edges-analysis Documentation

EDGES Team

Jul 12, 2023

CONTENTS

Calculate calibration coefficients for EDGES Calibration Observations

CHAPTER

ONE

INSTALLATION

Download/clone the repo and do

```
pip install .
```

in the top-level directory (optionally add an `-e` for develop-mode). Preferably, do this in an isolated python/conda environment.

CHAPTER

TWO

USAGE

2.1 CLI

There is a very basic CLI set up for running a full calibration pipeline over a set of data. To use it, do

```
$ edges-cal run --help
```

Multiple options exist, but the only ones required are `CONFIG` and `PATH`. The first should point to a YAML configuration for the run, and the second should point to a directory in which exists `S11`, `Resistance` and `Spectra` folders. Thus:

```
$ edges-cal run ~/config.yaml .
```

will work if you are in such a directory.

The `config.yaml` consists of a set of parameters passed to `edges_cal.CalibrationObservation`. See its docstring for more details.

In addition, you can run a "term sweep" over a given calibration, iterating over number of Cterms and Wterms until some threshold is met. This uses the same configuration as `edges-cal run`, but you can pass a maximum number of C and W-terms, along with a threshold at which to stop the sweep (this is a threshold in absolute RMS over degrees of freedom). This will write out a `Calibration` file for the "best" set of parameters.

You can also create full Jupyter notebook reports (and convert them to PDF!) using the CLI. To get this to work, you must install `edges-cal` with `pip install edges-cal[report]`. Then you must do the following:

1. Activate the environment you wish to use to generate the reports (usually `conda activate edges`)
2. Run `python -m ipykernel install --user --name edges --display-name "edges"`

Note that in the second command, calling it "edges" is necessary (regardless of the name of your environment!).

Now you can run

```
$ edges-cal report PATH --config ~/config.yaml
```

(obviously there are other parameters – use `edges-cal report --help` for help). The `PATH` should again be a calibration observation directory. The config can be the same file as in `edges-cal run`, and is optional. By default, both a notebook and a PDF will be produced, in the `outputs/` directory of the observation. You can turn off the PDF production with a `-R` flag.

Similarly, you can *compare* two observations as a report notebook with

```
$ edges-cal compare PATH COMPARE --config ~/config.yaml --config-cmp ~/config.yaml
```

This is intended to more easily show up what might be wrong in an observation, when compared to a "golden" observation, for example.

2.2 Using the Library

To import:

```
import edges_cal as ec
```

Most of the functionality is highly object-oriented, and objects exist for each kind of data/measurement. However, there is a container object for all of these, which manages them. Thus you will typically use

```
>>> calobs = ec.CalibrationObservation(path="path/to/top/level")
```

Several other options exist, and they have documentation that you can access interactively by using

```
>>> help(ec.CalibrationObservation)
```

The most relevant attributes are the (lazily-evaluated) calibration coefficient models:

```
>>> plt.plot(calobs.freq.freq, calobs.C1())
```

the various plotting routines, eg.

```
>>> calobs.plot_coefficients()
```

and the calibrate/decalibrate methods:

```
>>> calibrated_temp = calobs.calibrate("ambient")
```

Note that this final method can be applied to any `LoadSpectrum` – i.e. you can pass in field observations, or an antenna simulator.

CHAPTER

THREE

CONTENTS

3.1 License

3.2 Contributors

- Nivedita Mahesh @nmahesh1412
- David Lewis @dmlewis9
- Steven Murray @steven-murray

3.3 Changelog

3.3.1 v3.2.0

Added

- New `CompositeModel` class that combines modular linear models and behaves like a simple `Model`.

Changed

- Fully reworked `NoiseWaves` class to use the `CompositeModel` class.

3.3.2 v3.1.1

Changed

- New `model_filter` function that `xrfi_model` calls and can be called by other more general functions.

Added

- New tests of `ModelFilterInfo` and `ModelFilterInfoContainer`

3.3.3 v3.1.0

Added

- New semi-rigid S-parameters file from 2017.

3.3.4 v3.0.0

Fixed

- Tests were slow because they were using the wrong default for getting the standard deviation in `xrfi_model`. Now uses the fast kind.

Changed

- `modelling` module got an overhaul. It now consists of smaller, self-consistent classes that are based on `attrs` and are read-only (but clonable). There is an explicit split between a `Model` and a `FixedLinearModel` which is defined at some coordinates `x`, allowing the latter to provide speedups for fitting lots of data at the same coordinates. They are also YAML-read/writable.

3.3.5 v2.1.0

Added

- `FourierDay` linear model

Fixed

- models were being mutated in `ModelFit`. No more.

3.3.6 v2.0.0

Added

- Ability to pass through LNA S11 options
- Ability to specify `t_load` and `t_load_ns` in spectra
- Easier conversion to `Calibration` object from `CalibrationObservation`
- New `NoiseWaves` linear model
- Modular `InternalSwitch` class

Fixed

- Ensure fitted models are always copied

3.3.7 v1.0.0

Added

- Visualization of `xrfi_model` output.
- Allow `init_flags` as input to `xrfi_model`

Changed

- Performance boost for `xrfi_model_sweep`
- Faster modelling in detail
- Compatibility with `edges-io` v1.
- More consistency in API between different XRFI models.
- Allow not passing `spectrum` to `xrfi_explicit` ### Fixed
- Improvements to typing.
- Improvements to `xrfi` tests.

3.3.8 v0.7.0

Added

- Automatic notebook reports for calibration
- `xrfi_model_sweep` that actually works and is tested.
- Faster model evaluations

Fixed

- xRFI doesn't assume that input spectrum is all positive (could be residuals, and therefore have negatives).

3.3.9 Version 0.4.0

Changed

- Much faster modeling via re-use of basis function evaluations

3.3.10 Version 0.3.0

Changed

- load_name now always an alias (hot_load, ambient, short, open)
- Load.temp_ave now always the correct one (even for hot load)

3.3.11 Version 0.2.1

Added

- Basic tests
- Travis/tox/codecov setup

3.3.12 Version 0.2.0

Added

- Many many many new features, and complete modularisation of code
- Now based on `edges-io` package to do the hard work.
- Refined most modules to remove redundant code
- Added better package structure

3.3.13 Version 0.1.0

- First working version on github.

3.4 Tutorials

A good place to get a feel for how `edges-cal` works is by following some tutorials:

If you've covered the tutorials and still have questions about "how to do stuff" in `edges-cal`, consult the FAQs:

3.4.1 Miscellaneous FAQs

Can I eat soup for breakfast?

You should not.

3.5 API Reference

3.5.1 High Level Interface

edges_cal.cal_coefficients	The main user-facing module of `edges-cal`.

edges_cal.cal_coefficients

The main user-facing module of `edges-cal`.

This module contains wrappers around lower-level functions in other modules, providing a one-stop interface for everything related to calibration.

Functions

perform_term_sweep(calobs[, ...])	For a given calibration definition, perform a sweep over number of terms.

edges_cal.cal_coefficients.perform_term_sweep

edges_cal.cal_coefficients.**perform_term_sweep**(*calobs:* CalibrationObservation, *delta_rms_thresh:* *float* = *0*, *max_cterms:* *int* = *15*, *max_wterms:* *int* = *15*) → *CalibrationObservation*

For a given calibration definition, perform a sweep over number of terms.

