dimensions of the data arrays being combined. Default is None.

scale : function or numpy.ndarray-like or None, optional

Scaling factor to be used when combining images. 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.

mem_limit : float, optional

Maximum memory which should be used while combining (in bytes). Default is 16e9.

clip_extrema : bool, optional

Set to True if you want to mask pixels using an IRAF-like minmax clipping algorithm. The algorithm will mask the lowest nlow values and the highest nhigh values before combining the values to make up a single pixel in the resulting image. For example, the image will be a combination of Nimages-low-nhigh pixel values instead of the combination of Nimages.

Parameters below are valid only when clip_extrema is set to True, see Combiner.clip_extrema() for the parameter description:

- nlow : int or None, optional
- nhigh : int or None, optional

minmax_clip : bool, optional

Set to True if you want to mask all pixels that are below minmax_clip_min or above minmax_clip_max before combining. Default is False.

Parameters below are valid only when minmax_clip is set to True, see Combiner.minmax_clipping() for the parameter description:

- minmax_clip_min : float or None, optional
- minmax_clip_max : float or None, optional

sigma_clip : bool, optional

Set to True if you want to reject pixels which 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 sigma_clip_dev_func and the input data array. Any pixel with a deviation from the baseline value greater than that set by sigma_clip_high_thresh or lower than that set by sigma_clip_low_thresh will be rejected. Default is False.

Parameters below are valid only when sigma_clip is set to True. See Combiner.sigma_clipping() for the parameter description.

- sigma_clip_low_thresh : positive float or None, optional
- sigma_clip_high_thresh : positive float or None, optional
- sigma_clip_func : function, optional
- sigma_clip_dev_func : function, optional

dtype : str or numpy.dtype or None, optional

The intermediate and resulting `dtype` for the combined CCDs. See `ccdproc.Combiner`. If `None` this is set to `float64`. Default is `None`.

combine_uncertainty_function : callable, None, optional

If `None` use the default uncertainty func when using average, median or sum combine, otherwise use the function provided. Default is `None`.

ccdkwargs : Other keyword arguments for `astropy.nddata.fits_ccddata_reader`.

Returns

combined_image : `CCDData`

CCDData object based on the combined input of CCDData objects.

cosmicray_lacosmic

ccdproc.**cosmicray_lacosmic**(*ccd*, *sigclip=4.5*, *sigfrac=0.3*, *objlim=5.0*, *gain=1.0*, *readnoise=6.5*, *satlevel=65535.0*, *pssl=0.0*, *niter=4*, *sepmed=True*, *cleantype=u'meanmask'*, *fsmode=u'median'*, *psfmodel=u'gauss'*, *psffwhm=2.5*, *psfsize=7*, *psfk=None*, *psfbeta=4.765*, *verbose=False*)

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) *[R56]* as implemented by McCully (2014) *[R66]*. If you use this algorithm, please cite these two works.

Parameters

ccd : `CCDData` or `numpy.ndarray`

Data to have cosmic ray cleaned.

sigclip : float, optional

Laplacian-to-noise limit for cosmic ray detection. Lower values will flag more pixels as cosmic rays. Default: 4.5.

sigfrac : float, optional

Fractional detection limit for neighboring pixels. For cosmic ray neighbor pixels, a Laplacian-to-noise detection limit of sigfrac * sigclip will be used. Default: 0.3.

objlim : float, optional

Minimum contrast between Laplacian image and the fine structure image. Increase this value if cores of bright stars are flagged as cosmic rays. Default: 5.0.

pssl : float, optional

Previously subtracted sky level in ADU. We always need to work in electrons for cosmic ray detection, so we need to know the sky level that has been subtracted so we can add it back in. Default: 0.0.

gain : float, optional

Gain of the image (electrons / ADU). We always need to work in electrons for cosmic ray detection. Default: 1.0

readnoise : float, optional

Read noise of the image (electrons). Used to generate the noise model of the image. Default: 6.5.

satlevel : float, optional

Saturation level of the image (electrons). This value is used to detect saturated stars and pixels at or above this level are added to the mask. Default: 65535.0.

niter : int, optional

Number of iterations of the LA Cosmic algorithm to perform. Default: 4.

sepmed : bool, optional

Use the separable median filter instead of the full median filter. The separable median is not identical to the full median filter, but they are approximately the same and the separable median filter is significantly faster and still detects cosmic rays well. Default: True

cleantype : str, optional

Set which clean algorithm is used:

- `"median"`: An unmasked 5x5 median filter.
- `"medmask"`: A masked 5x5 median filter.
- `"meanmask"`: A masked 5x5 mean filter.
- `"idw"`: A masked 5x5 inverse distance weighted interpolation.

Default: `"meanmask"`.

fsmode : str, optional

Method to build the fine structure image:

- `"median"`: Use the median filter in the standard LA Cosmic algorithm.
- `"convolve"`: Convolve the image with the psf kernel to calculate the fine structure image.

Default: `"median"`.

psfmodel : str, optional

Model to use to generate the psf kernel if fsmode == 'convolve' and psfk is None. The current choices are Gaussian and Moffat profiles:

- `"gauss"` and `"moffat"` produce circular PSF kernels.
- The `"gaussx"` and `"gaussy"` produce Gaussian kernels in the x and y directions respectively.

