lfd Documentation

Release 1.0

Dino Bektesevic

Jan 18, 2019

Contents:

CHAPTER 1

Top level LFD overview

1.1 Setup

Certain modules depend on certain environmental variables to figure out where the data lives and how it's organized. It is recommended that conventional SDSS file naming and directory structures are followed. SDSS generally follows structure shown below:

```
boss
├ photo
│ └── redux
│ ├── photoRunAll-dr{DR}.par
│ └── runList.par
└ photoObj
      ├── 301
      │ └── [RUN]
      │ └── [CAMCOL]
      │ └── photoObj-{PADDED_RUN}-CAMCOL-{PADDED_FIELD}.fits
      └── frames
            └── [RUN]
                  └── [CAMCOL]
                        └── frame-{FILTER}-{PADDED_RUN}-CAMCOL-{PADDED_FIELD}.fits
```

Where the *RUN* marks the integer designation of the desired run (i.e. 94, 2888) and *PADDED_RUN* marks the integer run designation padded to 6 digits by preceding zeros (i.e. 000094, 002888). The slight difference between the *PADDED_RUN* and *PADDED_FIELD* is the fact that the field is only padded to 4 digits The square bracketted items (i.e. *[RUN]* means a directory named 94 or 2888). Finally there are two files requried to detect linear features. One is the image itself - these are the *frame* files (i.e. frame-i-000094-1-0168.fits) and the other required file is the catalog of all detected sources on that image - these are the *photoObj* files.

The files *photoRunAll-dr{DR}.par* and *runList.par* store the parameters of all the runs that SDSS made up to the DR (Data Release number). This is determined by the Data Release number from which the data was selected and downloaded from - f.e. for DR10 data release the appropriate file is *photoRunAll-dr10.par*. These files are downloadable from the SDSS servers.

The SDSS directory structure can be overriden by providing the required environmental variables. The environmental variables point to different folders and subfolders of this structure. They are listed and explained in the following table:

Env. Var. Name	Points to
BOSS	Top level *boss* directory
PHOTO_OBJ	*boss/photoObj* directory
PHOTO_REDUX	*boss/photo/redux* directory

These variables can be set through the terminal prior, or after, importing the *lfd* library by using the following commands:

```
export $ENVVAR=/some/path
```

or:

```
setenv $ENVVAR=/some/path
```

depending on the shell and OS used. Or they can be set by using top-level functionality in the *lfd* module.

lfd.**setup_detecttrails**(*bosspath=None*, *photoobjpath=None*, *photoreduxpath=None*, *debugpath=None*)

Sets up the environmental paths BOSS, BOSS_PHOTOOBJ, PHOTO_REDUX and DEBUG_PATH required for detecttrails package. Defining custom BOSS, BOSS_PHOTOOBJ and PHOTO_REDUX variables allows data not to follow the SDSS convention.

Parameters

- **bosspath** (*str*) – The path to which BOSS environmental variable will be set to. This is the "boss" top directory. Default is set to ~/Desktop/boss
- **photoobjpath** (*str*) – The path to which BOSS_PHOTOOBJ env. var. will be set to. Default is $BOSS/photoObj.
- **photoreduxpath** (*str*) – The path to which PHOTO_REDUX env. var. will be set to. By default set to $BOSS/photo/redux-
- **debugpath** (*str*) – The path to which DEBUG_PATH env. var. will be set to. The current dir by default. Used to run detecttrails module in debug mode.

lfd.**setup_createjobs**(*photoreduxpath=None*)

Sets up the environmental path of PHOTO_REDUX only! Minimum environment required for certain createjobs package functionality.

Parameters **photoreduxpath** (*str*) – The path to which PHOTO_REDUX env. var. will be set to. Defaults to '~/Desktop/boss/photoredux'.

1.2 Examples

Need to fill this out

Listed below are the packages contained in LFD and a description of their purpose:

- analysis - contains code required to create theoretical cross section profiles of defocused meteors and various plotting utilities
- createjobs - contains the code required to create PBS scripts that can be submited to a cluster using QSUB commands
- detecttrails - contains the image processing code that detects images containing linear features and outputs measured parameters
- errors - (not finished) contains functionality to parse any error messages outputed by detecttrails module whilst running on a cluster
- gui - contains the GUI interface to createjobs module and an image browser designed for faster and easier verification of results
- results - contains the required functionality to collate and parse the results outputed by detecttrails module.

CHAPTER 2

Linear Feature detection

2.1 Detection parameters

The detection execution parameters can be viewed/changed by accessing the params dictionaries of the `DetectTrails` class.

```
import lfd.detecttrails as detect
foo = detect.DetecTrails(run=2888, camcol=1, filter='i', field=139)
foo.params_bright
foo.params_dim
foo.params_removestars
```

A special *debug* key can be set to *True* for any of the individual steps of the algorithm which will produce a very verbose output to stdout as well as save snapshots of the processed image in intermediate steps to a location pointed to by *DEBUG_PATH* environmental variable.

```
import lfd.detecttrails as detect
foo = detect.DetecTrails(run=2888, debug=True)
```

or, individually

```
foo.params_removestars["debug"] = True
foo.params_bright["debug"] = True
foo.params_dim["debug"] = True
```

The verbose output contains printed statements of acceptance tests such as average angles, difference in image positions between fitted lines etc. Additionally a number of images that help identifying the detection limiting factors are saved as well. For the bright detection step the following images are saved:

- equBRIGHT - equalized 8bit int image with known objects masked out
- dilateBRIGHT - equalized and dilated image
- contoursBRIGHT - rectangles that pass the tests are drawn on the image
- boxhoughBRIGHT - rectangles with corresponding fitted lines (set 1)

- equhoughBRIGHT - processed image with second set of fitted lines (set 2)

while the dim detection step in debug mode produces the following set of images:

- equDIM - equlized, 8bit img, removed objects, increased brightness
- erodedDIM - equalized and eroded image without known objects
- openedDIM - equalized, eroded and dilated image without known objects
- contoursDIM - accepted rectangles
- equhoughDIM - processed img with with the first set of fitted lines
- boxhoughDIM - rectangles image with the second set of fitted lines

For the full list of parameters, their types and descriptions see the following tables sections. To get a better understanding of detection parameters contoursMode, contoursMethod, minAreaRectMinLen and lwTresh see *fit_minAreaRect()*. For nlinesInSet, thetaTresh, linesetTresh see *check_theta()*. For removestars params see doc *remove_stars()*.

2.2 Examples

The setup is mandatory for this package as it needs access to data, so either set the required environmental variables before running python interpreter or use one of the provided functions.

If SDSS convention is followed but it is sufficient to point to boss only.

```
import lfd
lfd.setup_detecttrails()
lfd.setup_detecttrails("~/path/boss")
lfd.setup_detecttrails("~/boss", photoreduxpath="/foo/bar")
```

The above examples are appropriate in the following cases:

- boss directory is located at *~/Desktop/boss* and SDSS conventions apply
- boss directory is not in its default position and SDSS conventions apply
- is a common use case when the run metadata (the .par files) are stored elsewhere but image data still follows SDSS convention.

This functionality is replicated in the `lfd.detecttrails.setup()` function but is not as practical as using the *lfd.setup_detecttrails()* because it does not declare default values or assumes SDSS convention. Using either allows a complete departure from SDSS conventions

```
lfd.detecttrails.setup("~/boss", "~/boss/photoObj", "~/boss/photo/redux")
```

After that instantiate the *DetectTrails* class, target desired data, and run `lfd.detecttrails.DetectTrails.process()` method. It's practical to import the module into the namespace. At least 1 data targeting keyword has to be given.

```
import lfd.detecttrails as detect
foo = detect.DetecTrails(filter="i")
foo = detect.DetectTrails(camcol=1)
foo = detect.DetectTrails(run=2888)
foo = detect.DetectTrails(run=2888, camcol=1)
foo = detect.DetectTrails(run=2888, camcol=1, filter='i')
foo = detect.DetectTrails(run=2888, camcol=1, filter='i', field=139)
foo.process()
```

The above examples will process different data. Listed in the same order:

- Process filter 'i' of all fields and filters of all runs
- Process camcol 1 of all fields and filters of all runs
- Process all fields, camcols and filters of run 2888
- Process all filters and fields of run 2888 but only for its first camera column
- Process only the 'i' filter of camcol 1 of the run 2888 for all fields
- Process this single specific field

Caution: The size of the selected dataset reduces from top to bottom of the list. The first example (selecting a single filter) will try to process dataset 1/5th the size of the entire SDSS dataset!

2.3 Table of removestars parameters

Parameter Name	Type	Default value	Description
pixscale	float	0.396	SDSS pixel scale in arcsec/pixel.
defaultxy	int	20	Half of the length of squares drawn over stars. Used when there's no Petrosian radius availible or if length/2 of square sides are larger than maxxy. In pixels.
maxxy	int	60	Max allowed half-length of square sides. Max covered area is 120x120
filter_caps	dict	**{u:22., g:22.2,** r:22.2, i:21.3, z:20.5}	Objects dimmer than filter_caps will not be removed (filter dependent)
magcount	int	3	max allowed number of filters in which magnitude difference is larger than maxmagdiff
maxmagdiff	float	3	Min number of votes required in Hough space to be considered a valid line
debug	bool	False	see above.

2.4 Table of bright parameters

Parameter Name	Type	Default value	Description
dilateKernel	array	np.ones((4,4))	Dilation kernel
contoursMode	CV2 const	cv2.RETR_LIST	All edges returned, see OpenCV of findContours for details.
contoursMethod	CV2 const	cv2.CHAIN_APPROX_NONE	All edge points returned, see OpenCV of findContours
minAreaRectMinlen	int	1	Once contours have been found minimal area rectangles are fitted. Length of sides are determined. If determined. If either side is shorter than this value, in pixels, the detection is dismissed.
lwTresh	int	5	ratio that longer/shorter rectangle side has to satisfy to be considered a possibly valid detection
houghMethod	int	20	Min number of votes required in Hough space to be considered a valid line
nlinesInSet	int	3	Hough transform returns a list of all possible lines. Only nlinesInSet top voted for lines are taken into consideration for futher checks. This is called a "set" of lines. There are 2 sets in total.
thetaTresh	float	0.15	if within a single set the dispe- rsion of maximal-minimal theta is larger than thetaTresh, detection is dismissed as False. In radians
linesetTresh	float	0.15	if both line sets pass thetaTresh, but their respective average thetas are larger than this, detection is dismissed. In radians.
dro	int	25	if r averages of both Hough line sets are larger than dro then detection is dismissed as false.
debug	bool	False	see above.

2.5 Table of dim parameters

Parameter Name	Type	Default value	Description
minFlux	float	0.02	for bright, all negative values are set to 0. For dim, all values below minFlux treshold are set to zero.
addFlux	float	0.5	value added to all remaining pixels with values still above 0.
erodeKernel	array	np.ones((3,3))	Erosion kernel
dilateKernel	array	np.ones((9,9))	Dilation kernel
contoursMode	CV2 const	cv2.RETR_LIST	All edges returned, see OpenCV of findContours for details.
contoursMethod	CV2 const	cv2.CHAIN_APPROX_NONE	All edge points returned, see OpenCV of findContours
minAreaRectMinlen	int	1	Once contours have been found minimal area rectangles are fitted. Length of sides are determined. If determined. If either side is shorter than this value, in pixels, the detection is dismissed.
lwTresh	int	5	ratio that longer/shorter rectangle side has to satisfy to be considered a possibly valid detection
houghMethod	int	20	Min number of votes required in Hough space to be considered a valid line
nlinesInSet	int	3	Hough transform returns a list of all possible lines. Only nlinesInSet top voted for lines are taken into consideration for futher checks. This is called a "set" of lines. There are 2 sets in total.
thetaTresh	float	0.15	if within a single set the dispe- rsion of maximal-minimal theta is larger than thetaTresh, detection is dismissed as False. In radians
linesetTresh	float	0.15	if both line sets pass thetaTresh, but their respective average thetas are larger than this, detection is dismissed. In radians.
dro	int	20	if r averages of both Hough line sets are larger than dro then detection is dismissed as false.
debug	bool	False	see above.

2.6 Module: removestars

Removestars module contains all the required functionality to read a catalog source and blot out objects in the image. Currently only SDSS photoObj files are supported as a catalog source. Old code supporting CSV catalog sources was left as an example how its relatively simple to add a different catalog source.

`lfd.detecttrails.removestars.`**`read_photoObj`**(*path_to_photoOBJ*)

Function that reads photoObj headers and returns following lists:

Returns

- **row** (*list(dict)*) – y coordinate of an object on the image. Each entry is a dictionary where keys are the filter designations and values the coordinates
- **col** (*list(dict)*) – x coordinate of an object on the image. Each entry is a dictionary where keys are the filter designations and values the coordinates

- **psfMag** (*list(dict)*) – Each entry is a dictionary where keys are the filter designations and values the magnitudes. See: http://www.sdss3.org/dr10/algorithms/magnitudes.php#mag_psf
- **nObserve** (*list(int)*) – Number of times that object was imaged by SDSS.
- **objctype** (*list(int)*) – SDSS type identifier, see cas.sdss.org/dr7/de/help/browser/enum.asp?n=ObjType
- **petro90** (*list(dict)*) – This is the radius, in arcsec, from the center of the object that contains 90% of its Petrosian flux. Each entry is a dictionary where keys are the filter designations and values the radii. See: http://www.sdss3.org/dr10/algorithms/magnitudes.php#mag_petro and http://data.sdss3.org/datamodel/files/BOSS_PHOTOOBJ/RERUN/RUN/CAMCOL/photoObj.html

Parameters **path_to_photoOBJ** (*str*) – string type system path to a photoObj*.fits file

lfd.detecttrails.removestars.**remove_stars**(*img*, *_run*, *_camcol*, *_filter*, *_field*, *defaultxy*, *filter_caps*, *maxxy*, *pixscale*, *magcount*, *maxmagdiff*, *debug*)

Removes all stars found in coordinate file from a given image by "blottings" out black squares at objects coordinats.

Size of the square is, if possible, determined according to its petrosian magnitude. The square is up/down-scaled by a scaling factor converting the object size from arcsec to pixel size. If the calculated radius is less than zero or bigger than maximal allowed size, a default square size is used.

Aditionally to determining size of the blocked out square, function tries to discriminate actual sources from false ones.

Squares will only be drawn:

- if there is a valid psfMag value.
- for those psfMag values that are under a certain user-set cap.
- if differences in measured magnitudes of all filters are less maxmagdiff in more than or equal to magcount filters. F.e. for maxmagdiff=5 magcount=3

won't pass: [2,8,8,8,8], [2,2,8,8,8], [1,1,7,8,9]
will pass: [5,6,7,8,9], [3,6,7,8,9], [3,3,7,8,9]

Idea is that psfMag values that contain only "opposite" extreme values (very bright/very dim) are most likely a false detection, they exist in one of the filters but not in any others. Such objects are not removed.

Parameters

- **img** (*np.array*) – grayscale 8 bit image
- **_run** (*int*) – run identifier
- **_camcol** (*int*) – camera column identifier 1-6
- **_filter** (*str*) – filter identifier (u,g,r,i,z)
- **_field** (*int*) – field identifier
- **defaultxy** (*int*) – default length of half of the total length of square sides
- **filter_caps** (*dict*) – dictionary (u,g,r,i,z) that defines the limiting magnitude under which objects are not erased
- **maxxy** – maximal allowed length of the half the length of square side, int

- **pixscale** – pixel scale. 1 pixel width represents pixscale arcsec.
- **dxy** – default side dimension of drawn rectangle.
- **magcount** – maximal allowed number of filters with differences of mag. greater than maxmagdiff.
- **maxmagdiff** – maximal allowed difference between two magnitudes.