Parameters

- **calobs** (*CalibrationObservation* instance) – The definition calibration class. The *cterms* and *wterms* in this instance should define the *lowest* values of the parameters to sweep over.

- **delta_rms_thresh** (*float*) – The threshold in change in RMS between one set of parameters and the next that will define where to cut off. If zero, will run all sets of parameters up to the maximum terms specified.
- **max_cterms** (*int*) – The maximum number of cterms to trial.
- **max_wterms** (*int*) – The maximum number of wterms to trial.

Classes

`CalibrationObservation`(loads, receiver, *[, ...])	A composite object representing a full Calibration Observation.
`Calibrator`(*, freq, cterms, wterms, C1, C2, ...)	
`HotLoadCorrection`(*, freq, raw_s11, ...[, ...])	Corrections for the hot load.
`Load`(*, spectrum, reflections[, loss_model, ...])	Wrapper class containing all relevant information for a given load.

edges_cal.cal_coefficients.CalibrationObservation

class edges_cal.cal_coefficients.**CalibrationObservation**(*loads: dict[str, edges_cal.cal_coefficients.Load]*, *receiver: Receiver*, *, *cterms: int = 5*, *wterms: int = 7*, *metadata: dict[str, Any] = _Nothing.NOTHING*)

A composite object representing a full Calibration Observation.

This includes spectra of all calibrators, and methods to find the calibration parameters. It strictly follows Monsalve et al. (2017) in its formalism. While by default the class uses the calibrator sources (“ambient”, “hot_load”, “open”, “short”), it can be modified to take other sources by setting `CalibrationObservation._sources` to a new tuple of strings.

Parameters

- **loads** (*dict[str, edges_cal.cal_coefficients.Load]*) – dictionary of load names to Loads
- **receiver** (*edges_cal.s11.Receiver*) – The object defining the reflection coefficient of the receiver.
- **cterms** (*int*) – The number of polynomial terms used for the scaling/offset functions
- **wterms** (*int*) – The number of polynomial terms used for the noise-wave parameters.
- **metadata** – Metadata associated with the data.

Methods

C1([f])	Scaling calibration parameter.
C2([f])	Offset calibration parameter.
Tcos([f])	Cosine noise-wave parameter.
Tsin([f])	Sine noise-wave parameter.
Tunc([f])	Uncorrelated noise-wave parameter.
__init__(loads, receiver, *[, cterms, ...])	Method generated by attrs for class CalibrationObservation.
calibrate(load[, q, temp])	Calibrate the temperature of a given load.
clone(**kwargs)	Clone the instance, updating some parameters.
decalibrate(temp, load[, freq])	Decalibrate a temperature spectrum, yielding uncalibrated T*.
from_io(io_obj, *[, semi_rigid_path, ...])	Create the object from an edges-io observation.
from_yaml(config[, obs_path])	Create the calibration observation from a YAML configuration.
get_K([freq])	Get the source-S11-dependent factors of Monsalve (2017) Eq.
get_linear_coefficients(load)	Calibration coefficients a,b such that T = aT* + b (derived from Eq.
get_load_residuals()	Get residuals of the calibrated temperature for a each load.
get_rms([smooth])	Return a dict of RMS values for each source.
inject([lna_s11, source_s11s, c1, c2, ...])	Make a new *CalibrationObservation* based on this, with injections.
new_load(load_name, io_obj[, ...])	Create a new load with the given load name.
plot_calibrated_temp(load[, bins, fig, ax, ...])	Make a plot of calibrated temperature for a given source.
plot_calibrated_temps([bins, fig, ax])	Plot all calibrated temperatures in a single figure.
plot_coefficients([fig, ax])	Make a plot of the calibration models, C1, C2, Tunc, Tcos and Tsin.
plot_raw_spectra([fig, ax])	Plot raw uncalibrated spectra for all calibrator sources.
plot_s11_models(**kwargs)	Plot residuals of S11 models for all sources.
to_calibrator()	Directly create a *Calibrator* object without writing to file.
with_load_calkit(calkit[, loads])	Return a new observation with loads having given calkit.
write(filename)	Write all information required to calibrate a new spectrum to file.

edges_cal.cal_coefficients.CalibrationObservation.C1

`CalibrationObservation.C1`(*f: Quantity | None = None*)

Scaling calibration parameter.

Parameters

f (*array-like*) – The frequencies at which to evaluate C1. By default, the frequencies of this instance.

edges_cal.cal_coefficients.CalibrationObservation.C2

CalibrationObservation.**C2**(*f: Quantity | None = None*)

Offset calibration parameter.

Parameters

f (*array-like*) – The frequencies at which to evaluate C2. By default, the frequencies of this instance.

edges_cal.cal_coefficients.CalibrationObservation.Tcos

CalibrationObservation.**Tcos**(*f: Quantity | None = None*)

Cosine noise-wave parameter.

Parameters

f (*array-like*) – The frequencies at which to evaluate Tcos. By default, the frequencies of this instance.

edges_cal.cal_coefficients.CalibrationObservation.Tsin

CalibrationObservation.**Tsin**(*f: Quantity | None = None*)

Sine noise-wave parameter.

Parameters

f (*array-like*) – The frequencies at which to evaluate Tsin. By default, the frequencies of this instance.

edges_cal.cal_coefficients.CalibrationObservation.Tunc

CalibrationObservation.**Tunc**(*f: Quantity | None = None*)

Uncorrelated noise-wave parameter.

Parameters

f (*array-like*) – The frequencies at which to evaluate Tunc. By default, the frequencies of thisinstance.

edges_cal.cal_coefficients.CalibrationObservation.__init__

CalibrationObservation.**__init__**(*loads: dict[str, edges_cal.cal_coefficients.Load], receiver: Receiver, *, cterms: int = 5, wterms: int = 7, metadata: dict[str, Any] = _Nothing.NOTHING*) → None

Method generated by attrs for class CalibrationObservation.

edges_cal.cal_coefficients.CalibrationObservation.calibrate

CalibrationObservation.**calibrate**(*load:* Load | *str*, *q=None*, *temp=None*)

Calibrate the temperature of a given load.

Parameters
load (*Load* or str) – The load to calibrate.

Returns
array (*calibrated antenna temperature in K, len(f).*)

edges_cal.cal_coefficients.CalibrationObservation.clone

CalibrationObservation.**clone**(***kwargs*)

Clone the instance, updating some parameters.

Parameters
kwargs – All parameters to be updated.

edges_cal.cal_coefficients.CalibrationObservation.decalibrate

CalibrationObservation.**decalibrate**(*temp:* ndarray, *load:* Load | *str*, *freq:* ndarray = *None*)

Decalibrate a temperature spectrum, yielding uncalibrated T*.

Parameters

- **temp** (*array_like*) – A temperature spectrum, with the same length as *freq.freq*.
- **load** (str or *Load*) – The load to calibrate.
- **freq** (*array-like*) – The frequencies at which to decalibrate. By default, the frequencies of the instance.

