VIS Documentation

Release 0.1

Sami-Matias Niemi

July 03, 2012

CONTENTS

Author Sami-Matias Niemi

Contact smn2@mssl.ucl.ac.uk

version 1.0

This Python package provides subpackages and methods related to the visible instrument (VIS) on board the Euclid satellite. The subpackages include methods to e.g. generate object catalogues, simulate VIS images, study radiation damage effects and fit new trap species, reduce and analyse data, and to include instrumental characteristics such as readout noise and CTI to "pristine" images generated with e.g. GREAT10 photon shooting code.

CHAPTER

ONE

CREATING OBJECT CATALOGS

The *sources* subpackage contains a script to generate object catalogs with random x and y positions for stars and galaxies. The magnitudes of stars and galaxies are drawn from input distributions that are based on observations. As the number of stars depends on the galactic latitude, the script allows the user to use three different (30, 60, 90 degrees) angles when generating the magnitude distribution for stars (see the example plot below).

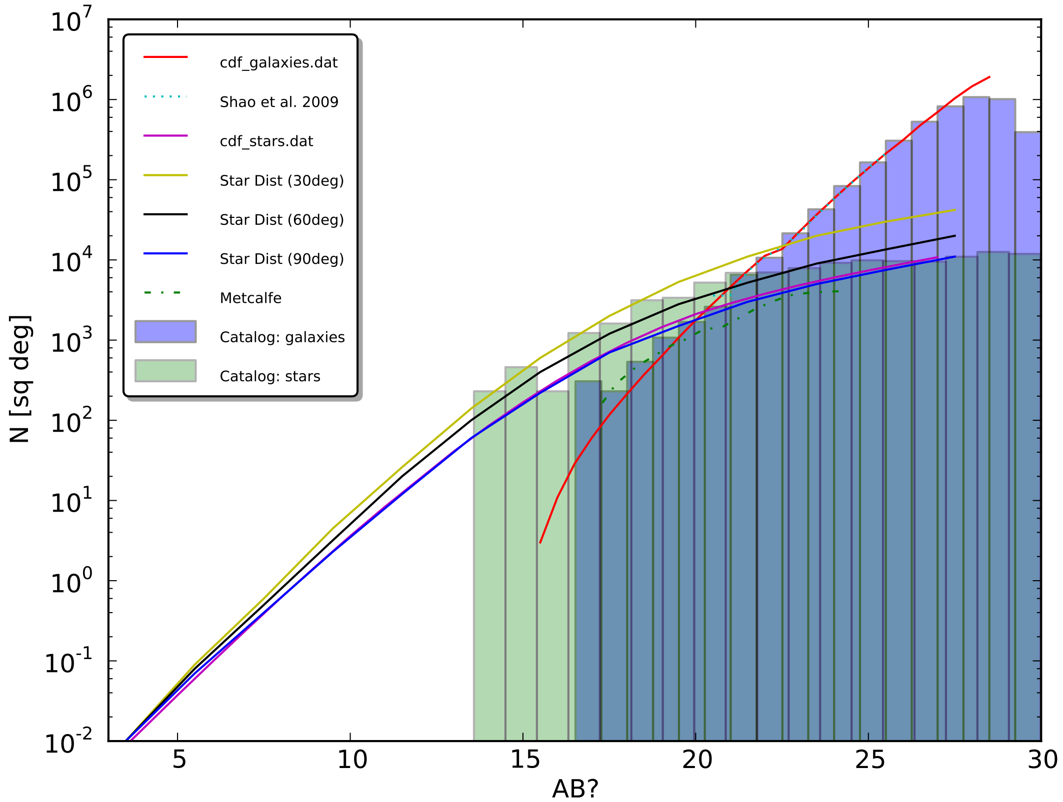

Figure 1.1: An example showing star and galaxy number counts in a source catalog suitable for VIS simulator. The solid lines show observations while the histograms show the distributions in the output catalogues.

For the Python code documentation, please see:

1.1 Generating Object Catalogue

This simple script can be used to generate an object catalogue that can then be used as an input for the VIS simulator.

To run:

```
python createObjectCatalogue.py
```

Please note that the script requires files from the data folder. Thus, you should place the script to an empty directory and either copy or link to the data directory.

requires NumPy

requires SciPy

requires matplotlib

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

sources.createObjectCatalogue.**drawFromCumulativeDistributionFunction**(*cpdf*, *x*, *number*)

Draw a number of random x values from a cumulative distribution function.