2.7 Module: detecttrails

Detecttrails module is the SDSS oriented wrapper that will call the correct order of operations on the targeted data to succesfully run LFD. The module also handles the IO operations required to succesfully store results and wraps the required functionality for easier debugging.

class lfd.detecttrails.detecttrails.**DetectTrails**(***kwargs*)

Convenience class that processes targeted SDSS frames.

Example usage

```
foo = DetectTrails(run=2888)
foo = DetectTrails(run=2888, camcol=1, filter='i')
foo = DetectTrails(run=2888, camcol=1, filter='i', field=139)
foo.process()
```

At least 1 keyword has to be sent!

See documentation for full details on detection parameters. Like results and errors file paths, detection parameters are optional too and can be set after instantiation through provided dictionaries.

```
foo.params_dim
foo.params_bright["debug"] = True
foo.params_removestars["filter_caps"]["i"] = 20
```

All errors are silenced and dumped to error file. Results are dumped to results file.

Parameters

- **run** (*int*) – run designation
- **camcol** (*int*) – camcol designation, 1 to 6
- **filter** (*str*) – filter designation, one of ugriz filters
- **field** (*int*) – field designation
- **params_dim** (*dict*) – detection parameters for detecting dim trails. See docs for details.
- **params_bright** (*dict*) – detection parameters for bright trails. See docs for details.
- **params_removestars** (*dict*) – detection parameters for tuning star removal. See docs for details.
- **debug** (*bool*) – turns on verbose and step-by-step image output visualizing the processing steps for all steps simultaneously. If $DEBUGPATH env. var. is not set errors will be raised.
- **results** (*str*) – path to file where results will be saved
- **errors** (*str*) – path to file where errors will be stored

_getRuns()

Reads runlist.par file and returns a list of all runs.

_load()
Parses the send kwargs to determine what selection users wants to process. Currently supported options are:

- run
- run-camcol
- run-filter
- run-filter-camcol
- camcol-filter
- camcol-frame
- field (full specification run-filter-camcol-frame)

_runInfo()
Reads runlist.par file and extracts startfield and endfield of a run. Runs are retrieved from self._run attribute of instance.

process()
Convenience function that runs process_field() for various inputs. Not using this function will void majority of error and exception handling in processing.

2.8 Module: processfield

Processfield is the image analysis workhorse module behind detecttrails. It contains all the functionality required to detect lines independend of the source of data, only the ingoing format and type.

Wrapping the functionality in this module for various different catalog and image sources so that the correct order of operations is ensured for different directory structures is what makes detecttrails capable of processing variety of different images.

lfd.detecttrails.processfield.**process_field_bright**(*img*, *lwTresh*, *thetaTresh*, *dilateKernel*, *contoursMode*, *contoursMethod*, *minAreaRectMinLen*, *houghMethod*, *nlinesInSet*, *lineSetTresh*, *dro*, *debug*)
Function detects bright trails in images. For parameters explanations see DetectTrails documentation for help.

1. pixels with values bellow minFlux are set to 0
2. Scale the image to 8 bit integer, 1 channel
3. histogram equalization
4. dilate to expand features
5. fit minimal area rectangles. Function aborts if no minAreaRect are found, see: fit_minAreaRect help
6. fit Hough lines on image and on found rectangles
7. compare lines, see: check_theta help
8. if evaluation passed write the results into file and return True, else returns False.

Parameters

- **img** (*np.array*) – numpy array representing gray 32 bit 1 chanel image.
- **lwTresh** (*float*) – treshold for the ratio of rectangle side lengths that has to be satistfied

- **thetatresh** (*float*) – treshold for the difference of angles, in radians, that each fitted set of lines has to satisfy
- **dilateKernel** (*np.array*) – kernel used to dilate the image
- **contoursMode** (*cv2.const*) – cv2.RETR_LIST should be used, but any return type of fitted contours supported by OpenCV is allowed
- **contoursMethod** (*cv2.const*) – cv2.CHAIN_APPROX_NONE should be used, but any return type of fitted contours supported by OpenCV is allowed
- **minAreaRectMinLen** (*int*) – treshold for minimal allowed length of fitted minimal area rectangles
- **houghMethod** (*int*) – treshold for minimum number of votes lines need to get in Hough space to be returned
- **nlinesInSet** (*int*) – number of most voted for lines that will be considered for colinearity tests
- **lineSetTresh** (*float*) – treshold for maximal allowed angle deviation between two sets of fitted lines
- **dro** (*int*) – treshold for maximal allowed distance between the average x-axis intersection coordinates of the two sets of lines

lfd.detecttrails.processfield.**process_field_dim**(*img*, *minFlux*, *addFlux*, *lwTresh*, *thetaTresh*, *erodeKernel*, *dilateKernel*, *contoursMode*, *contoursMethod*, *minAreaRectMinLen*, *houghMethod*, *nlinesInSet*, *dro*, *lineSetTresh*, *debug*)

Function detects dim trails in images. See DetectTrails documentation for more detailed explanation of parameters

1. pixels with values bellow minFlux are set to 0
2. addFlux is added to remaining pixels
3. Scale the image to 8 bit 1 chanel
4. histogram equalization
5. erode to kill noise
6. dilate to expand features that survived
7. fit minimal area rectangles. Function aborts if no minAreaRect are found, see: fit_minAreaRect help
8. fit Hough lines on image and on found rectangles
9. compare lines, see: help(check_theta)
10. if evaluation passed write the results into file and return True, else returns False.

Parameters

- **img** (*np.array*) – numpy array representing gray 32 bit 1 chanel image.
- **minFlux** (*float*) – treshold for maximal allowed pixel brightness value under which pixel values will be set to zero
- **addFlux** (*float*) – the brightness that will be added to all pixels above minFlux
- **lwTresh** (*float*) – treshold for the ratio of rectangle side lengths that has to be satistfied

- **thetatresh** (*float*) – treshold for the difference of angles, in radians, that each fitted set of lines has to satisfy
- **dilateKernel** (*np.array*) – kernel used to erode the image
- **dilateKernel** – kernel used to dilate the image
- **contoursMode** (*cv2.const*) – cv2.RETR_LIST should be used, but any return type of fitted contours supported by OpenCV is allowed
- **contoursMethod** (*cv2.const*) – cv2.CHAIN_APPROX_NONE should be used, but any return type of fitted contours supported by OpenCV is allowed
- **minAreaRectMinLen** (*int*) – treshold for minimal allowed length of fitted minimal area rectangles
- **houghMethod** (*int*) – treshold for minimum number of votes lines need to get in Hough space to be returned
- **nlinesInSet** (*int*) – number of most voted for lines that will be considered for colinearity tests
- **lineSetTresh** (*float*) – treshold for maximal allowed angle deviation between two sets of fitted lines
- **dro** (*int*) – treshold for maximal allowed distance between the average x-axis intersection coordinates of the two sets of lines

lfd.detecttrails.processfield.**setup_debug**()

Sets up the module global variables - paths to where the debug output is saved. Invoked when 'debug' key is set to True for any of the detection steps. Generally there would be no need to call this function otherwise.

lfd.detecttrails.processfield.**check_theta**(*hough1*, *hough2*, *navg*, *dro*, *thetaTresh*, *lineSetTresh*, *debug*)

Comapres colinearity between lines in each set of lines provided and between the two sets. If the lines in each set are nearly parallel and the two sets are nearly parallel and if the lines are intersecting the x axis at nearly the same point then they are nearly colinear.

> **Warning:** Function returns True when a test is satisfied - this means that the lines are not colinear.

Calculates the difference between max and min angle values of each set of fitted lines. If these diffs. are larger than thetaTresh, returns True.

```
dtheta = abs(theta.max()-theta.min())
if  dtheta2> theta_tresh:  return True
```

Difference of averages of angles in both sets are compared with linesetTresh. If the diff. is larger than linesetTresh a True is returned.

```
dtheta = abs(numpy.average(theta1-theta2))
if numpy.average(dtheta)> lineset_tresh: return True
```

If the average x axis intersection of the two line sets are not close enough True is returned

```
if abs(np.average(ro1)-np.average(ro2))>dro: return True
```

Parameters

- **hough1** (*cv2.HoughLines*) – 2D array of line parameters representing the first set of lines that will be compared. Line parameters are stored as tuples, i.e. hough1[0][i] –> tuple(ro, theta)
- **hough2** (*cv2.HoughLines*) – second set of lines to be compared
- **navg** (*int*) – number of lines that will be averaged together
- **thetaTresh** (*float*) – treshold, in radians, that each line set must not be bigger than
- **linesetTresh** (*float*) – treshold, in radians, that the difference of the two line sets should not be bigger than.
- **debug** (*bool*) – produces a verboose output of calculated values.

lfd.detecttrails.processfield.**draw_lines**(*hough*, *image*, *nlines*, *name*, *path=None*, *compression=0*, *color=(255, 0, 0)*)

Draws hough lines on a given image and saves it as a png.

Parameters

- **hough** (*cv2.HoughLines*) – 2D array of line parameters, line parameters are stored in [0][x] as tuples.
- **image** (*np.array or cv2.image*) – image will now be altered
- **nlines** (*int*) – number of lines to draw
- **name** (*str*) – name of the file without extension
- **path** (*str*) – the path will be set by default when DetectTrails debug params are set to True. Otherwise supply your own.
- **compression** (*int*) – cv2.IMWRITE_PNG_COMPRESSION parameter from 0 to 9. A higher value means a smaller size and longer compression time. Default value is 0.

lfd.detecttrails.processfield.**fit_minAreaRect**(*img*, *contoursMode*, *contoursMethod*, *minAreaRectMinLen*, *lwTresh*, *debug*)

Fits minimal area rectangles to the image. If no rectangles can be fitted it returns False. Otherwise returns True and an image with drawn rectangles.

1. finds edges using canny edge detection algorithm
2. finds all contours among the edges
3. fits a min. area rectangle to a contour only if:
   a. both sides of the rect. are longer than minAreaRectMinLen
   b. the ratio of longer vs, shorter rect. side is smaller than lwTresh

5. if no rectangles satisfying sent conditions are found function returns False and an empty (black) image

For more details on the parameters see documentation.

Parameters

- **img** (*np.array*) – numpy array representing gray 8 bit 1 chanel image.
- **lwTresh** (*float*) – ratio of longer/shorter rectangle side that needs to be satisfied
- **contoursMode** (*cv2.const*) – cv2.RETR_LIST should be used, but any of the contour return modes
- **contoursMethod** (*cv2.const*) – cv2.CHAIN_APPROX_NONE should be used, but any of the contour return modes supported by OpenCV can be used

- **minAreaRectMinLen** (*int*) – length, in pixels, of allowed shortest side to be fitted to contours

lfd.detecttrails.processfield.**dictify_hough**(*shape*, *houghVals*)
Function converts from hough line tuples (rho, theta) into a dictionary of pixel coordinates on the image.

Parameters

- **shape** (*tuple, list*) – shape (dimensions) of the image. Used in order to scale Hough-space coordinates into pixel-space coordinates
- **houghVals** (*tuple, list*) – (rho, theta) values of lines

---

Tip: Description of the algorithm's implementation, execution parameters and performance is availible in

Bektesevic & Vinkovic, 2017, Linear feature detection algorithm for astronomical surveys - I. Algorithm description

availible at arxiv.org/abs/1612.04748

This module contains all the basic functionality required to execute the linear feature detector interactively or in a sequential batch-like processing setup.

The module requires that all the required data exists and is linked to as described in the *setup section*.

Central construct of this module is the DetectTrails class which keeps track of the execution parameters and targeted data.

Broadly speaking the detection is a three step process:

- Removal of all known objects
- Detection of bright linear features
- Detection of dim linear features.

Each step is individually configurable through the *params* dictionaries of the class.

It is instructive to read the docstring of process_field() to see the steps algorithm does. For its implementation or mode details see *lfd.detecttrails.processfield* module of the package.

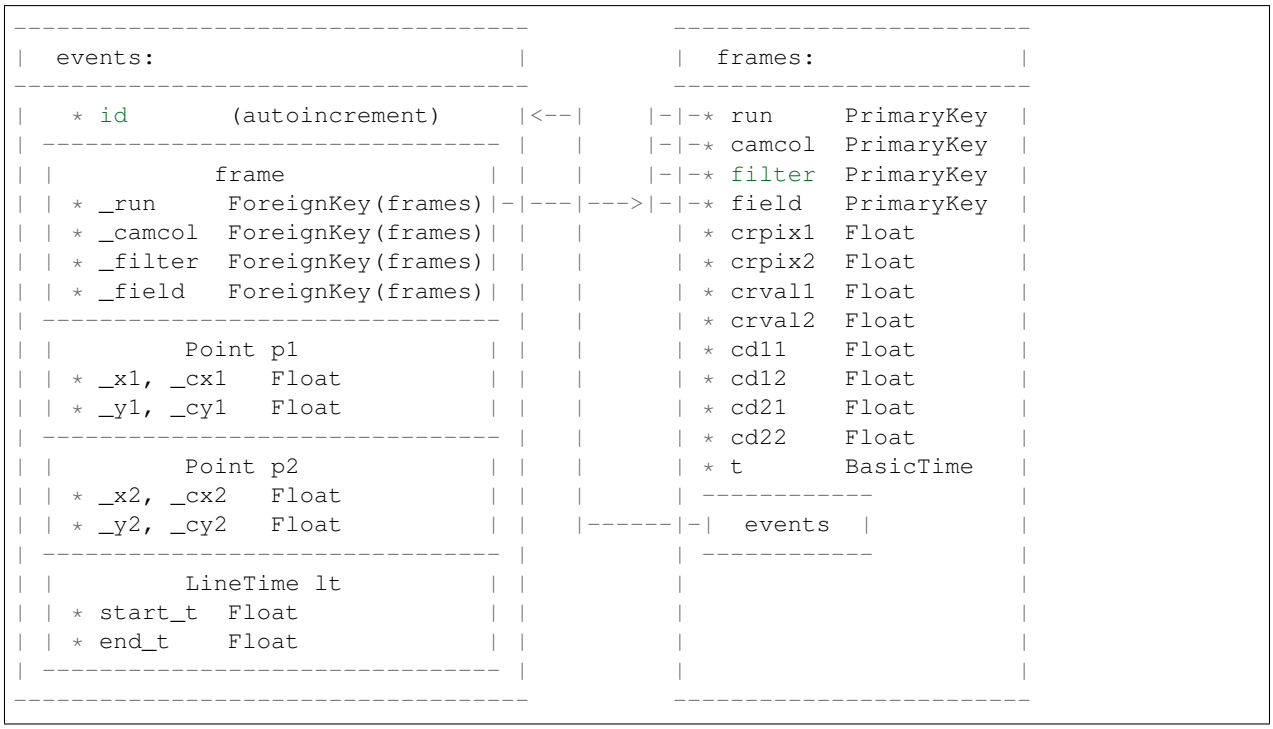
The DetectTrails is not SDSS data agnostic but the processing functionality in *lfd.detecttrails.processfield* is. That way this module can be used to run on other data given that the images are in FITS format and that the catalog for each image is also given as a header table of a FITS file. There should be one such fits catalog per image. Other catalog sources are possible but not impemented. It is instructive to see *removestars* module which should contain examples of old deprecated CSV catalog functionality.

CHAPTER 3

Handling the results

3.1 Database and setup

The underlaying database schema looks something like:



Event describes the occurence of a single line on a *Frame*. A Frame is the SDSS frame on which at least one Event (linear feature) was registered. If no Event is associated to a particular Frame, that Frame should be removed from the DB.

No event can exist without an being associated toa Frame. This constitutes a One-To-Many relationship from Frames to Events and Many-To-One in reverse:

```
Events ->|------- Frame
Frame  -|------<- Events
```

The awkward notation of the tables reflects the ORM aspects of the DB. Many of the attributes of the first table are marked with an underscore and I would advise not to access them directly. Even though a lot of effort has been put into assuring consistency of data no matter what is done to it, dealing directly with the values assumed understanding of the SDSS' CCD layout and usually comes with a loss of functionaity.