Returns
array_like (*T*, the normalised uncalibrated temperature.*)

edges_cal.cal_coefficients.CalibrationObservation.from_io

classmethod CalibrationObservation.**from_io**(*io_obj: ~edges_io.io.CalibrationObservation*, ***, *semi_rigid_path: str | ~pathlib.Path = ':semi_rigid_s_parameters_WITH_HEADER.txt'*, *freq_bin_size: int = 1*, *spectrum_kwargs: dict[str, dict[str, typing.Any]] | None = None*, *s11_kwargs: dict[str, dict[str, typing.Any]] | None = None*, *internal_switch_kwargs: dict[str, typing.Any] | None = None*, *lna_kwargs: dict[str, typing.Any] | None = None*, *f_low: ~astropy.units.quantity.Quantity = <Quantity 40. MHz>*, *f_high: ~astropy.units.quantity.Quantity = <Quantity inf MHz>*, *sources: tuple[str] = ('ambient', 'hot_load', 'open', 'short')*, *receiver_kwargs: dict[str, typing.Any] | None = None*, *restrict_s11_model_freqs: bool = True*, *hot_load_loss_kwargs: dict[str, typing.Any] | None = None*, ***kwargs*) → *CalibrationObservation*

Create the object from an edges-io observation.

Parameters

- **io_obj** – An calibration observation object from which all the data can be read.
- **semi_rigid_path** (*str or Path, optional*) – Path to a file containing S11 measurements for the semi rigid cable. Used to correct the hot load S11. Found automatically if not given.
- **freq_bin_size** – The size of each frequency bin (of the spectra) in units of the raw size.
- **spectrum_kwargs** – Keyword arguments used to instantiate the calibrator `LoadSpectrum` objects. See its documentation for relevant parameters. Parameters specified here are used for _all_ calibrator sources.
- **s11_kwargs** – Keyword arguments used to instantiate the calibrator `LoadS11` objects. See its documentation for relevant parameters. Parameters specified here are used for _all_ calibrator sources.
- **internal_switch_kwargs** – Keyword arguments used to instantiate the `InternalSwitch` objects. See its documentation for relevant parameters. The same internal switch is used to calibrate the S11 for each input source.
- **f_low** (*float*) – Minimum frequency to keep for all loads (and their S11's). If for some reason different frequency bounds are desired per-load, one can pass in full load objects through `load_spectra`.
- **f_high** (*float*) – Maximum frequency to keep for all loads (and their S11's). If for some reason different frequency bounds are desired per-load, one can pass in full load objects through `load_spectra`.
- **sources** – A sequence of strings specifying which loads to actually use in the calibration. Default is all four standard calibrators.
- **receiver_kwargs** – Keyword arguments used to instantiate the calibrator `Receiver` objects. See its documentation for relevant parameters. `lna_kwargs` is a deprecated alias.
- **restrict_s11_model_freqs** – Whether to restrict the S11 modelling (i.e. smoothing) to the given freq range. The final output will be calibrated only between the given freq range, but the S11 models themselves can be fit over a broader set of frequencies.

edges_cal.cal_coefficients.CalibrationObservation.from_yaml

classmethod CalibrationObservation.**from_yaml**(*config: str | Path | dict*, *obs_path: str | Path | None = None*)

Create the calibration observation from a YAML configuration.

edges_cal.cal_coefficients.CalibrationObservation.get_K

CalibrationObservation.**get_K**(*freq: Quantity | None = None*) → dict[str, tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]]

Get the source-S11-dependent factors of Monsalve (2017) Eq. 7.

edges_cal.cal_coefficients.CalibrationObservation.get_linear_coefficients

CalibrationObservation.**get_linear_coefficients**(*load:* Load | *str*)

Calibration coefficients a,b such that T = aT* + b (derived from Eq. 7).

Parameters

load (str or *Load*) – The load for which to get the linear coefficients.

edges_cal.cal_coefficients.CalibrationObservation.get_load_residuals

CalibrationObservation.**get_load_residuals**()

Get residuals of the calibrated temperature for a each load.

edges_cal.cal_coefficients.CalibrationObservation.get_rms

CalibrationObservation.**get_rms**(*smooth: int = 4*)

Return a dict of RMS values for each source.

Parameters

smooth (*int*) – The number of bins over which to smooth residuals before taking the RMS.

edges_cal.cal_coefficients.CalibrationObservation.inject

CalibrationObservation.**inject**(*lna_s11: ndarray = None*, *source_s11s: dict[str, numpy.ndarray] = None*, *c1: ndarray = None*, *c2: ndarray = None*, *t_unc: ndarray = None*, *t_cos: ndarray = None*, *t_sin: ndarray = None*, *averaged_spectra: dict[str, numpy.ndarray] = None*, *thermistor_temp_ave: dict[str, numpy.ndarray] = None*) → *CalibrationObservation*

Make a new *CalibrationObservation* based on this, with injections.

Parameters

- **lna_s11** – The LNA S11 as a function of frequency to inject.
- **source_s11s** – Dictionary of `{source: S11}` for each source to inject.
- **c1** – Scaling parameter as a function of frequency to inject.
- **c2** (*[type], optional*) – Offset parameter to inject as a function of frequency.
- **t_unc** – Uncorrelated temperature to inject (as function of frequency)
- **t_cos** – Correlated temperature to inject (as function of frequency)
- **t_sin** – Correlated temperature to inject (as function of frequency)
- **averaged_spectra** – Dictionary of `{source: spectrum}` for each source to inject.

Returns

CalibrationObservation – A new observation object with the injected models.

edges_cal.cal_coefficients.CalibrationObservation.new_load

CalibrationObservation.**new_load**(*load_name: str*, *io_obj: CalibrationObservation*, *reflection_kwargs: dict | None = None*, *spec_kwargs: dict | None = None*)

Create a new load with the given load name.

Uses files inside the current observation.

Parameters

- **load_name** (*str*) – The name of the load
- **run_num_spec** (*dict or int*) – Run number to use for the spectrum.
- **run_num_load** (*dict or int*) – Run number to use for the load's S11.
- **reflection_kwargs** (*dict*) – Keyword arguments to construct the `SwitchCorrection`.
- **spec_kwargs** (*dict*) – Keyword arguments to construct the `LoadSpectrum`.

edges_cal.cal_coefficients.CalibrationObservation.plot_calibrated_temp

CalibrationObservation.**plot_calibrated_temp**(*load: Load | str*, *bins: int = 2*, *fig=None*, *ax=None*, *xlabel=True*, *ylabel=True*, *label: str = ''*, *as_residuals: bool = False*, *load_in_title: bool = False*, *rms_in_label: bool = True*)

Make a plot of calibrated temperature for a given source.

Parameters

- **load** (`LoadSpectrum` instance) – Source to plot.
- **bins** (*int*) – Number of bins to smooth over (std of Gaussian kernel)
- **fig** (*Figure*) – Optionally provide a matplotlib figure to add to.
- **ax** (*Axis*) – Optionally provide a matplotlib Axis to add to.
- **xlabel** (*bool*) – Whether to write the x-axis label
- **ylabel** (*bool*) – Whether to write the y-axis label

Returns

fig – The matplotlib figure that was created.

edges_cal.cal_coefficients.CalibrationObservation.plot_calibrated_temps

CalibrationObservation.**plot_calibrated_temps**(*bins=64*, *fig=None*, *ax=None*, ***kwargs*)

Plot all calibrated temperatures in a single figure.

Parameters

bins (*int*) – Number of bins in the smoothed spectrum

Returns

fig – Matplotlib figure that was created.

edges_cal.cal_coefficients.CalibrationObservation.plot_coefficients

CalibrationObservation.**plot_coefficients**(*fig=None*, *ax=None*)

Make a plot of the calibration models, C1, C2, Tunc, Tcos and Tsin.

