INTERNAL REPORT

SRT Single-Dish Tools documentation

Matteo Bachetti,
Alberto Pellizzoni, Elise Egron, Simona Righini, Noemi Iacolina, Alessio Trois

Report n.58, released: 04/08/2016

Reviewer: Maura Pilia

Osservatorio Astronomico di Cagliari

Table of contents

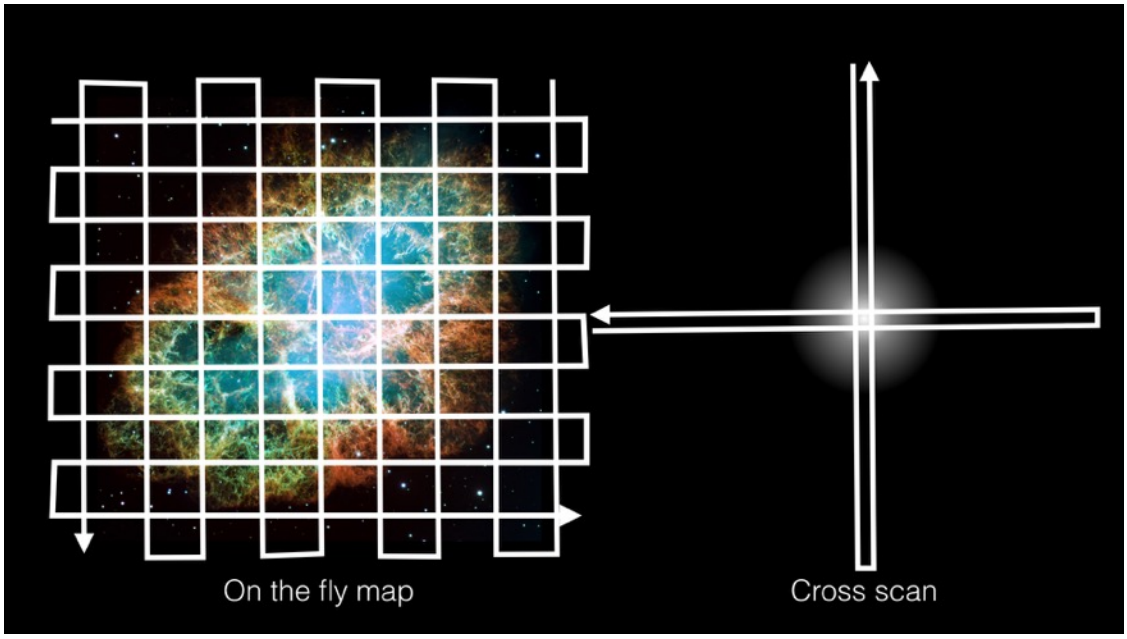

Figure 1. On-the-fly maps vs cross scan strategies for single dish observations. The first is able to produce images, the second is used to obtain quick flux measurements of point-like sources.

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 (private, for the time being) Github repository. The Continuous Integration service Travis CI is used for that. It will be released to the public as an Open Source library once a number of items in the roadmap are accomplished (see below), but it is already available upon request. The current version is 0.2-dev, indicating a pre-release of the 0.2 version.

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.

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: Full support of general coordinate systems, including Galactic
- v.0.5: Improved RFI support, using simple techniques of machine learning
- v.1.0: Compatibility with ALMA file format; code release.

Installation

Anaconda and virtual environment (recommended but optional)

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 py35 python=3.5
```

load the new environment:

```
$ source activate py35
```

and install the dependencies:

```
(py35) $ conda install matplotlib h5py astropy scipy numpy
(py35) $ pip install statsmodels # Optional
```

Cloning and installation

Clone the repository[1] (you will need to be a registered user of the code):

```
(py35) $ cd /my/software/directory/
(py35) $ git clone https://username@bitbucket.org/mbachett/srt-single-
dish-tools.git
```

or if you have deployed your SSH key to Bitbucket:

```
(py35) $ git clone git@bitbucket.org:mbachett/srt-single-dish-tools.git
```

Then:

```
(py35) $ cd srt-single-dish-tools
(py35) $ python setup.py install
(py35) $ python setup.py test
```

That's it. After installation has ended and tests have passed, you can verify that the software is installed correctly by executing:

```
(py35) $ SDTlcurve -h
```

If the help message appears, you're done!

Updating

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

```
(py35) $ git pull
(py35) $ python setup.py install
```

[1] A mirror on GitHub is also available, and will eventually become the main repository. For the time being, it is used only to trigger the automatic tests on Travis CI.

Tutorial

In this tutorial, we will see how to obtain calibrated images 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.

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:

```
(py35) $ 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.:

```
(py35) $ SDTinspect */
Group 0, Backend = ROACH2, Receiver = CCB
---------------
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/
(.....)
```

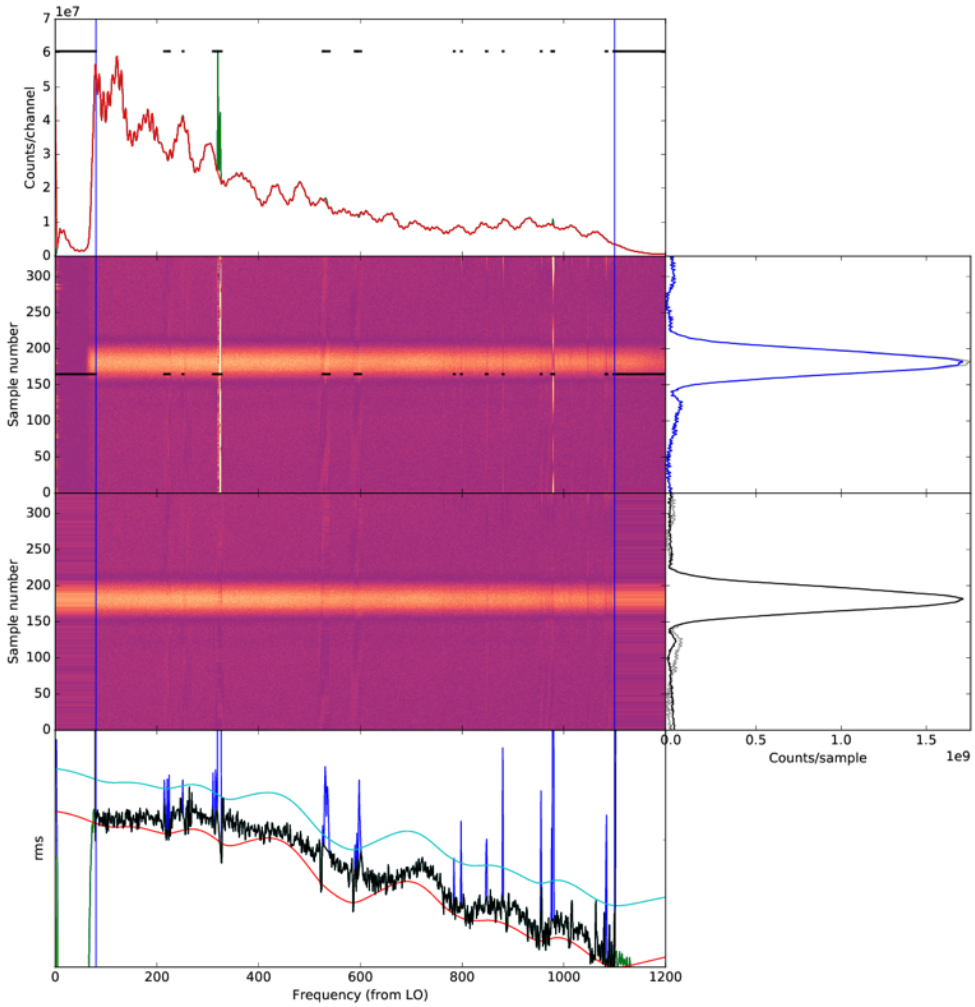

Figure 2. Output of the automatic filtering procedure for an OTF scan of a calibrator. Channels with an rms that is a multiple of the mean standard deviation of the scan (factor encoded in the `noise_threshold` key in the config file) are automatically filtered out. See bottom panel, where the cyan curved line is the running threshold to select noisy channels, and black lines in the middle-top and top panels. Optionally the user can choose the frequency interval (blue vertical lines). The rms of 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.

```
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:

```
(py35) $ SDTinspect -d */
Group 0, Backend = ROACH2, Receiver = CCB
(.....)
(py35) $ ls -alrt
```

```
CCB_ROACH_Src1_Obs0.ini
KKG_ROACH_Src1_Obs1.ini
```

Modify config files

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

```
(py35) $ SDTlcurve --sample-config
(py35) $ 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):

```
(py35) $ cat CCB_ROACH_Src1_Obs0.ini
[local]
workdir = .
datadir = .

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

`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.
`goodchans` is a list of channels that can be excluded from automatic filtering (for example, because they might contain an important spectral line.)

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

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 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 respectively. This process produces plots like Figure 2.

```
(py35) $ SDTpreprocess -c CCB_TP_Src1_Obs0.ini –splat 100:600
```

2. 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 makes a more precise fit. This procedure is very fast and aligns the vast majority of scans in a fraction of seconds. For more complicated scans, an interactive interface is also available (through the `--interactive` option).
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, he/she needs to delete these files first.

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:

```
(py35) $ SDTimage -c CCB_TP_Src1_Obs0.ini
```

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*:

```
(py35) $ SDTimage -c CCB_ROACH_Src1_Obs0.ini
```

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):

```
(py35) $ SDTimage -c CCB_ROACH_Src1_Obs0.ini --splat <freqmin>:<freqmax>
```

The above command will:

- Run through all the scans in the directories specified in the config file

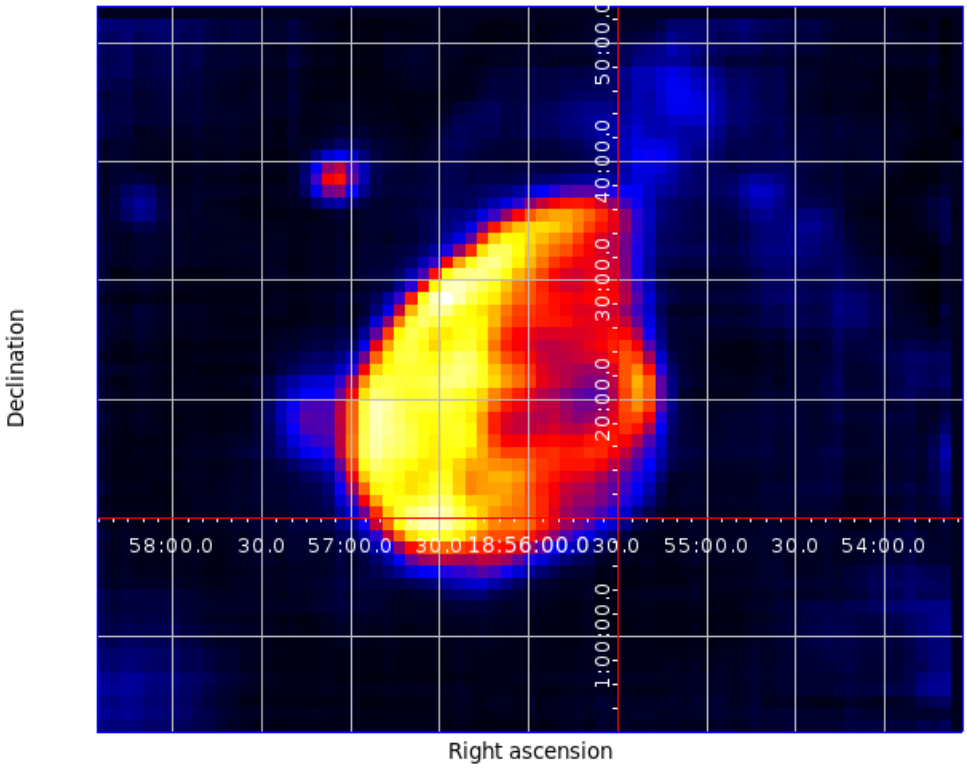

Figure 3. Map produced by `SDTimage`

- Clean them up if not already done in a previous step, in the same way of `SDTpreprocess`
- 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>.

The automatic RFI removal procedure might have missed some problematic scans. 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, 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 (option `-g`). It's called *global baseline subtraction*. This procedure makes a *global* fit 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 through the option `-e` followed by 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. A full usage of DS9-like regions is envisaged).

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

