ccdproc Documentation

Release 1.0.0

Steve Crawford and Matt Craig

March 15, 2016

Contents

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

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
- 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, simple 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 = ccdproc.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. For more details see

To make applying the same operations to a set of files in a directory easier, use an ImageFileCollection. It constructs, given a directory, an 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   # ignores the mask
array([ 2.,  4.,  6.])
```

You can force conversion to a numpy array with:

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

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

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

You can also write directly to a FITS file:

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

Masks and flags

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

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

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

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")
>>> result.header
OrderedDict()
```

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

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

Min/max clipping

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

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

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

Sigma clipping

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

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

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

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

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

A

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].

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.

[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

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. 2000 through 2031) and then subtracted row-by-row from the image. The `median` argument can be used to median combine instead.

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

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

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

Trim an image

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

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

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

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

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

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

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. These can be run together as:

```
>>> ccd = ccdproc.CCDData(img, unit=u.adu)
>>> nccd = ccdproc.ccd_process(ccd, oscan='[1:10,1:100]',
...                            trim='[10:100, 1:100]',
...                            error=True, gain=2.0*u.electron/u.adu,
...                            readnoise = 5*u.electron)
```

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 :ref:reprojection for more details.

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.

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='*')
```

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 or data

Three 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` and `data`.

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

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

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 bb
ccd_process(ccd[, oscan, trim, error, ...])	Perform basic processing on ccd data.
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.
fits_ccddata_reader(filename[, hdu, unit, ...])	Generate a CCDData object from a FITS file.
fits_ccddata_writer(ccd_data, filename[, ...])	Write CCDData object to FITS file.
flat_correct(ccd, flat[, min_value, add_keyword])	Correct the image for flat fielding.
gain_correct(ccd, gain[, gain_unit, add_keyword])	Correct the gain in the image.
rebin(ccd, newshape)	Rebin an array to have a new shape.
sigma_func(arr[, 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 th

background_deviation_box

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

Determine the background deviation with a box size of bbox. The algorithm steps through the image and

calculates the deviation within each box. It returns an array with the pixels in each box filled with the deviation value.

Parameters

data : ndarray or MaskedArray

Data to measure background deviation

bbox : int

Box size for calculating background deviation

Returns

background : ndarray or MaskedArray

An array with the measured background deviation in each pixel

Raises

ValueError

A value error is raised if bbox is less than 1

background_deviation_filter

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

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

Parameters

data : ndarray

Data to measure background deviation

bbox : int

Box size for calculating background deviation

Returns

background : ndarray or MaskedArray

An array with the measured background deviation in each pixel

Raises

ValueError

A value error is raised if bbox is less than 1

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*, *add_keyword=True*)

Perform basic processing on ccd data.

The following steps can be included: * overscan correction * trimming of the image * create deviation frame * gain correction * add a mask to the data * subtraction of master bias * subtraction of a dark frame * correction of flat field

The task returns a processed ccdproc.CCDData object.

Parameters

ccd: ‘~ccdproc.CCDData‘

Frame to be reduced

oscan: None, str, or, ‘~ccdproc.ccddata.CCDData‘

For no overscan correction, set to None. Otherwise proivde 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.

trim: None or str

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

error: boolean

If True, create an uncertainty array for ccd

master_bias: None or ‘~ccdproc.CCDData‘

A master bias frame to be subtracted from ccd.

dark_frame: None or ‘~ccdproc.CCDData‘

A dark frame to be subtracted from the ccd.

master_flat: None or ‘~ccdproc.CCDData‘

A master flat frame to be divided into ccd.

bad_pixel_mask: None or ‘~numpy.ndarray‘

A bad pixel mask for the data. The bad pixel mask should be in given such that bad pixels havea value of 1 and good pixels a value of 0.

gain: None or ‘~astropy.Quantity‘

Gain value to multiple the image by to convert to electrons

readnoise: None or ‘~astropy.Quantity‘

Read noise for the observations. The read noise should be in electrons.

oscan_median : bool, optional

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

oscan_model : `Model`, optional

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

min_value : None or float

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

dark_exposure : `Quantity`

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

data_exposure : `Quantity`

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

exposure_key : str or `Keyword`

Name of key in image metadata that contains exposure time.

exposure_unit : `Unit`

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

dark_scale: boolean

If True, scale the dark frame by the exposure times

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, ga
```

combine

ccdproc.**combine**(*img_list*, *output_file=None*, *method=u'average'*, *weights=None*, *scale=None*, *mem_limit=16000000000.0*, *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>*, ***ccdkwargs*)

