srttools Documentation

Release 0.5rc3

Matteo Bachetti and the SRT Imaging team

Sep 17, 2018

Contents

Part I

Introduction

The Sardinia Radio Telescope Single Dish Tools (SDT) are a set of Python tools designed for the quicklook and analysis of single-dish radio data, starting from the backends present at the Sardinia Radio Telescope. They are composed of a Python (2.7, 3.4+) library for developers and a set of command-line scripts to soften the learning curve for new users.

The Python library is written following the modern coding standards documented in the Astropy Coding Guidelines. Automatic tests cover a significant fraction of the code, and are launched each time a commit is pushed to the Github repository. The Continuous Integration service Travis CI is used for that. The current version is 0.5-devXXX, indicating the development version towards 0.5. See below the tentative roadmap.

In the current implementation, spectroscopic and total-power on-the-fly scans are supported, both as part of standalone flux measurements through "cross scans" and as parts of a map. Maps are formed through a series of scans that swipe the source region.

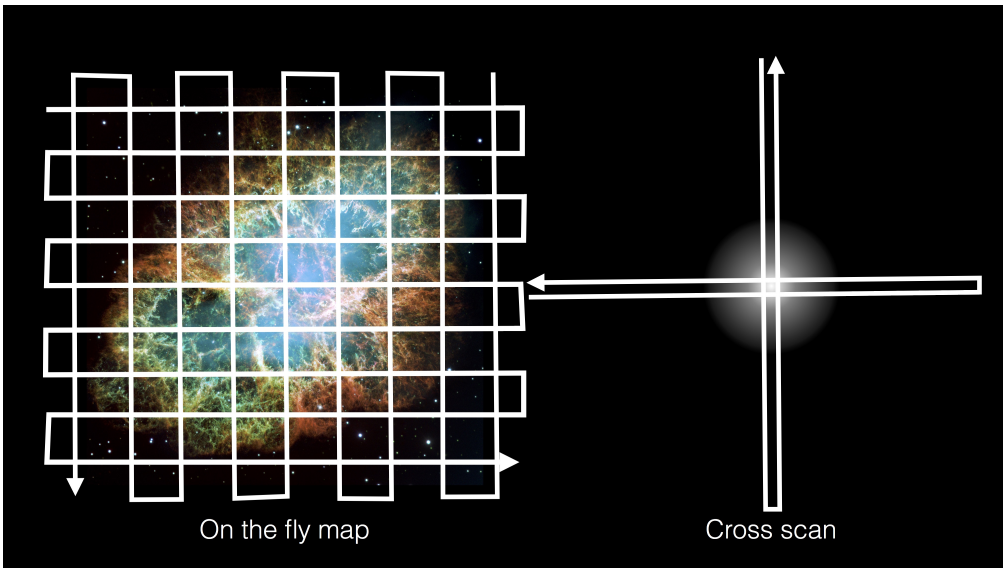

Fig. 1: **Figure 1.** On-the-fly maps vs cross scan strategies for single dish observations. The first is able to produce images, the second is used to obtain quick flux measurements of point-like sources.

Part II

Tentative Roadmap

- v.0.1: Simple map creation, draft calibrated fluxes
- v.0.2: Stable calibrated fluxes, use of multibeam in the K band
- v.0.3: Stabilization of interactive interface
- v.0.4: Generalized, user-supplied scanset filters
- v.**0.5**: Converters to MBFITS and CLASS
- v.0.6: Improved calibration, accept input gain curves
- v.0.7: Improved RFI support, using simple techniques of machine learning
- v.0.8: Full support of general coordinate systems, including Galactic
- v.1.0: code release.

Part III

Installation

CHAPTER 1

Prerequisites

1.1 Anaconda and virtual environment (recommended)

We strongly suggest to install the Anaconda Python distribution. Once the installation has finished, you should have a working conda command in your shell. First of all, create a new environment:

```
$ conda create -n py3 python=3
```

load the new environment:

```
$ source activate py3
```

and install the dependencies (including a few optional but recommended):

```
(py3) $ conda install astropy>=3 scipy numpy matplotlib pyyaml h5py statsmodels numba
```

```
$ pip install pyregion
```

1.2 Other Python distributions

Install the dependencies with pip (including a few optional but recommended):

```
$ pip install astropy>=3 scipy numpy matplotlib pyyaml h5py statsmodels numba pyregion
```

CHAPTER 2

Cloning and installation

Clone the repository:

```
(py3) $ cd /my/software/directory/
(py3) $ git clone https://github.com/matteobachetti/srt-single-dish-tools.git
```

or if you have deployed your SSH key to Github:

```
(py3) $ git clone git@github.com:matteobachetti/srt-single-dish-tools.git
```

Then:

```
(py3) $ cd srt-single-dish-tools
(py3) $ python setup.py install
```

That's it. After installation has ended, you can verify that software is installed by executing:

```
(py3) $ SDTimage -h
```

If the help message appears, you're done!

CHAPTER 3

Updating

To update the code, simply run `git pull` and reinstall:

```
(py3) $ git pull
(py3) $ python setup.py install
```

CHAPTER 4

Tutorials

4.1 Imaging Tutorial

In this tutorial, we will see how to obtain calibrated images and light curves from a set of on-the-fly (OTF) scans done with the SRT. Data are taken with the SARDARA ROACH2-based backend, with a bandwidth of 1024 MHz and 1024 channels. In this tutorial we will first learn how the software does a semi-automatic cleaning of the data from radio-frequency interferences (RFI), and how to tweak the relevant parameters to do the cleaning properly. Then, we will generate rough images with the default baseline subtraction algorithms. Afterwards, we will load a set of calibrators to perform the conversion from signal level to Janskys/pixel. Finally, we will apply the calibration to the previously generated images.

4.1.1 Inspect the observation

During a night of observations, we will in general observe a number of calibrators and sources, in random order. Our observation will be split into a series of directories:

```
(py3) $ ls
2016-05-04-220022_Src1/
2016-05-04-223001_Src1/
2016-05-04-230001_Cal1/
2016-05-04-230200_Cal2/
2016-05-04-230432_Src1/
2016-05-04-233523_Src1/
(....)
```

Some of these observations might have been done in different bands, or using different receivers, and you might have lost the list of observations (or the user was not the observer). The script `SDTinspector` is there to help, dividing the observations in groups based on observing time, backend, receiver, etc.:

```
(py3) $ SDTinspect */
Group 0, Backend = ROACH2, Receiver = CCB
------------------
```

(continues on next page)

(continued from previous page)

```
Src1, observation 0

Source observations:
2016-05-04-220022_Src1/
2016-05-04-223001_Src1/
2016-05-04-230432_Src1/

Calibrator observations:
2016-05-04-230001_Cal1/
2016-05-04-230200_Cal2/

Group 1, Backend = ROACH2, Receiver = KKG
---------------
Src1, observation 1

Source observations:
2016-05-04-233523_Src1/
(.....)

Calibrator observations:
(.....)
```

With the `-d` option, the script will also dump automatically a set of config files ready for the next step in the analysis:

```
(py3) $ SDTinspect */
Group 0, Backend = ROACH2, Receiver = CCB
(.....)
(py3) $ ls -alrt
CCB_ROACH_Src1_Obs0.ini
KKG_ROACH_Src1_Obs1.ini
```

4.1.2 Modify config files

If you did not pre-generate config files with the procedure above, you can generate a boilerplate config file with:

```
(py3) $ SDTlcurve --sample-config
(py3) $ ls
(...)
sample_config_file.ini
```

In the following, we will use the config files generated by SDTinspect, but it is very easy to adapt to the case of a custom-modified boilerplate.

Config files have this overall structure (slight changes might occur, like equals signs being changed to semicolons):

```
(py3) $ cat CCB_ROACH_Src1_Obs0.ini
[local]
workdir = .
datadir = .

[analysis]
projection = ARC
interpolation = spline
list_of_directories =
    2016-05-04-220022_Src1/
```

(continues on next page)

(continued from previous page)

```
    2016-05-04-223001_Src1/
    2016-05-04-230432_Src1/
calibrator_directories =
    2016-05-04-230001_Cal1/
    2016-05-04-230200_Cal2/
noise_threshold = 5
pixel_size = 1
goodchans =
```

You will likely not change the kind of interpolation or the projection in the plane of the sky (but if instead of `ARC` you want something different, all projections in this list are supported). `goodchans` is a list of channels that can be excluded from automatic filtering (for example, because they might contain an important spectral line.)

`pixel_size` is by default 1 arcminute. You might want to change this depending on the density of scans and the beam size at the observing frequency. Usually, 1/3 of the beam size is ok for dense OTF scan campaigns, while a larger value is better for sparse observations.

Also, you might know already that some observations were bad. In this case, it's sufficient to take them out of the list above.

4.1.3 Preprocess the files

This step is optional, because it can be merged with image production. However, for the sake of this tutorial we will proceed in this way for simplicity.

The easiest way to preprocess an observation is to call `SDTpreprocess` on a config file. The script will load all files, one by one, and do the following steps:

1. If the backend is spectroscopic, load each scan and filter out all channels whose that are more noisy than a given value of rms during the scan, then merge into a single channel. As an option (recommended), the user can specify a frequency interval that will be merged, otherwise the full frequency interval is taken: for this, one can use the option `--splat <minf:maxf>` where `minf`, `mmaxf` are in MHz referred to the *minimum* frequency of the interval. E.g. if our local oscillator is at 6900 MHz and we want to cut from 7000 to 7500, `minf` and `mmaxf` will be 100 and 600 resp. This process produces plots like the following:

```
(py3) $ SDTpreprocess -c CCB_TP_Src1_Obs0.ini --splat 80:1100 <more options>
```

2. About the `<more options>`: *If you select the option * ''–sub'', the single channels that are produced at step 1, or alternatively the single channels of a non-spectroscopic backend, will now be processed by a baseline subtraction routine. This routine, by default, applies an Asymmetric Least Squares Smoothing ('Eilers and Boelens 2005'_) to find the rough alignment of the scan, and then improves it by selecting the data that are closer to the baseline and making a standard least-square fit. This procedure is very fast and aligns the vast majority of scans in a fraction of a second. For more complicated scans, an interactive interface is also available, albeit with some portability issues that will be solved in future versions (use the ''–interactive'' option). It is possible to avoid regions with known strong sources. For now, they need to be specified by hand, with the ''-e'' option followed by a valid ds9-compatible region file containing *circular* regions in the `fk5` frame.*

3. The results of the first points are saved as `HDF5` files in the same directory as the original `fits` files. This makes it much faster to reload the scans for further use. **If the user wants to reprocess the files from scratch**, they need to delete these files first, or select the `--refilt` option.