Default: `"gauss"`.

psffwhm : float, optional

Full Width Half Maximum of the PSF to use to generate the kernel. Default: 2.5.

psfsize : int, optional

Size of the kernel to calculate. Returned kernel will have size psfsize x psfsize. psfsize should be odd. Default: 7.

psfk : `numpy.ndarray` (with float dtype) or None, optional

PSF kernel array to use for the fine structure image if `fsmode == 'convolve'`. If None and `fsmode == 'convolve'`, we calculate the psf kernel using `psfmodel`. Default: None.

psfbeta : float, optional

Moffat beta parameter. Only used if `fsmode=='convolve'` and `psfmodel=='moffat'`. Default: 4.765.

verbose : bool, optional

Print to the screen or not. Default: False.

Returns

nccd : `CCDData` or `numpy.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.

crmask : `numpy.ndarray`

If an `numpy.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

[R56], *[R66]*

Examples

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

```
>>> newdata, mask = cosmicray_lacosmic(data, sigclip=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, sigclip=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.

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`, `numpy.ndarray` or `numpy.ma.MaskedArray`

Data to have cosmic ray cleaned.

thresh : float, optional

Threshold for detecting cosmic rays. Default is 5.

error_image : numpy.ndarray, float or None, optional

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. Default is None.

mbox : int, optional

Median box for detecting cosmic rays. Default is 11.

gbox : int, optional

Box size to grow cosmic rays. If zero, no growing will be done. Default is 0.

rbox : int, optional

Median box for calculating replacement values. If zero, no pixels will be replaced. Default is 0.

Returns

nccd : CCDData or numpy.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 : numpy.ndarray

If an numpy.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.

create_deviation

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

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 or None, 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. Default is None.

readnoise : Quantity or None, optional

Read noise per pixel. Default is None.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to False or 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.

flat_correct

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

Correct the image for flat fielding.

The flat field image is normalized by its mean or a user-supplied value before flat correcting.

Parameters

ccd : CCDData

Data to be transformed.

flat : CCDData

Flatfield to apply to the data.

min_value : float or None, optional

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. Default is None.

norm_value : float or None, optional

If not None, normalize flat field by this argument rather than the mean of the image. This allows fixing several different flat fields to have the same scale. If this value is negative or 0, a ValueError is raised. Default is None.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to False or 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.

gain_correct

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

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 or None, optional

Unit for the gain; used only if gain itself does not provide units. Default is None.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to False or 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.

median_filter

ccdproc.**median_filter**(*data*, **args*, ***kwargs*)

See scipy.ndimage.median_filter for arguments.

If the data is a CCDData object the result will be another CCDData object with the median filtered data as data and copied unit and meta.

rebin

ccdproc.**rebin**(**args*, ***kwargs*)

Deprecated since version 1.1: The rebin function is deprecated and may be removed in a future version.

Rebin an array to have a new shape.

Parameters

ccd : CCDData or numpy.ndarray

Data to rebin.

newshape : tuple

Tuple containing the new shape for the array.

Returns

output : CCDData or numpy.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 dimension of the new shape is not equal to the data's.

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:

```
rebin(arr1, (20,20))
```

sigma_func

ccdproc.**sigma_func**(*arr*, *axis=None*)

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 numpy.ndarray

Array whose deviation is to be calculated.

axis : int, tuple of ints or None, optional

Axis or axes along which the function is performed. If None it is performed over all the dimensions of the input array. The axis argument can also be negative, in this case it counts from the last to the first axis. Default is None.

Returns

uncertainty : float

uncertainty of array estimated from median absolute deviation.

subtract_bias

ccdproc.**subtract_bias**(*ccd*, *master*, *add_keyword=True*)

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 False or 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.

subtract_dark

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

Subtract dark current from an image.

Parameters

ccd : CCDData

Image from which dark will be subtracted.

master : CCDData

Dark image.

dark_exposure : Quantity or None, optional

Exposure time of the dark image; if specified, must also provided data_exposure. Default is None.

data_exposure : Quantity or None, optional

Exposure time of the science image; if specified, must also provided dark_exposure. Default is None.

exposure_time : str or Keyword or None, optional

Name of key in image metadata that contains exposure time. Default is None.

exposure_unit : Unit or None, optional

Unit of the exposure time if the value in the meta data does not include a unit. Default is None.

scale: bool, optional

If True, scale the dark frame by the exposure times. Default is False.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to False or 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.

subtract_overscan

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

Subtract the overscan region from an image.

Parameters

ccd : CCDData

Data to have overscan frame corrected.

overscan : CCDData or None, optional

Slice from ccd that contains the overscan. Must provide either this argument or fits_section, but not both. Default is None.

overscan_axis : 0, 1 or None, 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.

If overscan_axis is explicitly set to None, the axis is set to the shortest dimension of the overscan section (or 1 in case of a square overscan). Default is 1.

fits_section : str or None, optional

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. Default is None.

median : bool, optional

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

model : Model or None, optional

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

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to False or 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

Creating a 100x100 array containing ones just for demonstration purposes:

```
>>> 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.

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.

transform_image

ccdproc.**transform_image**(*ccd*, *transform_func*, *add_keyword=True*, ***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 transformed.

transform_func : callable

Function to be used to transform the data, mask and uncertainty.

kwargs :

Additional keyword 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 False or 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 uncertainty 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.shift`:

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

trim_image

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

Trim the image to the dimensions indicated.

Parameters

ccd : `CCDData`

CCD image to be trimmed, sliced if desired.

fits_section : str or None, optional

Region of ccd from which the overscan is extracted; see `subtract_overscan` for details. Default is `None`.

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

Item(s) to add to metadata of result. Set to False or 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.

wcs_project

ccdproc.**wcs_project**(*ccd*, *target_wcs*, *target_shape=None*, *order=u'bilinear'*, *add_keyword=True*)

Given a CCDData image with WCS, project it onto a target WCS and return the reprojected data as a new CCDData image.