Parameters

- **cpdf** (*numpy array*) – cumulative distribution function
- **x** (*numpy array*) – values of the abscissa
- **number** (*int*) – number of draws

Returns randomly drawn x value

Return type ndarray

sources.createObjectCatalogue.**generateCatalog**(***kwargs*)

Generate a catalogue of stars and galaxies that follow realistic number density functions.

Parameters deg – galactic latitude, either 30, 60, 90

sources.createObjectCatalogue.**plotDistributionFunction**(*datax*, *datay*, *fitx*, *fity*, *output*)

Generates a simple plot showing the observed data points and the fit that was generated based on these data.

CHAPTER

TWO

GENERATING SIMULATED IMAGES

The *simulator* subpackage contains scripts to generate simulated VIS images. Two different methods of generating mock images is provided. One which takes observed images (say from HST) as an input and another in which analytical profiles are used for galaxies. The former code is custom made while the latter relies hevily on IRAF's artdata package and mkobjects task.

The VIS reference simulator is the custom made with real observed galaxies as an input. The IRAF based simulator can be used, for example, to train algorithms to derive elliptiticy of an object. For more detailed documentation, please see:

2.1 Simulation tools

2.1.1 The Euclid Visible Instrument Image Simulator

This file contains an image simulator for the Euclid VISible instrument.

The approximate sequence of events in the simulator is as follows:

1. Read in a configuration file, which defines for example, detector characteristics (bias, dark and readout noise, gain, plate scale and pixel scale, oversampling factor, exposure time etc.).
2. Read in another file containing charge trap definitions (for CTI modelling).
3. Read in a file defining the cosmic rays (trail lengths and cumulative distributions).
4. Read in CCD offset information, displace the image, and modify the output file name to contain the CCD and quadrant information (note that VIS has a focal plane of 6 x 6 detectors).
5. Read in source list and determine the number of different object types.
6. Read in a file which assigns data to a given object index.
7. Load the PSF model (a single 2D map or field dependent maps).

8. Generate a finemap (oversampled image) for each object type. If an object is a 2D image then calculate the shape tensor to be used for size scaling. Each type of an object is then placed onto its own finely sampled finemap.
9. Loop over the number of exposures to co-add and for each object in the object catalog:
   - determine the number of electrons an object should have by scaling the object's magnitude with the given zeropoint and exposure time.
   - determine whether the object lands on to the detector or not and if it is a star or an extended source (i.e. a galaxy).
   - if object is extended determine the size (using a size-magnitude relation) and scale counts, convolve with the PSF, and finally overlay onto the detector according to its position.
   - if object is a star, scale counts according to the derived scaling (first step), and finally overlay onto the detector according to its position.
10. Apply a multiplicative flat-field map [optional].
11. Add a charge injection line (horizontal and/or vertical) [optional].
12. Add cosmic ray streaks onto the CCD with random positions but known distribution [optional].
13. Add photon (Poisson) noise and constant dark current to the pixel grid [optional].
14. Add cosmetic defects from an input file [optional].
15. Add pre- and overscan regions in the serial direction [optional].
16. Apply the CDM03 radiation damage model [optional].
17. Add readout noise selected from a Gaussian distribution [optional].
18. Convert from electrons to ADUs using the given gain factor.
19. Add a given bias level and discretise the counts (16bit).
20. Finally the generated image is converted to a FITS file and saved to the working directory.

Warning: This code is still work in progress and new features are being added. Testing has been performed, but bugs may be lurking in corners, so be careful.

Dependencies

This script depends on the following packages.

requires PyFITS (tested with 3.0.3)

requires NumPy (tested with 1.6.2)

requires SciPy (tested with 0.10.0)

requires vissim-python package

Testing

Before trying to run the code, please make sure that you have compiled the cdm03.f90 Fortran code using f2py (f2py -c -m cdm03 cdm03.f90). For testing, please run the SCIENCE section from the test.config as follows:

```
python simulator.py -c data/test.config -s TESTSCIENCE -d
```

This will produce an image representing VIS lower left (0th) quadrant. Because noise and cosmic rays are randomised one cannot directly compare the science outputs but we must rely on the outputs that are free from random effects.