4.1.4 Let's produce some images now!

Finally, let us execute the map calculation. If data were taken with a Total Power-like instrument and they do not contain spectral information, it is sufficient to run

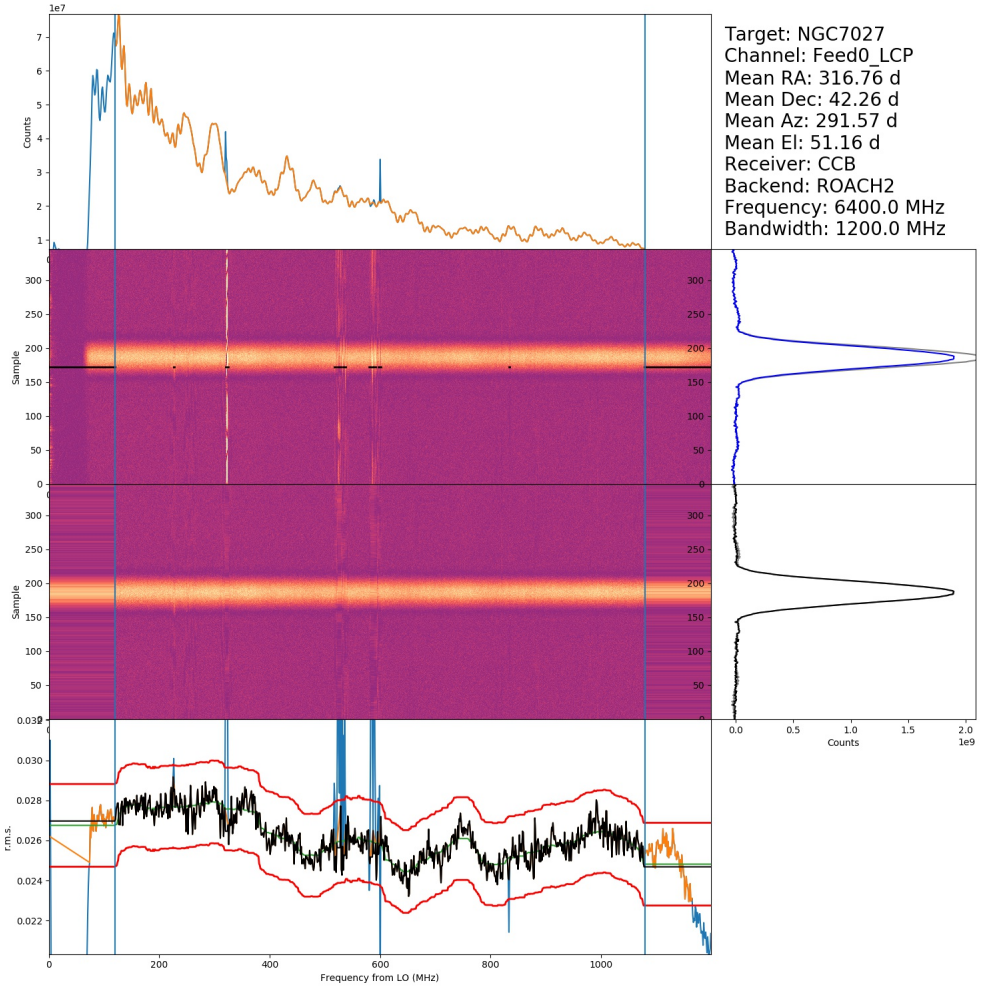

Fig. 1: **Figure 2.** Output of the automatic filtering procedure for an OTF scan of a calibrator. Channels where the root mean square of the signal is too high or too low are automatically filtered out. The threshold is encoded in the `noise_threshold` variable in the config file. This is the number of standard deviations from the median r.m.s. in a given interval. Optionally the user can choose the frequency interval (blue vertical lines). In the two right panels, one can see the scan before and after the cleaning. In the right-lower panel, the uncleaned scan is reported in grey to help the eye. The dynamical spectrum before and after the cleaning is shown in the two middle panels, and the effect of the cleaning on the scan is shown in the two right panels

```
(py3) $ SDTimage -c CCB_TP_Src1_Obs0.ini --sub
```

where CCB_TP_Src1_Obs0.ini should be substituted with the wanted config file. *This is also valid for spectroscopic scans that have already been preprocessed*

```
(py3) $ SDTimage -c CCB_ROACH_Src1_Obs0.ini --sub
```

Otherwise, if preprocessing were not executed before, specify the minimum and maximum frequency to select in the spectrum, with the `--splat` option (same as before)

```
(py3) $ SDTimage -c CCB_ROACH_Src1_Obs0.ini --splat <freqmin>:<freqmax> --sub
```

The above command will:

- Run through all the scans in the directories specified in the config file
- Clean them up if not already done in a previous step, in the same way of `SDTpreprocess`, including the baseline subtraction algorithm.
- Create a single frequency channel per polarization by summing the contributions between `freqmin` and `freqmax`, and discarding the remaining frequency channels, again if not already done in a previous step;
- Create the map in FITS format readable by DS9. The FITS extensions IMGCH0, IMGCH1, etc. contain an image for each polarization channel. The extensions IMGCH<no>-STD will contain the *error images* corresponding to IMGH<no>.

Note: When the user wants to reprocess the data from scratch, they have to remember the `--refilt` option. Otherwise, some steps like the spectral summation and the baseline subtraction are not repeated.

The automatic RFI removal procedure might have missed some problematic scan. The map might have, therefore, some residual "stripes" due to bad scans or wrong baseline subtraction.

The first thing to do, in these cases, is to go and look at the scans (by going through the PDF files produced by the calibration process in each subdirectory) and check that the noise threshold is appropriate for the level of noise found in scans. If it is not, as is often the case, and it is sufficient to re-run `SDTpreprocess` with the noise threshold changed in the config file to get a better cleaning of the data.

But `SDTimage` has an additional option to align the scans. It's called *global baseline subtraction*. This procedure makes a *global* fit (option `-g`) of all scans in an image, and tries to find the alignment of each scan that minimizes the *total rms* of the image. This procedure is only valid if the region that is fit is consistent with having zero average. This is, of course, not valid if the source is strong. In this case, together with the global fit option, we need to also specify a set of regions to neglect. This is done in two ways:

- through a ds9-compatible region file containing *circular* regions in *image* coordinates
- through the option `-e` followed by multiples of three numbers: X, Y and radius, in *image* coordinates (SAOimage ds9 or other imaging programs can create regions with these coordinates, one just needs to copy the numbers.).

In summary, to use the global fitting and discard the region centered at coordinates x,y=30,33 with radius 10 pixels, run

```
(py3) $ SDTimage -g -e 30 33 10 (...additional options)
```

4.1.5 Advanced imaging (TBC)

The automatic RFI removal procedure is often unable to clean all the data. The map might have some residual "stripes" due to bad scans. No worries! Launch the above command with the `--interactive` option

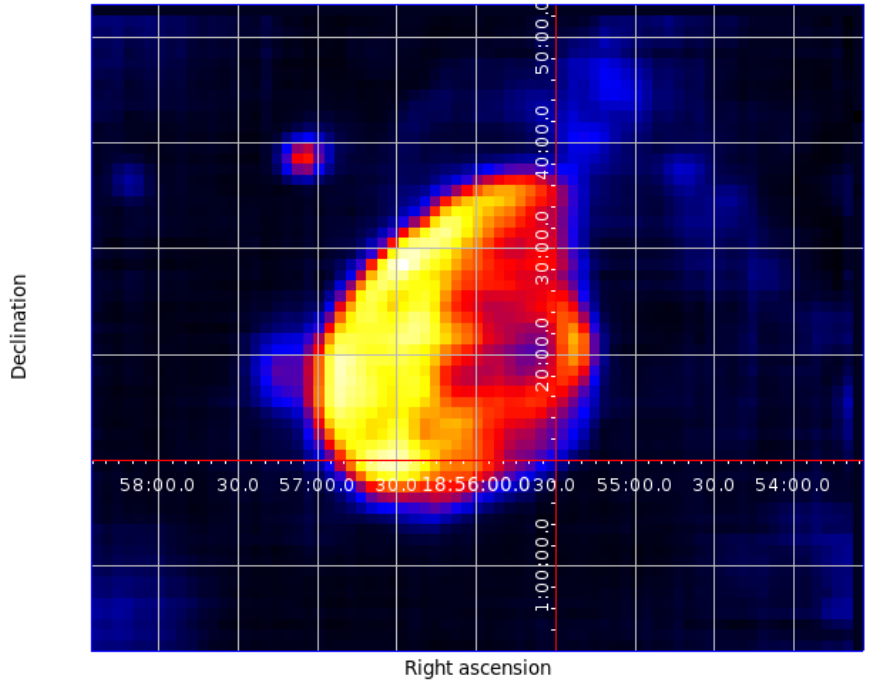

Fig. 2: **Figure 3.** Map produced by `SDTimage`

```
(py3) $ SDTimage -c MySource.ini --splat <freqmin>:<freqmax> --interactive
```

This will open a screen like this:

<placeholder>

where on the right you have the current status of the image, and on the left, larger, an image of the *standard deviation* of the pixels. Pixels with higher standard deviation might be due to a real source with high variability or high flux gradients, or to interferences. On this standard deviation image, you can point with the mouse and press 'A' on the keyboard to load all scans passing through that pixel. A second window will appear with a bunch of scans.

<placeholder>

Click on a bad scan and filter it according to the instructions printed in the terminal.