Parameters

- **fig** (*Figure*) – Optionally pass a matplotlib figure to add to.
- **ax** (*Axis*) – Optionally pass a matplotlib axis to pass to. Must have 5 axes.

edges_cal.cal_coefficients.CalibrationObservation.plot_raw_spectra

CalibrationObservation.**plot_raw_spectra**(*fig=None*, *ax=None*) → Figure

Plot raw uncalibrated spectra for all calibrator sources.

Parameters

- **fig** (`plt.Figure`) – A matplotlib figure on which to make the plot. By default creates a new one.
- **ax** (`plt.Axes`) – A matplotlib Axes on which to make the plot. By default creates a new one.

Returns

fig (`plt.Figure`) – The figure on which the plot was made.

edges_cal.cal_coefficients.CalibrationObservation.plot_s11_models

CalibrationObservation.**plot_s11_models**(***kwargs*)

Plot residuals of S11 models for all sources.

Returns

dict – Each entry has a key of the source name, and the value is a matplotlib fig.

edges_cal.cal_coefficients.CalibrationObservation.to_calibrator

CalibrationObservation.**to_calibrator**()

Directly create a *Calibrator* object without writing to file.

edges_cal.cal_coefficients.CalibrationObservation.with_load_calkit

CalibrationObservation.**with_load_calkit**(*calkit*, *loads: Sequence[str] = None*)

Return a new observation with loads having given calkit.

edges_cal.cal_coefficients.CalibrationObservation.write

CalibrationObservation.**write**(*filename: str | Path*)

Write all information required to calibrate a new spectrum to file.

Parameters

filename (*path*) – The filename to write to.

Attributes

C1_poly	*np.poly1d* object describing the Scaling calibration coefficient C1.
C2_poly	*np.poly1d* object describing the offset calibration coefficient C2.
Tcos_poly	*np.poly1d* object describing the cosine noise-wave parameter, Tcos.
Tsin_poly	*np.poly1d* object describing the sine noise-wave parameter, Tsin.
Tunc_poly	*np.poly1d* object describing the uncorrelated noise-wave parameter, Tunc.
freq	The frequencies at which spectra were measured.
internal_switch	
load_names	
metadata	Metadata associated with the object.
receiver_s11	The corrected S11 of the LNA evaluated at the data frequencies.
s11_correction_models	Dictionary of S11 correction models, one for each source.
source_thermistor_temps	Dictionary of input source thermistor temperatures.
t_load	
t_load_ns	
loads	
receiver	
cterms	
wterms	

edges_cal.cal_coefficients.CalibrationObservation.C1_poly

CalibrationObservation.**C1_poly**

np.poly1d object describing the Scaling calibration coefficient C1.

The polynomial is defined to act on normalized frequencies such that *freq.min* and *freq.max* map to -1 and 1 respectively. Use *C1()* as a direct function on frequency.

edges_cal.cal_coefficients.CalibrationObservation.C2_poly

CalibrationObservation.**C2_poly**

np.poly1d object describing the offset calibration coefficient C2.

The polynomial is defined to act on normalized frequencies such that *freq.min* and *freq.max* map to -1 and 1 respectively. Use *C2()* as a direct function on frequency.

edges_cal.cal_coefficients.CalibrationObservation.Tcos_poly

CalibrationObservation.**Tcos_poly**

np.poly1d object describing the cosine noise-wave parameter, Tcos.

The polynomial is defined to act on normalized frequencies such that *freq.min* and *freq.max* map to -1 and 1 respectively. Use *Tcos()* as a direct function on frequency.

edges_cal.cal_coefficients.CalibrationObservation.Tsin_poly

CalibrationObservation.**Tsin_poly**

np.poly1d object describing the sine noise-wave parameter, Tsin.

The polynomial is defined to act on normalized frequencies such that *freq.min* and *freq.max* map to -1 and 1 respectively. Use *Tsin()* as a direct function on frequency.

edges_cal.cal_coefficients.CalibrationObservation.Tunc_poly

CalibrationObservation.**Tunc_poly**

np.poly1d object describing the uncorrelated noise-wave parameter, Tunc.

The polynomial is defined to act on normalized frequencies such that *freq.min* and *freq.max* map to -1 and 1 respectively. Use *Tunc()* as a direct function on frequency.

edges_cal.cal_coefficients.CalibrationObservation.freq

CalibrationObservation.**freq**

The frequencies at which spectra were measured.

edges_cal.cal_coefficients.CalibrationObservation.internal_switch

property CalibrationObservation.**internal_switch**

edges_cal.cal_coefficients.CalibrationObservation.load_names

property CalibrationObservation.**load_names**

edges_cal.cal_coefficients.CalibrationObservation.metadata

property CalibrationObservation.**metadata**

Metadata associated with the object.

edges_cal.cal_coefficients.CalibrationObservation.receiver_s11

CalibrationObservation.**receiver_s11**

The corrected S11 of the LNA evaluated at the data frequencies.

edges_cal.cal_coefficients.CalibrationObservation.s11_correction_models

CalibrationObservation.**s11_correction_models**

Dictionary of S11 correction models, one for each source.

edges_cal.cal_coefficients.CalibrationObservation.source_thermistor_temps

CalibrationObservation.**source_thermistor_temps**

Dictionary of input source thermistor temperatures.

edges_cal.cal_coefficients.CalibrationObservation.t_load

property CalibrationObservation.**t_load**

edges_cal.cal_coefficients.CalibrationObservation.t_load_ns

property CalibrationObservation.**t_load_ns**

edges_cal.cal_coefficients.CalibrationObservation.loads

CalibrationObservation.**loads**: dict[str, *edges_cal.cal_coefficients.Load*]

edges_cal.cal_coefficients.CalibrationObservation.receiver

CalibrationObservation.**receiver**: Receiver

edges_cal.cal_coefficients.CalibrationObservation.cterms

CalibrationObservation.**cterms**: int

edges_cal.cal_coefficients.CalibrationObservation.wterms

CalibrationObservation.**wterms**: int

edges_cal.cal_coefficients.Calibrator

class edges_cal.cal_coefficients.**Calibrator**(*, *freq:* FrequencyRange, *cterms: int*, *wterms: int*, *C1: Callable[[ndarray], ndarray]*, *C2: Callable[[ndarray], ndarray]*, *Tunc: Callable[[ndarray], ndarray]*, *Tcos: Callable[[ndarray], ndarray]*, *Tsin: Callable[[ndarray], ndarray]*, *receiver_s11: Callable[[ndarray], ndarray]*, *internal_switch*, *t_load: float = 300*, *t_load_ns: float = 350*, *metadata: dict = _Nothing.NOTHING*)

Methods

__init__(*, freq, cterms, wterms, C1, C2, ...)	Method generated by attrs for class Calibrator.
calibrate_Q(freq, q, ant_s11)	Calibrate given power ratio spectrum.
calibrate_temp(freq, temp, ant_s11)	Calibrate given uncalibrated spectrum.
clone(**kwargs)	Clone the instance with new parameters.
decalibrate_temp(freq, temp, ant_s11)	De-calibrate given calibrated spectrum.
from_calfile(path)	Generate from calfile.
from_calobs(calobs)	Generate a `Calibration` from an in-memory observation.
from_calobs_file(path)	Generate from calobs file.
from_old_calfile(path)	Read from older calfiles.

edges_cal.cal_coefficients.Calibrator.__init__

Calibrator.**__init__**(*, *freq:* FrequencyRange, *cterms: int*, *wterms: int*, *C1: Callable[[ndarray], ndarray]*, *C2: Callable[[ndarray], ndarray]*, *Tunc: Callable[[ndarray], ndarray]*, *Tcos: Callable[[ndarray], ndarray]*, *Tsin: Callable[[ndarray], ndarray]*, *receiver_s11: Callable[[ndarray], ndarray]*, *internal_switch*, *t_load: float = 300*, *t_load_ns: float = 350*, *metadata: dict = _Nothing.NOTHING*) → None

Method generated by attrs for class Calibrator.

edges_cal.cal_coefficients.Calibrator.calibrate_Q

Calibrator.**calibrate_Q**(*freq: ndarray*, *q: ndarray*, *ant_s11: ndarray*) → ndarray

Calibrate given power ratio spectrum.