In the data subdirectory there is a file called "nonoisenocrQ0_00_00testscience.fits", which is the comparison image without any noise or cosmic rays. To test the functionality, please divide your nonoise and no cosmic ray track output image with the on in the data folder. This should lead to a uniformly unity image or at least very close given some numerical rounding uncertainties.

Benchmark using the SCIENCE section of the test.config input file:

```
Galaxy: 26753/26753 intscale=199.421150298 size=0.0329649781423
6772 objects were place on the detector

real    8m28.781s
user    8m24.792s
sys     0m1.361s
```

These numbers have been obtained with my laptop (2.2 GHz Intel Core i7) with 64-bit Python 2.7.2 installation.

Change Log

version 0.9

Version and change logs:

```
0.1: pre-development backbone.
0.4: first version with most pieces together.
0.5: this version has all the basic features present, but not fully tested.
0.6: implemented pre/overscan, fixed a bug when an object was getting close to t
     image it was not overlaid correctly. Included multiplicative flat fielding
0.7: implemented bleeding.
0.8: cleaned up the code and improved documentation. Fix a bug related to checki
     Improved the information that is being written to the FITS header.
0.9: fixed a problem with the CTI model swapping Q1 with Q2. Fixed a bug that ca
     be identical for each quadrant even though Q1 and 3 needs the regions to be
```

Future Work

Todo

1. start using oversampled PSF
2. implement spatially variable PSF
3. test that the cosmic rays are correctly implemented
4. implement CCD offsets (for focal plane simulations)
5. implement additive flat fielding (now only multiplicative pixel non-uniform effect is being simulated)
6. implement a Gaussian random draw from the size-magnitude distribution
7. move the cosmic ray track information file definitions to the input file (now hardcoded in the method)
8. double check the centering of objects, now the exact centering is not interpolated
9. charge injection line positions are now hardcoded to the code, read from the config file
10. CTI model values are not included to the FITS header
11. implement optional dithered offsets

Contact Information

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

class `simulator.simulator.`**`VISsimulator`**(*configfile*, *debug*, *section='SCIENCE'*)

Euclid Visible Instrument Image Simulator

The image that is being build is in:

```
self.image
```

Parameters

- **configfile** (*string*) – name of the configuration file
- **debug** (*boolean*) – debugging mode on/off
- **section** (*str*) – name of the section of the configuration file to process

`addChargeInjection`()

Add either horizontal or vertical charge injection line to the image.

`addCosmicRays`()

Add cosmic rays to the arrays based on a power-law intensity distribution for tracks.

Cosmic ray properties (such as location and angle) are chosen from random Uniform distribution.

addObjects()

Add objects from the object list to the CCD image (self.image).

Scale the object's brightness based on its magnitude. The size of the object is scaled using the brightness.

For fainter objects the convolved results are on the same sized grid as the input. However, for large values (intscale > 1e5) a box with sharp edges would be visible if convolution were not done on the full scale. The problem is that the values outside the input grid are not very reliable, so these objects should be treated with caution.

addPreOverScans()

Add pre- and overscan regions to the self.image. These areas are added only in the serial direction. Because the 1st and 3rd quadrant are read out in to a different serial direction than the nominal orientation, in these images the regions are mirrored.

The size of prescan and overscan regions are defined by the prescx and overscx keywords, respectively.

applyBias()

Adds a bias level to the image being constructed.

The value of bias is read from the configure file and stored in the information dictionary (key bias).

applyBleeding()

Apply bleeding along the CCD columns if the number of electrons in a pixel exceeds the full-well capacity.

Bleeding is modelled in the parallel direction only, because the CCD273s are assumed not to bleed in serial direction.

Returns None

applyCosmetics()

Apply cosmetic defects described in the input file.

Warning: This method does not work if the input file has exactly one line.

applyFlatfield()

Applies multiplicative and/or additive flat field.

Because the pixel-to-pixel non-uniformity effect (i.e. multiplicative) flat fielding takes place before CTI and other effects, the flat field file must be the same size as the pixels that see the sky. Thus, in case of a single quadrant (x, y) = (2048, 2066).

Note: The additive flat fielding effect has not been included yet.

applyNoise()

Apply dark current, the cosmic background, and Poisson noise. Scales dark and background with the exposure time.