4.1.6 Calibration of images

To calibrate the images, one needs to call `SDTcal` with the same config files used for the images if they were produced with `SDTinspect`. Otherwise, one can construct an alternative config file with

```
(py3) $ SDTcal  --sample-config
```

and modify the configuration file adding calibrator directories below `calibrator_directories`

```
calibrator_directories :
   datestring1-3C295/
   datestring2-3C295/
```

Then, call again `SDTcal` with the `--splat` option, using **the same frequency range** of the sources.

```
(py3) $ SDTcal -c CCB_ROACH_Src1_Obs0.ini--splat <freqmin>:<freqmax> -o calibration.hdf5
```

Finally, call `SDTimage` with the `--calibrate` option, e.g.

```
(py3) $ SDTimage --calibrate calibration.hdf5 -c CCB_ROACH_Src1_Obs0.ini --splat <freqmin>:<freqmax> --
↪interactive
```

... and that's it! The image values will be expressed in Jy instead of counts, so that applying a region with DS9 and calculating the total flux inside the given region will return the actual total flux contained in the region.

4.1.7 Calibrated light curves

Go to a directory close to your data set. For example

```
(py3) $ ls
observation1/
observation2/
calibrator1/
calibrator2/
observation3/
calibrator3/
calibrator4/
(....)
```

It is not required that scan files are directly inside `observation1` etc., they might be inside subdirectories. The important thing is to correctly point to them in the configuration file as explained below.

Produce a dummy calibration file, to be modified, with

```
(py3) $ SDTlcurve --sample-config
```

This produces a boilerplate configuration file, that we modify to point to our observations, and give the correct information to our program

```
(py3) $ mv sample_config_file.ini MySource.ini  # give a meaningful name!
(py3) $ emacs MySource.ini

(... modify file...)

(py3) $ cat sample_config_file.ini
(...)
[analysis]
(...)
list_of_directories :
;;Two options: either a list of directories:
    dir1
    dir2
    dir3
calibrator_directories :
    cal1
    cal2
noise_threshold : 5

;; Channels to save from RFI filtering. It might indicate known strong spectral
```

(continues on next page)

(continued from previous page)

```
;; lines
goodchans :
```

Finally, execute the light curve creation. If data were taken with a Total Power-like instrument and they do not contain spectral information, it is sufficient to run

```
(py3) $ SDTlcurve -c MySource.ini
```

Otherwise, specify the minimum and maximum frequency to select in the spectrum, with the `--splat` option

```
(py3) $ SDTlcurve -c MySource.ini --splat <freqmin>:<freqmax>
```

where `freqmin`, `freqmax` are in MHz referred to the *minimum* frequency of the interval. E.g. if our local oscillator is at 6900 MHz and we want to cut from 7000 to 7500, `freqmin` and `freqmax` will be 100 and 600 resp. The above command will:

- Run through all the scans in the directories specified in the config file
- Clean them up with a rough but functional algorithm for RFI removal that makes use of the spectral information
- Create a csv file for each source, containing three columns: time, flux, flux error for each cross scan

The light curve will also be saved in a text file.

4.2 Format converters

Italian antennas baded on the ACS control system save raw data in a FITS format called `fitszilla`. Users of other facilities might find it useful to have data converted in a known format. The SRT single dish tools have a convenient script for that, called `SDTconvert`.

4.2.1 Read logs and update information in fits files

Some information is not available in fits files produced with old versions of Nuraghe. Provided that an `acs.xml` file is available in the system, we can get a table of the full log in human-readable format with

```
(py3) $ SDTparselog acs.xml --to-csv
```

There are a few shortcuts to the information we want to retrieve. E.g. we can get a list of FITS files with the calibration mark on:

```
(py3) $ SDTparselog acs.xml --list-calon
```

The calibration mark ON is one of the missing pieces of information in some versions of Nuraghe.

We can get the list of files where the calibration mark was on like above, and then apply it with `SDTbulkchange`, with the following shortcut:

```
(py3) $ SDTparselog acs.xml --list-calon | grep fits | xargs SDTbulkchange --apply-cal-mark
```

4.2.2 CLASS format

To get the data in a calibrated CLASS format readable into GILDAS, provided that the observations had a compatible ON-OFF or ON-OFF-CAL sequence (for CAL, apply the information to fits files as described above), type

```
(py3) $ SDTconvert -f classfits directory_of_observation
```

This will save the calibrated data into a directory. We use the FITS format readable into CLASS, and for convenience we also save a small script that, launched from the user's version of CLASS, is able to convert the data into the native CLASS format. We do not make the direct conversion to the binary CLASS format for portability issues, but we found that in practice the FITS format is understood correctly across the last four years of GILDAS versions.

4.2.3 Simple feed coordinate conversion

The FITS format used at the SRT only saves the coordinates of the central feed, and the coordinates of the other feeds need to be calculated based on their offsets in the focal plane.

SDT knows how to treat this problem. However, users wanting to analyze the data with their own software can use `SDTconvert`:

```
(py3) $ SDTconvert -f fitsmod directory_of_observation
```

This will create a separate extension called ‘‘COORD‘‘*n* for each feed, where *n* is the number of the feed. Feed 0 will not need a separate extension. Each extension will contain the updated right ascension and declination of the sky region observed by each feed.

4.2.4 MBFITS

Many European facilities use MBFITS as their raw data format. `SDTconvert` can convert the raw data from Italian facilities to this format.

To get the data in the Hierarchical MBFITS format, with the scan divided in multiple files under a directory tree, use

```
(py3) $ SDTconvert -f mbfits directory_of_observation
```

To get a single MBFITS file for each Frontend-Backend combination, use instead

```
(py3) $ SDTconvert -f mbfitsw directory_of_observation
```

4.2.5 SDFITS

CASA and other software accept data in the SDFITS format.

To get the data in the SDFITS format, with the scan divided in multiple files under a directory tree, use

```
(py3) $ SDTconvert -f mbfits directory_of_observation
```

To get a single MBFITS file for each Frontend-Backend combination, use instead

```
(py3) $ SDTconvert -f mbfitsw directory_of_observation
```

Part IV

Command line interface

CHAPTER 5

Command line interface

5.1 SDTbulkchange

```
usage: SDTbulkchange [-h] [-k KEY] [-v VALUE] [--apply-cal-mark] [--recursive]
                     [--debug]
                     [files [files ...]]

Change all values of a given column or header keyword in fits files

positional arguments:
  files                 Single files to preprocess

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     Path to key or data column. E.g. "EXT,header,KEY" to
                        change key KEY in the headerin extension EXT;
                        EXT,data,COL to change columnCOL in the data of
                        extension EXT
  -v VALUE, --value VALUE
                        Value to be written
  --apply-cal-mark      Short for -k "DATA TABLE,data,flag_cal" -v 1
  --recursive           Look for file in up to two subdirectories
  --debug               Plot stuff and be verbose
```

5.2 SDTcal

```
usage: SDTcal [-h] [--sample-config] [--nofilt] [-c CONFIG] [--splat SPLAT]
              [-o OUTPUT] [--show] [--check]
              [file]

Load a series of cross scans from a config file and use them as calibrators.
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  file                  Input calibration file

optional arguments:
  -h, --help            show this help message and exit
  --sample-config       Produce sample config file
  --nofilt              Do not filter noisy channels
  -c CONFIG, --config CONFIG
                        Config file
  --splat SPLAT         Spectral scans will be scrunched into a single channel
                        containing data in the given frequency range, starting
                        from the frequency of the first bin. E.g. '0:1000'
                        indicates 'from the first bin of the spectrum up to
                        1000 MHz above'. ':' or 'all' for all the channels.
  -o OUTPUT, --output OUTPUT
                        Output file containing the calibration
  --show                Show calibration summary
  --check               Check consistency of calibration
```

5.3 SDTconvert

```
usage: SDTconvert [-h] [-f FORMAT] [--test] [--detrend] [files [files ...]]

Load a series of scans and convert them to variousformats

positional arguments:
  files                 Single files to process or directories

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Format of output files (options: mbfits, indicating
                        MBFITS v. 1.65; mbfitsw, indicating MBFITS v. 1.65
                        wrapped in asingle file for each FEBE; fitsmod
                        (default), indicating a fitszilla with converted
                        coordinates for feed number *n* in a separate COORDn
                        extensions); classfits, indicating a FITS file
                        readable into CLASS, calibrated when possible;sdfits,
                        for the SDFITS convention
  --test                Only to be used in tests!
  --detrend             Detrend data before converting to MBFITS
```

5.4 SDTfake

```
usage: SDTfake [-h] [-s SOURCE_FLUX] [-n NOISE_AMPLITUDE] [-b BASELINE]
               [-g GEOMETRY GEOMETRY GEOMETRY GEOMETRY]
               [--beam-width BEAM_WIDTH] [--spacing SPACING] [-o OUTDIR_ROOT]
               [--scan-speed SCAN_SPEED] [--integration-time INTEGRATION_TIME]
               [--spectral-bins SPECTRAL_BINS] [--no-cal] [--debug]
```

(continues on next page)

(continued from previous page)

```
Simulate a single scan or a map with a point source.

optional arguments:
  -h, --help            show this help message and exit
  -s SOURCE_FLUX, --source-flux SOURCE_FLUX
                        Source flux in Jy
  -n NOISE_AMPLITUDE, --noise-amplitude NOISE_AMPLITUDE
                        White noise amplitude
  -b BASELINE, --baseline BASELINE
                        Baseline kind: "flat", "slope" (linearly
                        increasing/decreasing), "messy" (random walk) or a
                        number (which gives an amplitude to the random-walk
                        baseline, that would be 20 for "messy")
  -g GEOMETRY GEOMETRY GEOMETRY GEOMETRY, --geometry GEOMETRY GEOMETRY GEOMETRY GEOMETRY
                        Geometry specification: length_ra, length_dec,
                        width_ra, width_dec, in arcmins. A square map of 2
                        degrees would be specified as 120 120 120 120. A
                        cross-like map, 2x2 degrees wide but only along
                        1-degree stripes, is specified as 120 120 60 60
  --beam-width BEAM_WIDTH
                        Gaussian beam width in arcminutes
  --spacing SPACING     Spacing between scans in arcminutes (default 0.5)
  -o OUTDIR_ROOT, --outdir-root OUTDIR_ROOT
                        Output directory root. Here, source and calibrator
                        scans/maps will be saved in outdir/gauss_ra,
                        outdir/gauss_dec, outdir/calibrator1,
                        outdir/calibrator2, where outdir is the outdir root
  --scan-speed SCAN_SPEED
                        Scan speed in arcminutes/second
  --integration-time INTEGRATION_TIME
                        Integration time in seconds
  --spectral-bins SPECTRAL_BINS
                        Simulate a spectrum with this number of bins
  --no-cal              Don't simulate calibrators
  --debug               Plot stuff and be verbose
```