Convenience function for combining multiple images

Parameters

img_list : `list`, 'string'

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

output_file: 'string', optional

Optional output fits filename to which the final output can be directly written.

method: 'string' (default average)

Method to combine images.
'average' : To combine by calculating average 'median' : To combine by calculating median

weights: '~numpy.ndarray', 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.

scale : function or array-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 (default 16e9)

Maximum memory which should be used while combining (in bytes).

minmax_clip : Boolean (default False)

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

Parameters below are valid only when minmax_clip is set to True.

minmax_clip_min: None, float
All pixels with values below minmax_clip_min will be masked.

minmax_clip_max: None or float
All pixels with values above minmax_clip_max will be masked.

sigma_clip : Boolean (default False)

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.

Parameters below are valid only when sigma_clip is set to True.

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

sigma_clip_high_thresh
[positive float or None] Threshold for rejecting pixels that deviate above the baseline value. If None, no rejection will be done based on sigma_clip_high_thresh.

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

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

****ccdkwargs: Other keyword arguments for CCD Object's fits 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=65536.0*, *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 iden-

tifying pixels based on a variation of the Laplacian edge detection. The algorithm is an implementation of the code describe in van Dokkum (2001) :ref:*[R1]* as implemented by McCully (2014) *[R2]*. If you use this algorithm, please cite these two works.

Parameters

ccd: ‘~ccdproc.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 lapacian-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 of 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: 65536.0.

niter : int, optional

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

sepmed : boolean, 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 : {‘median’, ‘medmask’, ‘meanmask’, ‘idw’}, optional

Set which clean algorithm is used:n ‘median’: An umasked 5x5 median filtern ‘medmask’: A masked 5x5 median filtern ‘meanmask’: A masked 5x5 mean filtern ‘idw’: A masked 5x5 inverse distance weighted interpolationn Default: “meanmask”.

fsmode : {‘median’, ‘convolve’}, optional

Method to build the fine structure image:n '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 : {'gauss', 'gaussx', 'gaussy', 'moffat'}, 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 : float numpy array, 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 : boolean, optional

Print to the screen or not. Default: False.

{log}

Returns

nccd : `CCDData` or `ndarray`

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

crmask : `ndarray`

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

Notes

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

References

[R1], *[R2]*

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 or numpy.ndarray or numpy.MaskedArary

Data to have cosmic ray cleaned

thresh : float

Threshold for detecting cosmic rays

error_image : None, float, or ndarray

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

mbox : int

Median box for detecting cosmic rays

gbox : int

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

rbox : int

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

{log}

Returns

nccd : CCDData or ndarray

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

nccd : ndarray

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

Notes

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

Examples

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

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

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

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

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

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

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, optional

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

readnoise : Quantity

Read noise per pixel.

add_keyword : str, Keyword or dict-like, optional

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

fits_ccddata_reader

ccdproc.**fits_ccddata_reader**(*filename*, *hdu=0*, *unit=None*, *hdu_uncertainty=u'UNCERT'*, *hdu_mask=u'MASK'*, *hdu_flags=None*, ***kwd*)

Generate a CCDData object from a FITS file.

Parameters

filename : str

Name of fits file.

hdu : int, optional

FITS extension from which CCDData should be initialized. If zero and and no data in the primary extention, it will search for the first extension with data. The header will be added to the primary header.

unit : astropy.units.Unit, optional

Units of the image data. If this argument is provided and there is a unit for the image in the FITS header (the keyword `BUNIT` is used as the unit, if present), this argument is used for the unit.

hdu_uncertainty : str or None, optional

FITS extension from which the uncertainty should be initialized. If the extension does not exist the uncertainty of the CCDData is `None`. Default is `'UNCERT'`.

hdu_mask : str or None, optional

FITS extension from which the mask should be initialized. If the extension does not exist the mask of the CCDData is `None`. Default is `'MASK'`.

hdu_flags : str or None, optional

Currently not implemented. Default is `None`.

kwd :

Any additional keyword parameters are passed through to the FITS reader in `astropy.io.fits`; see Notes for additional discussion.