Additionally saves the image without noise to a FITS file.

applyRadiationDamage()

Applies CDM03 radiation model to the image being constructed.

See Also:

Class :*CDM03*

applyReadoutNoise()

Applies readout noise to the image being constructed.

The noise is drawn from a Normal (Gaussian) distribution. Mean = 0.0, and std = sqrt(readout noise).

configure()

Configures the simulator with input information and creates and empty array to which the final image will be build on.

discretise(*max=65535*)

Converts a floating point image array (self.image) to an integer array with max values defined by the argument max.

Parameters **max** (*float*) – maximum value the the integer array may contain [default 65k]

Returns None

electrons2ADU()

Convert from electrons to ADUs using the value read from the configuration file.

generateFinemaps()

Generates finely sampled images of the input data.

> **Warning:** This should be rewritten. Now a direct conversion from FORTRAN, and thus not probably very effective. Assumes the PSF sampling for other finemaps.

readObjectlist()

Reads object list using numpy.loadtxt, determines the number of object types, and finds the file that corresponds to a given object type.

This method also displaces the object coordinates based on the quadrant and the CCD to be simulated.

If even a single object type does not have a corresponding input then this method forces the program to exit.

readPSFs()

Reads in a PSF from a FITS file.

Note: at the moment this method supports only a single PSF file.

simulate()

Create a single simulated image of a quadrant defined by the configuration file. Will do all steps defined in the config file sequentially.

Returns None

`writeOutputs()`

Writes out a FITS file using PyFITS and converts the image array to 16bit unsigned integer as appropriate for VIS.

Updates header with the input values and flags used during simulation.

2.1.2 Generating Mock Objects with IRAF

This script provides a class that can be used to generate objects such as galaxies using IRAF.

requires PyRAF

requires PyFITS

requires NumPy

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.1

class `simulator.generateGalaxies.generateFakeData(log, **kwargs)`

Generates an image frame with stars and galaxies using IRAF's artdata.

`addObjects(inputlist='galaxies.dat')`

Add object(s) from inputlist to the output image.

Parameters **inputlist** (*str*) – name of the input list

`createGalaxylist(ngalaxies=150, output='galaxies.dat')`

Generates an ascii file with uniform random x and y positions. The magnitudes of galaxies are taken from an isotropic and homogeneous power-law distribution.

The output ascii file contains the following columns: xc yc magnitude model radius ar pa <save>

Parameters

- **ngalaxies** (*int*) – number of galaxies to include
- **output** (*str*) – name of the output ascii file

`createStarlist(nstars=20, output='stars.dat')`

Generates an ascii file with uniform random x and y positions. The magnitudes of stars are taken from an isotropic and homogeneous power-law distribution.

The output ascii file contains the following columns: xc yc magnitude

Parameters

- **nstars** (*int*) – number of stars to include
- **output** (*str*) – name of the output ascii file

maskCrazyValues(*filename=None*)

For some reason mkobjects sometimes adds crazy values to an image. This method tries to remove those values and set them to more reasonable ones. The values > 65k are set to the median of the image.

Parameters filename (*str*) – name of the input file to modify [default = self.settings['output']]

Returns None

runAll(*nostars=True*)

Run all methods sequentially.

Figure 2.1: An example image generated with the reference simulator. The image shows a part of a single CCD.

CHAPTER

THREE

INSTRUMENT CHARACTERISTICS

The *postproc* subpackage contains methods related to either generating a CCD mosaics from simulated data that is in quadrants like the VIS reference simulator produces or including instrument characteristics to simulated images that contain only Poisson noise and background. For more detailed documentation of the Python classes, please see:

3.1 Postprocessing tools

3.1.1 Inserting instrument characteristics

This file provides a class to insert instrument specific features to a simulated image. Supports multiprocessing.

Note: The output images will be compressed with gzip to save disk space.

Warning: The logging module used does not work well with multiprocessing, but starts to write multiple entries after a while. This should be fixed.

requires PyFITS

requires NumPy

requires CDM03 (FORTRAN code, f2py -c -m cdm03 cdm03.f90)

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.8

class `postproc.postprocessing.`**`PostProcessing`**(*values*, *work_queue*, *result_queue*, *seed*)

Euclid Visible Instrument postprocessing class. This class allows to add radiation damage (as defined by the CDM03 model) and add readout noise to a simulated image.