5.5 SDTimage

```
usage: SDTimage [-h] [--sample-config] [-c CONFIG] [--refilt] [--altaz]
                [--sub] [--interactive] [--calibrate CALIBRATE] [--nofilt]
                [-g] [-e EXCLUDE [EXCLUDE ...]] [--chans CHANS] [-o OUTFILE]
                [-u UNIT] [--destripe] [--npix-tol NPIX_TOL] [--debug]
                [--quick] [--scrunch-channels] [--bad-chans BAD_CHANS]
                [--splat SPLAT]
                [file]

Load a series of scans from a config file and produce a map.

positional arguments:
  file                  Load intermediate scanset from this file

optional arguments:
  -h, --help            show this help message and exit
  --sample-config       Produce sample config file
```

(continues on next page)

(continued from previous page)

```
  -c CONFIG, --config CONFIG
                        Config file
  --refilt              Re-run the scan filtering
  --altaz               Do images in Az-El coordinates
  --sub                 Subtract the baseline from single scans
  --interactive         Open the interactive display
  --calibrate CALIBRATE
                        Calibration file
  --nofilt              Do not filter noisy channels
  -g, --global-fit      Perform global fitting of baseline
  -e EXCLUDE [EXCLUDE ...], --exclude EXCLUDE [EXCLUDE ...]
                        Exclude region from global fitting of baseline and
                        baseline subtraction. It can be specified as X1, Y1,
                        radius1, X2, Y2, radius2 in image coordinates or as a
                        ds9-compatible region file in image or fk5 coordinates
                        containing circular regions to be excluded. Currently,
                        baseline subtraction only takes into account fk5
                        coordinates and global fitting image coordinates. This
                        will change in the future.
  --chans CHANS         Comma-separated channels to include in global fitting
                        (Feed0_RCP, Feed0_LCP, ...)
  -o OUTFILE, --outfile OUTFILE
                        Save intermediate scanset to this file.
  -u UNIT, --unit UNIT  Unit of the calibrated image. Jy/beam or Jy/pixel
  --destripe            Destripe the image
  --npix-tol NPIX_TOL   Number of pixels with zero exposure to tolerate when
                        destriping the image, or the full row or column is
                        discarded. Default None, meaning that the image will
                        be destriped as a whole
  --debug               Plot stuff and be verbose
  --quick               Calibrate after image creation, for speed (bad when
                        calibration depends on elevation)
  --scrunch-channels    Sum all the images from the single channels into one.
  --bad-chans BAD_CHANS
                        Channels to be discarded when scrunching, separated by
                        a comma (e.g. --bad-chans Feed2_RCP,Feed3_RCP )
  --splat SPLAT         Spectral scans will be scrunched into a single channel
                        containing data in the given frequency range, starting
                        from the frequency of the first bin. E.g. '0:1000'
                        indicates 'from the first bin of the spectrum up to
                        1000 MHz above'. ':' or 'all' for all the channels.
```

5.6 SDTinspect

```
usage: SDTinspect [-h] [-g GROUP_BY [GROUP_BY ...]] [--options OPTIONS] [-d]
                  [--only-after ONLY_AFTER] [--only-before ONLY_BEFORE]
                  directories [directories ...]

From a given list of directories, read the relevant information and link
observations to calibrators. A single file is read for each directory.

positional arguments:
  directories           Directories to inspect
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -g GROUP_BY [GROUP_BY ...], --group-by GROUP_BY [GROUP_BY ...]
  --options OPTIONS     Options to be written in config files; they have to be
                        specified as a string defining a dictionary. For
                        example,'{"pixel_size": 0.6, "noise_threshold": 5}'
  -d, --dump-config-files
  --only-after ONLY_AFTER
                        Only after a certain date and time, e.g. ``--only-
                        after 20150510-111020`` to indicate scans done after
                        11:10:20 UTC on May 10th, 2015
  --only-before ONLY_BEFORE
                        Only before a certain date and time, e.g. ``--only-
                        before 20150510-111020`` to indicate scans done before
                        11:10:20 UTC, May 10th, 2015
```

5.7 SDTlcurve

```
usage: SDTlcurve [-h] [-s SOURCE [SOURCE ...]] [--sample-config] [--nofilt]
                 [-c CONFIG] [--splat SPLAT] [-o OUTPUT]
                 [file]

Load a series of cross scans from a config file and obtain a calibrated curve.

positional arguments:
  file                  Input calibration file

optional arguments:
  -h, --help            show this help message and exit
  -s SOURCE [SOURCE ...], --source SOURCE [SOURCE ...]
                        Source or list of sources
  --sample-config       Produce sample config file
  --nofilt              Do not filter noisy channels
  -c CONFIG, --config CONFIG
                        Config file
  --splat SPLAT         Spectral scans will be scrunched into a single channel
                        containing data in the given frequency range, starting
                        from the frequency of the first bin. E.g. '0:1000'
                        indicates 'from the first bin of the spectrum up to
                        1000 MHz above'. ':' or 'all' for all the channels.
  -o OUTPUT, --output OUTPUT
                        Output file containing the calibration
```

5.8 SDTmonitor

```
usage: SDTmonitor [-h] [-c CONFIG] [--test] directory

Run the SRT quicklook in a given directory.

positional arguments:
  directory             Directory to monitor
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Config file
  --test                Only to be used in tests!
```

5.9 SDTopacity

```
usage: SDTopacity [-h] [--tatm TATM] [--tau0 TAU0] [--t0 T0] files [files ...]

Calculate opacity from a skydip scan and plot the fit results

positional arguments:
  files        File to inspect

optional arguments:
  -h, --help   show this help message and exit
  --tatm TATM  Atmospheric temperature
  --tau0 TAU0  Initial value for tau (to be fit)
  --t0 T0      Initial value for Tsys (to be fitted)
```

5.10 SDTparselog

```
usage: SDTparselog [-h] [--to-csv] [--list-calon] [files [files ...]]

Read ACS logs and return useful information

positional arguments:
  files         Single files to preprocess

optional arguments:
  -h, --help    show this help message and exit
  --to-csv      Save a CSV file with the results
  --list-calon  List files with calibration mark on
```

5.11 SDTpreprocess

```
usage: SDTpreprocess [-h] [-c CONFIG] [--sub] [--interactive] [--nofilt]
                     [--debug] [--splat SPLAT] [-e EXCLUDE [EXCLUDE ...]]
                     [files [files ...]]

Load a series of scans from a config file and preprocess them, or preprocess a
single scan.

positional arguments:
  files                 Single files to preprocess
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Config file
  --sub                 Subtract the baseline from single scans
  --interactive         Open the interactive display for each scan
  --nofilt              Do not filter noisy channels
  --debug               Plot stuff and be verbose
  --splat SPLAT         Spectral scans will be scrunched into a single channel
                        containing data in the given frequency range, starting
                        from the frequency of the first bin. E.g. '0:1000'
                        indicates 'from the first bin of the spectrum up to
                        1000 MHz above'. ':' or 'all' for all the channels.
  -e EXCLUDE [EXCLUDE ...], --exclude EXCLUDE [EXCLUDE ...]
                        Exclude region from global fitting of baseline and
                        baseline subtraction. It can be specified as X1, Y1,
                        radius1, X2, Y2, radius2 in image coordinates or as a
                        ds9-compatible region file in image or fk5 coordinates
                        containing circular regions to be excluded. Currently,
                        baseline subtraction only takes into account fk5
                        coordinates and global fitting image coordinates. This
                        will change in the future.
```

Part V

API documentation

CHAPTER 6

srttools package

6.1 Submodules

6.2 srttools.calibration module

Produce calibrated light curves.

`SDTlcurve` is a script that, given a list of cross scans from different sources, is able to recognize calibrators and use them to convert the observed counts into a density flux value in Jy.

class srttools.calibration.**CalibratorTable**(**args*, ***kwargs*)

Bases: `astropy.table.Table`

Table composed of fitted and tabulated fluxes.

Initialize the object.

Jy_over_counts(*channel=None*, *elevation=None*, *map_unit='Jy/beam'*, *good_mask=None*)

Compute the Jy/Counts conversion corresponding to a given map unit.

Parameters

channel : str

Channel name (e.g. 'Feed0_RCP', 'Feed0_LCP' etc.)

elevation : float or array-like

The elevation or a list of elevations

map_unit : str

A valid unit for the calibrated map (See the keys of FLUX_QUANTITIES)

good_mask : array of bools, default None

This mask can be used to specify the valid entries of the table. If None, the mask is set to an array of True values

Returns

fc : float or array-like

One conversion value for each elevation

fce : float or array-like

the uncertainties corresponding to each `fc`

Jy_over_counts_rough(*channel=None*, *map_unit='Jy/beam'*, *good_mask=None*)

Get the conversion from counts to Jy.

Returns

fc : float

flux density /count ratio

fce : float

uncertainty on `fc`

Other Parameters

channel : str

Name of the data channel

map_unit : str

A valid unit for the calibrated map (See the keys of FLUX_QUANTITIES)

good_mask : array of bools, default None

This mask can be used to specify the valid entries of the table. If None, the mask is set to an array of True values

beam_width(*channel=None*)

Calculate the (weighted) mean beam width, in radians.

Checks for invalid (nan and such) values.

calculate_src_flux(*channel=None*, *map_unit='Jy/beam'*, *source=None*)

Calculate source flux and error, pointing by pointing.

Uses the conversion factors calculated from the tabulated fluxes for all sources but the current, and the fitted Gaussian amplitude for the current source. Updates the calibrator table and returns the average flux

Parameters

channel : str or list of str

Data channel

map_unit : str

Units in the map (default Jy/beam)

source : str

Source name. Must match one of the sources in the table. Default

Returns

mean_flux : array of floats

Array with as many channels as the input ones

mean_flux_err : array of floats

Uncertainties corresponding to mean_flux

calibrate()
Calculate the calibration constants.