```
(py35) $ SDTimage -g -e 30 33 10 (… + usual options)
```

Now, have fun with DS9 (Figure 3)!

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

```
(py35) $ 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** used for the sources:

```
(py35) $ SDTcal -c CCB_ROACH_Src1_Obs0.ini --splat <freqmin>:<freqmax> \
-o calibration.hdf5
```

Finally, call SDTimage with the --calibrate option, as follows:

```
(py35) $ 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.

Command line interface

Here follows a schematic view on the options for the command line scripts.

SDTcal

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

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

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
```

SDTimage

```
usage: SDTimage [-h] [--sample-config] [-c CONFIG] [--refilt] [--sub]
                [--interactive] [--calibrate CALIBRATE] [--nofilt] [-g]
                [-e EXCLUDE [EXCLUDE ...]] [--chans CHANS] [-o OUTFILE]
                [--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)

optional arguments:
  -h, --help            show this help message and exit
```

```
  --sample-config       Produce sample config file
  -c CONFIG, --config CONFIG
                        Config file
  --refilt              Re-run the scan filtering
  --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.
                        The region is indicated by three numbers: X, Y
                        and radius. e.g. -e 30 31 10

  --chans CHANS         Comma-separated channels to include in global
                        fitting
                        (Ch0, Ch1, ...)
  -o OUTFILE, --outfile OUTFILE
                        Save intermediate scanset to this 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.
```

SDTinspect

```
usage: SDTinspect [-h] [-g GROUP_BY [GROUP_BY ...]] [-d]
                  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

optional arguments:
  -h, --help            show this help message and exit
  -g GROUP_BY [GROUP_BY ...], --group-by GROUP_BY [GROUP_BY ...]
  -d, --dump-config-files
```

SDTlcurve

```
usage: SDTlcurve [-h] [--sample-config] [-c CONFIG]
                 [--pickle-file PICKLE_FILE] [--splat SPLAT] [--refilt]
```

```
Load a series of cross scans from a config file and produce a calibrated
light curve.

optional arguments:
  -h, --help            show this help message and exit
  --sample-config       Produce sample config file
  -c CONFIG, --config CONFIG
                        Config file
  --pickle-file PICKLE_FILE
                        Name for the intermediate pickle 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.
  --refilt              Re-run the scan filtering
```

SDTpreprocess

```
usage: SDTpreprocess [-h] [-c CONFIG] [--sub] [--interactive] [--nofilt]
                     [--splat SPLAT]
                     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

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
  --nofilt              Do not filter noisy channels
  --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.
```

API documentation

Here follows the detailed description of the API, for developers.

srttools.core package

Submodules

srttools.core.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.core.calibration.CalibratorTable`**(**args*, ***kwargs*)¶

Bases: `srttools.core.calibration.SourceTable`

Class containing all information and functions about calibrators.

Initialize the object.

`Jy_over_counts`(*channel*, *elevation=None*)¶

`Jy_over_counts_rough`(*channel=None*)¶

Get the conversion from counts to Jy.

Other Parameters:

channel : str

Name of the data channel

`calibrate`()

Calculate the calibration constants.

`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()
Compute the conversion between Jy and counts.

Try to get a meaningful fit over elevation. Revert to the rough function `Jy_over_counts_rough` in case `statsmodels` is not installed.