Any flags, weight, or uncertainty are ignored in doing the reprojection.

Parameters

ccd : `CCDData`

Data to be projected.

target_wcs : `WCS` object

WCS onto which all images should be projected.

target_shape : two element list-like or None, optional

Shape of the output image. If omitted, defaults to the shape of the input image. Default is None.

order : str, optional

Interpolation order for re-projection. Must be one of:

- 'nearest-neighbor'
- 'bilinear'
- 'biquadratic'
- 'bicubic'

Default is `'bilinear'`.

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

Item(s) to add to metadata of result. Set to False or 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.

2.4.2 Classes

`Combiner`(ccd_list[, dtype])	A class for combining CCDData objects.
`ImageFileCollection`([location, keywords, ...])	Representation of a collection of image files.
`Keyword`(name[, unit, value])	

Combiner

class ccdproc.**Combiner**(*ccd_list*, *dtype=None*)

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.

dtype : str or `numpy.dtype` or None, optional

Allows user to set dtype. See `numpy.array` `dtype` parameter description. If `None` it uses `np.float64`. Default is `None`.

Raises

TypeError

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

Examples

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

```
>>> import numpy as np
>>> import astropy.units as u
>>> from ccdproc import Combiner, CCDData
>>> ccddata1 = CCDData(np.ones((4, 4)), unit=u.adu)
>>> ccddata2 = CCDData(np.zeros((4, 4)), unit=u.adu)
>>> ccddata3 = CCDData(np.ones((4, 4)), unit=u.adu)
>>> c = Combiner([ccddata1, ccddata2, ccddata3])
>>> ccdall = c.average_combine()
>>> ccdall
CCDData([[ 0.66666667,  0.66666667,  0.66666667,  0.66666667],
         [ 0.66666667,  0.66666667,  0.66666667,  0.66666667],
         [ 0.66666667,  0.66666667,  0.66666667,  0.66666667],
         [ 0.66666667,  0.66666667,  0.66666667,  0.66666667]])
```

Attributes Summary

dtype	
scaling	Scaling factor used in combining images.
weights	Weights used when combining the CCDData objects.

Methods Summary

average_combine([scale_func, scale_to, ...])	Average combine together a set of arrays.
clip_extrema([nlow, nhigh])	Mask pixels using an IRAF-like minmax clipping algorithm.
median_combine([median_func, scale_to, ...])	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 threshold values.
sum_combine([sum_func, scale_to, ...])	Sum combine together a set of arrays.

Attributes Documentation

dtype

scaling

Scaling factor used in combining images.

Parameters

scale : function or numpy.ndarray-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.

weights
Weights used when combining the CCDData objects.

Parameters
weight_values : numpy.ndarray or None

An array with the weight values. The dimensions should match the the dimensions of the data arrays being combined.

Methods Documentation

average_combine(*scale_func=<function average>*, *scale_to=None*, *uncertainty_func=<numpy.ma.core._frommethod instance>*)
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 average. 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.

Parameters
scale_func : function, optional

Function to calculate the average. Defaults to numpy.ma.average.

scale_to : float or None, optional

Scaling factor used in the average combined image. If given, it overrides scaling. Defaults to None.

uncertainty_func : function, optional

Function to calculate uncertainty. Defaults to numpy.ma.std.

Returns
combined_image: CCDData

CCDData object based on the combined input of CCDData objects.

clip_extrema(*nlow=0*, *nhigh=0*)
Mask pixels using an IRAF-like minmax clipping algorithm. The algorithm will mask the lowest nlow values and the highest nhigh values before combining the values to make up a single pixel in the resulting image. For example, the image will be a combination of Nimages-nlow-nhigh pixel values instead of the combination of Nimages.

Parameters
nlow : int or None, optional

If not None, the number of low values to reject from the combination. Default is 0.

nhigh : int or None, optional

If not None, the number of high values to reject from the combination. Default is 0.

Notes

Note that this differs slightly from the nominal IRAF imcombine behavior when other masks are in use. For example, if nhigh>=1 and any pixel is already masked for some other reason, then this algorithm will count the masking of that pixel toward the count of nhigh masked pixels.

Here is a copy of the relevant IRAF help text *[R23]*:

nlow = 1, nhigh = (minmax)
The number of low and high pixels to be rejected by the "minmax" algorithm. These numbers are converted to fractions of the total number of input images so that if no rejections have taken place the specified number of pixels are rejected while if pixels have been rejected by masking, thresholding, or nonoverlap, then the fraction of the remaining pixels, truncated to an integer, is used.

References

[R23]

median_combine(*median_func=<function median>*, *scale_to=None*, *uncertainty_func=<function sigma_func>*)
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.MaskedArray`. Default is `numpy.ma.median`.

scale_to : float or None, optional

Scaling factor used in the average combined image. If given, it overrides `scaling`. Defaults to None.

uncertainty_func : function, optional

Function to calculate uncertainty. Defaults is `sigma_func`.

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 : float or None, optional

If not None, all pixels with values below min_clip will be masked. Default is `None`.

max_clip : float or None, optional

If not None, all pixels with values above min_clip will be masked. Default is `None`.

sigma_clipping(*low_thresh=3*, *high_thresh=3*, *func=<numpy.ma.core._frommethod instance>*, *dev_func=<numpy.ma.core._frommethod instance>*)
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, optional

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. Default is 3.

high_thresh : positive float or None, optional

Threshold for rejecting pixels that deviate above the baseline value. If None, no rejection will be done based on high_thresh. Default is 3.

func : function, optional

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

dev_func : function, optional

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

sum_combine(*sum_func=<numpy.ma.core._frommethod instance>*, *scale_to=None*, *uncertainty_func=<numpy.ma.core._frommethod instance>*)

Sum combine together a set of arrays.