The following conversion functions are calculated for each tabulated cross scan belonging to a calibrator:

- 'Flux/Counts' and 'Flux/Counts Err': Tabulated flux density divided by the _height_ of the fitted Gaussian. This is used, e.g. to calibrate images in Jy/beam, as it calibrates the local amplitude to the flux density
- 'Flux Integral/Counts' and 'Flux Integral/Counts Err': Tabulated flux density divided by the _volume_ of the 2D Gaussian corresponding to the fitted cross scans, assuming a symmetrical beam (which is generally not the case, but a good approximation). This is used, e.g., to perform the calibration in Jy/pixel: Each pixel will be normalized to the expected total flux in the corresponding pixel area

See also:

srttools.calibration.CalibratorTable.from_scans

check_consistency(*channel=None*, *epsilon=0.05*)
Check the consistency of calculated and fitted flux densities.

For each source in the srttools' calibrator list, use calculate_src_flux to calculate the source flux ignoring the tabulated value, and compare the calculated and tabulated values.

Returns
retval : bool

True if, for all calibrators, the tabulated and calculated values of the flux are consistent. False otherwise.

check_not_empty()
Check that table is not empty.

Returns
good : bool

True if all checks pass, False otherwise.

check_up_to_date()
Check that the calibration information is up to date.

Returns
good : bool

True if all checks pass, False otherwise.

compute_conversion_function(*map_unit='Jy/beam'*, *good_mask=None*)
Compute the conversion between Jy and counts.

Try to get a meaningful second-degree polynomial fit over elevation. Revert to the rough function Jy_over_counts_rough in case statsmodels is not installed. In this latter case, only the baseline value is given for flux conversion and error. These values are saved in the calibration_coeffs and calibration_uncerts attributes of CalibratorTable, and a dictionary called calibration is also created. For each channel, this dictionary contains either None or an object. This object is the output of a fit procedure in statsmodels. The method object.predict(X) returns the calibration corresponding to elevation X.

counts_over_Jy(*channel=None*, *elevation=None*)
Get the conversion from Jy to counts.

from_scans(*scan_list=None*, *debug=False*, *freqsplat=None*, *config_file=None*, *nofilt=False*, *plot=False*)
Load source table from a list of scans.

For each scan, a fit is performed. Since we are assuming point-like sources here, the fit is a Gaussian plus a slope. The centroid, width and amplitude of the fit fill out new rows of the CalibratorTable ('Fit RA' or 'Fit Dec', 'Width' and 'Counts' respectively).

Parameters

scan_list : list of str

List of files containing cross scans to be fitted

config_file : str

File containing the configuration (list of directories etc.)

Returns

retval : bool

True if at least one scan was correctly processed

Other Parameters

debug : bool

Throw debug information

freqsplat : str

List of frequencies to be merged into one. See `srttools.scan.interpret_frequency_range()`

nofilt : bool

Do not filter the noisy channels of the scan. See `srttools.scan.clean_scan_using_variability`

plot : bool

Plot diagnostic plots? Default False, True if debug is True.

See also:

`srttools.scan.interpret_frequency_range`

get_fluxes()

Get the tabulated flux of the source, if listed as calibrators.

Updates the table.

plot_two_columns(*xcol*, *ycol*, *xerrcol=None*, *yerrcol=None*, *ax=None*, *channel=None*, *xfactor=1*, *yfactor=1*, *color=None*, *test=False*)

Plot the data corresponding to two given columns.

show()

Show a summary of the calibration.

update()

Update the calibration information.

Execute `get_fluxes`, `calibrate` and `compute_conversion_function`

write(*fname*, **args*, ***kwargs*)

Same as Table.write, but adds path information for HDF5.

srttools.calibration.**read_calibrator_config**()

Read the configuration of calibrators in data/calibrators.

Returns

configs : dict

Dictionary containing the configuration for each calibrator. Each key is the name of a calibrator. Each entry is another dictionary, in one of the following formats: 1) {'Kind' : 'FreqList', 'Frequencies' : [...], 'Bandwidths' : [...], 'Fluxes' : [...], 'Flux Errors' : [...]} where 'Frequencies' is the list of observing frequencies in GHz, 'Bandwidths' is the list of bandwidths in GHz, 'Fluxes' is the list of flux densities in Jy from the literature and 'Flux Errors' are the uncertainties on those fluxes. 2) {'Kind' : 'CoeffTable', 'CoeffTable': {'coeffs' : 'time, a0, a0e, a1, a1e, a2, a2e, a3, a3e

2010.0,0 ...}}

where the 'coeffs' key contains a dictionary with the table of coefficients a la Perley & Butler ApJS 204, 19 (2013), as a comma-separated string.

Examples

```
>>> calibs = read_calibrator_config()
>>> calibs['DummyCal']['Kind']
'CoeffTable'
>>> 'coeffs' in calibs['DummyCal']['CoeffTable']
True
```

6.3 srttools.fit module

Useful fitting functions.

srttools.fit.**contiguous_regions**(*condition*)

Find contiguous True regions of the boolean array "condition".

Return a 2D array where the first column is the start index of the region and the second column is the end index.

Parameters

condition : boolean array

Returns

idx : [[i0_0, i0_1], [i1_0, i1_1], ...]

A list of integer couples, with the start and end of each True blocks in the original array

Notes

From http://stackoverflow.com/questions/4494404/

find-large-number-of-consecutive-values-fulfilling- condition-in-a-numpy-array

srttools.fit.**ref_std**(*array*, *window*)

Minimum standard deviation along an array.

If a data series is noisy, it is difficult to determine the underlying standard deviation of the original series. Here, the standard deviation is calculated in a rolling window, and the minimum is saved, because it will likely be the interval with less noise.

Parameters

array : numpy.array object or list

Input data

window : int or float

Number of bins of the window

Returns

ref_std : float

The reference Standard Deviation

srttools.fit.**ref_mad**(*array*, *window*)

Ref. Median Absolute Deviation of an array, rolling median-subtracted.

If a data series is noisy, it is difficult to determine the underlying statistics of the original series. Here, the MAD is calculated in a rolling window, and the minimum is saved, because it will likely be the interval with less noise.

Parameters

array : `numpy.array` object or list

Input data

window : int or float

Number of bins of the window

Returns

ref_std : float

The reference MAD

srttools.fit.**linear_fun**(*x*, *q*, *m*)

A linear function.

Parameters

x : float or array

The independent variable

m : float

The slope

q : float

The intercept

Returns

y : float or array

The dependent variable

srttools.fit.**linear_fit**(*x*, *y*, *start_pars*, *return_err=False*)

A linear fit with any set of data.

Parameters

x : array-like

y : array-like

start_pars : [q0, m0], floats

Intercept and slope of linear function

Returns

par : [q, m], floats

Fitted intercept and slope of the linear function

srttools.fit.**offset**(*x*, *off*)
An offset.

srttools.fit.**offset_fit**(*x*, *y*, *offset_start=0*, *return_err=False*)
Fit a constant offset to the data.

Parameters
x : array-like

y : array-like

offset_start : float

Constant offset, initial value

Returns
offset : float

Fitted offset

srttools.fit.**baseline_rough**(*x*, *y*, *start_pars=None*, *return_baseline=False*, *mask=None*)
Rough function to subtract the baseline.

Parameters
x : array-like

the sample time/number/position

y : array-like

the data series corresponding to x

start_pars : [q0, m0], floats

Intercept and slope of linear function

Returns
y_subtracted : array-like, same size as y

The initial time series, subtracted from the trend

baseline : array-like, same size as y

Fitted baseline

Other Parameters
return_baseline : bool

return the baseline?

mask : array of bools

Mask indicating the good x and y data. True for good, False for bad

srttools.fit.**purge_outliers**(*y*, *window_size=5*, *up=True*, *down=True*, *mask=None*, *plot=False*)
Remove obvious outliers.

Attention: This is known to throw false positives on bona fide, very strong Gaussian peaks

srttools.fit.**baseline_als**(*x*, *y*, *lam=None*, *p=None*, *niter=40*, *return_baseline=False*, *offset_correction=True*, *mask=None*, *outlier_purging=True*)
Baseline Correction with Asymmetric Least Squares Smoothing.

Parameters
x : array-like

the sample time/number/position

y : array-like

the data series corresponding to x

lam : float

the lambda parameter of the ALS method. This control how much the baseline can adapt to local changes. A higher value corresponds to a stiffer baseline

p : float

the asymmetry parameter of the ALS method. This controls the overall slope tollerated for the baseline. A higher value correspond to a higher possible slope

Returns

y_subtracted : array-like, same size as y

The initial time series, subtracted from the trend

baseline : array-like, same size as y

Fitted baseline. Only returned if return_baseline is True

Other Parameters

niter : int

The number of iterations to perform

return_baseline : bool

return the baseline?

offset_correction : bool

also correct for an offset to align with the running mean of the scan

outlier_purging : bool

Purge outliers before the fit?

mask : array of bools

Mask indicating the good x and y data. True for good, False for bad

srttools.fit.**fit_baseline_plus_bell**(*x*, *y*, *ye=None*, *kind='gauss'*)

Fit a function composed of a linear baseline plus a bell function.

Parameters

x : array-like

the sample time/number/position

y : array-like

the data series corresponding to x

Returns

mod_out : `Astropy.modeling.model` object

The fitted model

fit_info : dict

Fit info from the Astropy fitting routine.

Other Parameters

ye : array-like

the errors on the data series

kind: str

Can be 'gauss' or 'lorentz'

srttools.fit.**total_variance**(*xs*, *ys*, *params*)

Calculate the total variance of a series of scans.

This functions subtracts a linear function from each of the scans (excluding the first one) and calculates the total variance.

Parameters

xs : list of array-like [array1, array2, ...]

list of arrays containing the x values of each scan

ys : list of array-like [array1, array2, ...]

list of arrays containing the y values of each scan

params : list of array-like [[q0, m0], [q1, m1], ...]

list of arrays containing the parameters [m, q] for each scan.

Returns

total_variance : float

The total variance of the baseline-subtracted scans.

srttools.fit.**align**(*xs*, *ys*)

Given the first scan, it aligns all the others to that.