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

get_fluxes()
Get the tabulated flux of the calibrator.

plot_two_columns(*xcol*, *ycol*, *xerrcol=None*, *yerrcol=None*, *ax=None*, *channel=None*, *xfactor=1*, *yfactor=1*, *color=None*)
Plot the data corresponding to two given columns.

show()
Show a summary of the calibration.

update()
Update the calibration information.

***class* srttools.core.calibration.SourceTable(**args*, ***kwargs*)**
Bases: `astropy.table.Table`

Class containing all information and functions about sources.

Initialize the object.

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

srttools.core.calibration.decide_symbol(*values*)
Decide symbols for plotting.

Assigns different symbols to RA scans, Dec scans, backward and forward.

srttools.core.calibration.flux_function(*start_frequency*, *bandwidth*, *coeffs*, *ecoeffs*)
Flux function from Perley & Butler ApJS 204, 19 (2013).

srttools.core.calibration.get_fluxes(*basedir*, *scandir*, *channel='Ch0'*, *feed=0*, *plotall=False*, *verbose=True*, *freqsplat=None*)
Get fluxes from all scans in path.

srttools.core.calibration.get_full_table(*config_file*, *channel='Ch0'*, *feed=0*, *plotall=False*, *picklefile=None*, *verbose=True*, *freqsplat=None*)
Get all fluxes in the directories specified by the config file.

srttools.core.calibration.main_calibrator(*args=None*)
Main function.

srttools.core.calibration.main_lc_calibrator(*args=None*)
Main function.

srttools.core.calibration.read_calibrator_config()
Read the configuration of calibrators in data/calibrators.

srttools.core.calibration.show_calibration(*full_table*, *feed=0*, *plotall=False*)
Show the results of calibration.

srttools.core.calibration.test_calibration_roach()
Test that the calibration executes completely, ROACH version.

srttools.core.calibration.test_calibration_tp()
Test that the calibration executes completely.

srttools.core.fit module

Useful fitting functions.

srttools.core.fit.align(*xs*, *ys*)
Given the first scan, it aligns all the others to that.

srttools.core.fit.baseline_als(*x*, *y*, *lam=None*, *p=None*, *niter=10*, *return_baseline=False*, *offset_correction=True*, *outlier_purging=True*)
Baseline Correction with Asymmetric Least Squares Smoothing.

srttools.core.fit.baseline_rough(*time*, *lc*, *start_pars=None*, *return_baseline=False*)
Rough function to subtract the baseline.

srttools.core.fit.fit_baseline_plus_bell(*x*, *y*, *ye=None*, *kind='gauss'*)
Fit a function composed of a linear baseline plus a bell function.

kind: ‘gauss’ or ‘lorentz’

srttools.core.fit.linear_fit(*time*, *lc*, *start_pars*, *return_err=False*)
A linear fit with any set of data. Return the parameters.

srttools.core.fit.linear_fun(*x*, *q*, *m*)
A linear function.

srttools.core.fit.minimize_align(*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.

srttools.core.fit.objective_function(*params*, *args*)
Put the parameters in the right order to use with scipy’s minimize.

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

srttools.core.fit.offset_fit(*time*, *lc*, *offset_start=0*, *return_err=False*)
A linear fit with any set of data. Return the parameters.

srttools.core.fit.purge_outliers(*y*)
srttools.core.fit.ref_std(*array*, *window*)
Minimum standard deviation along an array.

srttools.core.global_fit module

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

srttools.core.global_fit.counter()
srttools.core.global_fit.display_intermediate(*scanset*, *chan='Ch0'*, *feed=0*, *excluded=None*, *parfile=None*, *factor=1*)
Get a clean image by subtracting linear trends from the initial scans.

Parameters:

scanset : a :class:`ScanSet` instance

The scanset to be fit

Other Parameters:

chan : str

channel of the scanset to be fit. Defaults to `"Ch0"`

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 _save_iteration

srttools.core.global_fit.fit_full_image(*scanset*, *chan='Ch0'*, *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

Other Parameters:

chan : str

channel of the scanset to be fit. Defaults to `"Ch0"`

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.core.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.core.histograms.histogram2d(*x*, *y*, *bins=10*, *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).