`applyLinearCorrection`(*image*)

Applies a linear correction after one forward readout through the CDM03 model.

Bristow & Alexov (2003) algorithm further developed for HST data processing by Massey, Rhodes et al.

Parameters **image** (*ndarray*) – radiation damaged image

Returns corrected image after single forward readout

Return type ndarray

`applyRadiationDamage`(*data*, *iquadrant=0*)

Apply radian damage based on FORTRAN CDM03 model. The method assumes that input data covers only a single quadrant defined by the iquadrant integer.

Parameters

- **data** (*ndarray*) – imaging data to which the CDM03 model will be applied to.
- **iquandrant** (*int*) – number of the quadrant to process

cdm03 - Function signature: sout = cdm03(sinp,iflip,jflip,dob,rdose,in_nt,in_sigma,in_tr,[xdim,y

Required arguments: sinp : input rank-2 array('f') with bounds (xdim,ydim) iflip : input int jflip : input int dob : input float rdose : input float in_nt : input rank-1 array('d') with bounds (zdim) in_sigma : input rank-1 array('d') with bounds (zdim) in_tr : input rank-1 array('d') with bounds (zdim)

Optional arguments: xdim := shape(sinp,0) input int ydim := shape(sinp,1) input int zdim := len(in_nt) input int

Return objects: sout : rank-2 array('f') with bounds (xdim,ydim)

Note: Because Python/NumPy arrays are different row/column based, one needs to be extra careful here. NumPy.asfortranarray will be called to get an array laid out in Fortran order in memory. Before returning the array will be laid out in memory in C-style (row-major order).

Returns image that has been run through the CDM03 model

Return type ndarray

`applyReadoutNoise`(*data*)

Applies readout noise. The noise is drawn from a Normal (Gaussian) distribution. Mean = 0.0, and std = sqrt(readout).

Parameters **data** (*ndarray*) – input data to which the readout noise will be added to

Returns updated data, noise image

Return type dict

compressAndRemoveFile(*filename*)
This method compresses the given file using gzip and removes the parent from the file system.

Parameters **filename** (*str*) – name of the file to be compressed

Returns None

cutoutRegion(*data*)
Cuts out a region from the imaging data. The cutout region is specified by xstart/stop and ystart/stop that are read out from the self.values dictionary. Also checks if there are values that are above the given cutoff value and sets those pixels to a max value (default=33e3).

Parameters

- **data** (*ndarray*) – image array
- **max** (*int or float*) – maximum allowed value [default = 33e3]

Returns cut out image from the original data

Return type ndarray

discretisetoADUs(*data*)
Convert floating point arrays to integer arrays and convert to ADUs. Adds bias level after converting to ADUs.

Parameters **data** (*ndarray*) – data to be discretised to.

Returns discretised array in ADUs

Return type ndarray

generateCTImap(*CTIed*, *originalData*)
Calculates a map showing the CTI effect. This map is being generated by dividing radiation damaged image with the original data.

Parameters

- **CTIed** (*ndarray*) – Radiation damaged image
- **originalData** (*ndarray*) – Original image before any radiation damage

Returns CTI map (ratio of radiation damaged image and original data)

Return type ndarray

loadFITS(*filename*, *ext=0*)
Loads data from a given FITS file and extension.

Parameters

- **filename** (*str*) – name of the FITS file
- **ext** (*int*) – FITS header extension [default=0]

Returns data, FITS header, xsize, ysize

Return type dict

radiateFullCCD (*fullCCD*, *quads=(0, 1, 2, 3)*, *xsize=2048*, *ysize=2066*)
This routine allows the whole CCD to be run through a radiation damage mode. The routine takes into account the fact that the amplifiers are in the corners of the CCD. The routine assumes that the CCD is using four amplifiers.

Parameters

- **fullCCD** (*ndarray*) – image of containing the whole CCD
- **quads** (*list*) – quadrants, numbered from lower left

Returns radiation damaged image

Return type ndarray

run ()
This is the method that will be called when multiprocessing.

writeFITSfile (*data*, *output*, *unsigned16bit=True*)
Write out FITS files using PyFITS.