Parameters

xs : list of array-like [array1, array2, ...]

list of arrays containing the x values of each scan

ys : list of array-like [array1, array2, ...]

list of arrays containing the y values of each scan

Returns

qs : array-like

The list of intercepts maximising the alignment, one for each scan

ms : array-like

The list of slopes maximising the alignment, one for each scan

6.4 srttools.global_fit module

Functions to clean images by fitting linear trends to the initial scans.

srttools.global_fit.**fit_full_image**(*scanset*, *chan='Feed0_RCP'*, *feed=0*, *excluded=None*, *par=None*)

Get a clean image by subtracting linear trends from the initial scans.

Parameters

scanset : a :class:`ScanSet` instance

The scanset to be fit

Returns

new_counts : array-like

The new Counts column for scanset, where a baseline has been subtracted from each scan to produce the cleanest image background.

Other Parameters

chan : str

channel of the scanset to be fit. Defaults to `"Feed0_RCP"`

feed : int

feed of the scanset to be fit. Defaults to 0

excluded : [[centerx0, centery0, radius0]]

List of circular regions to exclude from fitting (e.g. strong sources that might alter the total rms)

par : [m0, q0, m1, q1, . . .] or None

Initial parameters – slope and intercept for linear trends to be subtracted from the scans

`srttools.global_fit.`**`display_intermediate`**(*scanset*, *chan='Feed0_RCP'*, *feed=0*, *excluded=None*, *parfile=None*, *factor=1*)

Display the intermediate steps of global_fitting.

Parameters

scanset : a :class:`ScanSet` instance

The scanset to be fit

Other Parameters

chan : str

channel of the scanset to be fit. Defaults to `"Feed0_RCP"`

feed : int

feed of the scanset to be fit. Defaults to 0

excluded : [[centerx0, centery0, radius0]]

List of circular regions to exclude from fitting (e.g. strong sources that might alter the total rms)

parfile : str

File containing the parameters, in the same format saved by _callback

6.5 srttools.histograms module

This module contains a fast replacement for numpy's histogramdd and histogram2d. Two changes were made. The first was replacing

np.digitize(a, b)

with

np.searchsorted(b, a, "right")

This performance bug is explained on https://github.com/numpy/numpy/issues/2656 The speedup is around 2x for big number of bins (roughly >100). It assumes that the bins are monotonic.

The other change is to allow lists of weight arrays. This is advantageous for resampling as there is just one set of coordinates but several data arrays (=weights). Therefore repeated computations are prevented.

srttools.histograms.**histogram2d**(*x*, *y*, *bins=10*, *bin_range=None*, *normed=False*, *weights=None*)

Compute the bi-dimensional histogram of two data samples.

Parameters

x : array_like, shape (N,)

An array containing the x coordinates of the points to be histogrammed.

y : array_like, shape (N,)

An array containing the y coordinates of the points to be histogrammed.

bins : int or [int, int] or array_like or [array, array], optional

The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
- If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

bin_range : array_like, shape(2,2), optional

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the `bins` parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

normed : bool, optional

If False, returns the number of samples in each bin. If True, returns the bin density `bin_count / sample_count / bin_area`.

weights : array_like, shape(N,), optional

An array of values `w_i` weighing each sample `(x_i, y_i)`. Weights are normalized to 1 if `normed` is True. If `normed` is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin. Weights can also be a list of (weight arrays or None), in which case a list of histograms is returned as H.

Returns

H : ndarray, shape(nx, ny)

The bi-dimensional histogram of samples `x` and `y`. Values in `x` are histogrammed along the first dimension and values in `y` are histogrammed along the second dimension.

xedges : ndarray, shape(nx,)

The bin edges along the first dimension.

yedges : ndarray, shape(ny,)

The bin edges along the second dimension.

See also:

histogram
1D histogram

histogramdd
Multidimensional histogram

Notes

When `normed` is True, then the returned histogram is the sample density, defined such that the sum over bins of the product `bin_value * bin_area` is 1.

Please note that the histogram does not follow the Cartesian convention where `x` values are on the abscissa and `y` values on the ordinate axis. Rather, `x` is histogrammed along the first dimension of the array (vertical), and `y` along the second dimension of the array (horizontal). This ensures compatibility with `histogramdd`.

Examples

```
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

Construct a 2D-histogram with variable bin width. First define the bin edges:

```
>>> xedges = [0, 1, 1.5, 3, 5]
>>> yedges = [0, 2, 3, 4, 6]
```

Next we create a histogram H with random bin content:

```
>>> x = np.random.normal(3, 1, 100)
>>> y = np.random.normal(1, 1, 100)
>>> H, xedges, yedges = np.histogram2d(y, x, bins=(xedges, yedges))
```

Or we fill the histogram H with a determined bin content:

```
>>> H = np.ones((4, 4)).cumsum().reshape(4, 4)
>>> print((H[::-1]))  # This shows the bin content in the order as plotted
[[ 13.  14.  15.  16.]
 [  9.  10.  11.  12.]
 [  5.   6.   7.   8.]
 [  1.   2.   3.   4.]]
```

Imshow can only do an equidistant representation of bins:

```
>>> fig = plt.figure(figsize=(7, 3))
>>> ax = fig.add_subplot(131)
>>> _ = ax.set_title('imshow: equidistant')
>>> im = plt.imshow(H, interpolation='nearest', origin='low',
...                 extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])
```

pcolormesh can display exact bin edges:

```
>>> ax = fig.add_subplot(132)
>>> _ = ax.set_title('pcolormesh: exact bin edges')
>>> X, Y = np.meshgrid(xedges, yedges)
>>> _ = ax.pcolormesh(X, Y, H)
>>> _ = ax.set_aspect('equal')
```

NonUniformImage displays exact bin edges with interpolation:

```
>>> ax = fig.add_subplot(133)
>>> _ = ax.set_title('NonUniformImage: interpolated')
>>> im = mpl.image.NonUniformImage(ax, interpolation='bilinear')
```

(continues on next page)

(continued from previous page)

```
>>> xcenters = xedges[:-1] + 0.5 * (xedges[1:] - xedges[:-1])
>>> ycenters = yedges[:-1] + 0.5 * (yedges[1:] - yedges[:-1])
>>> _ = im.set_data(xcenters, ycenters, H)
>>> _ = ax.images.append(im)
>>> _ = ax.set_xlim(xedges[0], xedges[-1])
>>> _ = ax.set_ylim(yedges[0], yedges[-1])
>>> _ = ax.set_aspect('equal')
>>> _ = plt.close(fig)
```

srttools.histograms.**histogramdd**(*sample*, *bins=10*, *bin_range=None*, *normed=False*, *weights=None*)

Compute the multidimensional histogram of some data.

Parameters

sample : array_like

The data to be histogrammed. It must be an (N,D) array or data that can be converted to such. The rows of the resulting array are the coordinates of points in a D dimensional polytope.

bins : sequence or int, optional

The bin specification:

- A sequence of arrays describing the bin edges along each dimension.
- The number of bins for each dimension (nx, ny, ... =bins)
- The number of bins for all dimensions (nx=ny=...=bins).

bin_range : sequence, optional

A sequence of lower and upper bin edges to be used if the edges are not given explicitly in `bins`. Defaults to the minimum and maximum values along each dimension.

normed : bool, optional

If False, returns the number of samples in each bin. If True, returns the bin density `bin_count / sample_count / bin_volume`.

weights : array_like (N,), optional

An array of values `w_i` weighing each sample `(x_i, y_i, z_i, ...)`. Weights are normalized to 1 if normed is True. If normed is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin. Weights can also be a list of (weight arrays or None), in which case a list of histograms is returned as H.

Returns

H : ndarray

The multidimensional histogram of sample x. See normed and weights for the different possible semantics.

edges : list

A list of D arrays describing the bin edges for each dimension.

See also:

`histogram`

1-D histogram

histogram2d
2-D histogram

Examples

```
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5, 8, 4), 6, 9, 5)
```

6.6 srttools.imager module

Produce calibrated images.

SDTimage is a script that, given a list of cross scans composing an on-the-fly map, is able to calculate the map and save it in FITS format after cleaning the data.

class srttools.imager.**ScanSet**(*data=None*, *norefilt=True*, *config_file=None*, *freqsplat=None*, *nofilt=False*, *nosub=False*, ***kwargs*)

Bases: astropy.table.Table

Class obtained by a set of scans.

Once the scans are loaded, this class contains all functionality that will be used to produce (calibrated or uncalibrated) maps with WCS information.

Parameters

data : str or None

data can be one of the following: + a config file, containing the information on the scans to load + an HDF5 archive, containing a former scanset + another ScanSet or an Astropy Table

config_file : str

Config file containing the parameters for the images and the directories containing the image and calibration data

norefilt : bool

See srttools.scan.Scan

freqsplat : str

See srttools.scan.interpret_frequency_range

nofilt : bool

See srttools.scan.clean_scan_using_variability

nosub : bool

See srttools.scan.Scan

Other Parameters

kwargs : additional arguments

These will be passed to Scan initializers

Examples