Instead many wrappers (i.e. Point, BasicTime etc.) are offered around those attributes that expand on their functionality and ensure data consistency.

Relationships 'frame' and 'events' can be accessed as attributes of the class. This allows for simpler, more object oriented access to immediatelly related objects.

Point is another special SQLAlchemy construct, a mutable composite. Point class ensures data consistency in an interactive session while managing coordinate transformations between multiple coordinate systems.

The SQL type BasicTime was implemented to allow for more powerful astronomy focused date and time management. It is essentially the SDSS modified TAI time stamp stored as a Float value in the DB but wrapped in Astropy's Time object.

By default, SQL database that will be used is SQLite but this is not mandatory. Other DBs can be used in its stead, preferentially ones that enforce referential integrity to avoid loss of consistency.

Todo: MetaEvent - when an object passes over the entire CCD array it creates a lot of Events, but is still the same object. A MetaEvent table linking multiple Events into a singular object is still required.

3.2 Connecting to the DB

The tables and metadata are registered on import. Connecting to the Db and acquiring an engine, session and connection is done similarly to how setup of detecttrails module works. All three are availible as module global variables, but it is recommended that context managers found in utilities module are used.

```
1 import lfd
2 lfd.setup_results(URI="sqlite:///", dbpath="~/Desktop", name="foo.db", echo=False)
```

Again, as before this functionality is replciated and accessible from the module itself as well.

```
1 from lfd import results
2 results.connect2db(uri="sqlite:////home/user/Desktop/bar.db", echo=False)
```

If the DB at the desired URI does not exists a new empty DB is created and connected to instead. By default the connection is attempted to URI: *sqlite:////home/$USER/Desktop/foo.db*. Additionally if the used DB is SQLite, *PRAGMA FOREIGN_KEYS* is set to ON and *echo* is set to False.

If this is not flexible enough a DB can be created manually and the package's engine and session, availible as attributes of *results* module, can be set to the engine and session created with that DB directly.

lfd.**setup_results**(*URI='sqlite:///'*, *dbpath='~/Desktop'*, *name='foo.db'*, *echo=False*)

Either connects to an existing DB (that has to be mappable by results package) or creates a new empty DB with the required tables.

Parameters

- **URI** (*str*) – an URI to the DB. Defaults to 'sqlite:///$USER_HOME/food.db'.
- **dbpath** (*str*) – location of a directory with an existing DB, or a directory where a new DB will be created. Set to '~' by default.
- **name** (*str*) – name of the existing, or newly created, DB. Default: 'foo.db'
- **echo** (*bool*) – verbosity, Frue by default.

lfd.results.**connect2db**(*uri*, *echo=False*)

Connects to an existing DB or creates a new empty DB to connect too. The DB has to be mappable by the results package.

Parameters

- **URI** (*an URI used to connect to a DB. By default set to:*) – 'sqlite:///$USER_HOME/foo.db'.
- **name** (*the name of the existing, or newly created, DB. Default: 'foo.db'*) –
- **echo** (*verbosity of the DB. False by default.*) –

3.3 Examples

This is not a tutorial on the full usage of SqlAlchemy, DB's or on how to write queries that return your desired data. This tutorial just showcases some common practices recommended when using this package.

3.3.1 Basic access

Connect to a new database with and populate it with some test sample data (see *create_test_sample()*)

```
from lfd import results
results.connect2db("~/Desktop/foo.db")
results.create_test_sample()
```

Results module will have, by now, bound a sessionmaker to the name *Session*. Session is used as a staging area for all transactions towards the DB. It is recommended that Session is constructed at the beginning of any operation in which DB access is expected and then immediately closed afterwards:

```
s = results.Session()
s.query(results.Event).all()
s.close()
```

3.3.2 Rolling back the changes

If a situation arises in which operation might fail a rollback should be issued

```
s = results.Session()
frame = results.Frame(2888, 1, "i", 139, 1, 1, 1, 1, 1, 1, 1, 4412911)
s.add_all([frame, frame])
s.commit()

# will fail due to primary key restrictions - same frame added twice
# this renders the session unusable. To fix it issue a rollback:
```

(continues on next page)

(continued from previous page)

```
s.rollback()
s.close()
```

3.3.3 Session context manager

Since open-commit-rollback-close is a very common pattern of use when inspecting data, results module has a context manager (see *session_scope()*) for the Session that will return an opened session, commit the changes made to objects while it was opened and automatically close it at the end of the block. If a problem was encountered, a rollback is automatically issued.

```
with results.session_scope() as s:
    s.add_all([frame, frame])
```

There is, in essence, nothing that can be done by managing your own Session that could not be done using the context manager. The downside of context managers is the fact that objects fetched by, but not expunged from, the session will not be availible after the session closes.

3.3.4 Adding and expunging objects

Expunging loads objects data and detaches it from the session, which allows it to be used after the session has closed. Expunged objects can be manipulated and have their values changed but if they are not added back to the session the changes will not persist.

```
with results.session_scope() as s:
   frame = s.query(results.Frame).first()
# will raise an DetachedInstanceError
frame

with results.session_scope() as s:
   frame = s.query(results.Frame).first()
   s.expunge(frame)
# will not raise error
frame
# will raise an error
frame.events
```

If access to the object(s) relationships is needed then that relationship has to be expunged as well. Note: Event uses Frame in its repr string, ergo it must not be printed or an error will be raised. In the same lines, expunging *frame.events* will raise an error as it's a list of *Event* objects, not an *Event* object itself.

```
# when working with single InstrumentedAttribute
with results.session_scope() as s:
   event = s.query(results.Event).first()
   s.expunge(event.frame)
   s.expunge(event)
# will not raise error
event
event.frame

# loading the entire InstrumentedList has a workaround
with results.session_scope() as s:
   frame = s.query(results.Frame).first()
   _ = frame.events
```

(continues on next page)

(continued from previous page)

```
    s.expunge_all()
# will not raise an error
frame
frame.events

# make a change and persist it in the DB
frame.events[0].x1 = 999
with results.session_scope() as s:
    s.add_all(frame.events)
    s.add(frame)
```

The frames workaround works by forcing non-lazy load of all associated events into the session - therefore *expunge_all* will work. Constantly expunging can be a bit tiresome the *deep_expunge()* and *deep_expunge_all()* can be used to expunge all first level relationships of given item/s, sometimes.

3.3.5 Querying

The main purpose of results package querying for events and inspection of their mutual relationships. There are two ways queries can be issued, one was just shown in the examples above

1) Using the session scope

```
with results.session_scope() as s:
    e = s.query(results.Event.run).first()
    print(e)
    print(e.frame)
```

2) Using one of the query methods

```
results.Event.query("frames.run==2888").first()
results.Frame.query("frame.t>4702185918").all()
```

Using Session and session_scope will make sure connections are opened or closed appropriately. Using <*Table*>.*query* method will implicitly open a session and a connection and carry it along as long as it lives.

The second method has no particular real benefit, in fact it only has negatives, compared to the first, except that it's faster to type when interactively experimenting with the data in the terminal/idle/etc. Because it skips both the filter and the session scope statements. It also accepts string-like SQL queries and performs an implicit join between the tables so even the complicated queries can be written quickly. I discourage the use of this approach, except sometimes in interactive use, for various reasons some of which are described bellow.

Attention: There are three different things to be aware of when using SQLAlchemy:

- the *Engine*,
- the *Connection*,
- and the *Session*.

There should be only one engine per DB URI. Depending on the type of connection pool, there can be 1 or many Connections. There can always be many sessions.

The Engine is created and made availible after the call to the `connect2db()`, usually immediately after importing the module. Session is a factory that will create a session when called (i.e. *Session()* from early examples). Users are **ALWAYS** encouraged to use the Session because of its many advantages, but mainly because, as stated

above, there can be many or just 1 Connection that is shared among many sessions. If not properly managed, and it is hard to properly manage, errors occur. Alongside that very important fact, there are some other benefits:

- a Session is a factory, since the same factory will create our sessions, it is guaranteed that all sessions will have the same configuration.
- Session manages its Connections, which otherwise can be hard to do
- automatic construction of SQL queries from OO like expressions
- guaranteed connection creation and release
- Identity map
- Unit of Work pattern

For more, see: https://docs.sqlalchemy.org/en/latest/orm/session_transaction.html

Queries can be 'filtered' to select results contrained by some parameters.

```
with results.session_scope() as s:
    # Selecting all events from specific run
    s.query(Event).filter(Event.run=2888).all()

    # same query as previous section, example 2 - querying Event on Frame
    # attribute requires join
    query = s.query(lfd.results.Event).join(lfd.results.Frame)
    fquery = query.filter(lfd.results.Frame.t > 4702185918)
    f = fquery.all()

    # a demo on why this is still better than querying tables directly

    # Selecting all Events with known line start time
    s.query(Event).filter(Event.start_t != None).all()

    # selecting by using human readable data formats
    fquery = query.filter(lfd.results.Frame.t.iso > '2009-09-27 10:06:10.430
↪')
    f = fquery.all()
    results.deep_expunge_all(f, s)
```

Many attributes, like the *BasicTime* in Frame, have wrappers that allow them to be used more expressively and clearly in the code. That is why their use is so heavily recommended. By using them you don't have to know about the implementation details and caveats. Counting the total number of Events in results DB:

```
s.query(Event).count()
```

These examples should cover most of the situations.

3.4 Common misuse patterns

3.4.1 DetachedInstanceError

```
with results.session_scope() as s:
    e = s.query(results.Event).first()
```

(continues on next page)

(continued from previous page)

```
e.run
>>> Traceback : sqlalchemy.orm.exc.DetachedInstanceError
```

- Incorrect use: session scope is meant to be used as a Unit of Work pattern, grouping multiple operations into a single transaction. Outside of that session scope all additional database operations should not be possible.
- Inconsistency: Primary purpose of the reusults package is to offer users interactive introspection of data. If data does not live outside a narrow scope it's very hard to inspect interactively.

Solution is to do all required work in the scope, or expunge the items.

Todo: Create a auto-expunge scope. Very hard to generalize expunging, especially for circular definitions.

3.4.2 Passing values in/out

```
<in a script>
<some code>
e = results.Event.query().first()
<some additional code, perhaps even changing e>
```

Incorrect use: the session persist untill the session is automatically deleted. This should not be done as explained in the Examples section. If only bits of data need to go out either store them somewhere, or query directly only for them and not the whole row.

```
<in a script>
<some code>
with results.session_scope() as s:
    e = s.query(results.Event).first()
    x1, x2 = e.x1, e.x2
<some further code using x1, x2>

# or
with results.session_scope() as s:
    x1, x2 = s.query(results.Event.x1, results.Event.x2).first()
<some further code using x1, x2>
```

or if modification of DB data is required:

```
<code calculating new values of x1, x2>
with results.session_scope() as s:
    e = s.query(results.Event).first()
    e.x1, e.x2 = x1, x2
<some further code>
```

3.5 Event

Event contains a single linear feature detected with all its measured parameters on a single Frame. Event is intended to be used as the basic object on which work is done, as it encompases all information (times, frame, points...) required. While that may be true, Event is still composed of multiple smaller movimg pieces with which it can have complex relationships with so there are some things worth remembering when working with Event.

In short:

1. Non-prefixed coordinates are properties of the Event class (i.e. x1, y1...). The column names are prefixed with an underscore (i.e. _x1, _cx1...)
2. always change column values through properties since many of them have additional sanity and caveats checking imposed on them
3. Points interpret the coordinates in frame-referenced mode by default
4. Frame properties can not be changed through Event
5. If Frame is changed the changes won't reflect on Event untill DB commit is made.

Longer explanation:

Point objects handle the coord conversions and maintain consistency among the coordinates during interactive work. The table values are hot-wired to Point objects through composites and properties. It is easy to commit a mistake to DB when working with column attributes directly and leave the DB in inconsistent state that can't be fixed without manually editing the value.

Point objects are dependent on the Frame object to provide a reference point for conversion between the two coordinate systems. Frame should update the Event's coordinates, but this is only enforced at instantiation time. Enforcing consistency only at instantiation time lets us leave the well-defined CCD coord system into ccd gaps and then *snap2ccd* before commiting. But at the same time is possible not to see the changed values update immediately when a Frame attribute is changed. The coordinates will be updated once a DB commit is made. Additionally, the Event to Frame relationship is many to one, which means there can be many events on a Frame. Updating a Frame requires reflecting that change to all Events on it.

Point objects will issue warnings or errors if inconsistent situations arise. When a warning is issued, unless it's clearly understood and expected, the best course of action is to issue a rollback. Otherwise DB could be sent to an inconsistent state.

The start/end times are stored in the DB in the SDSS-TAI format. There are some caveats when converting this time to MJD. See: https://www.sdss.org/dr12/help/glossary/#tai

class lfd.results.event.**Event**(*frame*, *x1=None*, *y1=None*, *x2=None*, *y2=None*, *cx1=None*, *cy1=None*, *cx2=None*, *cy2=None*, *start_t=None*, *end_t=None*, *coordsys='frame'*, ***kwargs*)

Class Event maps table 'events'. Corresponds to a single linear feature detection. Contains the measured properties of the feature and links to the Frame on which it was detected.

Parameters

- **id** (*int*) – PrimaryKey, autoincremental
- **_run** (*int*) – run id (ForeignKey)
- **_camcol** (*int*) – camcol id (ForeignKey)
- **_filter** (*str*) – filter id (ForeignKey)
- **_field** (*int*) – field id (ForeignKey)
- **frame** (*sql.relationshitp*) – Frame on which the linear feature was detected, a many to one relationship to frames
- **x1** (*float*) – x frame coordinate of point 1 of the linear feature
- **y1** (*float*) – y frame coordinate of point 1 of the linear feature
- **x2** (*float*) – x frame coordinate of point 2 of the linear feature
- **y2** (*float*) – y frame coordinate of point 2 of the linear feature
- **cx1** (*float*) – x ccd coordinate of point 1 of the linear feature
- **cy2** (*float*) – x ccd coordinate of point 1 of the linear feature

- **cx2** (*float*) – x ccd coordinate of point 2 of the linear feature
- **cy2** – y ccd coordinate of point 2 of the linear feature
- **p1** – see class Point composite Column mapping of x1, y1 to a point p1
- **p2** – see class Point,composite Column mapping of x2, y2 to a point p2
- **start_t** – if possible, the time of the first detection of the linear feature on the frame, see class BasicTime
- **end_t** – if possible, the time of the last detection of the linear feature on the frame, see class BasicTime
- **lt** – not very usefull, see class LineTime

Examples

A Frame object reference is required. See lfd.results.frame.Frame for documentation. At minimum, supply the Frame object and coordinates of two points defining a line:

```
>>> foo = Event(Frame, 1, 1, 2, 2)
```

By default the coordinates are considered to be in "frame" coordinate sys. Optionally specify the "ccd" as the reference coordinate system:

```
>>> foo = Event(Frame, 1, 1, 2, 2, coordsys="ccd")
```

Optionaly all values could be supplied:

```
>>> foo = Event(Frame, 1, 1, 2, 2, 1, 1, 2, 2)
```

or on an example of a different CCD (4, 'i'):

```
>>> foo = Event(f, 1, 1, 1, 1, 11377, 27010, 11377, 2710)
```

or verbosely:

```
>>> foo = Event(frame=Frame, x1=1, y1=1, x2=2, y2=2, cx1=1, cy1=1, cx2=2,
                cy2=2, coordsys="frame")
```

Caution: When specifying coordinates in the ccd coordinate system be mindful of the fact that coordinates (0, 0) in 'ccd' frame are upper left corner of the ccd array (camcol 1, filter 'r'). It is not possible to have CCD coordinates (1, 1) or (2, 2) except on the first chip of the CCD-array. If the Frame reference is with respect to some other camcol and filter the ccd coordinates must, otherwise an Error will be raised.