Notes

FITS files that contained scaled data (e.g. unsigned integer images) will be scaled and the keywords used to manage scaled data in `astropy.io.fits` are disabled.

fits_ccddata_writer

ccdproc.**fits_ccddata_writer**(*ccd_data*, *filename*, *hdu_mask=u'MASK'*, *hdu_uncertainty=u'UNCERT'*, *hdu_flags=None*, ***kwd*)

Write CCDData object to FITS file.

Parameters

filename : str

Name of file

hdu_mask, hdu_uncertainty, hdu_flags : str or None, optional

If it is a string append this attribute to the HDUList as `ImageHDU` with the string as extension name. Flags are not supported at this time. If `None` this attribute is not appended. Default is `'MASK'` for mask, `'UNCERT'` for uncertainty and `None` for flags.

kwd :

All additional keywords are passed to `astropy.io.fits`

Raises

ValueError

- If `self.mask` is set but not a `ndarray`.
- If `self.uncertainty` is set but not a `StdDevUncertainty`.
- If `self.uncertainty` is set but has another unit then `self.data`.

NotImplementedError

Saving flags is not supported.

flat_correct

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

Correct the image for flat fielding.

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

Parameters

ccd : `CCDData`

Data to be flatfield corrected

flat : `CCDData`

Flatfield to apply to the data

min_value : None or float

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

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

Item(s) to add to metadata of result. Set to 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, optional

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

add_keyword : str, Keyword or dict-like, optional

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

rebin

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

Rebin an array to have a new shape.

Parameters

data : CCDData or ndarray

Data to rebin

newshape : tuple

Tuple containing the new shape for the array

Returns

output : CCDData or ndarray

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

Raises

TypeError

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

ValueError

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

Notes

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

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

Examples

Given an array that is 100x100,

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

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

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

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 ndarray

Array whose deviation is to be calculated.

axis : None or int or tuple of ints, optional

Axis or axes along which the function is performed. If None (the default) 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.

Returns

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

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

data_exposure : Quantity

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

exposure_time : str or Keyword

Name of key in image metadata that contains exposure time.

exposure_unit : Unit

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

scale: boolean

If True, scale the dark frame by the exposure times

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

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

overscan_axis : None, 0 or 1, optional

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

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).

fits_section : str

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

median : bool, optional

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

model : `Model`, optional

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

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

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

```
>>> 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 flatfield corrected

transform_func : function

Function to be used to transform the data

kwargs: dict

Dictionary of arguments to be used by the transform_func.

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to 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 uncertainy data but it will only transform the data. This will not properly handle uncertainties that arise due to correlation between the pixels.

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

Examples

Given an array that is 100x100,

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

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

```
>>> from scipy.ndimage.interpolation import shift
>>> 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

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

add_keyword : str, Keyword or dict-like, optional

Item(s) to add to metadata of result. Set to 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 : astropy.wcs.WCS object

WCS onto which all images should be projected.

target_shape : two element list-like, optional

Shape of the output image. If omitted, defaults to the shape of the input image.

order : str, optional

Interpolation order for re-projection. Must be one of: + 'nearest-neighbor' + 'bilinear' + 'biquadratic' + 'bicubic'

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

CCDData(*args, **kwd)	A class describing basic CCD data
Combiner(ccd_list[, dtype])	A class for combining CCDData objects.
ImageFileCollection([location, keywords, ...])	Representation of a collection of image files.
Keyword(name[, unit, value])	

CCDData

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

Bases: astropy.nddata.NDDataArray

A class describing basic CCD data

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

Parameters

data : `ndarray` or `CCDData`

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

uncertainty : `StdDevUncertainty` or `ndarray`,

optional Uncertainties on the data.

mask : `ndarray`, optional

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

flags : `ndarray` or `FlagCollection`, optional

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

wcs : `WCS` object, optional

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

meta : `dict`-like object, optional

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

unit : `Unit` instance or str, optional

The units of the data.

Raises

ValueError

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

Notes

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

For example:

```
>>> from ccdproc import CCDData
>>> import numpy as np
>>> x = CCDData([1,2,3], unit='adu')
>>> np.asarray(x)
array([1, 2, 3])
```

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

```
>>> from ccdproc import CCDData
>>> from matplotlib import pyplot as plt
>>> x = CCDData([[1,2,3], [4,5,6]], unit='adu')
>>> plt.imshow(x)
```

Methods

read(*args, **kwargs)	Classmethod to create an CCDData instance based on a FITS file. This method uses fits_ccddata_reader() with the provided parameters.
write(*args, **kwargs)	Writes the contents of the CCDData instance into a new FITS file. This method uses fits_ccddata_writer() with the provided parameters.