```
>>> scanset = ScanSet()  # An empty scanset
>>> isinstance(scanset, ScanSet)
True
```

analyze_coordinates(*altaz=False*)
Save statistical information on coordinates.

apply_user_filter(*user_func=None*, *out_column=None*)
Apply a user-supplied function as filter.

Parameters
user_func : function

This function needs to accept a `scanset` as only argument. `ScanSet` object. It has to return an array with the same length of a column of `scanset`

out_column : str

column where the results will be stored

Returns
retval : array

the result of user_func

barycenter_times()
Create barytime column with observing times converted to TDB.

calculate_beam_fom(*im*, *cm=None*, *radius=0.3*, *label=None*, *use_log=False*, *show_plot=False*)
Calculate various figures of merit (FOMs) in an image.

These FOMs are useful to single out asymmetries in a beam shape: for example, when characterizing the beam of the radio telescope using a map of a calibrator, it is useful to understand if there are lobes appearing only in one direction.

Parameters
im : 2-d array

The image to be analyzed

Returns
results_dict : dict

Dictionary containing the results

Other Parameters
cm : [int, int]

'Center of mass' of the image

radius : float

The radius around the center of mass, in percentage of the image size (0 <= radius <= 0.5)

use_log: bool

Rescale the image to a log scale before calculating the coefficients. The scale is the same documented in the ds9 docs, for consistency. After normalizing the image from 0 to 1, the log-rescaled image is log(ax + 1) / log a, with `x` the normalized image and `a` a constant fixed here at 1000

show_plot : bool, default False

show the plots immediately

calculate_delta_altaz()
Construction of delta altaz coordinates.

Calculate the delta of altazimutal coordinates wrt the position of the source

calculate_images(*no_offsets=False*, *altaz=False*, *calibration=None*, *elevation=None*, *map_unit='Jy/beam'*, *calibrate_scans=False*, *direction=None*, *onlychans=None*)
Obtain image from all scans.

no_offsets: use positions from feed 0 for all feeds. direction: 0 if horizontal, 1 if vertical

calculate_zernike_moments(*im*, *cm=None*, *radius=0.3*, *norder=8*, *label=None*, *use_log=False*)
Calculate the Zernike moments of the image.

These moments are useful to single out asymmetries in the image: for example, when characterizing the beam of the radio telescope using a map of a calibrator, it is useful to calculate these moments to understand if the beam is radially symmetric or has distorted side lobes.

Parameters

im : 2-d array

The image to be analyzed

cm : [int, int]

'Center of mass' of the image

radius : float

The radius around the center of mass, in percentage of the image size (0 <= radius <= 0.5)

norder : int

Maximum order of the moments to calculate

Returns

moments_dict : dict

Dictionary containing the order, the sub-index and the moment, e.g. {0: {0: 0.3}, 1: {0: 1e-16}, 2: {0: 0.95, 2: 6e-19}, ... } Moments are symmetrical, so only the unique values are reported.

calibrate_images(*calibration*, *elevation=0.7853981633974483*, *map_unit='Jy/beam'*, *direction=None*)
Calibrate the images.

convert_coordinates(*altaz=False*)
Convert the coordinates from sky to pixel.

create_wcs(*altaz=False*)
Create a wcs object from the pointing information.

destripe_images(*niter=10*, *npix_tol=None*, ***kwargs*)

find_scans_through_pixel(*x*, *y*, *test=False*)
Find scans passing through a pixel.

fit_full_images(*chans=None*, *fname=None*, *save_sdev=False*, *no_offsets=False*, *altaz=False*, *calibration=None*, *excluded=None*, *par=None*, *map_unit='Jy/beam'*)
Flatten the baseline with a global fit.

Fit a linear trend to each scan to minimize the scatter in an image

get_coordinates(*altaz=False*)
Give the coordinates as pairs of RA, DEC.

get_obstimes()
Get `astropy.Time` object for time at the telescope location.

get_opacity(*datadir=None*, *dirlist=None*)
List all scans contained in the directory listed in config.

interactive_display(*ch=None*, *recreate=False*, *test=False*)
Modify original scans from the image display.

list_scans(*datadir=None*, *dirlist=None*)
List all scans contained in the directory listed in config.

load_scans(*scan_list*, *freqsplat=None*, *nofilt=False*, ***kwargs*)
Load the scans in the list one by ones.

read_images_from_meta()

reprocess_scans_through_pixel(*x*, *y*, *test=False*)
Given a pixel in the image, find all scans passing through it.

rerun_scan_analysis(*x*, *y*, *key*, *test=False*)
Rerun the analysis of single scans.

save_ds9_images(*fname=None*, *save_sdev=False*, *scrunch=False*, *no_offsets=False*, *altaz=False*, *calibration=None*, *map_unit='Jy/beam'*, *calibrate_scans=False*, *destripe=False*, *npix_tol=None*, *bad_chans=[]*)
Save a ds9-compatible file with one image per extension.

scrunch_images(*bad_chans=[]*)
Sum the images from all channels.

update_meta_with_images()

update_scan(*sname*, *sid*, *dim*, *zap_info*, *fit_info*, *flag_info*, *test=False*)
Update a scan in the scanset after filtering.

write(*fname*, ***kwargs*)
Same as Table.write, but adds path information for HDF5.

Moreover, saves the scan list to a txt file, that will be read when data are reloaded. This is a *temporary solution*

6.7 srttools.interactive_filter module

Interactive operations.

exception srttools.interactive_filter.**TestWarning**
Bases: `UserWarning`

exception srttools.interactive_filter.**PlotWarning**
Bases: `UserWarning`

srttools.interactive_filter.**mask**(*xs*, *border_xs*, *invert=False*)
Create mask from a list of interval borders.

Parameters
xs : array

the array of values to filter

border_xs : array

the list of borders. Should be an even number of positions

Returns
mask : array

Array of boolean values, that work as a mask to xs

Other Parameters
invert : bool

Mask value is False if invert = False, and vice versa. E.g. for zapped intervals, invert = False. For baseline fit selections, invert = True

class srttools.interactive_filter.**intervals**
Bases: object

A list of xs and ys of the points taken during interactive selection.

Initialize.

add(*point*)
Append points.

clear()
Clear.

class srttools.interactive_filter.**DataSelector**(*xs*, *ys*, *ax1*, *ax2*, *masks=None*, *xlabel=None*, *title=None*, *test=False*)
Bases: object

Plot and process scans interactively.

Initialize.

align_all()
Given the selected scan, aligns all the others to that.

base(*event*)
Add an interval to the ones that will be used by baseline sub.

flag(*value=True*)

on_click(*event*)
Dummy function, in case I want to do something with a click.

on_key(*event*)
Do something when the keyboard is used.

on_pick(*event*)
Do this when I pick a line in the plot.

plot_all(*silent=False*)
Plot everything.

print_info()
Print info on the current scan.

Info includes zapped intervals and fit parameters.

print_instructions()
Print to terminal some instructions for the interactive window.

quit()

reset()

subtract_baseline()
Subtract the baseline based on the selected intervals.

subtract_model(*channel*)
Subtract the model from the scan.

zap(*event*)
Create a zap interval.

class srttools.interactive_filter.**ImageSelector**(*data*, *ax*, *fun=None*, *test=False*)
Bases: object

Return xs and ys of the image, and the key that was pressed.

Attributes

img	(array) the image
ax	(pyplot.axis instance) the axis where the image will be plotted
fun	(function) the function to call when a key is pressed. It must accept three arguments: x, y and key

Initialize an ImageSelector class.

Parameters
data : array

the image

ax : pyplot.axis instance

the axis where the image will be plotted

fun : function, optional

the function to call when a key is pressed. It must accept three arguments: x, y and key

on_key(*event*)
Do this when the keyboard is pressed.

plot_img()
Plot the image on the interactive display.

srttools.interactive_filter.**create_empty_info**(*keys*)

srttools.interactive_filter.**select_data**(*xs*, *ys*, *masks=None*, *title=None*, *xlabel=None*, *test=False*)
Open a DataSelector window.

6.8 srttools.io module

Input/output functions.

srttools.io.**mkdir_p**(*path*)
Safe mkdir function.

Parameters
path : str

Name of the directory/ies to create

Notes

Found at http://stackoverflow.com/questions/600268/mkdir-p-functionality-in-python

srttools.io.**detect_data_kind**(*fname*)
Placeholder for function that recognizes data format.

srttools.io.**correct_offsets**(*obs_angle*, *xoffset*, *yoffset*)
Correct feed offsets for derotation angle.

All angles are in radians.

Examples

```
>>> x = 2 ** 0.5
>>> y = 2 ** 0.5
>>> angle = np.pi / 4
>>> xoff, yoff = correct_offsets(angle, x, y)
>>> np.allclose([xoff, yoff], 2 ** 0.5)
True
```

srttools.io.**observing_angle**(*rest_angle*, *derot_angle*)
Calculate the observing angle of the multifeed.

If values have no units, they are assumed in radians

Parameters
rest_angle : float or Astropy quantity, angle

rest angle of the feeds

derot_angle : float or Astropy quantity, angle

derotator angle

Examples

```
>>> observing_angle(0 * u.rad, 2 * np.pi * u.rad).to(u.rad).value
0.0
>>> observing_angle(0, 2 * np.pi).to(u.rad).value
0.0
```

srttools.io.**get_rest_angle**(*xoffsets*, *yoffsets*)
Calculate the rest angle for multifeed.

The first feed is assumed to be at position 0, for it the return value is 0

Examples