A `CCDData` object is returned with the data property set to the sum of the arrays. If the data was masked or any data have been rejected, those pixels will not be included in the sum. 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 multiplication of summation of standard deviation of the input by square root of number of images. Because sum_combine returns 'pure sum' with masked pixels ignored, if re-scaled sum is needed, average_combine have to be used with multiplication by number of images combined.

Parameters

sum_func : function, optional

Function to calculate the sum. Defaults to `numpy.ma.sum`.

scale_to : float or None, optional

Scaling factor used in the sum combined image. If given, it overrides `scaling`. Defaults to None.

uncertainty_func : function, optional

Function to calculate uncertainty. Defaults to `numpy.ma.std`.

Returns

combined_image: `CCDData`

CCDData object based on the combined input of CCDData objects.

ImageFileCollection

class ccdproc.**ImageFileCollection**(*location=None*, *keywords=None*, *info_file=None*, *filenames=None*, *glob_include=None*, *glob_exclude=None*, *ext=0*)

Bases: `object`

Representation of a collection of image files.

The class offers a table summarizing values of keywords in the FITS headers of the files in the collection and offers convenient methods for iterating over the files in the collection. The generator methods use simple filtering syntax and can automate storage of any FITS files modified in the loop using the generator.

Parameters

location : str or None, optional

Path to directory containing FITS files. Default is `None`.

keywords : list of str, '*' or None, optional

Keywords that should be used as column headings in the summary table. If the value is or includes '*' then all keywords that appear in any of the FITS headers of the files in the collection become table columns. Default value is '*' unless `info_file` is specified. Default is `None`.

info_file : str or None, optional

Path to file that contains a table of information about FITS files. In this case the keywords are set to the names of the columns of the `info_file` unless `keywords` is explicitly set to a different list. Default is `None`.

Deprecated since version 1.3.

filenames: str, list of str, or None, optional

List of the names of FITS files which will be added to the collection. The filenames are assumed to be in `location`. Default is `None`.

glob_include: str or None, optional

Unix-style filename pattern to select filenames to include in the file collection. Can be used in conjunction with `glob_exclude` to easily select subsets of files in the target directory. Default is `None`.

glob_exclude: str or None, optional

Unix-style filename pattern to select filenames to exclude from the file collection. Can be used in conjunction with `glob_include` to easily select subsets of files in the target directory. Default is `None`.

ext: str or int, optional

The extension from which the header and data will be read in all files. Default is `0`.

Raises

ValueError

Raised if keywords are set to a combination of '*' and any other value.

Attributes Summary

`ext`	str or int, The extension from which the header and data will
`files`	list of str, Unfiltered list of FITS files in location.
`glob_exclude`	str or None, Unix-style filename pattern to select filenames to exclude
`glob_include`	str or None, Unix-style filename pattern to select filenames to include

Continued on next page

Table 2.5 – continued from previous page

keywords	list of str, Keywords currently in the summary table.
location	str, Path name to directory containing FITS files.
summary	Table of values of FITS keywords for files in the
summary_info	Table of values of FITS keywords for files in the

Methods Summary

ccds([ccd_kwargs])	Generator that yields each CCDData in the collection.
data([do_not_scale_image_data])	Generator that yields each image in the collection.
files_filtered(**kwd)	Determine files whose keywords have listed values.
hdus([do_not_scale_image_data])	Generator that yields each HDUList in the collection.
headers([do_not_scale_image_data])	Generator that yields each header in the collection.
refresh()	Refresh the collection by re-reading headers.
sort(keys)	Sort the list of files to determine the order of iteration.
values(keyword[, unique])	List of values for a keyword.

Attributes Documentation

ext
str or int, The extension from which the header and data will be read in all files.

files
list of str, Unfiltered list of FITS files in location.

glob_exclude
str or None, Unix-style filename pattern to select filenames to exclude in the file collection.

glob_include
str or None, Unix-style filename pattern to select filenames to include in the file collection.

keywords
list of str, Keywords currently in the summary table.

Setting the keywords causes the summary table to be regenerated unless the new keywords are a subset of the old.

Changed in version 1.3: Added `deleter` for `keywords` property.

location
str, Path name to directory containing FITS files.

summary
Table of values of FITS keywords for files in the collection.

Each keyword is a column heading. In addition, there is a column called `file` that contains the name of the FITS file. The directory is not included as part of that name.

The first column is always named `file`.

The order of the remaining columns depends on how the summary was constructed.

If a wildcard, `*` was used then the order is the order in which the keywords appear in the FITS files from which the summary is constructed.

If an explicit list of keywords was supplied in setting up the collection then the order of the columns is the order of the keywords.

summary_info
Table of values of FITS keywords for files in the collection.

Each keyword is a column heading. In addition, there is a column called 'file' that contains the name of the FITS file. The directory is not included as part of that name.

Deprecated since version 0.4.

Methods Documentation

ccds(*ccd_kwargs=None*, ***kwd*)
Generator that yields each CCDData in the collection.

If any of the parameters `save_with_name`, `save_location` or `overwrite` evaluates to `True` the generator will write a copy of each FITS file it is iterating over. In other words, if `save_with_name` and/or `save_location` is a string with non-zero length, and/or `overwrite` is `True`, a copy of each FITS file will be made.