Attributes Summary

data
dtype
header
meta
shape
size
uncertainty
unit
wcs

Methods Summary

add(other[, compare_wcs])	
copy()	Return a copy of the CCDData object.
divide(other[, compare_wcs])	
multiply(other[, compare_wcs])	
subtract(other[, compare_wcs])	
to_hdu([hdu_mask, hdu_uncertainty, hdu_flags])	Creates an HDUList object from a CCDData object.

Attributes Documentation

data

dtype

header

meta

shape

size

uncertainty

unit

wcs

Methods Documentation

add(*other*, *compare_wcs=u'first_found'*)

copy()
Return a copy of the CCDData object.

divide(*other*, *compare_wcs=u'first_found'*)

multiply(*other*, *compare_wcs=u'first_found'*)

subtract(*other*, *compare_wcs=u'first_found'*)

to_hdu(*hdu_mask=u'MASK'*, *hdu_uncertainty=u'UNCERT'*, *hdu_flags=None*)
Creates an HDUList object from a CCDData object.

Parameters
hdu_mask, hdu_uncertainty, hdu_flags : str or None, optional

If it is a string append this attribute to the HDUList as `ImageHDU` with the string as extension name. Flags are not supported at this time. If `None` this attribute is not appended. Default is `'MASK'` for mask, `'UNCERT'` for uncertainty and `None` for flags.

Returns
hdulist : astropy.io.fits.HDUList object

Raises
ValueError

- If `self.mask` is set but not a `ndarray`.
- If `self.uncertainty` is set but not a `StdDevUncertainty`.
- If `self.uncertainty` is set but has another unit then `self.data`.

NotImplementedError

Saving flags is not supported.

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 : 'numpy dtype'

Allows user to set dtype.