It is possible to submit start or end time of the linear feature in any the following formats: Astropy Time object, float number in mjd format, float number in sdss-tai format:

Astropy Time Object - supply an instantiated Astropy Time Object:

```
>>> st = astropy.time.Time(58539.0, format="mjd")
>>> et = astropy.time.Time("2019-02-25 00:00:10.000")
>>> foo = Event(frame, 1, 1, 2, 2, start_t=st, end_t=et)
```

Any Astropy Time supported format such as mjd, iso, isot, jd etc…

```
>>> foo = Event(frame, 1, 1, 2, 2, end_t=63072064.184, format="cxcsec")
```

or in the SDSS modified tai format:

```
>>> foo = Event(frame, 1, 1, 2, 2, start_t=4575925956.49, format="sdss-tai")
```

_findPointsOnSides(*m*, *b*)

Looking for an intersection of a horizontal/vertical borders with a line equation does not neccessarily return a point within the range we're looking for. It is easier and faster to do it manually. Each individual border will be checked manually and if it satisfies, two coordinates (defining a Point) will be appended to a list. Special cases of (0, 0) and (2048, 2048) will satisfy both border conditions and so will be duplicated in the result. Interestingly, if working from 'frame' reference system it's not neccessary to know which reference frame we're looking at.

Parameters

- **m** (*float*) – line slope
- **b** (*float*) – line y intercept

classmethod query(*condition=None*)

A class method that can be used to query the Event table. Appropriate for interactive work, not as appropriate for large codebase usage. See package help for more details on how the Session is kept open. Will return a query object, not the query result.

If condition is supplied it is interpreted as a common string SQL. It's sufficient to use the names of mapped classes and their attributes as they will automatically be replaced by the correct table and column names.

Examples: Event.query().all() Event.query().first() Event.query("run == 3").all() Event.query("Event.run > 3").all() Event.query("Frame.t > 4412911072y").all() Event.query("events.cx1 > 10000").all() Event.query("frames.filter == 'i'").all()

snap2ccd()

Snap the curent coordinates to the points of intersection of the reference frame CCD border and the linear feature.

A negatively sloped 45° linear feature passes diagonally across the first CCD in the array (1, 'r'), cutting through both its corners. Such feature could be defined by P1(-1000, -1000) and P2(10000, 10000). Snap will determine the two border points P1(0,0) and P2(2048, 2048).

3.6 Frame

Frame represents a single SDSS image. On a single CCD there can be up to 3 such frames simultaneously, slightly overlapped on top and bottom edges. It is not clear to me how to resolve a Frame's position within a CCD due to the SDSS' drift-scan method.

Frames can only exist within a CCD and CCD's are placed in an fiter x camcol grid in the CCD plane where the distance between camcols is only slightly less than the size of the CCD itself so that the gaps can be filled in by another run.

A single Frame can contain many Events.

class lfd.results.frame.**Frame**(*run*, *camcol*, *filter*, *field*, *crpix1*, *crpix2*, *crval1*, *crval2*, *cd11*, *cd12*, *cd21*, *cd22*, *t*, ***kwargs*)

Class Frame maps table 'frames'. Corresponds to an SDSS frame. A frame is uniquely defined by the set (run, camcol, filter, field). For in-depth see datamodel: https://data.sdss.org/datamodel/files/BOSS_PHOTOOBJ/frames/RERUN/RUN/CAMCOL/frame.html

Parameters

- **run** (*int*) – run id (composite PrimaryKey)
- **camcol** (*int*) – camcol id (composite PrimaryKey)
- **filter** (*str*) – filter id (composite PrimaryKey)
- **field** (*int*) – field id (composite PrimaryKey)
- **crpix1** (*int*) – x frame coordinate of the reference central pixel
- **crpix2** (*int*) – y frame coordinate of the reference central pixel
- **crval1** (*float*) – RA on-sky coordinates of the reference pixel (degrees)
- **crval2** (*float*) – DEC on-sky coordinate of the reference pixel
- **cd11** (*float*) – change of RA per column pixel
- **cd12** (*float*) – change of RA per row pixel
- **cd21** (*float*) – change of DEC per column pixel
- **cd22** (*float*) – chage of DEC per row pixel
- **t** (BasicTime) – time of start of frame exposure
- **events** (*sql.relationship*) – list of all event(s) registered on this frame, a one to many relationship to events

Example

Almost everything is not nullable so supply everything:

```
>>> foo = Frame(run, camcol, filter, field, crpix1, cprix2, crval1, crval2,
                cd11, cd21, cd22, t)
```

i.e.

```
>>> foo = Frame(2888, 1, 'i', 139, 741, 1024, 119, 23, 1, 1, 1,
                4575925956.49)
```

Time can be given as SDSS TAI or any of the other formats supported by Astropy Time Object. In the DB itself it is always forced to the SDSS TAI time format.

classmethod query(*condition=None*)

Class method that used to query the Frame ('frames') table. Returns a Query object. Appropriate for interactive work as Session remains open for the lifetime of the Query. See results package help to see details.

If condition is supplied it is interpreted as an SQL string query. It's suffiecient to use mapped class names and their attributes as the translation to table and column names will be automaticall performed.

Example

Frame.query("Frame.run > 2").first() Frame.query("field == 1 and filter == 'i'").all() Frame.query("Event.y1 > 2).all()

3.7 Point

Point class provides a more comfortable interface for handling coordinates and coord transformations. Point is capable of moving and setting of new coordinate values in any frame while maintaning consistency across all coordinates.

There are two coordinate systems to manage: 'frame' and 'ccd'. Both are expressed in units of pixels and both have an inverted y axis (increases from from top to bottom such that the y coordinate remains positive within the CCD array).

- 'ccd' coordinate system represents the whole CCD array, the area from (0, 0) to (MAX_W_CCDARRAY, MAX_H_CCDARRAY). This area contains all CCDs. One CCD is defined by (camcol, filter). There are 6 camcols and 5 filters. A point *anywhere* in this area can be represented in the 'ccd' coordinate system. The names of 'ccd' coordinates are (cx, cy).
- The 'frame' coordinate system is used to represent image-coordinates. These are the values measured by the detecttrails package. The origin of the coordinate system is set to be reference CCD's upper left corner. Coordinates (x, y) change if the chosen reference CCD changes. The 'frame' coordinate system is a slight misnomer as within each CCD there can be up to 3 frames-xxxx-xx-x-xxx.fits images at once. The frame dimensions are 2048x2048 but images are cut to 1489px height with 139px of overlap between following images. A point *within* any CCD can be represented in the 'frame' coordinate system.

To see the values defining both coordinate systems and how to convert between them look at the ccd_dimensions and coord_conversion modules.

Warnings will be produced in an inconsistent situations to warn that, logically, the performed operations do not make sense, but will not enforce consistency at all times.

3.7.1 Important Notes

Point is capable of interpreting the coordinates in two different context. One is as an absolute coordinate, usually in ccd coordinate system, and the other as a frame-referenced coordinates in which case the coordinates remember a reference to the frame they're defined in. In the latter case it is possible to have 'frame' coordinates with numerically determined values even when they are not within a CCD bounary, while in the former it is not because no reference frame has been set. This can be bit confusing:

```
p = res.Point(3790, 5415, coordsys="ccd")
<lfd.results.point.Point(x=None, y=None, cx=3790, cy=5415)>

p2 = res.Point(-1, -1, camcol=2, filter="u")
<lfd.results.point.Point(x=-1, y=-1, cx=3790.820956, cy=5415.8870802)>
```

Both points reference approximately the same coordinate except that the frame-referenced Point understands which frame's (0, 0) point to use to calculate the 'frame' coordinates x and y from. The same point can be expressed referenced from a different frame:

```
p2 = res.Point(-7584.64, 2707, camcol=4, filter="i")
<lfd.results.point.Point(x=-7584, y=2707, cx=3790.88, cy=5415.44)>
```

This makes more sense in the context of using the Point class to represent the linear feature in the Event class, but it has some merrits as a feature on its own.

class lfd.results.point.**Point**(*x=None*, *y=None*, *cx=None*, *cy=None*, *camcol=None*, *filter=None*, *coordsys='frame'*)

Provides a more comfortable interface for handling coordinates and coord transformations while maintaning consistency across all coordinates.

Parameters

- **x** (*float*) – 'ccd' or 'frame' coordinate, depending on coordsys

- **y** (*float*) – 'ccd' or 'frame' coordinate, depending on coordsys
- **coordsys** (*str*) – 'ccd' or 'frame', designation of coordinate system
- **inCcd** (*bool*) – True if the current coordinate is within a CCD
- **camcol** (*int*) – the camera column of reference frame
- **filter** – the filter row of the reference frame

Example

```
>>> p = Point(10, 10, coordsys="ccd")
>>> p = Point(10, 10, camcol=2, filter='r')
```

To switch between coordinate system:

```
>>> p.useCoordSys('ccd')
>>> p.switchSys()
```

Check if current Point is in CCD or not

```
>>> p.inCcd
```

Moving the point can be by assignment or method. Method will try to resolve intended coordsys, or default to current coordsys, but assignment always uses currentl coordsys. Mixing coord. sys. is possible:

```
>>> p.move(x=10, y=10)
>>> p.move(cx=100, y=10)
>>> p.y = 100
```

_check_sensibility(*attr*)

For a given attribute attr checks for a series of conditions that indicate an illogical operation or in some cases operations that could leave an Event in an inconsistent state. Is not always correct, but that's why it's a warning not Error.

Attrs	Conditions
camcol, filter	if coord. sys. is ccd providing these makes no sense
frame	if frame is sent as attr and self.inCcd is False the frame coordinate system coordinates are not defined.

move(**args*, ***kwargs*)

Move a point to different coordinates. Supports a more flexible way to change Point coordinates than direct assignment. Checks are are performed and if the ingoing coordiantes do not match the coordsys, the Point is *NOT* moved and a warning is issued. This makes move very oppinionated and annoying, but it forces explicit statements about where and how the point is moved in the code.

Example

Implicitly resolves coordinates

```
>>> move(1, 1, "frame") # --> x=1, y=1
>>> move((1, 1), "frame") # --> x=1, y=1
>>> move(1, 1, "ccd") # --> cx=1, cy=1
>>> move((1, 1), "ccd") # --> cx=1, cy=1
```

Refuses to work without coordsys specification

```
>>> move(1, 1) # --> will always fail with error, no coordsys sent
```

Missmatched coordinates and coordinate systems issue warnings

```
>>> move(x=1, y=1, coordsys="ccd") # --> not moved, warning issued
>>> move(cx=1, cy=1, coordsys="frame") # --> not moved, warning issued
```

Partially specified coordinates will work only if coordsys is properly matched to the given coordiante which must be explicitly specified

```
>>> move(1, "frame") # --> fails
>>> move(x=1, coordsys="frame") # --> moves the x coordinate only
>>> move(cx=1, coordsys="frame") # --> isses a warning, doesn't move x
```

Parameters

- ****kwargs** (*dict*) – x, y or cx, cy or coordsys and their values
- ****kwargs** – x, y or cx, cy and coordsys and their values

switchSys()

Switch to the other coordinate system ('frame'-> 'ccd' and vice-versa). Not particularily useful as both are usually accessible, but practical to state which coordinate system we are currently in.

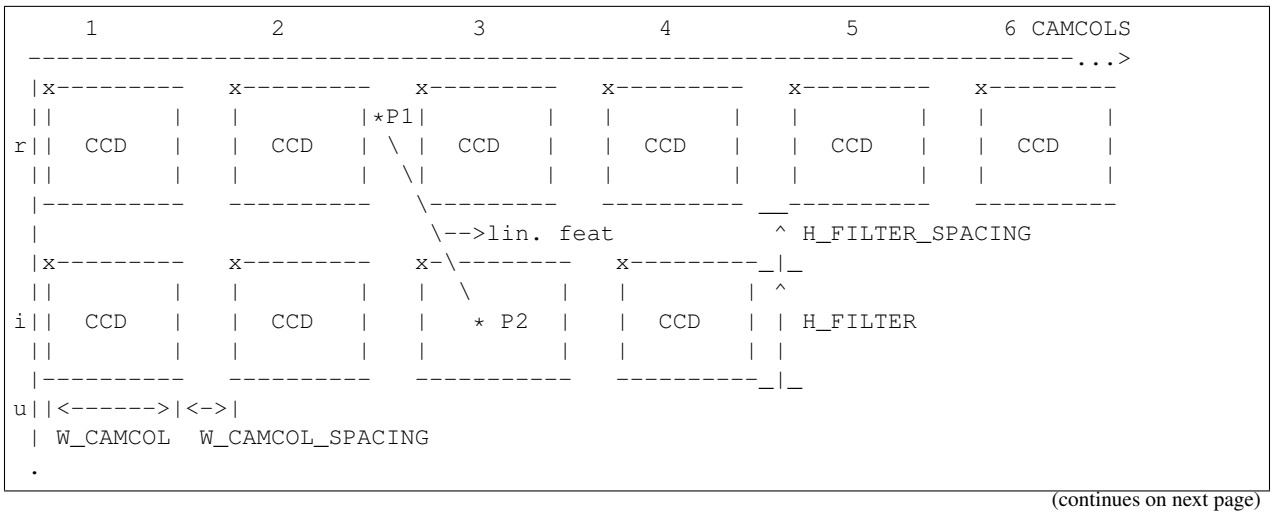
useCoordSys(*coordsys*)

Use a particular coordinate system, either 'frame' or 'ccd'.

Parameters **coordsys** (*str*) – coordsys designation, 'frame' or 'ccd'

3.7.2 Dimensions & Constants

This module contains all neccessary constants to define a usable coordinate system on, and between, the images:



(continues on next page)

(continued from previous page)

```
.
. FILTERS (riuzg)
`
```

Widths and heights were fixed to the official values and while the values of spaces in between the CCD dimensions were availible in the literature they did not correspond exactly to calculable quantities - so they have been replaced by these new values that seem to match the data better than the ones provided in literature.

CONSTANTS

W_CAMCOL [float] 2048, the width of one CCD, in pixels, corresponds to image width

H_FILTER [float] 2048, the height one one CCD, in pixels, images are cut to 1489px height with 139px of overlap between them, there are usually 3 images in a CCD at a given time, although only 2 is also possible.

W_CAMCOL_SPACING [float] 1743.820956, the width the gap beween two camera columns

H_FILTER_SPACING [float] 660.4435401, the spacing between two rows of filters, these gaps can not be noticed on the images but it takes some time for the sky to drift through them.

MAX_W_CCDARRAY [float] 21008.0, unrounded: 21007.10478, the lower edge of the CCD array, in pixels, if all W_CAMCOL and W_CAMCOL_SPACINGS were added together

MAX_H_CCDARRAY : 12882.0, unrounded: 12881.77416, the lower edge of the CCD array, in pixels, if all H_FILTER and H_FILTER_SPACINGS were added together

ARCMIN2PIX [float] 0.0066015625, arcminutes/pixel, pixel scale, corresponds to0.396 arcsec/pixel , but expressed in minutes because of how values were given in the tables.

MM2ARCMIN [float] 3.63535503, detector image scale, mm/arcminute.

3.7.3 Coordinate Conversion Functions

exception lfd.results.coord_conversion.**CoordinateConversionError**(*incoords*, *outcoords*, *msg=None*, **args*)

Generic Error to be called when no sollution to coordinate conversions between CCD and frame coordinate systems are not possible. A light wrapper around ArithmeticError.