```
>>> xoffsets = [0.0, -0.0382222, -0.0191226, 0.0191226, 0.0382222,
...             0.0191226, -0.0191226]
>>> yoffsets = [0.0, 0.0, 0.0331014, 0.0331014, 0.0, -0.0331014,
...             -0.0331014]
>>> np.allclose(get_rest_angle(xoffsets, yoffsets).to(u.deg).value,
...             np.array([0., 180., 120., 60., 360., 300., 240.]))
True
```

srttools.io.**print_obs_info_fitszilla**(*fname*)
Placeholder for function that prints out oberving information.

srttools.io.**read_data_fitszilla**(*fname*)

srttools.io.**read_data**(*fname*)
Read the data, whatever the format, and return them.

srttools.io.**root_name**(*fname*)
Return the file name without extension.

srttools.io.**get_chan_columns**(*table*)

6.9 srttools.opacity module

srttools.opacity.**calculate_opacity**(*file*, *plot=True*, *tatm=None*, *tau0=None*, *t0=None*)
Calculate opacity from a skydip scan.

Atmosphere temperature is fixed, from Buffa et al.'s calculations.

Parameters
file : str

File name of the skydip scan in Fits format

plot : bool

Plot diagnostics about the fit

Returns
opacities : dict

Dictionary containing the opacities calculated for each channel, plus the time in the middle of the observation.

Other Parameters
tatm : float

Atmospheric temperature (fixed in the fit). The default value is calculated from an empyrical formula

tau0 : float

Initial opacity in the fit. The default value is np.log(2 / (1 + np.sqrt(1 - 4 * (t30 - t90) / tatm))), where t30 and t90 are the Tsys values calculated at 30 and 90 degrees of elevation respectively.

t0 : float

Initial value for Tsys in the fit.

srttools.opacity.**exptau**(*airmass*, *tatm*, *tau*, *t0*)
Function to fit to the T vs airmass data.

srttools.opacity.**main_opacity**(*args=None*)

6.10 srttools.read_config module

Read the configuration file.

srttools.read_config.**sample_config_file**(*fname='sample_config_file.ini'*)
Create a sample config file, to be modified by hand.

srttools.read_config.**get_config_file**()
Get the current config file.

srttools.read_config.**read_config**(*fname=None*)
Read a config file and return a dictionary of all entries.

6.11 srttools.scan module

Scan class.

class srttools.scan.**Scan**(*data=None*, *config_file=None*, *norefilt=True*, *interactive=False*, *nosave=False*, *debug=False*, *freqsplat=None*, *nofilt=False*, *nosub=False*, *avoid_regions=None*, *save_spectrum=False*, ***kwargs*)
Bases: astropy.table.Table

Class containing a single scan.

Load a Scan object

Parameters
data : str or None

data can be one of the following: None, in which case an empty Scan object is created; a FITS or HDF5 archive, containing an on-the-fly or cross scan in one of the accepted formats; another Scan or astropy.Table object

config_file : str

Config file containing the parameters for the images and the directories containing the image and calibration data

norefilt : bool

If an HDF5 archive is present with the same basename as the input FITS file, do not re-run the filtering (default True)

freqsplat : str

See srttools.scan.interpret_frequency_range

nofilt : bool

See srttools.scan.clean_scan_using_variability

nosub : bool

Do not run the baseline subtraction.

Other Parameters

kwargs : additional arguments

These will be passed to `astropy.Table` initializer

baseline_subtract(*kind='als'*, *plot=False*, *avoid_regions=None*, ***kwargs*)

Subtract the baseline.

Parameters

kind : str

If 'als', use the Asymmetric Least Square fitting in `srttools.fit.baseline_als()`, using a very stiff baseline (lam=1e11). If 'rough', use `srttools.fit.baseline_rough()` instead.

Other Parameters

plot : bool

Plot diagnostic information in an image with the same basename as the fits file, an additional label corresponding to the channel, in PNG format.

avoid_regions: [[r0_ra, r0_dec, r0_radius], [r1_ra, r1_dec, r1_radius]]

Avoid these regions from the fit

chan_columns()

List columns containing samples.

check_order()

Check that times in a scan are monotonically increasing.

clean_and_splat(*good_mask=None*, *freqsplat=None*, *noise_threshold=5*, *debug=True*, *save_spectrum=False*, *nofilt=False*)

Clean from RFI.

Very rough now, it will become complicated eventually.

Parameters

good_mask : boolean array

this mask specifies intervals that should never be discarded as RFI, for example because they contain spectral lines

freqsplat : str

List of frequencies to be merged into one. See `srttools.scan.interpret_frequency_range()`

noise_threshold : float

The threshold, in sigmas, over which a given channel is considered noisy

Returns

masks : dictionary of boolean arrays

this dictionary contains, for each detector/polarization, True values for good spectral channels, and False for bad channels.

Other Parameters

save_spectrum : bool, default False

Save the spectrum into a 'ChX_spec' column

debug : bool, default True

Save images with quicklook information on single scans

nofilt : bool

Do not filter noisy channels (see `clean_scan_using_variability()`)

get_info_string(*ch*)

interactive_filter(*save=True*, *test=False*)
Run the interactive filter.

save(*fname=None*)
Call self.write with a default filename, or specify it.

write(*fname*, **args*, ***kwargs*)
Same as Table.write, but adds path information for HDF5.

srttools.scan.**interpret_frequency_range**(*freqsplat*, *bandwidth*, *nbin*)
Interpret the frequency range specified in freqsplat.

Parameters

freqsplat : str

Frequency specification. If None, it defaults to the interval 10%-90% of the bandwidth. If ':', it considers the full bandwidth. If 'f0:f1', where f0 and f1 are floats/ints, f0 and f1 are interpreted as start and end frequency in MHz, *referred to the local oscillator* (LO; e.g., if '100:400', at 6.9 GHz this will mean the interval 7.0-7.3 GHz)

bandwidth : float

The bandwidth in MHz

nbin : int

The number of bins in the spectrum

Returns

freqmin : float

The minimum frequency in the band (ref. to LO), in MHz

freqmax : float

The maximum frequency in the band (ref. to LO), in MHz

binmin : int

The minimum spectral bin

binmax : int

The maximum spectral bin

Examples

```
>>> interpret_frequency_range(None, 1024, 512)
(102.4, 921.6, 51, 459)
>>> interpret_frequency_range('default', 1024, 512)
(102.4, 921.6, 51, 459)
>>> interpret_frequency_range(':', 1024, 512)
(0, 1024, 0, 511)
>>> interpret_frequency_range('all', 1024, 512)
```

(continues on next page)

(continued from previous page)

```
(0, 1024, 0, 511)
>>> interpret_frequency_range('200:800', 1024, 512)
(200.0, 800.0, 100, 399)
```

srttools.scan.**clean_scan_using_variability**(*dynamical_spectrum*, *length*, *bandwidth*, *good_mask=None*, *freqsplat=None*, *noise_threshold=5.0*, *debug=True*, *nofilt=False*, *outfile='out'*, *label=''*, *smoothing_window=0.05*, *debug_file_format='pdf'*, *info_string='Empty info string'*)

Clean a spectroscopic scan using the difference of channel variability.

From the dynamical spectrum, i.e. the list of spectra obtained in each sample of a scan, we calculate the rms variability of each frequency channel. This forms a sort of rms spectrum. We calculate the baseline of this spectrum, and all channels whose rms is above above noise_threshold times the reference median absolute deviation (`srttools.fit.ref_mad()`), calculated with a minimum window of 20 samples, are cut and assigned an interpolated value between the closest valid points. The baseline is calculated with `srttools.fit.baseline_als()`, using a lambda value depending on the number of channels, with a formula that has been shown to work in a few standard cases but might be modified in the future.

Parameters

dynamical_spectrum : 2-d array

Array of shape MxN, with M spectra of N elements each.

length : float

Duration in seconds of the scan (assumed to have constant sample time)

bandwidth : float

Bandwidth in MHz

Returns

results : object

The attributes of this object are:

lc
[array-like] The cleaned light curve

freqmin
[float] Minimum frequency in MHz, referred to local oscillator

freqmax
[float] Maximum frequency in MHz, referred to local oscillator

Other Parameters

good_mask : boolean array

this mask specifies channels that should never be discarded as RFI, for example because they contain spectral lines

freqsplat : str

List of frequencies to be merged into one. See `srttools.scan.interpret_frequency_range()`

noise_threshold : float

The threshold, in sigmas, over which a given channel is considered noisy

debug : bool

Print out debugging information

nofilt : bool

Do not filter noisy channels (set noise_threshold to 1e32)

outfile : str

Root file name for the diagnostics plots (outfile_label.png)

label : str

Label to append to the filename (outfile_label.png)

smoothing_window : float

Width of smoothing window, in fraction of spectral length

See also:

srttools.fit.baseline_als, srttools.fit.ref_mad

srttools.scan.**list_scans**(*datadir*, *dirlist*)

List all scans contained in the directory listed in config.

6.12 srttools.simulate module

Functions to simulate scans and maps.

srttools.simulate.**simulate_scan**(*dt=0.04*, *length=120.0*, *speed=4.0*, *shape=None*, *noise_amplitude=1.0*, *center=0.0*, *baseline='flat'*, *nbin=1*, *calon=False*, *nsamples=None*)

Simulate a scan.

Parameters

dt : float

The integration time in seconds

length : float

Length of the scan in arcminutes

speed : float

Speed of the scan in arcminutes / second

shape : function

Function that describes the shape of the scan. If None, a constant scan is assumed. The zero point of the scan is in the *center* of it

noise_amplitude : float

Noise level in counts

center : float

Center coordinate in degrees

baseline : str, number or tuple

"flat", "slope" (linearly increasing/decreasing), "messy" (random walk), a number (which gives an amplitude to the random-walk baseline, that is 20 for "messy"), or

a tuple (m, q, messy_amp) giving the maximum and minimum absolute-value slope and intercept, and the random-walk amplitude.

srttools.simulate.**save_scan**(*times*, *ra*, *dec*, *channels*, *filename='out.fits'*, *other_columns=None*, *other_keywords=None*, *scan_type=None*, *src_ra=None*, *src_dec=None*, *srcname='Dummy'*, *counts_to_K=0.03*)

Save a simulated scan in fitszilla format.

Parameters

times : iterable

times corresponding to each bin center, in seconds

ra : iterable

RA corresponding to each bin center

dec : iterable

Dec corresponding to each bin center

channels : {'Ch0': array([...]), 'Ch1': array([...]), ...}

Dictionary containing the count array. Keys represent the name of the channel

filename : str

Output file name

srcname : str

Name of the source

counts_to_K : float, array or dict

Conversion factor between counts and K. If array, it has to be the same length as channels.keys()

srttools.simulate.**simulate_map**(*dt=0.04*, *length_ra=120.0*, *length_dec=120.0*, *speed=4.0*, *spacing=0.5*, *count_map=None*, *noise_amplitude=1.0*, *width_ra=None*, *width_dec=None*, *outdir='sim/'*, *baseline='flat'*, *mean_ra=180*, *mean_dec=70*, *srcname='Dummy'*, *channel_ratio=1*, *nbin=1*)

Simulate a map.

Parameters

dt : float

The integration time in seconds

length : float

Length of the scan in arcminutes

speed : float

Speed of the scan in arcminutes / second

shape : function

Function that describes the shape of the scan. If None, a constant scan is assumed. The zero point of the scan is in the *center* of it

noise_amplitude : float

Noise level in counts

spacing : float

Spacing between scans, in arcminutes

baseline : str

"flat", "slope" (linearly increasing/decreasing), "messy" (random walk) or a number (which gives an amplitude to the random-walk baseline, that is 20 for "messy").

count_map : function

Flux distribution function, centered on zero

outdir : str or iterable (str, str)

If a single string, put all files in that directory; if two strings, put RA and DEC scans in the two directories.

channel_ratio : float

Ratio between the counts in the two channels

Part VI

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

G

H

I

J

L

M

O

P

Q

R

S

T

U

W

Z