Raises

TypeError

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

Notes

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

```
>>> 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.
`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 threshol

Attributes Documentation

dtype

scaling
Scaling factor used in combining images.

Parameters
scale : function or array-like or None, optional

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

weights
Weights used when combining the `CCDData` objects.

Parameters
weight_values : `ndarray`

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 `average`.

scale_to : float, optional

Scaling factor used in the average combined image. If given, it overrides `CCDData.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.

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 `masked_array`. Default is to use `numpy.ma.median` to calculate median.

scale_to : float, optional

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

uncertainty_func : function, optional

Function to calculate uncertainty. Defaults to `ccdproc.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 : None or float

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

max_clip : None or float

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

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

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

high_thresh : positive float or None

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

func : function

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

dev_func : function

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

ImageFileCollection

class ccdproc.**ImageFileCollection**(*location=None*, *keywords=None*, *info_file=None*)
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, optional

path to directory containing FITS files

keywords : list of str or '*', 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.

info_file : str, 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.

Raises

ValueError

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

Attributes Summary

files	list of str, Unfiltered list of FITS files in location.
keywords	list of str, Keywords currently in the summary table.
location	str, Path name to directory containing FITS files
summary	
summary_info	Deprecated – use summary instead – astropy.table.Table of values of FITS keywords for files in the collection.

Methods Summary

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 HDU 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

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

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.

location
str, Path name to directory containing FITS files

summary

summary_info
Deprecated – use summary instead – astropy.table.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.

Methods Documentation

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

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`

save_location : str

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.

overwrite : bool

If `True`, overwrite input FITS files.

do_not_scale_image_data : bool

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

return_fname : bool, default is False

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.

kwd : dict

Any additional keywords are used to filter the items returned; see 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.

`**kwd` is list of keywords and values the files must have.

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

The value '*' represents any value.
A missing keyword is indicated by value ''

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

NOTE: Value comparison is case *insensitive* for strings.

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

Generator that yields each HDU 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

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`

save_location : str

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 HDU with `save_location` set.

overwrite : bool

If `True`, overwrite input FITS files.

do_not_scale_image_data : bool

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

return_fname : bool, default is False

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.

kwd : dict

Any additional keywords are used to filter the items returned; see Examples for details.

Returns

astropy.io.fits.HDU

If `return_fname` is `False`, yield the next HDU in the collection

(astropy.io.fits.HDU, str)

If `return_fname` is `True`, yield a tuple of (HDU, `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

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`

save_location : str

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.

overwrite : bool

If `True`, overwrite input FITS files.

do_not_scale_image_data : bool

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

return_fname : bool, default is False

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.

kwd : dict

Any additional keywords are used to filter the items returned; see 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=None*)

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 or 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.

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

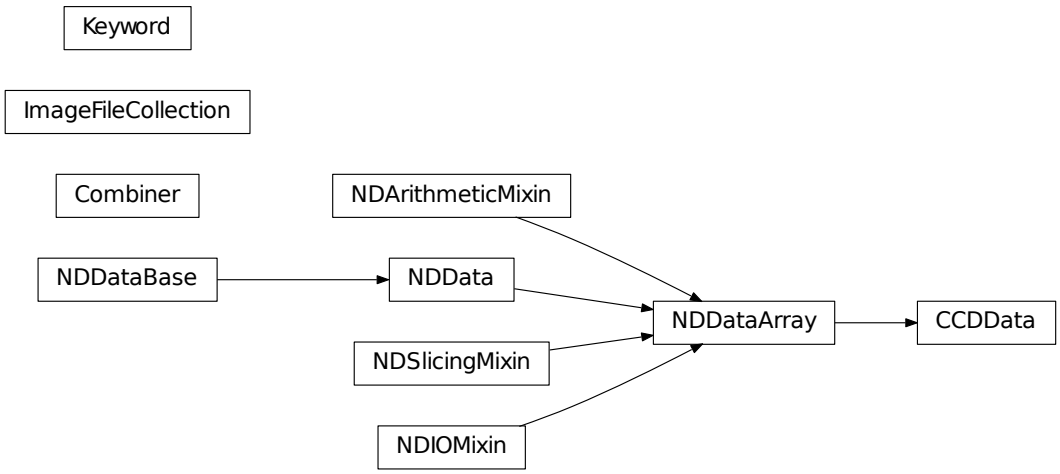
Methods Documentation

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



CHAPTER 3

Contributors

3.1 Authors and Credits

3.1.1 ccdproc Project Contributors

Project Coordinators

- Matt Craig (@mwcraig)
- Steve Crawford (@crawfordsm)

Alphabetical list of contributors

- Christoph Deil (@cdeil)
- Carlos Gomez (@carlgogo)
- Hans Moritz Günther (@hamogu)
- Forrest Gasdia (@EP-Guy)
- Nathan Heidt (@heidtha)
- Anthony Horton (@AnthonyHorton)
- Jennifer Karr (@JenniferKarr)
- Stefan Nelson (@pulsestaysconstant)
- Joe Philip Ninan (@indiajoe)
- Punyaslok Pattnaik (@Punyaslok)
- Evert Rol (@evertrol)
- William Schoenell (@wschoenell)
- Michael Seifert (@MSeifert04)
- Sourav Singh (@souravsingh)
- Brigitta Sipocz (@bsipocz)
- Connor Stotts (@stottsco)
- Ole Streicher (@olebole)

- Erik Tollerud (@eteq)
- Nathan Walker (@walkerna22)

(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 0.4.0 (2016-03-15)

4.1.1 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

4.1.2 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]

4.1.3 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]

4.1.4 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]

4.2 0.3.3 (2015-10-24)

4.2.1 New Features

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

4.2.2 Other Changes and Additions

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

4.2.3 Bug Fixes

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

4.3 0.3.2 (never released)

There was no 0.3.2 release because of a packaging error.

4.4 0.3.1 (2015-05-12)

4.4.1 New Features

4.4.2 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]

4.4.3 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]

4.5 0.3.0 (2015-03-17)

4.5.1 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]

4.5.2 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.

4.5.3 Bug Fixes

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

4.6 0.2.2 (2014-11-05)

4.6.1 New Features

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

4.6.2 Other Changes and Additions

- Added Changes to the docs [#183]

4.6.3 Bug Fixes

- Allow the unit string “adu” to be upper or lower case in a FIS header [#182]

4.7 0.2.1 (2014-09-09)

4.7.1 New Features

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

4.7.2 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]

4.7.3 Bug Fixes

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

4.8 0.2 (2014-07-28)

- Initial release.

Bibliography

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

[R2] McCully, C., 2014, "Astro-SCRAPPY", https://github.com/astropy/astroscrappy

C

U

V

W