Example

```
>>> raise CoordinateConversionError(incoords, outcoords)
```

Parameters

- **incoords** (*tuple, list*) – a set of ingoing to-be-converted coordinates
- **outcoords** (*tuple, list*) – set of (miss)calculated coordinates
- **msg** (*str*) – customize the error message
- ***args** (*tuple, list*) – any additional args can be supplemented and are appended to the end of the error message

- **are (cx, cy) CCD coordinates and outcoords are the frame** (*incoords*) –
- **y, camcol, filter) coordinates or vice-versa.** ((*x,*) –

lfd.results.coord_conversion.**convert_ccd2frame**(*x*, *y*)

Conversts the coordinate pair (x, y) from the CCD coordinate system to a frame coordinate system by applying the following formulae:

```
x = cx - {camcol} * (W_CAMCOL + W_CAMCOL_SPACING)
y = cy - {filter} * (H_FILTER + H_FILTER_SPACING)
```

Where camcol and filter are iterated over untill a possible sollution is found. A sollution is possible if it is contained within the width and height of a single CCD respective to its (0, 0) point in the frame coord. system. Only one such sollution is guaranteed to exists.

lfd.results.coord_conversion.**convert_frame2ccd**(*x*, *y*, *camcol*, *filter*)

Converts from frame coordinates (x, y, camcol, filter) to CCD coordinates (cx, cy) via the following formulae:

```
cx = x + (camcol-1) * (W_CAMCOL + W_CAMCOL_SPACING)
cy = y +  filter    * (H_FILTER + H_FILTER_SPACING)
```

Filter can be sent as an integer or a string from {riuzg}.

lfd.results.coord_conversion.**get_filter_from_int**(*filterint*)

Provides translation between an integer row value of the filter and its string value where the top row is indexed with a zero:

```
0 -> r
1 -> i
2 -> u
3 -> z
4 -> g
```

lfd.results.coord_conversion.**get_filter_int**(*filter*)

Provides the mapping between filter in string form and integer value based on the row the searched for filter is in, starting from the top:

```
r -> 0
i -> 1
u -> 2
z -> 3
g -> 4
```

3.8 Time

Dealing with time-stamps has always been a challenging aspect of APIs. This module, hopefully, alleviates some of issues wen dealing with timestamps by defining a new database type decorator that will wrap the awkward SDSS TAI timestamps into a more pleasant OO interface by using Astropy Time objects.

class lfd.results.basictime.**BasicTime**(**args*, ***kwargs*)

A class that will force storing time stamps in the SDSS TAI format and will force read-out of the timestamp from the DB as an Astropy Time object.

Assignments, comparisons and other interactions will be the same as any other Astropy Time object.

impl
alias of `sqlalchemy.sql.sqltypes.Float`

process_bind_param(*value*, *dialect*)
Used to map value to the desired storage format within the DB. Expects an Astropy Time object. Converts the time to SDSS TAI format for storage.

process_result_value(*value*, *dialect*)
Used to map read values from the DB to a format appropriate for interactive work. Expects an SDSS TAI format float value and returns an Astropy Time object.

class lfd.results.basictime.**LineTime**(*t_start*, *t_end*, *t_format='tai'*)
Used to map two BasicTime timestamps onto a singular object. Potential for expansion to a point where Events can be queried on statistical properties of the duration of the linear feature and other such features.

Currently useless as many of the advanced interfaces are not implemented.

3.9 Utilities

Contains various utilities that expand and suplement the results module functionality and make it easier to work with.

lfd.results.utils.**from_file**(*path*)
Read a detecttrails formatted CSV file into a database.

lfd.results.utils.**create_test_sample**()
Creates a test sample of mock Events for testing, demonstration and learning purposes.

lfd.results.utils.**session_scope**()
Provide a transactional scope around a series of operations.

lfd.results.utils.**pprint**(*objlist*, **args*, ***kwargs*)
Pretty-prints a list of Frame or Event objects in a table-like format.

Parameters

- **short** (*bool*) – True by default. The shortened format corresponds to:

```
run camcol filter field time x1 y1 x2 y2
```

  if short is false, the long table format is printed:

```
run camcol filter field time x1 y1 x2 y2 cx1 cy1 cx2 cy2
```

- **digits** (*int*) – 2 by default. Controls the number of printed significant digits,

lfd.results.utils.**deep_expunge**(*item*, *session*)
For a given item, expunges all first order sql relationships from given session.

Parameters

- **item** (*object*) – any OO mapped sqlalchemy ORM object to be expunged completely (Event, Frame etc..)
- **session** (*sql.Session()*) – active session from which we want to expunge the item from

lfd.results.utils.**deep_expunge_all**(*items*, *session*)
For a given list of items, expunges all first order sql relationships from given session.

Parameters

- **items** (*list(object) or tuple(object)*) – set of OO mapped sqlalchemy ORM object to be expunged completely (Event, Frame etc..)
- **session** (*sql.Session()*) – active session from which we want to expunge the items from

In interactive use it is not very likely detecttrails will output unmanageable quantities of results. However, it LFD was designed to be able to reprocess the entire SDSS database of images in which case the size of the results can become hard to handle if left in its original CSV output format.

Main purpose of results package is to allow collating the default outputs and easy interactive inspection of the results.

It is written using SQLAlchemy, a Python SQL toolkit and a ORM. Because of this before beginning to work with the module it is neccessary to create, or connect to, a database. See *connect2db* function from this module or the *setup_db* from lfd library root on how to do that.

Results module is more than just a ORM interface to a DB. It is aware of the complex SDSS camera array geometry and capable of translating raw result on-image coordinates to on-ccdarray coordinates without loss of consistency of the data, it provides wrappers between the modified SDSS MJD and other time formats as well as various results-colating, table-printing and database utilities.

CHAPTER 4

Creating DQS cluster jobs

4.1 Setup

Createjobs module depends on run parameter files to solve targeted data id information. As described in *Setup* .par files are found in the *photo/redux* directory to which the env. var. *PHOTO_REDUX* points. The minimal setup for this module requires that *PHOTO_REDUX* variable is set. As before, this can be done in terminal, or by using the provided functionality.

`lfd.`**`setup_createjobs`**(*photoreduxpath=None*)
Sets up the environmental path of PHOTO_REDUX only! Minimum environment required for certain createjobs package functionality.

Parameters **`photoreduxpath`** (*str*) – The path to which PHOTO_REDUX env. var. will be set to. Defaults to '~/Desktop/boss/photoredux'.

`lfd.createjobs.`**`setup`**(*photoreduxpath=None*)
Sets up the required environmental paths for createjobs module.

Parameters **`photoreduxpath`** (*str*) – The path to which PHOTO_REDUX env. var. will be set to. Defaults to $BOSS/photoObj

4.2 Templates

By default the opened template is called "generic" and can be found in the same folder this module resides in. Since it's neccesary to change a lot of parameters, especially paths, template can be edited, or a new one can be provided in its place.

Specifying *template_path* at Job class instantiation time will instruct the writer to substitute the template used.

When writing your own template, to avoid error reports, you have to specify all parameters in the new template that this class can change. Parameters are uppercase single words, i.e: JOBNAME, QUEUE, NODEFLAG

```
#!/usr/bin/ksh
#PBS -N JOBNAME
#PBS -S /usr/bin/ksh
#PBS -q QUEUE
#PBS -l nodes=NODEFLAG:ppn=PPN
#PBS -l walltime=WALLCLOCK,cput=CPUTIME
```

Not all enviroment paths in the template are changable throught this class. This was done to avoid confusion and additional complexity of how working paths are handles, since on a cluster greater flexibility is provided by the filesystem, that is usually tricky to nicely wrap in Python. Additionally, most of the used directories will often share the same top directory path, while individual jobs will only differ at the level of particular target subdirectory inside the targeted top directory. Such paths can be edited in place in the template, for example:

```
cp *.txt /home/fermi/$user/run_results/$JOB_ID/
```

Bellow is the full content of the generic template provided with the module:

```
#!/usr/bin/ksh
#PBS -N JOBNAME
#PBS -S /usr/bin/ksh
#PBS -q QUEUE
#PBS -l nodes=NODEFLAG:ppn=PPN
#PBS -l walltime=WALLCLOCK,cput=CPUTIME
#PBS -m e
#QSUB -eo -me

SAVEFOLDER=RESULTSPATH

cd ~
user=`whoami`
hss=`hostname`

if  [ "$PBS_ENVIRONMENT" != "" ] ; then
 TMPJOB_ID=$PBS_JOBID.$$
 JOB_ID=${TMPJOB_ID%%[!0-9]*}.$$
 ARC=`uname`
fi

nodefile=$PBS_NODEFILE
if [ -r $nodefile ] ; then
    nodes=$(sort $nodefile | uniq)
else
    nodes=localhost
fi

##Export paths user has to change as instructed by help(createjobs)
##fitsdm is the fits unpack path, can be anywhere
##BOSS should point to root boss folder with the files
##that copies the sdss tree:
##    boss/photo/redux/runList.par
##    boss/photoObj/301/..... photoObj files
##    boss/photoObj/frames/301/..... frames files
export FITSDMP=/scratch/$hss/$user/fits_dump
export BOSS=/scratch1/fermi-node02/dr10/boss
export PHOTO_REDUX=$BOSS/photo/redux
export BOSS_PHOTOOBJ=$BOSS/photoObj
```

(continues on next page)

(continued from previous page)

```
##Make sure we have all the necessary folders in place
mkdir -p  /scratch/$hss/$user
mkdir -p  /scratch/$hss/$user/test_trails
mkdir -p /scratch/$hss/$user/fits_dump
mkdir -p /home/fermi/$user/$SAVEFOLDER/
mkdir -p /home/fermi/$user/$SAVEFOLDER/$JOB_ID

cd /scratch/$hss/$user/test_trails
mkdir -p  /scratch/$hss/$user/test_trails/$JOB_ID

cd $JOB_ID
echo $nodes >nodes #contains node identifier
echo $PBS_EXEC_HOST >aaa2 #contains various host parameters
set >aaa3 #contains host parameters

cp /home/fermi/$user/run_detect/*.py* /scratch/$hss/$user/test_trails/$JOB_ID/
mkdir sdss
cp -r /home/fermi/$user/run_detect/sdss/* sdss/

source ~/.bashrc #get the right python interp.

COMMAND

##Copy the results back to fermi, delete what you don't need anymore
cp *.txt /home/fermi/$user/$SAVEFOLDER/$JOB_ID
#cp nodes /home/fermi/$user/$SAVEFOLDER/$JOB_ID
#cp a*   /home/fermi/$user/$SAVEFOLDER/$JOB_ID

##Remove everything
rm a* nodes *py*
rm -rf sdss
```

4.3 Examples

The simplest one is just specifying the number of jobs you want. Jobs will then take the runs found in runlist.par (only reruns 301 are considered, special reprocessings are not included), resolve the targeted data and create the jobs.

```
jobs = cj.Jobs(500)
jobs.create()
There are no runs to create jobs from.
  Creating jobs for all runs in runlist.par file.

Creating:
  765 jobs with 1 runs per job
  Queue:     standard
    Wallclock: 24:00:00
    Cputime:   48:00:00
    Ppn:       3
    Path:      /home/user/Desktop/.../jobs
```

User will be notified about all important parameters that were set. Notice that the default save path, queue, wallclock, ppn and cputime are set by default. All parameters can be sent as keyword arguments at instantiation.

Notice also that we specified 500 jobs to be created but 765 jobs were created instead. This is intentional. Jobs looks for the next larger whole number divisor divisor to split the jobs between. It's better to send in more jobs than to risk

jobs failing.

Specifying certain runs by hand is possible by sending a list of runs of interest:

```
runs = [125, 99, 2888, 1447]
jobs = cj.Jobs(2, runs=runs)
jobs.create()
Creating:
     2 jobs with 2 runs per job
     Queue:     standard
 Wallclock: 24:00:00
 Cputime:   48:00:00
 Ppn:       3
 Path:      /home/user/Desktop/.../jobs
```

In both examples so far what is actually being written as a command that will be executed at job submission:

```
python3 -c "import detecttrails as dt; dt.DetectTrails(run=2888).process()"
```

It is possible to edit the command that is going to be executed. Specifying aditional keyword arguments to Jobs class helps you utilize DetectTrails class run options. Sent kwargs are applied globaly across every job. It's not, however, possible to specify separate kwargs for each command individually.

```
runs = [125, 99, 2888, 1447]
jobs = cj.Jobs(2, runs=runs, camcol=1)
jobs.create()
```

would create 2 jobs with 2 runs per job as the above example. The invocation of the DetectTrails class would now look like:

```
python3 -c "import detect_trails as dt; dt.DetectTrails(run=125,camcol=1).process()"
```

Which would process only the camcol 1 of run 125. Actual written job#.dqs file is not as readable/friendly as above examples. Another example:

```
jobs = cj.Jobs(2, runs=runs, camcol=1, filter="i")
```

would execute 2 jobs with following 2 calls to DetectTrails class:

```
python3 -c "import detect_trails as dt;
            dt.DetectTrails(run=125,camcol=1,filter=i).process()"
```

See help on DetectTrails class for a complete set of allowable options.

To further fine tune your job behaviour it's possible to change the default execution command to supply additional execution parameters. By default keyword argument command is set to:

```
python3 -c "import detecttrails as dt; dt.DetectTrails($).process()"
```

Where "$" sign gets automatically expanded by the writer module. There should also **ALWAYS** be a "$" character present in a command. "$" replaces arguments of DetectTrails class at instantiation, sort of as ***kwargs* usually do. Example:

```
jobs = cj.Jobs(2, runs=runs, camcol=1, filter="i")
jobs.command = 'python3 -c "import detecttrails as dt;' +\\
               'x = dt.DetectTrails($);'                 +\\
               'x.params_bright[\'debug\'] = True;'      +\\
```

(continues on next page)

(continued from previous page)

```
                'x.process()"\n'
jobs.create()
```

which will get written as:

```
python3 -c "import detecttrails as dt; +\\
            x = dt.DetectTrails(run=125,camcol=1,filter=i); +\\
            x.params_bright['debug'] = True; +\\
            x.process()"
```

Again, the actual written command in the job#.dqs file would not look as user friendly and readabe as it is here. In the above example notice that quotation marks are trice nested as follows:

```
'    ("   (\'   \')    ")     '
```

where:

- outter ‘: declares a python string, this string becomes the command attribute of Jobs class.
- inner “: encloses the string that will be executed by the python -c command in the actual job#.dqs file.
- innermost ’: mark a new string that will get interpreted as an argument inside the string you’re sending to python -c.

This complication is here because the command has to be sent as a string therefore the quotation marks used inside should not escape the outsidemost quotations. General usefull guidelines:

1) the outter-most quotation as single ‘’ marks
2) everything past “-c” flag in double quotation marks “”
3) further quotation marks should be escaped single quotations.
4) A EXPLICIT newline character should ALWAYS be present at the end.

Tip: It is usually much simpler to declare the command as a multiline python comment by using the triple-quote mechanism:

```
command = """python3 -c "import detecttrails as dt;
           x = dt.DetectTrails($);
           x.params_bright['debug']=True;
           x.process()
"""
jobs.command = command
jobs.create()
```

because there is no need to escape any of the special characters at all.

Same applies when executing a custom command for all runs:

```
jobs = cj.Jobs(500)
jobs.command = 'python3 -c "import detecttrails as dt;' +\\
               'x = dt.DetectTrails($);'                +\\
               'x.params_bright[\'debug\'] = True;'     +\\
               'x.process()"  \n\'
jobs.create()
```

would produce jobs for all runs as in the first usage case, where each job would execute the following command(s):

```
python3 -c "import detecttrails as dt; +\\
           x = dt.DetectTrails(run=273); +\\
           x.params_bright['debug'] = True; +\\
           x.process()"
```

To see the list of all changable execution parameters of DetectTrails class see the tables of detection parameters described *Table of removestars parameters*, *Table of bright parameters*, and *Table of dim parameters*.