Parameters

save_with_name : str, optional

string added to end of file name (before extension) if FITS file should be saved after iteration. Unless `save_location` is set, files will be saved to location of the source files `self.location`. Default is `''`.

save_location : str, optional

Directory in which to save FITS files; implies that FITS files will be saved. Note this provides an easy way to copy a directory of files–loop over the CCDData with `save_location` set. Default is `''`.

overwrite : bool, optional

If `True`, overwrite input FITS files. Default is `False`.

clobber : bool, optional

Alias for `overwrite`. Default is `False`.

do_not_scale_image_data : bool, optional

If `True`, prevents fits from scaling images. Default is `True`. Default is `True`.

return_fname : bool, optional

If True, return the tuple (header, file_name) instead of just header. The file name returned is the name of the file only, not the full path to the file. Default is `False`.

ccd_kwargs : dict, optional

Dict with parameters for `fits_ccddata_reader`. For instance, the key `'unit'` can be used to specify the unit of the data. If `'unit'` is not given then `'adu'` is used as the default unit. See `fits_ccddata_reader` for a complete list of parameters that can be passed through `ccd_kwargs`.

****kwd :**

Any additional keywords are used to filter the items returned; see `files_filtered` examples for details.

Returns

`astropy.nddata.CCDData`

If `return_fname` is `False`, yield the next CCDData in the collection.

(`astropy.nddata.CCDData`, str)

If `return_fname` is `True`, yield a tuple of (CCDData, `file name`) for the next item in the collection.

data(*do_not_scale_image_data=False*, ***kwd*)

Generator that yields each image in the collection.

If any of the parameters `save_with_name`, `save_location` or `overwrite` evaluates to `True` the generator will write a copy of each FITS file it is iterating over. In other words, if `save_with_name` and/or `save_location` is a string with non-zero length, and/or `overwrite` is `True`, a copy of each FITS file will be made.

Parameters

save_with_name : str, optional

string added to end of file name (before extension) if FITS file should be saved after iteration. Unless `save_location` is set, files will be saved to location of the source files `self.location`. Default is `''`.

save_location : str, optional

Directory in which to save FITS files; implies that FITS files will be saved. Note this provides an easy way to copy a directory of files–loop over the image with `save_location` set. Default is `''`.

overwrite : bool, optional

If `True`, overwrite input FITS files. Default is `False`.

clobber : bool, optional

Alias for `overwrite`. Default is `False`.

do_not_scale_image_data : bool, optional

If `True`, prevents fits from scaling images. Default is `False`. Default is `True`.

return_fname : bool, optional

If True, return the tuple (header, file_name) instead of just header. The file name returned is the name of the file only, not the full path to the file. Default is `False`.

ccd_kwargs : dict, optional

Dict with parameters for `fits_ccddata_reader`. For instance, the key `'unit'` can be used to specify the unit of the data. If `'unit'` is not given then `'adu'` is used as the default unit. See `fits_ccddata_reader` for a complete list of parameters that can be passed through `ccd_kwargs`.

****kwd :**

Any additional keywords are used to filter the items returned; see `files_filtered` examples for details.

Returns

`numpy.ndarray`

If `return_fname` is `False`, yield the next image in the collection.

(`numpy.ndarray`, str)

If `return_fname` is `True`, yield a tuple of (image, `file name`) for the next item in the collection.

files_filtered(***kwd*)
Determine files whose keywords have listed values.

Parameters
include_path : bool, keyword-only

If the keyword `include_path=True` is set, the returned list contains not just the file-name, but the full path to each file. Default is `False`.

****kwd :**

`**kwd` is dict of keywords and values the files must have. The value '*' represents any value. A missing keyword is indicated by value ''.

Returns
filenames : list

The files that satisfy the keyword-value restrictions specified by the `**kwd`.

Notes

Value comparison is case *insensitive* for strings.

Examples

Some examples for filtering:

```
>>> keys = ['imagetyp','filter']
>>> collection = ImageFileCollection('test/data', keywords=keys)
>>> collection.files_filtered(imagetyp='LIGHT', filter='R')
>>> collection.files_filtered(imagetyp='*', filter='')
```

In case you want to filter with keyword names that cannot be used as keyword argument name, you have to unpack them using a dictionary. For example if a keyword name contains a space or a `-`:

```
>>> add_filters = {'exp-time': 20, 'ESO TPL ID': 1050}
>>> collection.files_filtered(imagetyp='LIGHT', **add_filters)
```

hdus(*do_not_scale_image_data=False*, ***kwd*)
Generator that yields each HDUList in the collection.

If any of the parameters `save_with_name`, `save_location` or `overwrite` evaluates to `True` the generator will write a copy of each FITS file it is iterating over. In other words, if `save_with_name` and/or `save_location` is a string with non-zero length, and/or `overwrite` is `True`, a copy of each FITS file will be made.

Parameters
save_with_name : str, optional

string added to end of file name (before extension) if FITS file should be saved after iteration. Unless `save_location` is set, files will be saved to location of the source files `self.location`. Default is `''`.

save_location : str, optional

Directory in which to save FITS files; implies that FITS files will be saved. Note this provides an easy way to copy a directory of files–loop over the HDUList with `save_location` set. Default is `''`.

overwrite : bool, optional

If `True`, overwrite input FITS files. Default is `False`.

clobber : bool, optional

Alias for `overwrite`. Default is `False`.

do_not_scale_image_data : bool, optional

If `True`, prevents fits from scaling images. Default is `False`. Default is `True`.

return_fname : bool, optional

If True, return the tuple (header, file_name) instead of just header. The file name returned is the name of the file only, not the full path to the file. Default is `False`.

ccd_kwargs : dict, optional

Dict with parameters for `fits_ccddata_reader`. For instance, the key `'unit'` can be used to specify the unit of the data. If `'unit'` is not given then `'adu'` is used as the default unit. See `fits_ccddata_reader` for a complete list of parameters that can be passed through `ccd_kwargs`.