Parameters

- **freq** (*np.ndarray*) – The frequencies at which to calibrate
- **q** (*np.ndarray*) – The power ratio to calibrate.
- **ant_s11** (*np.ndarray*) – The antenna S11 for the load.

Returns

temp (*np.ndarray*) – The calibrated temperature.

edges_cal.cal_coefficients.Calibrator.calibrate_temp

Calibrator.**calibrate_temp**(*freq: ndarray*, *temp: ndarray*, *ant_s11: ndarray*)

Calibrate given uncalibrated spectrum.

Parameters

- **freq** (*np.ndarray*) – The frequencies at which to calibrate
- **temp** (*np.ndarray*) – The temperatures to calibrate (in K).
- **ant_s11** (*np.ndarray*) – The antenna S11 for the load.

Returns

temp (*np.ndarray*) – The calibrated temperature.

edges_cal.cal_coefficients.Calibrator.clone

Calibrator.**clone**(***kwargs*)

Clone the instance with new parameters.

edges_cal.cal_coefficients.Calibrator.decalibrate_temp

Calibrator.**decalibrate_temp**(*freq*, *temp*, *ant_s11*)

De-calibrate given calibrated spectrum.

Parameters

- **freq** (*np.ndarray*) – The frequencies at which to calibrate
- **temp** (*np.ndarray*) – The temperatures to calibrate (in K).
- **ant_s11** (*np.ndarray*) – The antenna S11 for the load.

Returns

temp (*np.ndarray*) – The calibrated temperature.

Notes

Using this and then **:method:`calibrate_temp`** immediately should be an identity operation.

edges_cal.cal_coefficients.Calibrator.from_calfile

classmethod Calibrator.**from_calfile**(*path: str | Path*) → *Calibrator*

Generate from calfile.

edges_cal.cal_coefficients.Calibrator.from_calobs

classmethod Calibrator.**from_calobs**(*calobs:* CalibrationObservation) → *Calibrator*

Generate a Calibration from an in-memory observation.

edges_cal.cal_coefficients.Calibrator.from_calobs_file

classmethod Calibrator.**from_calobs_file**(*path: str | Path*) → *Calibrator*

Generate from calobs file.

edges_cal.cal_coefficients.Calibrator.from_old_calfile

classmethod Calibrator.**from_old_calfile**(*path: str | Path*) → *Calibrator*

Read from older calfiles.

Attributes

freq
cterms
wterms
t_load
t_load_ns
metadata

edges_cal.cal_coefficients.Calibrator.freq

Calibrator.**freq**: *FrequencyRange*

edges_cal.cal_coefficients.Calibrator.cterms

Calibrator.**cterms**: int

edges_cal.cal_coefficients.Calibrator.wterms

Calibrator.**wterms**: int

edges_cal.cal_coefficients.Calibrator.t_load

Calibrator.**t_load**: float

edges_cal.cal_coefficients.Calibrator.t_load_ns

Calibrator.**t_load_ns**: float

edges_cal.cal_coefficients.Calibrator.metadata

Calibrator.**metadata**: dict

edges_cal.cal_coefficients.HotLoadCorrection

class edges_cal.cal_coefficients.**HotLoadCorrection**(*, *freq: ~edges_cal.tools.FrequencyRange*, *raw_s11: ~numpy.ndarray*, *raw_s12s21: ~numpy.ndarray*, *raw_s22: ~numpy.ndarray*, *model: ~edges_cal.modelling.Model = Polynomial(parameters=None, n_terms=21, transform=IdentityTransform(), offset=0.0)*, *complex_model: type[edges_cal.modelling.ComplexRealImagModel] | type[edges_cal.modelling.ComplexMagPhaseModel] = <class 'edges_cal.modelling.ComplexMagPhaseModel'>*, *use_spline: bool = False*)

Corrections for the hot load.

Measurements required to define the HotLoad temperature, from Monsalve et al. (2017), Eq. 8+9.

Parameters

- **path** – Path to a file containing measurements of the semi-rigid cable reflection parameters. A preceding colon (:) indicates to prefix with DATA_PATH. The default file was measured in 2015, but there is also a file included that can be used from 2017: ":semi_rigid_s_parameters_2017.txt".
- **f_low, f_high** – Lowest/highest frequency to retain from measurements.
- **n_terms** – The number of terms used in fitting S-parameters of the cable.

Methods

__init__(*, freq, raw_s11, raw_s12s21, raw_s22)	Method generated by attrs for class HotLoadCorrection.
from_file([path, f_low, f_high, ...])	Instantiate the HotLoadCorrection from file.
get_power_gain(semi_rigid_sparams, hot_load_s11)	Define Eq.
power_gain(freq, hot_load_s11)	Calculate the power gain.

edges_cal.cal_coefficients.HotLoadCorrection.__init__

HotLoadCorrection.**__init__**(*, *freq: ~edges_cal.tools.FrequencyRange*, *raw_s11: ~numpy.ndarray*, *raw_s12s21: ~numpy.ndarray*, *raw_s22: ~numpy.ndarray*, *model: ~edges_cal.modelling.Model = Polynomial(parameters=None, n_terms=21, transform=IdentityTransform(), offset=0.0)*, *complex_model: type[edges_cal.modelling.ComplexRealImagModel] | type[edges_cal.modelling.ComplexMagPhaseModel] = <class 'edges_cal.modelling.ComplexMagPhaseModel'>*, *use_spline: bool = False*) → None

Method generated by attrs for class HotLoadCorrection.

edges_cal.cal_coefficients.HotLoadCorrection.from_file

classmethod HotLoadCorrection.**from_file**(*path: str | ~pathlib.Path = ':semi_rigid_s_parameters_WITH_HEADER.txt'*, *f_low: ~astropy.units.quantity.Quantity = <Quantity 0. MHz>*, *f_high: ~astropy.units.quantity.Quantity = <Quantity inf MHz>*, *set_transform_range: bool = True*, ***kwargs*)

Instantiate the HotLoadCorrection from file.

Parameters

- **path** – Path to the S-parameters file.
- **f_low, f_high** – The min/max frequencies to use in the modelling.

edges_cal.cal_coefficients.HotLoadCorrection.get_power_gain

static HotLoadCorrection.**get_power_gain**(*semi_rigid_sparams: dict*, *hot_load_s11: ndarray*) → ndarray

Define Eq. 9 from M17.