Parameters

- **data** (*ndarray*) – data to write to a FITS file
- **output** (*string*) – name of the output file
- **unsigned16bit** (*bool*) – whether to scale the data using bzero=32768

Returns None

3.1.2 Generating a mosaic

This file contains a class to create a single VIS CCD image from separate files one for each quadrant.

requires NumPy

requires PyFITS

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

To execute:

```
python tileCCD.py -f 'Q*science.fits' -e 1
```

where -f argument defines the input files to be tiled and the -e argument marks the FITS extension from which the imaging data are being read.

version 0.3

class postproc.tileCCD.**tileCCD** (*inputs*, *log*)
Class to create a single VIS CCD image from separate quadrants files.

readData()

Reads in data from all the input files and the header from the first file. Input files are taken from the input dictionary given when class was initiated.

Subtracts the pre- and overscan regions if these were simulated.

runAll()

Wrapper to perform all class methods.

tileCCD(*xsize=2048*, *ysize=2066*)

Tiles quadrants to form a single CCD image.

Assume that the input file naming convention is Qx_CCDX_CCDY_name.fits.

Parameters

- **xsize** (*int*) – length of a quadrant in column direction
- **ysize** (*int*) – length of a quadrant in row direction

Returns image array of size (ysize*2, xsize*2)

Return type dnarray

writeFITSfile(*data=None*, *unsigned16bit=True*)

Write out FITS files using PyFITS.

Parameters

- **data** (*ndarray*) – data to write to a FITS file, if None use self.data
- **unsigned16bit** (*bool*) – whether to scale the data using bzero=32768

Returns None

CHAPTER

FOUR

DATA REDUCTION

The *reduction* subpackage contains a simple script to reduce VIS data. For more detailed documentation of the classes, please see:

4.1 Data reduction tools

4.1.1 VIS Data Reduction and Processing

This simple script can be used to reduce (simulated) VIS data.

Does the following steps:

```
1 Bias correction
2 Flat fielding
3 CTI correction (conversion to electrons and back to ADUs)
```

To Run:

```
python reduceVISdata.py -i VISCCD.fits -b superBiasVIS.fits -f SuperFlatField.fi
```

requires PyFITS

requires NumPy

requires CDM03 (FORTRAN code, f2py -c -m cdm03 cdm03.f90)

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.4

Todo

1. FITS extension should probably be read from the command line
2. implement background/sky subtraction

---

class `reduction.reduceVISdata.`**`reduceVISdata`**(*values*, *log*)

Simple class to reduce VIS data.

`applyCTICorrection`()

Applies a CTI correction in electrons using CDM03 CTI model. Converts the data to electrons using the gain value given in self.values. The number of forward reads is defined by self.values['order'] parameter.

Bristow & Alexov (2003) algorithm further developed for HST data processing by Massey, Rhodes et al.

There is probably an excess of .copy() calls here, but I had some problems when calling the Fortran code so I added them for now.

`flatfield`()

Take into account pixel-to-pixel non-uniformity through multipicative flat fielding.

`subtractBias`()

Simply subtracts self.bias from the input data.

`writeFITSfile`()

Write out FITS files using PyFITS.

CHAPTER

FIVE

DATA ANALYSIS

The *analysis* subpackage contains classes and scripts related to data analysis. A simple source finder and shape measuring classes are provided together with a wrapper to analyse reduced VIS data. For more detailed documentation of the classes, please see:

5.1 VIS data analysis tools

5.1.1 Object finding and measuring ellipticity

This script provides a class that can be used to analyse VIS data. One can either choose to use a Python based source finding algorithm or give a SExtractor catalog as an input. If an input catalog is provided then the program assumes that X_IMAGE and Y_IMAGE columns are present in the input file.

requires PyFITS

requires NumPy

requires matplotlib

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.2

class `analysis.analyse.`**`analyseVISdata`** (*filename*, *log*, ***kwargs*)

Simple class that can be used to find objects and measure their ellipticities.

One can either choose to use a Python based source finding algorithm or give a SExtractor catalog as an input. If an input catalog is provided then the program assumes that X_IMAGE and Y_IMAGE columns are present in the input file.

Parameters

- **filename** (*string*) – name of the FITS file to be analysed.
- **log** (*instance*) – logger
- **kwargs** (*dict*) – additional keyword arguments

Settings dictionary contains all parameter values needed for source finding and analysis.

doAll()

Run all class methods sequentially.

findSources()

Finds sources from data that has been read in when the class was initiated. Saves results such as x and y coordinates of the objects to self.sources. x and y coordinates are also available directly in self.x and self.y.

measureEllipticity()

Measures ellipticity for all objects with coordinates (self.x, self.y).