****kwd :**

Any additional keywords are used to filter the items returned; see `files_filtered` examples for details.

Returns

`astropy.io.fits.HDUList`

If `return_fname` is `False`, yield the next HDUList in the collection.

(`astropy.io.fits.HDUList`, str)

If `return_fname` is `True`, yield a tuple of (HDUList, `file name`) for the next item in the collection.

headers(*do_not_scale_image_data=True*, ***kwd*)

Generator that yields each header in the collection.

If any of the parameters `save_with_name`, `save_location` or `overwrite` evaluates to `True` the generator will write a copy of each FITS file it is iterating over. In other words, if `save_with_name` and/or `save_location` is a string with non-zero length, and/or `overwrite` is `True`, a copy of each FITS file will be made.

Parameters

save_with_name : str, optional

string added to end of file name (before extension) if FITS file should be saved after iteration. Unless `save_location` is set, files will be saved to location of the source files `self.location`. Default is `''`.

save_location : str, optional

Directory in which to save FITS files; implies that FITS files will be saved. Note this provides an easy way to copy a directory of files–loop over the header with `save_location` set. Default is `''`.

overwrite : bool, optional

If `True`, overwrite input FITS files. Default is `False`.

clobber : bool, optional

Alias for `overwrite`. Default is `False`.

do_not_scale_image_data : bool, optional

If `True`, prevents fits from scaling images. Default is `True`. Default is `True`.

return_fname : bool, optional

If True, return the tuple (header, file_name) instead of just header. The file name returned is the name of the file only, not the full path to the file. Default is `False`.

ccd_kwargs : dict, optional

Dict with parameters for `fits_ccddata_reader`. For instance, the key `'unit'` can be used to specify the unit of the data. If `'unit'` is not given then `'adu'` is used as the default unit. See `fits_ccddata_reader` for a complete list of parameters that can be passed through `ccd_kwargs`.

****kwd :**

Any additional keywords are used to filter the items returned; see `files_filtered` examples for details.

Returns

`astropy.io.fits.Header`

If `return_fname` is `False`, yield the next header in the collection.

(`astropy.io.fits.Header`, str)

If `return_fname` is `True`, yield a tuple of (header, `file name`) for the next item in the collection.

refresh()

Refresh the collection by re-reading headers.

sort(*keys*)

Sort the list of files to determine the order of iteration.

Sort the table of files according to one or more keys. This does not create a new object, instead is sorts in place.

Parameters

keys : str, list of str

The key(s) to order the table by.

values(*keyword*, *unique=False*)

List of values for a keyword.

Parameters

keyword : str

Keyword (i.e. table column) for which values are desired.

unique : bool, optional

If True, return only the unique values for the keyword. Default is `False`.

Returns

list

Values as a list.

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

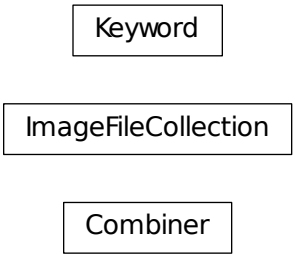
Methods Documentation

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

Parameters
header : Header

FITS header containing a value for this keyword.

2.4.3 Class Inheritance Diagram



2.5 ccdproc.utils.slices Module

Define utility functions and classes for ccdproc

2.5.1 Functions

slice_from_string(string[, fits_convention])	Convert a string to a tuple of slices.

slice_from_string

ccdproc.utils.slices.**slice_from_string**(*string*, *fits_convention=False*)

Convert a string to a tuple of slices.

Parameters

string : str

A string that can be converted to a slice.

fits_convention : bool, optional

If True, assume the input string follows the FITS convention for indexing: the indexing is one-based (not zero-based) and the first axis is that which changes most rapidly as the index increases.

Returns

slice_tuple : tuple of slice objects

A tuple able to be used to index a numpy.array

Notes

The `string` argument can be anything that would work as a valid way to slice an array in Numpy. It must be enclosed in matching brackets; all spaces are stripped from the string before processing.

Examples

```
>>> import numpy as np
>>> arr1d = np.arange(5)
>>> a_slice = slice_from_string('[2:5]')
>>> arr1d[a_slice]
array([2, 3, 4])
>>> a_slice = slice_from_string('[ : : -2] ')
>>> arr1d[a_slice]
array([4, 2, 0])
>>> arr2d = np.array([arr1d, arr1d + 5, arr1d + 10])
>>> arr2d
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a_slice = slice_from_string('[1:-1, 0:4:2]')
>>> arr2d[a_slice]
array([[5, 7]])
>>> a_slice = slice_from_string('[0:2,0:3]')
>>> arr2d[a_slice]
array([[0, 1, 2],
       [5, 6, 7]])
```

CHAPTER 3

Contributors

3.1 Authors and Credits

3.1.1 ccdproc Project Contributors

Project Coordinators

- Matt Craig (@mwcraig)
- Steve Crawford (@crawfordsm)
- Michael Seifert (@MSeifert04)

Alphabetical list of contributors

- Yoonsoo P. Bach (@ysBach)
- Kyle Barbary (@kbarbary)
- Javier Blasco (@javierblasco)
- Christoph Deil (@cdeil)
- Carlos Gomez (@carlgogo)
- Hans Moritz Günther (@hamogu)
- Forrest Gasdia (@EP-Guy)
- Nathan Heidt (@heidtha)
- Elias Holte (@Sondanaa)
- Anthony Horton (@AnthonyHorton)
- Jennifer Karr (@JenniferKarr)
- James McCormac (@jmccormac01)