Parameters

- **semi_rigid_sparams** (*dict*) – A dictionary of reflection coefficient measurements as a function of frequency for the semi-rigid cable.
- **hot_load_s11** (*array-like*) – The S11 measurement of the hot_load.

Returns

gain (*np.ndarray*) – The power gain.

edges_cal.cal_coefficients.HotLoadCorrection.power_gain

HotLoadCorrection.**power_gain**(*freq: Quantity*, *hot_load_s11: LoadS11*) → ndarray

Calculate the power gain.

Parameters

- **freq** (*np.ndarray*) – The frequencies.
- **hot_load_s11** (LoadS11) – The S11 of the hot load.

Returns

gain (*np.ndarray*) – The power gain as a function of frequency.

Attributes

s11_model	The reflection coefficient.
s12s21_model	The transmission coefficient.
s22_model	The reflection coefficient from the other side.
freq	
raw_s11	
raw_s12s21	
raw_s22	
model	
complex_model	
use_spline	

edges_cal.cal_coefficients.HotLoadCorrection.s11_model

HotLoadCorrection.**s11_model**

The reflection coefficient.

edges_cal.cal_coefficients.HotLoadCorrection.s12s21_model

HotLoadCorrection.**s12s21_model**

The transmission coefficient.

edges_cal.cal_coefficients.HotLoadCorrection.s22_model

HotLoadCorrection.**s22_model**

The reflection coefficient from the other side.

edges_cal.cal_coefficients.HotLoadCorrection.freq

HotLoadCorrection.**freq**: *FrequencyRange*

edges_cal.cal_coefficients.HotLoadCorrection.raw_s11

HotLoadCorrection.**raw_s11**: ndarray

edges_cal.cal_coefficients.HotLoadCorrection.raw_s12s21

HotLoadCorrection.**raw_s12s21**: ndarray

edges_cal.cal_coefficients.HotLoadCorrection.raw_s22

HotLoadCorrection.**raw_s22**: ndarray

edges_cal.cal_coefficients.HotLoadCorrection.model

HotLoadCorrection.**model**: *Model*

edges_cal.cal_coefficients.HotLoadCorrection.complex_model

HotLoadCorrection.**complex_model**: type[*edges_cal.modelling.ComplexRealImagModel*] | type[*edges_cal.modelling.ComplexMagPhaseModel*]

edges_cal.cal_coefficients.HotLoadCorrection.use_spline

HotLoadCorrection.**use_spline**: bool

edges_cal.cal_coefficients.Load

class edges_cal.cal_coefficients.**Load**(*, *spectrum: LoadSpectrum*, *reflections: LoadS11*, *loss_model: Callable[[ndarray], ndarray]* | HotLoadCorrection | *None* = *None*, *ambient_temperature: float* = *298.0*)

Wrapper class containing all relevant information for a given load.

Parameters

- **spectrum** (LoadSpectrum) – The spectrum for this particular load.
- **reflections** (SwitchCorrection) – The S11 measurements for this particular load.
- **hot_load_correction** (*HotLoadCorrection*) – If this is a hot load, provide a hot load correction.
- **ambient** (LoadSpectrum) – If this is a hot load, need to provide an ambient spectrum to correct it.

Methods

__init__(*, spectrum, reflections[, ...])	Method generated by attrs for class Load.
from_io(io_obj, load_name[, f_low, f_high, ...])	Define a full *Load* from a path and name.
get_temp_with_loss([freq])	Calculate the temperature of the load accounting for loss.
with_calkit(calkit)	Return a new Load with updated calkit.

edges_cal.cal_coefficients.Load.__init__

Load.**__init__**(*, *spectrum: LoadSpectrum*, *reflections: LoadS11*, *loss_model: Callable[[ndarray], ndarray]* | HotLoadCorrection | *None* = *None*, *ambient_temperature: float* = *298.0*) → None

Method generated by attrs for class Load.

edges_cal.cal_coefficients.Load.from_io

classmethod Load.**from_io**(*io_obj: ~edges_io.io.CalibrationObservation*, *load_name: str*, *f_low: ~astropy.units.quantity.Quantity* = *<Quantity 40. MHz>*, *f_high: ~astropy.units.quantity.Quantity* = *<Quantity inf MHz>*, *reflection_kwargs: dict | None* = *None*, *spec_kwargs: dict | None* = *None*, *loss_kwargs: dict | None* = *None*, *ambient_temperature: float | None* = *None*)

Define a full *Load* from a path and name.

Parameters

- **path** (*str or Path*) – Path to the top-level calibration observation.
- **load_name** (*str*) – Name of a load to define.
- **f_low, f_high** (*float*) – Min/max frequencies to keep in measurements.
- **reflection_kwargs** (*dict*) – Extra arguments to pass through to `SwitchCorrection`.
- **spec_kwargs** (*dict*) – Extra arguments to pass through to `LoadSpectrum`.
- **ambient_temperature** – The ambient temperature to use for the loss, if required (required for new hot loads). By default, read an ambient load's actual temperature reading from the io object.

Returns

load (*Load*) – The load object, containing all info about spectra and S11's for that load.

edges_cal.cal_coefficients.Load.get_temp_with_loss

Load.**get_temp_with_loss**(*freq: Quantity | None* = *None*)

Calculate the temperature of the load accounting for loss.

edges_cal.cal_coefficients.Load.with_calkit

Load.**with_calkit**(*calkit:* Calkit)

Return a new Load with updated calkit.

Attributes

averaged_Q	The average spectrum power ratio, Q (over time).
averaged_spectrum	The average uncalibrated spectrum (over time).
freq	Frequencies of the spectrum.
load_name	The name of the load.
loss_model	The loss model as a callable function of frequency.
s11_model	Callable S11 model as function of frequency.
t_load	The assumed temperature of the internal load.
t_load_ns	The assumed temperature of the internal load + noise source.
temp_ave	The average temperature of the thermistor (over frequency and time).
spectrum	
reflections	
ambient_temperature	

edges_cal.cal_coefficients.Load.averaged_Q

property Load.**averaged_Q: ndarray**

The average spectrum power ratio, Q (over time).

edges_cal.cal_coefficients.Load.averaged_spectrum

property Load.**averaged_spectrum: ndarray**

The average uncalibrated spectrum (over time).

edges_cal.cal_coefficients.Load.freq

property Load.**freq:** *FrequencyRange*

Frequencies of the spectrum.

edges_cal.cal_coefficients.Load.load_name

property Load.load_name: str
The name of the load.

edges_cal.cal_coefficients.Load.loss_model

property Load.loss_model
The loss model as a callable function of frequency.

edges_cal.cal_coefficients.Load.s11_model

property Load.s11_model: Callable[[ndarray], ndarray]
Callable S11 model as function of frequency.

edges_cal.cal_coefficients.Load.t_load

property Load.t_load: float
The assumed temperature of the internal load.

edges_cal.cal_coefficients.Load.t_load_ns

property Load.t_load_ns: float
The assumed temperature of the internal load + noise source.

edges_cal.cal_coefficients.Load.temp_ave

Load.temp_ave
The average temperature of the thermistor (over frequency and time).

edges_cal.cal_coefficients.Load.spectrum

Load.spectrum: LoadSpectrum

edges_cal.cal_coefficients.Load.reflections

Load.reflections: LoadS11

edges_cal.cal_coefficients.Load.ambient_temperature

Load.**ambient_temperature**: float

3.5.2 Lower Level Methods

edges_cal.reflection_coefficient	Functions for working with reflection coefficients.
edges_cal.receiver_calibration_func	Functions for calibrating the receiver.
edges_cal.modelling	Functions for generating least-squares model fits for linear models.
edges_cal.xrfi	Functions for excising RFI.
edges_cal.tools	Tools to use in other modules.

edges_cal.reflection_coefficient

Functions for working with reflection coefficients.