Former approach covers most basics about how to get the most out of DetectTrails class on QSUB. However, described approaches still do not let you create jobs per frames. Sollution for this problem is to send in a list of Event or Frame objects. Read docs of results package to see how to instatiate those objects. Using them with createjobs module is quite straight-forward.

```
# mixing them in the same list/tuple is allowed
r = [Event, Event, Frame, Event... ]

jobs = cj.Jobs(5, runs=r)
jobs.create()
  Creating:
      6 jobs with 1372 runs per job
  Queue:     standard
  Wallclock: 24:00:00
  Cputime:   48:00:00
  Ppn:       3
  Path:      /home/user/Desktop/.../jobs
```

This time it's not runs you're executing but frames, therefore you can let a larger number of them per job; i.e. the invocation of DetectTrails now looks like:

```
python3 -c "import detect_trails as dt;
            dt.DetectTrails(run=125,camcol=1,filter='i', field=69).process()"
```

4.4 Jobs

Job class is the interface to the writer which stores all parameters required to sucessfully populate the templte. It also wraps additional functionality such as directory and script IO that will try and prevent overwriting of previously created scripts.

class lfd.createjobs.createjobs.**Jobs**(*n*, *runs=None*, *template_path=None*, *save_path=None*, *queue='standard'*, *wallclock='24:00:00'*, *ppn='3'*, *cputime='48:00:00'*, *pernode=False*, *command='python3 -c "import detect-trails as dt; dt.DetectTrails($).process()"n'*, *res_path='run_results'*, ***kwargs*)

Class that holds all the important functions for making Qsub jobs.

Template is located inside this package in "createjobs" folder under the name "generic". Location where final results are saved on Fermi cluster by default is:

```
/home/fermi/$user/$res_path/$JOB_ID.
```

Can be changed by editing the template or providing a new one. One can also be provided in string format as a kwargs named "template".

Parameters

- **n** (*int*) – number of jobs you want to start.
- **save_path** (*str*) – path to directory where jobs will be stored. By default set to ~/Desktop/createjobs
- **res_path** (*str*) – path to subdirectory on cluster master where results will be copied once the job is finished.
- **template_path** (*str*) – path to the desired template
- **template** (*str*) – a full template text as a string
- **queue** (*str*) – sets the QSUB queue type: serial, standard or parallel. Defaults to standard. Your local QSUB setup will limit wallclock, cputime and queue name differently than assumed here.
- **wallclock** (*str*) – set maximum wallclock time allowed for a job in hours. Default: 24:00:00
- **cputime** (*str*) – set maximum cpuclock time allowed for a job in hours. Default: 48:00:00
- **ppn** (*str*) – maximum allowed processors per node. Default: 3
- **command** (*str*) – command that will be invoked by the job. Default: python -c "import detect_trails as dt; dt.DetectTrails($).process()" where "$" gets expanded depending on kwargs.
- ****kwargs** (*dict*) – named parameters that will be forwarded to command. Allow for different targeting of data. See documentation for examples
- **runs** – if runs are not specified, all SDSS runs found in runlist.par file will be used. If runs is a list of runs only those runs will be sorted into jobs. If runs is a list of Event or Frame instances, only those frames will be sorted into jobs. See docs on detailed usage.

create()

Creates job#.dqs files from runlst. runlst is a list(list()) in which inner list contains all runs per job. Length of outter list is the number of jobs started. See class help.

getAllRuns()

Returns a list of all runs found in runlist.par file.

makeRunlst(*runs=None*)

Create a runlst from a list of runs or Results instance. Recieves a list of runs: [N1,N2,N3,N4,N5...] and returns a runlst:

```
[
  (N1, N2...N( n_runs / n_jobs)) # idx = 0
  ...
  (N1, N2...N( n_runs / n_jobs)) # idx = n_jobs
]
```

Runlst is a list of lists. Inner lists contain runs that will be executed in a single job. Lenght of outter list matches the number of jobs that will be started, f.e.:

```
runls = list(
               (2888, 2889, 2890)
               (3001, 3002, 3003)
            )
```

will start 2 jobs (job0.dqs, job1.dqs), where job0.dqs will call DetectTrails.process on 3 runs: 2888, 2889, 2890.

If (optionally) a list of runs is supplied a run list will be produced fom that list, instead of the runs attribute.

lfd.createjobs.writer.**get_node_with_files**(*job*, *run*)
Deprecated since version 1.0.

Reads lst-lnk file to retrieve nodes on which fits files of run are stored. Returns the node number. In cases where error occured while reading node number returns the first, "01", node.

lfd.createjobs.writer.**writeJob**(*job*, *verbose=True*)
Writes the job#.dqs files. Takes in a Jobs instance and processes the "generic" template replacing any/all keywords using values from Jobs instance. For each entry in runlst it creates a new job#.dqs file, which contains commands to execute detecttrails processing for each entry of entry in runlst.

Parameters

- **job** (*lfd.createjobs.Job*) – Job object from which job scripts are to be created.
- **verbose** (*bool*) – deprecated to alleviate clutter

Note: There is a GUI interace for this module that might be easier to understand, at a minimal cost of functionality loss.

Note: This package was written for the Fermi cluster at the Astronomical Observatory Belgrade but it should be easy to adapth to any other QSUB cluster.

Used to create .dqs files necessary to run a job on QSUB system. Main idea was to create a class that can take care of writing large job(s) without having to resort to manually editing the templates or produced job script(s) post-fact. Should alleviate a lot of work and unavoidable confusion, when such jobs are created manually.

Jobs are created through a Job instance, holding all required parameters. Job instance uses a writer to populate a template of **.dqs* scripts that can then be submitted to the cluster interface.

CHAPTER 5

GUIs

5.1 Verifying results

5.1.1 Image Checker

ImageChecker.**__init__**()

There are several configurable parameters that are imporant for this class:

- resize_x - the reduction factor describing how much has the width been reduced from the original to the displayed image.
- resize_y - the reduction factor describing how much has the height reduced between the original and displayed image.

Optionally you can rebind the key functionality or change the color scheme of the table in the top right by editing the TopRight Frame of the rightFrame. Additionally, the displayed keys in the TopRight Frame are editable through that class.

class lfd.gui.imagechecker.imagechecker.**ImageChecker**

GUI app that allows for visual inspection of Events. To run the app instantiate the class and run its mainloop method or invoke run function located in this module.

The App itself does not manage the data. Data loading and management is handled by the EventBrowser class stored in self.data attribute.

The GUI consits of 2 Frames - left and right. Left frame is used to display information on the Event and the right hand side displays the image representation of the Event if availible. GUI binds the following shortcut keys:

- <Left> - move to previous image without saving any changes
- <Right> - continue to the next image without saving any changes
- <Up> - continue to the next image but set the verified and false positive flags of the current Event to True and False respectively. Persist the changes to the DB
- <Down> - continue to the next image but set the verified and false positive flags of the current Event to True and True respectively and persist the change to the DB

- <LMB> - when clicked on the image will move the first point of the linear feature to that location and persist the changes to the database
- <RMB> - when clicked on the image will move the second point of the linear feature to that location and persist the changes to the database

The colors in the data table on the right frame indicate the following:

- Yellow - the Event was never visually inspected
- Green - the Event was visually inspected and confirmed as true
- Red - the Event was visually inspected and was determined to be a false detection

failedUpdate()
Redraw left and right Frames and display their failure screens.

initGUI()
Will initialize the GUI for the first time by prompting user for the location of the Database to connect to and the location of the images. The order of operations here is not particulary important because the update function will be called to clean and redisplay everything on the screen.

initImages()
Prompt user for the directory containing all the images in the DB.

initResults()
Prompt user for the database file from which Events will be read.

update()
Redraw right and left Frames. The order is important, updating left frame before loading new event will not load the data required to draw the line over the canvas.

class lfd.gui.imagechecker.leftframe.**LeftFrame**(*parent*)
Represents the left frame of the GUI. Contains the Canvas within which the image is displayed and additional functionality that allows the users to chage the line parameters, and persist those changes to the DB. The following mouse actions are bound to the canvas:

- <Button-1> - on click of the left mouse button (LMB) will bind the current coordinates of the mouse pointer and convert the on-canvas coordinates to the frame-coordinate system using the resize_x and resize_y resizing reduction factors defined in the root class of the app. These converted coordinates are then set as a new x1, y1 coordinates of the p1 Point of the Event.
- <Button-3> - on right mouse button (RMB) click records the coordinates of the pointer, scales them to frame coord. sys. and persists the change as the x2, y2 coordinates of p2 Point of the Event.

drawline(*delete=False*)
Draws the line defined by the current Event's Points p1 and p2. If delete is set to True then it will delete any existing line. It is important that the resize scaling factors are correctly set.

failedImageLoadScreen()
Clears the canvas and displays the Error image.

lmb(*event*)
Callback, records and updates the x1, y1 coordinates of the Event.

rmb(*event*)
Callback, records and updates the x2, y2 coordinates of the Event.

update()
Updates the canvas and handles the errors.

updateLine(*sx*, *sy*, *which*)
Function that will scale the canvas coordinates to correspond to the frame-coordinate system and sets the

new coordinates as the p1 or p2 coordinates of the Event. It is important that the resize scaling factors used in the App are correct if the output is to be trusted.

Parameters

- **sx** (*int*) – x coordinate in canvas coordinate system
- **sy** (*int*) – y coordinate in canvas coordinate system
- **which** (*str*) – used to determine whether the coordinates belong to point 1 or point 2 of the Event. Either '1' or '2'.

class lfd.gui.imagechecker.rightframe.**RightFrame**(*parent*)

Represents the right part of the frame containing all the action buttons and displaying the data of he Event from the database. The right frame is split into two sub-frames one used to display the Event in question and the other one containing all the action elements (next, true, false, previous, find, change data source etc.)

failedEventLoadScreen()

Redraws the right frame displaying appropriate error messages in case of failure.

update()

Calls the update methods of each subframe in the correct order and handles failures.

class lfd.gui.imagechecker.topright.**TopRight**(*parent*)

Top right part of the right frame. Used to display the data on currently selected Event.

Contains several customizable attributes such as:

- unverified_color- the color to display when the Event's verified flag is False (DarkGoldenrod1 by default)
- falsepositive_color - the color to display when the Event is verified as false positive (red by default)
- positive_color - color to display when the Event is verified as a positive detection (DarkOliveGreen3)
- displayKeys - keys that will be displayed in the information table of the Event. Any valid column name of Event is accepted, by default will be: *[run, camcol, filter, field, frame.t.iso]*

updateImageData()

Clears the currently displayed table, and draws a new table displaying the data of currently loaded Event. If there is no Event currently loaded, raises an IndexError (since the index of current Event is None).

class lfd.gui.imagechecker.botright.**BottomRight**(*parent*)

Bottom right Frame of the RightFrame of the app. This section contains all the active elements of the app, such as Buttons for selecting the DB, directory of images, moving to the next or previous image or changing the DB entries by verifying their truthfulness.

false(**args*)

Callback that sets the false_positive attribute of the current Event to True, persists the change to the DB, moves the current data index to the following data instance and updates the whole GUI.

nextimg(**args*)

Callback function that moves the current data index to the following one and updates the whole GUI.

previmg(**args*)

Callback function that moves to the previous data instance and updates the whole GUI.

search()

Opens a new window that allows user to input the run, camcol, filter and field designations of the Frame they would like to jump to. The search will jump to the first Event with the correct Frame designation.

As there can be multiple Events on the same Frame, user can provide the ordinal number of the Event they are interested in.

selectimages()
Re-initializes the apps selection of directory containing images and refreshes the whole GUI.

selectresults()
Re-initializes the apps selection of the Event database and refreshes the whole GUI.

true(**args*)
Callback that sets the false_positive attribute of the current Event to False, persists the change to the DB, moves the current data index to the following data instance and updates the whole GUI.

5.1.2 Data Browser

Data Browsers are classes that maintain index consistency between two Indexers and provide the functionality required to browse two indexers simultaneously. Since the source databases for results and images could be completely disjointed this means one of the Indexers is designated as a primary indexer. Browsing follows primary indexer while the secondary indexer is queried for the corresponding item.

class lfd.gui.imagechecker.databrowser.**Browser**(*primaryIndexer=None*, *secondaryIndexer=None*)
Browser is the generic abstraction of a browser that iterates over the pairs of (primary, secondary) items. Given objects capable of itemizing, i.e. indexing, the primary and secondary items Browser will ensure the consistency of the browsing index between the two indexers. For example, if we wanted to browse to the next value of primary indexer the following actions are performed:

1) invokes the next method of the primary indexer

2) identifies that item

3) invokes the get method of the secondary indexer.

This is neccessary since the sets of items indexed by primary and secondary can be completely disjoint.

The attributes and methods of this class are mainly private or hidden. By inheriting this class and declaring a dictionary class attribute "rename" on that class it is possible to rename the methods of this class into something more appropriate such as assigning the name 'images' to '_primary' when dealing with ImageBrowser class etc.

Parameters

- **primaryIndexer** (*lfd.gui.imagechecker.Indexer*) – the primary Indexer (EventIndexer or ImageIndexer)
- **secondaryIndexer** (*lfd.gui.imagechecker.Indexer*) – the secondary Indexer

get(*run*, *camcol*, *filter*, *field*, *which=0*)
Given frame specifiers (run, camcol, filter, field) select and advance both indexers to the item if possible. Relationship from primary to secondary indexer can be many to one, so providing 'which' allows selection on a particular secondary of interest.

getNext()
Advance the index of the primary by a step and then find if the secondary contains the newly selected object.

getPrevious()
Regress the index of the primary by a step and then find if the secondary contains the newly selected object.

class lfd.gui.imagechecker.databrowser.**EventBrowser**(*resdbURI=None*, *imgdbURI=None*)
A Browser which primary set of items to browse through are Events. For each Event indexed it will attempt to find a corresponding image. This Browser guarantees that all Events will be Browsed, but not all indexed images will be browsed through.

Parameters

- **resdbURI** (*str*) – URI of the database of Events (i.e. results)
- **imgdbURI** (*str*) – URI of the database of Images

event
The item, of the primary indexer, pointed to by the current index.

events
The items indexed by the primary indexer.

image
The item, of the secondary indexer, pointed to by the current index.

images
The items indexed by the secondary indexer.

initEvents(*URI*)
Instantiate the primary indexer.

initImages(*URI*)
Instantiate the secondary indexer.

class lfd.gui.imagechecker.databrowser.**GenericBrowser**
GenericBrowser metaclass offers the ability to rename the values of attributes being browsed through into something more appropriate. In the case of an EventBrowser for example that would be:

```
primary --> Event
secondary --> Images
```

and the other way around for ImageBrowser. This is completely superfluous and here more because I wanted to tr something out than out of any real neccessity.

class lfd.gui.imagechecker.databrowser.**ImageBrowser**(*resdbURI=None*, *imgdbURI=None*)
A Browser which primary set of items to browse through are Images. For each Image indexed it will attempt to find a corresponding Event. This Browser guarantees that all Images will be browsed, but not all indexed events will be browsed through.

Parameters

- **resdbURI** (*str*) – URI of the database of Events (i.e. results)
- **imgdbURI** (*str*) – URI of the database of Images

event
The item, of the secondary indexer, pointed to by the current index.

events
The items indexed by the secondary indexer.

image
The item, of the primary indexer, pointed to by the current index.

images
The items indexed by the primary indexer.

initEvents(*URI*)
Instantiate the secondary indexer.

initImages(*URI*)
Instantiate the primary indexer.

5.1.3 Indexers

Indexers establish order amongs items

class lfd.gui.imagechecker.indexers.**EventIndexer**(*URI=None*)
Indexes Events database providing a convenient way to establish order among the items.

_getFromFrameId(*run*, *camcol*, *filter*, *field*, *which=0*)
Queries the database for the Event and returns it. In case multiple Events correspond to the same frame identifier, supplying which selects one of the returned results.

The returned Event is expunged from the database session.

Parameters

- **run** (*int*) – run identifier
- **camcol** (*int*) – camcol identifier
- **filter** (*str*) – string identifier
- **field** (*int*) – field identifier
- **which** (*int*) – if multiple Events are returned, which one in particular is wanted

_getFromItemId(*eventid*)
Given an unique Event id queries the database for the row and returns it as an appropriate object. The returned object is expunged from the database session.