- Stefan Nelson (@stefannelson)
- Joe Philip Ninan (@indiajoe)
- Punyaslok Pattnaik (@Punyaslok)
- Adrian Price-Whelan (@adrn)
- Evert Rol (@evertrol)
- William Schoenell (@wschoenell)
- Sourav Singh (@souravsingh)
- Brigitta Sipocz (@bsipocz)
- Connor Stotts (@stottsco)
- Ole Streicher (@olebole)
- JVSN Reddy (@janga1997)
- Erik Tollerud (@eteq)
- Zè Vinícius (@mirca)
- Josh Walawender (@joshwalawender)
- Nathan Walker (@walkerna22)
- Benjamin Weiner (@bjweiner)
- Jiyong Youn (@hletrd)

(If you have contributed to the ccdproc project and your name is missing, please send an email to the coordinators, or open a pull request for this page in the ccdproc repository)

CHAPTER 4

Full Changelog

4.1 1.3.0 (2017-11-1)

4.1.1 New Features

- Add representation for ImageFileCollection. [#475, #515]
- Added ext parameter and property to ImageFileCollection to specify the FITS extension. [#463]
- Add keywords.deleter method to ImageFileCollection. [#474]
- Added `glob_include` and `glob_exclude` parameter to `ImageFileCollection`. [#484]
- Add `bitfield_to_boolean_mask` function to convert a `bitfield` to a boolean mask (following the numpy conventions). [#460]
- Added `gain_corrected` option in ccd_process so that calibration files do not need to previously been gain corrected. [#491]
- Add a new `wcs_relax` argument to `CCDData.to_header()` that is passed through to the `WCS` method of the same name to allow more flexible handing of headers with SIP distortion. [#501]
- `combine` now accepts `numpy.ndarray` as the input `img_list`. [#493, #503]
- Added `sum` option in method for `combime`. [#500, #508]
- Add `norm_value` argument to `flat_correct` that allows the normalization of the flat frame to be manually specified. [#584, #577]

4.1.2 Other Changes and Additions

- removed ability to set unit of CCDData to None. [#451]
- deprecated `summary_info` property of `ImageFileCollection` now raises a deprecation warning. [#486]
- Logging will include the abbreviation even if the `meta` attribute of the processed `CCDData` isn't a `fits.Header`. [#528]

- The CCDData class and the functions fits_ccddata_reader and fits_ccddata_writer will be imported from astropy.nddata if astropy >= 2.0 is installed (instead of the one defined in ccdproc). [#528]
- Building the documentation requires astropy >= 2.0. [#528]
- When reading a CCDData from a file the WCS-related keywords are removed from the header. [#568]
- The info_file argument for ImageFileCollection is now deprecated. [#585]

4.1.3 Bug Fixes

- ImageFileCollection now handles Headers with duplicated keywords (other than COMMENT and HISTORY) by ignoring all but the first. [#467]
- The ccd method of ImageFileCollection will raise an NotImplementedError in case the parameter overwrite=True or clobber=True is used instead of silently ignoring the parameter. [#527]
- The sort method of ImageFileCollection now requires an explicitly given keys argument. [#534]
- Fixed a problem with CCDData.read when the extension wasn't given and the primary HDU contained no data but another HDU did. In that case the header were not correctly combined. [#541]
- Suppress errors during WCS creation in CCDData.read(). [#552]
- The generator methods in ImageFileCollection now don't leave open file handles in case the iterator wasn't advanced or an exception was raised either inside the method itself or during the loop. [#553]
- Allow non-string columns when filtering an ImageFileCollection with a string value. [#567]

4.2 1.2.0 (2016-12-13)

ccdproc has now the following additional dependency:

- scikit-image.

4.2.1 New Features

- Add an optional attribute named filenames to ImageFileCollection, so that users can pass a list of FITS files to the collection. [#374, #403]
- Added block_replicate, block_reduce and block_average functions. [#402]
- Added median_filter function. [#420]
- combine now takes an additional combine_uncertainty_function argument which is passed as uncertainty_func parameter to Combiner.median_combine or Combiner.average_combine. [#416]
- Added ccdmask function. [#414, #432]

4.2.2 Other Changes and Additions

- ccdprocs core functions now explicitly add HIERARCH cards. [#359, #399, #413]
- combine now accepts a dtype argument which is passed to Combiner.__init__. [#391, #392]
- Removed CaseInsensitiveOrderedDict because it is not used in the current code base. [#428]

4.2.3 Bug Fixes

- The default dtype of the `combine`-result doesn't depend on the dtype of the first CCDData anymore. This also corrects the memory consumption calculation. [#391, #392]
- `ccd_process` now copies the meta of the input when subtracting the master bias. [#404]
- Fixed `combine` with `CCDData` objects using `StdDevUncertainty` as uncertainty. [#416, #424]
- `ccds` generator from `ImageFileCollection` now uses the full path to the file when calling `fits_ccddata_reader`. [#421 #422]

4.3 1.1.0 (2016-08-01)

4.3.1 New Features

- Add an additional combination method, `clip_extrema`, that drops the highest and/or lowest pixels in an image stack. [#356, #358]

4.3.2 Other Changes and Additions

- `cosmicray_lacosmic` default `satlevel` changed from 65536 to 65535. [#347]
- Auto-identify files with extension `fts` as FITS files. [#355, #364]
- Raise more explicit exception if unit of uncalibrated image and master do not match in `subtract_bias` or `subtract_dark`. [#361, #366]
- Updated the `Combiner` class so that it could process images with >2 dimensions. [#340, #375]

4.3.3 Bug Fixes

- `Combiner` creates plain array uncertainties when using``average_combine`` or `median_combine`. [#351]
- `flat_correct` does not properly scale uncertainty in the flat. [#345, #363]
- Error message in weights setter fixed. [#376]

4.4 1.0.1 (2016-03-15)

The 1.0.1 release was a release to fix some minor packaging issues.