Most of the functions in this module follow the formalism/notation of

> Monsalve et al., 2016, "One-Port Direct/Reverse Method for Characterizing VNA Calibration Standards", IEEE Transactions on Microwave Theory and Techniques, vol. 64, issue 8, pp. 2631-2639, https://arxiv.org/pdf/1606.02446.pdf

They represent basic relations between physical parameters of circuits, as measured with internal standards.

Functions

CalkitMatch([resistance])	Create a Match standard.
CalkitOpen(**kwargs)	Factory function for creating Open standards, with resistance=inf.
CalkitShort(**kwargs)	Factor function for creating Short standards, with resistance=0.
agilent_85033E(f, resistance_of_match[, ...])	Generate open, short and match standards for the Agilent 85033E.
de_embed(gamma_open_intr, gamma_short_intr, ...)	Obtain network S-parameters from OSL standards and intrinsic reflections of DUT.
gamma2impedance(gamma, z0)	Convert reflection coeffiency to impedance.
gamma_de_embed(s11, s12s21, s22, gamma_ref)	Obtain the intrinsic reflection coefficient.
gamma_embed(s11, s12s21, s22, gamma)	Obtain the intrinsic reflection coefficient.
get_calkit(base[, resistance_of_match, ...])	Get a calkit based on a provided base calkit, with given updates.
get_sparams(gamma_open_intr, ...)	Obtain network S-parameters from OSL standards and intrinsic reflections of DUT.
impedance2gamma(z, z0)	Convert impedance to reflection coefficient.
input_impedance_transmission_line(z0, gamma, ...)	Calculate the impedance of a terminated transmission line.

edges_cal.reflection_coefficient.CalkitMatch

edges_cal.reflection_coefficient.**CalkitMatch**(*resistance=<Quantity 50. Ohm>*, ***kwargs*) → *CalkitStandard*

Create a Match standard.

See *CalkitStandard* for all possible parameters.

edges_cal.reflection_coefficient.CalkitOpen

edges_cal.reflection_coefficient.**CalkitOpen**(***kwargs*) → *CalkitStandard*

Factory function for creating Open standards, with resistance=inf.

See *CalkitStandard* for all parameters available.

edges_cal.reflection_coefficient.CalkitShort

edges_cal.reflection_coefficient.**CalkitShort**(***kwargs*) → *CalkitStandard*

Factor function for creating Short standards, with resistance=0.

See *CalkitStandard* for all parameters available.

edges_cal.reflection_coefficient.agilent_85033E

edges_cal.reflection_coefficient.**agilent_85033E**(*f: ndarray*, *resistance_of_match: float*, *match_delay: bool = True*, *md_value_ps: float = 38.0*)

Generate open, short and match standards for the Agilent 85033E.

Note: this function is deprecated. Please use the methods of the Calkit objects instead!

Parameters

- **f** (*np.ndarray*) – Frequencies in MHz.
- **resistance_of_match** (*float*) – Resistance of the match standard, in Ohms.
- **match_delay** (*bool*) – Whether to match the delay offset.
- **md_value_ps** (*float*) – Some number that does something to the delay matching.

Returns

o, s, m (*np.ndarray*) – The open, short and match standards.

edges_cal.reflection_coefficient.de_embed

edges_cal.reflection_coefficient.**de_embed**(*gamma_open_intr: ndarray | float*, *gamma_short_intr: ndarray | float*, *gamma_match_intr: ndarray | float*, *gamma_open_meas: ndarray*, *gamma_short_meas: ndarray*, *gamma_match_meas: ndarray*, *gamma_ref*) → tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

Obtain network S-parameters from OSL standards and intrinsic reflections of DUT.

See Eq. 3 of Monsalve et al., 2016.

Parameters

- **gamma_open_intr** – The intrinsic reflection of the open standard (assumed as true) as a function of frequency.
- **gamma_shrt_intr** – The intrinsic reflection of the short standard (assumed as true) as a function of frequency.
- **gamma_load_intr** – The intrinsic reflection of the load standard (assumed as true) as a function of frequency.
- **gamma_open_meas** – The reflection of the open standard measured at port 1 as a function of frequency.
- **gamma_shrt_meas** – The reflection of the short standard measured at port 1 as a function of frequency.
- **gamma_load_meas** – The reflection of the load standard measured at port 1 as a function of frequency.
- **gamma_ref** – The reflection coefficient of the device under test (DUT) at the reference plane.

Returns

- *gamma* – The intrinsic reflection coefficient of the DUT.
- *s11* – The S11 of the network.
- *s12s21* – The product *S12*S21* of the network
- *s22* – The S22 of the network.

edges_cal.reflection_coefficient.gamma2impedance

edges_cal.reflection_coefficient.**gamma2impedance**(*gamma: float | ndarray*, *z0: float | ndarray*) → float | ndarray

Convert reflection coeffiency to impedance.

See Eq. 19 of Monsalve et al. 2016.

Parameters

- **gamma** – Reflection coefficient.
- **z0** – Reference impedance.

Returns

z – The impedance.

edges_cal.reflection_coefficient.gamma_de_embed

edges_cal.reflection_coefficient.**gamma_de_embed**(*s11: np.typing.ArrayLike*, *s12s21: np.typing.ArrayLike*, *s22: np.typing.ArrayLike*, *gamma_ref: np.typing.ArrayLike*) → np.typing.ArrayLike

Obtain the intrinsic reflection coefficient.

See Eq. 2 of Monsalve et al., 2016.

Obtains the instrinsic reflection coefficient from the one measured at the reference plane, given a set of reflection coefficients.

Parameters

- **s11** – The S11 parameter of the two-port network for the port facing the calibration plane.
- **s12s21** – The product of `S12*S21` of the two-port network.
- **s22** – The S22 of the two-port network for the port facing the device under test (DUT)
- **gamma_ref** – The reflection coefficient of the device under test (DUT) measured at the reference plane.

Returns

gamma – The intrinsic reflection coefficient of the DUT.

See also:

gamma_embed
The inverse function to this one.

edges_cal.reflection_coefficient.gamma_embed

edges_cal.reflection_coefficient.**gamma_embed**(*s11: np.typing.ArrayLike*, *s12s21: np.typing.ArrayLike*, *s22: np.typing.ArrayLike*, *gamma: np.typing.ArrayLike*) → np.typing.ArrayLike

Obtain the intrinsic reflection coefficient.

See Eq. 1 of Monsalve et al., 2016.

Obtains the instrinsic reflection coefficient from the one measured at the reference plane, given a set of reflection coefficients.

Parameters

- **s11** – The S11 parameter of the two-port network for the port facing the calibration plane.
- **s12s21** – The product of `S12*S21` of the two-port network.
- **s22** – The S22 of the two-port network for the port facing the device under test (DUT)
- **gamma** – The intrinsic reflection coefficient of the device under test (DUT);.