Parameters **eventid** (*int*) – unique id of the desired Event

commit()
Add the object back to the session and commit any changes made to it.

event
Returns the event pointed to by the current index.

initFromDB(*uri*)
Connects to a database and indexes all Events therein.

class lfd.gui.imagechecker.indexers.**ImageIndexer**(*URI=None*)
Indexes Image database providing a convenient way to establish order among the items.

_getFromFrameId(*run*, *camcol*, *filter*, *field*, *which=0*)
Queries the database for the Image and returns it. In case multiple Imagess correspond to the same frame identifier, supplying which selects one of the returned results.

The returned Image is expunged from the database session.

Parameters

- **run** (*int*) – run identifier
- **camcol** (*int*) – camcol identifier
- **filter** (*str*) – string identifier
- **field** (*int*) – field identifier
- **which** (*int*) – if multiple items are returned, which one in particular is wanted

_getFromItemId(*imageid*)
Given an unique image id queries the database for the row and returns it as an appropriate object. The returned object is expunged from the database session.

Parameters **eventid** (*int*) – unique id of the desired Event

image
Returns the image pointed to by the current index.

initFromDB(*uri*)
Connects to a database and indexes all Images therein.

class lfd.gui.imagechecker.indexers.**Indexer**(*items=None*)
Generic indexer of items in a database. Given a list of items or a database connection, session and table will produce an order-maintained list of item ids. Indexer allows us to move through the rows of the db in sequentiall or non-sequential manner while loading the currently pointed to object dynamically from the database.

Parameters **items** (*list, tuple*) – list of unique ids of the objects from the db

current
current position of the index

Type int

maxindex
the maximal value of the index

Type int

items
unique ids of DB rows, the item[current] points to the currently selected row of the DB

Type list

item
currently selected dynamically loaded from the DB as an object

Type object

_Indexer__step(*step*)
Makes a positive (forward) or negative (backwards) step in the list of items and loads the newly pointed to object.

_getFromFrameId(**args*, ***kwargs*)
Given a frame identifier (run, camcol, filter, field), and possibly *which*, queries the database for the object and returns it. Should be implementation specific perogative of classes that inehrit from Indexer

_getFromItemId(**args*, ***kwargs*)
Given an unique object id queries the database for the row and returns it as an appropriate object. The returned object is expunged from the database session.

get(*run=None*, *camcol=None*, *filter=None*, *field=None*, *which=0*, *itemid=None*)
If nothing is provided, jumps to the current index and loads the db row into item. If frame identifiers (run, camcol, filter, field) are given loads the object from the database and jumps the index to its position. If itemid is provided, loads the desired object and jumps to its index.

In some cases the frame identifiers can be shared among multiple rows (i.e. multiple Events on a Frame) in which case providing 'which' makes it possible to select a particular item from the returned set.

The selected item is expunged from the session.

Parameters

- **run** (*int*) – run identifier
- **camcol** (*int*) – camcol identifier
- **filter** (*str*) – string identifier
- **field** (*int*) – field identifier

- **which** (*int*) – if multiple items are returned, which one in particular is wanted
- **itemid** (*int*) – unique id of the desired item

goto (*index=None*, *itemid=None*)
If an 'index' is provided jumps to the given index. If 'itemid' is given, jumps to the index of the provided item id. If neither are given reloads the current item.

initFromDB (*dbURI*)
Connects to the given DB URI and indexes wanted rows from it.

next ()
Makes a step forward and retrieves the item.

previous ()
Makes a step backwards and retrieves the item.

skip (*steps*)
Skips 'steps' number of steps. Value of 'steps' can be positive or negative indicating forward or backward skip respectively.

5.1.4 Utilities

lfd.gui.imagechecker.utils.**create_imageDB** (*filenamestr*, *saveURI*, *echo=False*)
Finds all paths to matching files given by filenamestr, extracts their frame identifiers and stores them in a database given by saveURI.

Examples

Filenamestr can contain wildcards, f.e.:

```
>>> create_imageDB("/path/to/dir_containing_subdirs/*/*.png",
    "sqlite:///foo.db")
```

will find all /path/to/dir/subdirs/frame-run-camcol-filter-frame.png styled filenames and add their frame identifiers and paths to foo DB.

Parameters

- **filenamestr** (*str*) – wildcarded string that will be used to match all desired image files
- **saveURI** (*str*) – URI containing type and location of the images database

lfd.gui.imagechecker.utils.**eventId2Filename** (*eventid*, *type='.png'*)
From an event id constructs a SDSS styled filename via frameId2Filename function.

Parameters

- **eventid** (*int*) – unique Event identifier
- **type** (*str*) – file extension (.png, .jpeg etc...)

lfd.gui.imagechecker.utils.**eventId2FrameId** (*eventid*)
Returns frame identifiers (run, camcol, filter, field) for an Event identified by given event id.

Parameters **eventid** (*int*) – unique Event identifier

lfd.gui.imagechecker.utils.**filename2frameId** (*filename*)
From an SDSS style filename of the:

```
frame-{filter}-{run:06d}-{camcol}-{field:04}.fits.{type}
```

format extracts frame identifiers (run, camcol, filter, field).

Parameters **filename** (*str*) – just the filename, no prepended path

lfd.gui.imagechecker.utils.**filepath2frameId**(*filepath*)
From a filepath extracts SDSS frame identifiers. Filepath must be of the followng format:

```
/path/to/frame-{filter}-{run:06d}-{camcol}-{field:04}.fits.{type}
```

Parameters **filepath** (*str*) – path-like string

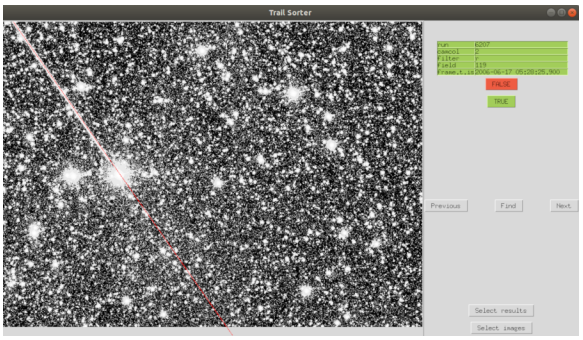
lfd.gui.imagechecker.utils.**frameId2Filename**(*run*, *camcol*, *filter*, *field*, *type='.png'*)
Translates between frame identifiers and SDSS style filename of the:

```
frame-{filter}-{run:06d}-{camcol}-{field:04}.fits.{type}
```

format, where type represents the .png, .jpg or other extensions.

Parameters

- **run** (*int*) – run identifier
- **camcol** (*int*) – camcol identifier
- **filter** (*str*) – string identifier
- **field** (*int*) – field identifier
- **type** (*str*) – file extension (.png, .jpeg etc. . .)



ImageChecker is a image browser designed to speed up results verification.

After processing large ammount of data, depending on the selected detection parameters, there would be none to a lot of false positive detections in the dataset. These potential detections would need to be manually checked in some situations to ensure the quality of the results.

The frames of interest, i.e. the results or its subset, would be converted, keeping in mind to scale and preprocess the images appropriately for their intended use, to one of the supported formats (png, jpeg, gif, ppm/pgm) and their frame identifiers and paths stored in a database. ImageCheker would connect to the results and images database and would be able to browse through keyed either on the images, or detected events.

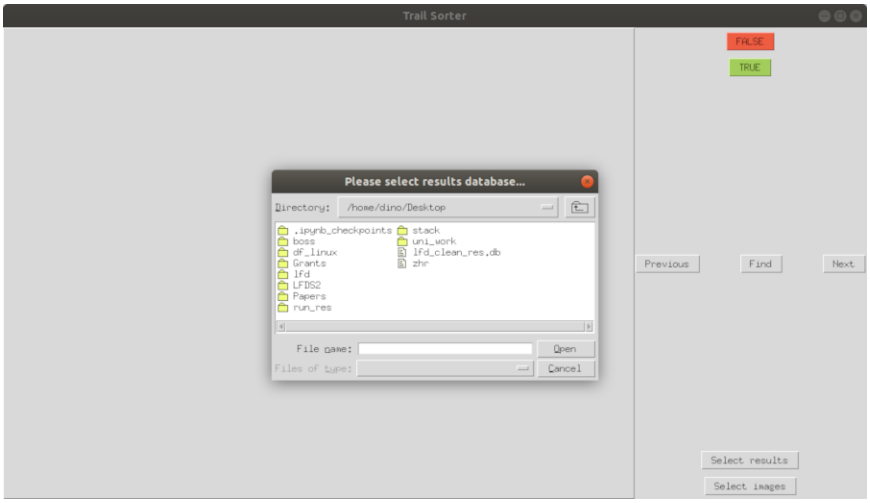
While this is possible to do on the cluster itself, using X forwarding, it is not recomended due to large potential latency and possibly long image download times. For fastest possible verification it is best to run ImageChecker locally. ImageChecker was designed to facilitate fast verification and correction of results and was not intended to be, yet another, FITS viewer.

To run the GUI import the package and use its *run* method or invoke the mainloop of the app itself:

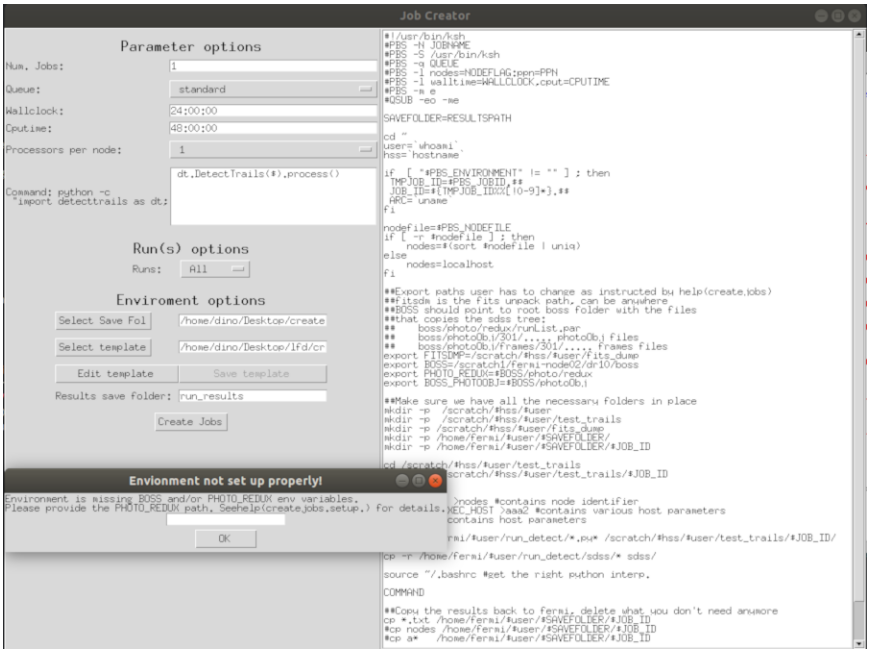
```
import lfd
lfd.gui.imagechecker.run()

# or optionally if one wants to potentially change default parameters
import lfd.gui.imagechecker as imgchkr
app = ImageChecker()
app.mainloop()
```

The user should be greeted with popup windows instructing them to select the results database first and the image database second. After successful connections to the databases were made (notice that there is no setup required as described in the *Database and setup*, the values are inferred at startup time from the selected DB's instead) user will be able to browse images using directional keys and a mouse.



5.2 Creating DQS jobs



To run the GUI import the package and use its *run* method or invoke the mainloop of the app itself:

```
import lfd
lfd.gui.jobcreator.run()

# or optionally if one wants to potentially change default parameters
import lfd.gui.imagechecker as imgchkr
app = JobCreator()
app.mainloop()
```

JobCreator is a GUI interface to createjobs module. It provides an easier to manage graphical interace to the functionality contained in the createjobs module and has additional practical features such as in-place template editing.

class lfd.gui.jobcreator.rightframe.**RightFrame**(*parent*)
RightFrame of the jobcreator gui. Contains the template from which jobs will be created. The template is not editable unless its state is changed by the eddittemplate function found in the leftbotframe module.

class lfd.gui.jobcreator.leftframe.**LeftFrame**(*parent*)
LeftFrame of the jobcreator gui. Contains 3 subframes: top, mid and bot. In order they control the folowing settings for job creation:

- Top - global execution settings for the jobs (f.e. wallclock, cputime, ppn...)
- Mid - invocation settings (f.e. from all runs, lists, results...)
- Bot - global environment settings (f.e. template, save paths, copy paths...)

Has to inherit from the root frame because job access is required. Will spawn additional windows promting user for settings for any particularily complex configurations.

getConf()
Reads the complete configuration selected by the user.

class lfd.gui.jobcreator.lefttopframe.**TopFrame**(*parent*, *row=0*, *col=0*)
Part of the LeftFrame of the GUI. Contains fields to select the: - number of jobs - queue type - wallclock limits - CPU time limits - processors per node limit (ppn) - command editor

Requires a parent Frame that has access to Job object so that the settings can propagate to it.

getcommand()
Reads the curent command TextBox and reduces it to a form compatible with the createjobs module. Separating lines in the TextBox by using <Return> key is allowed.

getn()
Reads the current value from the Entry for number of jobs and performs basic sanity checks. Will raise an error if the value in the box is unreadable or if the number of jobs is zero. Technically all these boxes should have verifiers but I can't really be bothered and this one is the one that will likely get changed the most.

class lfd.gui.jobcreator.leftmidframe.**MidFrame**(*parent*, *row=1*, *col=0*)
Part of the LeftFrame of the GUI. Contains the drop-down menu that selects the runs that will be processed. Currently the option to select from results database are a bit wonky.

Requires a parent Frame that has access to Job object so that the settings can propagate to it.

getResultsDBPath(*parent*, *update*)
Opens a file dialog window that enables user to navigate through the filesystem to select their desired database of results.

Expects to receive the parent window of the binding object and a StringVar that is used to represent this path. It will update its value which triggers its trace method, which updates the class attribute used to store the path to the database.

readRes(*parent*)
Callback that connects to the database given by the URI and path provided by the user and selects all existing Frames in that database. Selected frames are propagated to the runs attribute of the job object inherited from root. Expects to receive the parent window containing the binding object. The parent window is destroyed once all results are read in.

runFromSingle(*parent*, *runs*)
Callback function for the case when a 'Single' run source is chosen.

Parameters

- **– the parent window that contains the widget that registers this** (*parent*) – callback. This window will be destroyed at the end of this func.
- **– an Entry or a Text widget from which the value will be read out** (*runs*) – as.

runsFromList(*parent*, *runs*)
Callback function to convert a coma separated string of runs into a list of integers. Propagates the list to the job object inherited from root. Expects to receive the parent window containing the binding object. The parent window is destroyed once the conversion is complete.

selectRuns(*selection*)
Callback that will execute everytime the drop-down menu selection changes. For each of the option in the menu here we define an action that will triger the appropriate additional menus required to configure the job.

- All - the runs are allowed to be undefined, createjobs package will read the runlistAll file in $PHOTO_REDUX env var location for runs
- Single - a pop-up with an entry field is displayed where only 1 run id is permitted
- List - a pop-up with a textbox is displayed where a list of runs is given as a comma separated string
- Results - a pop-up window that lets user select the DB from which jobs will be created.

setResPath(**tmp*)
Callback 'observer' function used to track when the contents of an Entry changes. Specifically, tracks when the text value of an Entry used to select path to results database has changed. Updates the stored path to the results.

Expects the arguments corresponding to the invocation of trace method of a StringVar.

setUriPath(**tmp*)
Callback 'observer' function used to track when the contents of an Entry changes. Specifically, tracks when the text value of an Entry used to select URI path to results database has changed. Updates the stored URI path to the results.