4.5 1.0.0 (2016-03-15)

4.5.1 General

- ccdproc has now the following requirements:
  - Python 2.7 or 3.4 or later.
  - astropy 1.0 or later

  - numpy 1.9 or later
  - scipy
  - astroscrappy
  - reproject

4.5.2 New Features

- Add a WCS setter for `CCDData`. [#256]
- Allow user to set the function used for uncertainty calculation in `average_combine` and `median_combine`. [#258]
- Add a new keyword to ImageFileCollection.files_filtered to return the full path to a file [#275]
- Added ccd_process for handling multiple steps. [#211]
- CCDData.write now writes multi-extension-FITS files. The mask and uncertainty are saved as extensions if these attributes were set. The name of the extensions can be altered with the parameters `hdu_mask` (default extension name `'MASK'`) and `hdu_uncertainty` (default `'UNCERT'`). CCDData.read can read these files and has the same optional parameters. [#302]

4.5.3 Other Changes and Additions

- Issue warning if there are no FITS images in an `ImageFileCollection`. [#246]
- The overscan_axis argument in subtract_overscan can now be set to None, to let subtract_overscan provide a best guess for the axis. [#263]
- Add support for wildcard and reversed FITS style slicing. [#265]
- When reading a FITS file with CCDData.read, if no data exists in the primary hdu, the resultant header object is a combination of the header information in the primary hdu and the first hdu with data. [#271]
- Changed cosmicray_lacosmic to use astroscrappy for cleaning cosmic rays. [#272]
- CCDData arithmetic with number/Quantity now preserves any existing WCS. [#278]
- Update astropy_helpers to 1.1.1. [#287]
- Drop support for Python 2.6. [#300]
- The `add_keyword` parameter now has a default of `True`, to be more explicit. [#310]
- Return name of file instead of full path in `ImageFileCollection` generators. [#315]

4.5.4 Bug Fixes

- Adding/Subtracting a CCDData instance with a Quantity with a different unit produced wrong results. [#291]
- The uncertainty resulting when combining CCDData will be divided by the square root of the number of combined pixel [#309]
- Improve documentation for read/write methods on `CCDData` [#320]
- Add correct path separator when returning full path from `ImageFileCollection.files_filtered`. [#325]

4.6 0.3.3 (2015-10-24)

4.6.1 New Features

- add a `sort` method to ImageFileCollection [#274]

4.6.2 Other Changes and Additions

- Opt in to new container-based builds on travis. [#227]
- Update astropy_helpers to 1.0.5. [#245]

4.6.3 Bug Fixes

- Ensure that creating a WCS from a header that contains list-like keywords (e.g. `BLANK` or `HISTORY`) succeeds. [#229, #231]

4.7 0.3.2 (never released)

There was no 0.3.2 release because of a packaging error.

4.8 0.3.1 (2015-05-12)

4.8.1 New Features

- Add CCDData generator for ImageCollection [#405]

4.8.2 Other Changes and Additions

- Add extensive tests to ensure `ccdproc` functions do not modify the input data. [#208]
- Remove red-box warning about API stability from docs. [#210]
- Support astropy 1.0.5, which made changes to `NDData`. [#242]

4.8.3 Bug Fixes

- Make `subtract_overscan` act on a copy of the input data. [#206]
- Overscan subtraction failed on non-square images if the overscan axis was the first index, `0`. [#240, #244]

4.9 0.3.0 (2015-03-17)

4.9.1 New Features

- When reading in a FITS file, the extension to be used can be specified. If it is not and there is no data in the primary extension, the first extension with data will be used.
- Set wcs attribute when reading from a FITS file that contains WCS keywords and write WCS keywords to header when converting to an HDU. [#195]

4.9.2 Other Changes and Additions

- Updated CCDData to use the new version of NDDATA in astropy v1.0. This breaks backward compatibility with earlier versions of astropy.

4.9.3 Bug Fixes

- Ensure `dtype` of combined images matches the `dtype` of the `Combiner` object. [#189]

4.10 0.2.2 (2014-11-05)

4.10.1 New Features

- Add dtype argument to `ccdproc.Combiner` to help control memory use [#178]

4.10.2 Other Changes and Additions

- Added Changes to the docs [#183]

4.10.3 Bug Fixes

- Allow the unit string "adu" to be upper or lower case in a FITS header [#182]

4.11 0.2.1 (2014-09-09)

4.11.1 New Features

- Add a unit directly from BUNIT if it is available in the FITS header [#169]

4.11.2 Other Changes and Additions

- Relaxed the requirements on what the metadata must be. It can be anything dict-like, e.g. an astropy.io.fits.Header, a python dict, an OrderedDict or some custom object created by the user. [#167]

4.11.3 Bug Fixes

- Fixed a new-style formating issue in the logging [#170]

4.12 0.2 (2014-07-28)

- Initial release.

CHAPTER 5

Licenses

5.1 Ccdproc License

Ccdproc is licensed under a 3-clause BSD style license:

Copyright (c) 2011-2017, Astropy-ccdproc Developers All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Astropy Team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliography

[R56] 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

[R66] McCully, C., 2014, "Astro-SCRAPPY", https://github.com/astropy/astroscrappy

[R23] image.imcombine help text. http://stsdas.stsci.edu/cgi-bin/gethelp.cgi?imcombine

Python Module Index

C

Index

T

U

V

W