Ellipticity is measured using Guassian weighted quadrupole moments. See shape.py and especially the ShapeMeasurement class for more details.

plotEllipticityDistribution()

Creates a simple plot showing the derived ellipticity distribution.

readSources()

Reads in a list of sources from an external file. This method assumes that the input source file is in SExtractor format. Input catalog is saves to self.sources. x and y coordinates are also available directly in self.x and self.y.

writeResults()

Outputs results to an ascii file defined in self.settings. This ascii file is in SExtractor format and contains the following columns:

```
1. X coordinate
2. Y coordinate
3. ellipticity
4. R_{2}
```

5.1.2 Measuring a shape of an object

Simple class to measure quadrupole moments and ellipticity of an object.

requires NumPy

requres PyFITS

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.2

class analysis.shape.**shapeMeasurement**(*data*, *log*, ***kwargs*)

Provides methods to measure the shape of an object.

Parameters

- **data** (*ndarray*) – name of the FITS file to be analysed.

- **log** (*instance*) – logger
- **kwargs** (*dict*) – additional keyword arguments

Settings dictionary contains all parameter values needed.

`circular2DGaussian` (*x*, *y*, *sigma*)
Create a circular symmetric Gaussian centered on x, y.

Parameters

- **x** (*float*) – x coordinate of the centre
- **y** (*float*) – y coordinate of the centre
- **sigma** (*float*) – standard deviation of the Gaussian, note that sigma_x = sigma_y = sigma

Returns circular Gaussian 2D profile and x and y mesh grid

Return type dict

`measureRefinedEllipticity` ()
Derive a refined iterated ellipticity measurement for a given object.

By default ellipticity is defined in terms of the Gaussian weighted quadrupole moments. If self.settings['weigted'] is False then no weighting scheme is used. The number of iterations is defined in self.settings['iterations'].

:return centroids, ellipticity (including projected e1 and e2), and R2 :rtype: dict

`quadrupoles` (*image*)
Derive quadrupole moments and ellipticity from the input image.

Parameters **image** (*ndarray*) – input image data

Returns quadrupoles, centroid, and ellipticity (also the projected components e1, e2)

Return type dict

`writeFITS` (*data*, *output*)
Write out a FITS file using PyFITS.

Parameters

- **data** (*ndarray*) – data to write to a FITS file
- **output** (*string*) – name of the output file

Returns None

5.1.3 Object finding

Simple source finder that can be used to find objects from astronomical images.

reqiures NumPy

requires SciPy

requires matplotlib

author Sami-Matias Niemi

contact smn2@mssl.ucl.ac.uk

version 0.2

class `analysis.sourceFinder.`**`sourceFinder`**(*image*, *log*, ***kwargs*)

This class provides methods for source finding.

Parameters

- **image** (*numpy.ndarray*) – 2D image array
- **log** (*instance*) – logger
- **kwargs** (*dictionary*) – additional keyword arguments

`cleanSample()`

Cleans up small connected components and large structures.

`find()`

Find all pixels above the median pixel after smoothing with a Gaussian filter.

Note: maybe one should use mode instead of median?

`generateOutput()`

Outputs the found positions to an ascii and a DS9 reg file.

Returns None

`getCenterOfMass()`

Finds the center-of-mass for all objects using numpy.ndimage.center_of_mass method.

Note: these positions are zero indexed!

Returns xposition, yposition, center-of-masses

Return type list

`getContours()`

Derive contours using the diskStructure function.

`getFluxes()`

Derive fluxes or counts.

`getSizes()`

Derives sizes for each object.

`plot()`

Generates a diagnostic plot.

Returns None

runAll()

Performs all steps of source finding at one go.

Returns source finding results such as positions, sizes, fluxes, etc.

Return type dictionary

The *data* subfolder contains the supporting data, such as cosmic ray distributions, cosmetics maps, flat fielding files, PSFs, and an example configuration file.

CHAPTER

SIX

CHARGE TRANSFER INEFFICIENCY

The *fitting* subpackage contains a simple script that can be used to fit trap species so that the Charge Transfer Inefficiency (CTI) trails forming behind charge injection lines agree with measured data.