Expects the arguments corresponding to the invocation of trace method of a StringVar.

class lfd.gui.jobcreator.leftbotframe.**BotFrame**(*parent*, *row=2*, *col=0*)
Bottom part of the LeftFrame of the GUI. Handles all of the paths involved in the creation of a Job and updates the template in RightFrame when the template path changes.

editTemplate()
A callback of a Button action that will change the state of the RightFrame Text box and make it editable.

saveTemplate()
A Button callback that will save the current template to a file. Spawns a file dialog to retrieve the save location. Changes the state of the RightFrame Text box back to un-editable.

setSavePathWithPrompt()
Callback that will spawn a directory selector through which a new path, where the job DQS files will be saved, can be selected.

setTemplatePath(**args*)
Callback that will track the Entry box of the template path and upon modification will cause the RightFrame template display to reload the new template.

setTemplatePathWithPrompt()
Callback that will spawn a file selector window through which a new template can be selected. See setTemplatePath. Will cause an update of the RightFrame to redisplay the newly selected template.

updateTemplatePath(*path*, *showerr=False*)
Updates the RightFrame's template display and replaces the current content with content read from a file at the provided path. If showerr is supplied an error will be raised if the given path does not exist. This is useful if the directory will be created after the path selection or if the update is called from a callback tied to a StringVar/Entry trace methods as a way to silence errors untill the full path has been manually inputed.

Parameters

- **path** (*str*) – path to the new template
- **showerr** (*bool*) – if False no error will be raised even if path does not exist, usefull when error needs to be raised later, on a callback initiated by a button click

LFD contains 2 GUI interfaces:

- Imagechecker is an image browser developed specifically to help with results verification.
- JobCreator is a GUI interface to createjobs module that helps with the creation of cluster job scritps because it's capable of displaying all selected parameters on-screen in a way that's not possible with the createjobs module.

CHAPTER 6

Analysis

Note: A small subset of detected linear features will belong to trails left by meteors when flying through the SDSS field of view (FOV). Most of this module is oriented to providing tools for analysis and additional functionality specifically suited to processing meteor trails.

Various miscelaneous functionality usefull in analyzing or understanding the behaviour of trails on images is provided in this module.

6.1 Profiles

6.1.1 Convolution

This module contains various convolution functionality specifically suited to work with provided convolutional objects.

lfd.analysis.profiles.convolution.**largest_common_scale**(**args*)

Finds the new appropriate common scale between given objects such that the new boundaries start at the leftmost and end at the rightmost object and the step between two points is the smallest step value for all objects.

It is very important that the scale of the two convolved functions is the same! F.e. convolving a functions with scale lengths of 1 arcsecond and 1 degree is drastically different than convolving functions with the same scale length.

Parameters ***args** (*list, tuple*) – set of objects we wish to reduce to the largest common scale

Example

```
>>> newscale = largest_common_scale(obj1, obj2, obj3, ...)
```

lfd.analysis.profiles.convolution.**convolve_seeing**(*obj*, *seeing*, *name='seeing-convolved'*)

Rescales, normalizes and convolves object and seeing.

Parameters

- **obj** (*lfd.analysis.profile.ConvolutionObject*) – brightness profile of object to be convolved
- **obj2** (*lfd.analysis.profile.ConvolutionObject*) – seeing profile that will be convolved with object
- **name** (*str*) – ConvolutionObjects can be assigned names, or will default to an appropriate name based on instantiation, which helps with tracking the current object status and with plotting.

lfd.analysis.profiles.convolution.**convolve_defocus**(*obj*, *defocus*, *name='defocus-convolved'*)

Rescales, normalizes and convolves object and defocus.

Parameters

- **obj** (*lfd.analysis.profile.ConvolutionObject*) – brightness profile of object to be convolved
- **defocus** (*lfd.analysis.profile.ConvolutionObject*) – defocus profile that will be convolved with object
- **name** (*str*) – ConvolutionObjects can be assigned names, or will default to an appropriate name based on instantiation, which helps with tracking the current object status and with plotting.

lfd.analysis.profiles.convolution.**convolve_seeing_defocus**(*obj*, *seeing*, *defocus*, *name='seeing-defocus-convolved'*)

Rescales, normalizes and then convolves object, seeing and defocus.

Parameters

- **obj** (*lfd.analysis.profile.ConvolutionObject*) – brightness profile of object to be convolved
- **defocus** (*lfd.analysis.profile.ConvolutionObject*) – defocus profile that will be convolved with object
- **seeing** (*lfd.analysis.profile.ConvolutionObject*) – seeing profile that will be convolved with the previous convolution result
- **name** (*str*) – ConvolutionObjects can be assigned names, or will default to an appropriate name based on instantiation, which helps with tracking the current object status and with plotting.

lfd.analysis.profiles.convolution.**convolve**(**args*, *name=None*)

Rescales, normalizes and convolves the provided ConvolutionObjects. Functionality applied in convolve_seeing/defocus or convolve_seeing_defocus functions is used when two or three ConvolutionObjects are provided. Otherwise the convolution is performed recursively which can be slow.

Parameters

- ***args** (*tuple, list*) – set of objects that will be convolved
- **name** (*str*) – ConvolutionObjects can be assigned names, or will default to an appropriate name based on instantiation, which helps with tracking the current object status and with plotting. If left None will default to "convolved"

Example

```
>>> convolve(obj, seeing, defocus)
```

class lfd.analysis.profiles.convolutionobj.**ConvolutionObject**(*obj*, *scale*, *name=None*)

Represents an object that can be convolved with other ConvolutionObjects or functions. Practical because the objects can be created with or without knowing their analytical expressions. ConvolutionObject can be instantiated from two arrays - scale and obj. Scale represents the "x-axis" and the obj array the function value at some x-coordinate. The function is then constructed by interpolating between the obj points. If an object's analytical light-curve is known then user can override the method 'f' such that it returns the value of the analytical expression at a given coordinate.

obj
brightness values of the object, its profile

Type list, tuple or np.array

scale
the "x" coordinates against which obj was evaluated, note that objects "center" (central maximal value) is generally centered on the 0 of the scale

Type list, tuple or np.array

__guessf
the function used for interpolation in case analytical form is unknown

Type np.interp1d

scaleleft
leftmost points of the scale

Type float

scaleright
rightmost points of the scale

Type float

objleft
rightmost points at which obj>0

Type float

objright
rightmost points at which obj>0

Type float

step
the difference between two scale points (fineness of the coordinates)

Type float

name
by default assigned to the class name instantiating this object. Useful to keep track of the function it represents but also for plotting.

Type str

Parameters

- **obj** (*list, tuple or np.array*) – brightness values of the object, its profile

- **scale** (*list, tuple or np.array*) – the "x" coordinates against which obj was evaluated, note that objects "center" (central maximal value) is generally centered on the 0 of the scale

calc_fwhm()
Calculates Full-Width-Half-Maximum of the object.

f(*r=None*)
Returns the value of the profile function at a point/array of points r. If analytical expression is not know interpolation between nearest points is attempted. If this occurs a warning is raised.

classmethod fromConvolution(*obj*, *scale*, *name=None*)
From given arrays obj and scale create a ConvolutionObject.

norm()
Renormalizes values so that maximal value is 1.

rescale(*x*, *y=None*, *step=None*)
Given a scale x recalculate the brightness values of obj over the new points. If only x and y are supplied, then a new scale is created with the new boundaries [x, y> and previously used step If x, y and step are provided then a new scale is created with new boundaries [x, y> where the distance between two points equals to step.

update()
Check and update the scale left and rightmost points. Find the first and last coordinate at which object is still brighter than 0. Recalcualtes step.

6.1.2 Object Profiles

Object profiles contains commonly used profiles of different types of source objects such as PointSource, Disk etc...

class lfd.analysis.profiles.objectprofiles.**PointSource**(*h*, *res=0.001*)
Simple point like source. Point-like sources are not resolved, therefore regardless of scale and size only a single element of obj will have a value

Parameters

- **h** (*float*) – height of the object, in meters
- **res** (*float*) – desired resolution in arcseconds. The scale step (the difference between two neighbouring "x-axis" points) is then determined by scaling the appropriate angular size of the object by the resolution, so that the object remains unresolved at the required resolution.

f(*r*)
Returns the intensity value of the object at a point. Evaluating a point source over only handfull of points is not well defined. The function may not behave properly if number of points is very small, i.e. 2 or 3 points only.

class lfd.analysis.profiles.objectprofiles.**GaussianSource**(*h*, *fwhm*, *res=0.001*, *units='meters'*)
Simple gaussian intensity profile.

Parameters

- **h** (*float*) – height of the object, in meters
- **fwhm** (*float*) – FWHM of the gaussian profile
- **res** (*float*) – desired resolution, in arcseconds

- **units** (*string*) – spatial units (meters by default) - currently not very well supported

f(*r*)
Evaluate the gaussian at a point.

class lfd.analysis.profiles.objectprofiles.**DiskSource**(*h*, *radius*, *res=0.001*)
Brightness profile of a disk-like source.

Parameters

- **h** (*float*) – height of the object, in meters
- **radius** (*float*) – radius of the objects disk, in meters
- **res** (*float*) – desired resolution, in arcseconds

f(*r*, *units='arcsec'*)
Returns the brightness value of the object at a point. By default the units of the scale are arcseconds but radians are also accepted.

width()
FWHM is not a good metric for measuring the end size of the object because disk-like profiles do not taper off towards the top. Instead width of the object (difference between first and last point with brightness above zero) is a more appropriate measure of the size of the object.

class lfd.analysis.profiles.objectprofiles.**RabinaSource**(*imgpath*, *h*)

```
Bektesevic & Vinkovic et. al. 2017 (arxiv: 1707.07223) Eq. (9)
```

a 1D integrated projection of fiducial 3D meteor head model as given by:

```
Rabina J., et al. 2016, J. Quant. Spectrosc. Radiat. Transf,178, 295
```

The integration of this profile is complicated and depends on variety of parameters, such as the angle of the observer linesight and meteor direction of travel. It is not practical to preform the integration every time brightness value needs to be estimated (too slow). A set of pregenerated projections of this profile to a plane were created where the angle between meteor direction of travel and observer linesight varies in the range from 0-1.5 radians (0-86 degrees) which are then used to produce this 1D profile. The 1D profile is created from a crosssection perpendicular to the meteors direction of travel. These 2D profiles are avilable as images in the rabina directory along with the required code to generate them. The default profile is then scaled appropriately to the desired height and any missing brightness values are then interpolated between the points.

Parameters

- **h** (*float*) – height of the object, in meters
- **imgpath** (*str*) – path to the image of the 2D integrated profile, the precalculated profiles can be found in lfd/analysis/profiles/rabina alongside with the code required to generate new ones.

f(*r*, *units='arcsec'*)
Returns the brightness value at a desired point.

6.1.3 Seeing

A slight missnomer of the module hides the fact that seeing contains both the 1D seeing profiles that reproduce the effects of atmospheric seeing on a object integrated brightness profile but also the defocus function as used and described in:

```
Bektesevic & Vinkovic et. al. 2017 (arxiv: 1707.07223)
```

class lfd.analysis.profiles.seeing.**GausKolmogorov**(*fwhm*, *scale=None*, *res=0.001*)

Simple Gaus-Kolmogorov seeing. Convolving with this function has the effect of blurring the original object.

Parameters

- **fwhm** (*float*) – FWHM of the Gaus-Kolmogorov profile
- **scale** (*list, tuple or np.array*) – if scale is given then GK will be evaluated over it
- **res** (*float*) – if scale is not given one will be created from the estimated width of the GK profile and resolution in arcseconds

f(*r*, *sigma=None*)

Evaluates the GK at a point r. Providing sigma estimates GK at r for a different GK distribution with a FWHM= sigma*2.436/1.035 - for convenience only.

class lfd.analysis.profiles.seeing.**FluxPerAngle**(*d*, *Ro*, *Ri*, *scale=None*, *units='arcsec'*, *res=0.001*)

The defocusing model as given by Equation (6) in:

```
Bektesevic & Vinkovic et. al. 2017 (arxiv: 1707.07223)
```

Parameters

- **d** (*float*) – the width of the object, in meters(?)
- **Ro** (*float*) – the diameter of the primary mirror of used instrument, in milimeters
- **Ri** (*float*) – the diameter of secondary mirror of the used instrument, in milimeters
- **scale** (*list, tuple or np.array*) – if scale is not given, appropriate scale will be created from d, Ro and Ri
- **units** (*str*) – in arcseconds by default - not very well supported
- **res** (*float*) – desired resolution in arcseconds, if scale is not given

f(*r*, *d=None*, *Ro=None*, *Ri=None*, *units='arcsec'*)

Estimate the value of the function at a point r. For convenience and quick calculations d, Ro, Ri and units can be provided too, otherwise the values used when creating an object are used.

6.1.4 Utilities

A collection of carious miscelaneous functionality that helps visualize the profiles and results of convolutions and their numerical values.

lfd.analysis.profiles.plotutils.**plot_profiles**(*ax*, *profiles*, **args*, *normed=True*, ***kwargs*)

Normalizes all given profiles and then plots them on a given axis. Set normed to False if normalization is not desired. Lables are determined from the name attribute of the profile. **args* and ***kwargs* are forwarded to the matplotlib plot function.

Profiles module consists of classes and functions neccessary to reproduce the plots from Bektesevic & Vinkovic et. al. 2017 (arxiv: 1707.07223).

The module provides classes for various different types of object brightness profiles (PointSource, GaussianSource, DiskSource, RabinaSource), seeing (Gaus- Kolmogorov), and defocusing effects (FluxPerAngle).

Sources represent 1D integrated brightness profiles of different source distributions as well as various parameters describing these distributions (such as FWHM, width, distance from instrument, variance etc.) in a common lookalike interface. They usually carry with them their resolution, scale or units (units are more for personal reference than actually useful).

Convolving Sources with Seeing and/or Defocus profile produce a new Source, which brightness profile corresponds to that of an source which is affected by these efects. So a convolution of a PointSource and seeing represents a source in focus observed through an atmoshere and PointSource convolved with Defocus corresponds to a defocused point source in ideal seeing (no seeing effects) etc

The module also provides a light wrapper around plotting utilities in matplotlib (but this is by no means extensive).

Examples

```
from lfd.analysis import profiles

point = profiles.PointSource(100)
seeing = profiles.GausKolmogorov(profiles.SDSSSEEING)
defocus = profiles.FluxPerAngle(100, *profiles.SDSS)

a = profiles.convolve(point, seeing, defocus)

import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1)
profiles.plot_profiles(ax, (point, seeing, defocus, a))
plt.legend()
plt.show()
```

Linear Feature Detector (LFD) library is a collection of packages that enable users to detect and analyze linear features on astronomical images. The code was designed to be run interactively, or at scale on a cluster.

LFD is a more complete version of LFDS that contains all of the never-published features of LFDS. Except for the linear feature detection code in the detecttrails module, most of the LFDS code was recoded from scratch and made compatible with Python 3 and OpenCv 3.0.

While most of the code is flexible enough to be appropriated for use with various different data sources it's primarily intended to be used to analyize SDSS images on a Sun Grid Engine (SGE) cluster for the purpose of detecting meteor streaks. For more details on its application in this area of astronomy refer to:

Bektesevic & Vinkovic, 2017, MNRAS, 1612.04748, Linear Feature Detection Algorithm for Astronomical Surveys - I. Algorithm description

Some of the previoulsy unreleased features include GUI interfaces to the PBS job creation module, a package for colation and analysis of results, a image browser designed to speed up the verification of results, a package for calculating and predicting the expected cross section of defocused trails and other common case plotting and analysis tools are provided.

CHAPTER 7

Dependencies

- Python 3+
- OpenCv 3+
- fitsio 0.9+
- numpy
- SqlAlchemy
- Astropy

Erin Sheldon's SDSS utilities come bundled with the provided code.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

B

C

D

E

F

G

I

J

L

M

T

U

W