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')
>>> 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.show()
```

srttools.core.histograms.histogramdd(*sample*, *bins=10*, *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).

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)
```

srttools.core.imager module

Produce calibrated light curves.

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

Bases: `astropy.table.Table`

Class containing a set of scans.

Initialize a ScanSet object.

analyze_coordinates(*altaz=False*)

Save statistical information on coordinates.

calculate_images(*scrunch=False*, *no_offsets=False*, *altaz=False*, *calibration=None*)

Obtain image from all scans.

Parameters:

scrunch : bool, default False

Sum all channels

no_offsets : bool, default False

use positions from feed 0 for all feeds

calibrate_images(*calibration*)

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.

find_scans_through_pixel(*x*, *y*)

Find scans passing through a pixel.

fit_full_images(*chans=None*, *fname=None*, *save_sdev=False*, *scrunch=False*, *no_offsets=False*, *altaz=False*, *calibration=None*, *excluded=None*, *par=None*)
Fit a linear trend to each scan to minimize the scatter in the image.

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

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

list_scans(*datadir*, *dirlist*)
List all scans contained in the directory listed in config.

load(*fname*, ***kwargs*)
Set default path and call Table.read.

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

reprocess_scans_through_pixel(*x*, *y*)
Reprocess interactively all scans passing through a pixel.

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

save_ds9_images(*fname=None*, *save_sdev=False*, *scrunch=False*, *no_offsets=False*, *altaz=False*, *calibration=None*)
Save a ds9-compatible file with one image per extension.

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

write(*fname*, ***kwargs*)
Set default path and call Table.write.

srttools.core.imager.main_imager(*args=None*)
Main function.

srttools.core.imager.main_preprocess(*args=None*)
Preprocess the data.

srttools.core.interactive_filter module

Interactive operations.

class **srttools.core.interactive_filter.DataSelector**(*xs*, *ys*, *ax1*, *ax2*, *masks=None*, *xlabel=None*, *title=None*)
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 of data to the ones that will be used by baseline sub.

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()
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.

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.core.interactive_filter.ImageSelector(*data*, *ax*, *fun=None*)**
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.

class **`srttools.core.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.

`srttools.core.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

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

srttools.core.io module

Input/output functions.

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

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

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

srttools.core.io.profile_coords()
Same test above, with profiling.

srttools.core.io.read_data(*fname*)
Read the data, whatever the format, and return them as an Astropy Table.

srttools.core.io.read_data_fitszilla(*fname*)
Open a fitszilla FITS file and read all relevant information.

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

srttools.core.read_config module

Read the configuration file.

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

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

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

srttools.core.scan module

Scan class.

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

Class containing a single scan.

Initialize a Scan object.

Freqsplat is a string, freqmin:freqmax, and gives the limiting frequencies of the interval to splat in a single channel.

baseline_subtract(*kind='als'*)
Subtract the baseline.

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

noise_threshold : float

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

freqsplat : str

Specification of frequency interval to merge into a single channel

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

`interactive_filter`(*save=True*)
Run the interactive filter.

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

`make_single_channel`(*freqsplat*, *masks=None*)
Transform a spectrum into a single-channel count rate.

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

`write`(*fname*, ***kwargs*)
Set default path and call Table.write.

`zap_birdies`()
Zap bad intervals.

`srttools.core.scan.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.core.scan.list_scans`(*datadir*, *dirlist*)
List all scans contained in the directory listed in config.

srttools.core.simulate module

Functions to simulate scans and maps.

srttools.core.simulate.save_scan(*times*, *ra*, *dec*, *channels*, *filename='out.fits'*, *other_columns=None*, *scan_type=None*)
Save a simulated scan in fitszilla format.

Parameters:

times : iterable

times corresponding to each bin center

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

srttools.core.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'*)
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) or “messy” (random walk)

srttools.core.simulate.simulate_scan(*dt=0.04*, *length=120.0*, *speed=4.0*, *shape=None*, *noise_amplitude=1.0*, *center=0.0*)

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