6.1 Fortran code for CTI

The *fortran* folder contains a CDM03 CTI model Fortran code. For speed the CDM03 model has been written in Fortran because it contains several nested loops. One can use f2py to compile the code to a format that can be imported from Python.

CHAPTER

SEVEN

SUPPORTING METHODS AND FILES

7.1 Objects

A few postage stamps showing observed galaxies have been placed to the *objects* directory. These FITS files can be used for, e.g., testing the shape measurement code.

7.2 Code

The *support* subpackage contains some support classes and methods related to generating log files and read in data.

CHAPTER

EIGHT

PHOTOMETRIC ACCURACY

The reference simulator code has been tested against photometric accuracy (without aperture correction). A simulated image was generated with the reference simulator, sources were identified and photometry performed using SExtractor, and finally the extracted magnitudes were compared against the input catalog. The following figure shows that the photometric accuracy with realistic noise and the end-of-life radiation damage is around 0.1 mag without aperture correction. Please note, however, that the derived magnitudes are based on a single 565 second exposure. Because of this the faint galaxies have low signal-to-noise ratio and therefore the derived magnitudes are inaccurate.

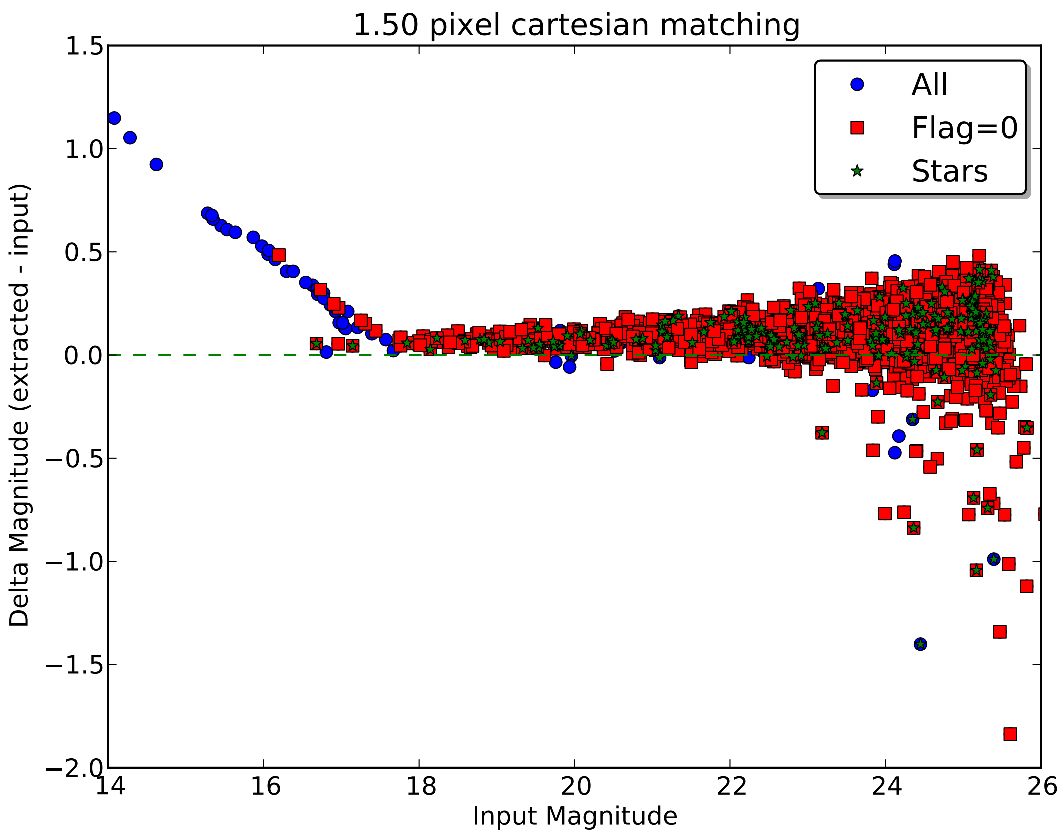

Figure 8.1: Example showing the recovered photometry from a reference simulator image with realistic noise and end-of-life radiation damage, but without aperture correction. The offset is around 0.1mag.

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

A

C

D

E

F

G

L

M

P

Q

R

S

T

V

W