Returns

gamma_ref – The reflection coefficient of the DUT measured at the reference plane.

See also:

gamma_de_embed
The inverse function to this one.

edges_cal.reflection_coefficient.get_calkit

edges_cal.reflection_coefficient.**get_calkit**(*base*, *resistance_of_match: Quantity | None = None*, *open: dict | None = None*, *short: dict | None = None*, *match: dict | None = None*)

Get a calkit based on a provided base calkit, with given updates.

Parameters

- **base** – The base calkit to use, eg. AGILENT_85033E

- **resistance_of_match** – The resistance of the match, overwrites default from the base.
- **open** – Dictionary of parameters to overwrite the open standard.
- **short** – Dictionary of parameters to overwrite the short standard.
- **match** – Dictionary of parameters to overwrite the match standard.

edges_cal.reflection_coefficient.get_sparams

edges_cal.reflection_coefficient.**get_sparams**(*gamma_open_intr: ndarray | float*, *gamma_short_intr: ndarray | float*, *gamma_match_intr: ndarray | float*, *gamma_open_meas: ndarray*, *gamma_short_meas: ndarray*, *gamma_match_meas: ndarray*) → tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Obtain network S-parameters from OSL standards and intrinsic reflections of DUT.

See Eq. 3 of Monsalve et al., 2016.

Parameters

- **gamma_open_intr** – The intrinsic reflection of the open standard (assumed as true) as a function of frequency.
- **gamma_shrt_intr** – The intrinsic reflection of the short standard (assumed as true) as a function of frequency.
- **gamma_load_intr** – The intrinsic reflection of the load standard (assumed as true) as a function of frequency.
- **gamma_open_meas** – The reflection of the open standard measured at port 1 as a function of frequency.
- **gamma_shrt_meas** – The reflection of the short standard measured at port 1 as a function of frequency.
- **gamma_load_meas** – The reflection of the load standard measured at port 1 as a function of frequency.

Returns

- *s11* – The S11 of the network.
- *s12s21* – The product *S12*S21* of the network
- *s22* – The S22 of the network.

edges_cal.reflection_coefficient.impedance2gamma

edges_cal.reflection_coefficient.**impedance2gamma**(*z: float | ndarray*, *z0: float | ndarray*) → float | ndarray

Convert impedance to reflection coefficient.

See Eq. 19 of Monsalve et al. 2016.

Parameters

- **z** – Impedance.
- **z0** – Reference impedance.

Returns
gamma – The reflection coefficient.

edges_cal.reflection_coefficient.input_impedance_transmission_line

edges_cal.reflection_coefficient.**input_impedance_transmission_line**(*z0: ndarray*, *gamma: ndarray*, *length: float*, *z_load: ndarray*) → ndarray

Calculate the impedance of a terminated transmission line.

Parameters

- **z0** (*array-like*) – Complex characteristic impedance
- **gamma** (*array-like*) – Propagation constant
- **length** (*float*) – Length of transmission line
- **z_load** (*array-like*) – Impedance of termination.

Returns
Impedance of the transmission line.

Classes

Calkit(open, short, match)	
CalkitStandard(*, resistance[, ...])	Class representing a calkit standard.

edges_cal.reflection_coefficient.Calkit

class edges_cal.reflection_coefficient.**Calkit**(*open: CalkitStandard*, *short: CalkitStandard*, *match: CalkitStandard*)

Methods

__init__(open, short, match)	Method generated by attrs for class Calkit.
clone(*[, short, open, match])	Return a clone with updated parameters for each standard.

edges_cal.reflection_coefficient.Calkit.__init__

Calkit.**__init__**(*open:* CalkitStandard, *short:* CalkitStandard, *match:* CalkitStandard) → None

Method generated by attrs for class Calkit.

edges_cal.reflection_coefficient.Calkit.clone

Calkit.**clone**(*, *short=None*, *open=None*, *match=None*)

Return a clone with updated parameters for each standard.

Attributes

open
short
match

edges_cal.reflection_coefficient.Calkit.open

Calkit.**open:** *CalkitStandard*

edges_cal.reflection_coefficient.Calkit.short

Calkit.**short:** *CalkitStandard*

edges_cal.reflection_coefficient.Calkit.match

Calkit.**match:** *CalkitStandard*

edges_cal.reflection_coefficient.CalkitStandard

class edges_cal.reflection_coefficient.**CalkitStandard**(*, *resistance*, *offset_impedance=<Quantity 50. Ohm>*, *offset_delay=<Quantity 30. ps>*, *offset_loss=<Quantity 2.2 GOhm / s>*, *capacitance_model: ~edges_cal.modelling.Polynomial | None = None*, *inductance_model: ~edges_cal.modelling.Polynomial | None = None*)

Class representing a calkit standard.

The standard could be open, short or load/match. See the Appendix of Monsalve et al. 2016 for details.

For all parameters, 'offset' refers to the small transmission line section of the standard (not an offset in the parameter).

Parameters

- **resistance** (*float | astropy.units.quantity.Quantity*) – The resistance of the standard termination, either assumed or measured.
- **offset_impedance** (*float | astropy.units.quantity.Quantity*) – Impedance of the transmission line, in Ohms.
- **offset_delay** (*float | astropy.units.quantity.Quantity*) – One-way delay of the transmission line, in picoseconds.
- **offset_loss** (*float | astropy.units.quantity.Quantity*) – One-way loss of the transmission line, unitless.

Methods

__init__(*, resistance[, offset_impedance, ...])	Method generated by attrs for class CalkitStandard.
gl(freq)	Obtain the product gamma*length.
lossy_characteristic_impedance(freq)	Obtain the lossy characteristic impedance of the transmission line (offset).
offset_gamma(freq)	Obtain reflection coefficient of the offset.
reflection_coefficient(freq)	Obtain the combined reflection coefficient of the standard.
termination_gamma(freq)	Reflection coefficient of the termination.
termination_impedance(freq)	The impedance of the termination of the standard.

edges_cal.reflection_coefficient.CalkitStandard.__init__

CalkitStandard.**__init__**(*, *resistance*, *offset_impedance=<Quantity 50. Ohm>*, *offset_delay=<Quantity 30. ps>*, *offset_loss=<Quantity 2.2 GOhm / s>*, *capacitance_model: ~edges_cal.modelling.Polynomial | None = None*, *inductance_model: ~edges_cal.modelling.Polynomial | None = None*) → None

Method generated by attrs for class CalkitStandard.

edges_cal.reflection_coefficient.CalkitStandard.gl

CalkitStandard.**gl**(*freq: Quantity*) → ndarray

Obtain the product gamma*length.

gamma is the propagation constant of the transmission line (offset) and l is its length. See Eq. 21 of Monsalve et al. 2016.

edges_cal.reflection_coefficient.CalkitStandard.lossy_characteristic_impedance

CalkitStandard.**lossy_characteristic_impedance**(*freq: Quantity*) → Quantity

Obtain the lossy characteristic impedance of the transmission line (offset).

See Eq. 20 of Monsalve et al., 2016

edges_cal.reflection_coefficient.CalkitStandard.offset_gamma

CalkitStandard.**offset_gamma**(*freq: Quantity*) → Quantity

Obtain reflection coefficient of the offset.

Eq. 19 of M16.

edges_cal.reflection_coefficient.CalkitStandard.reflection_coefficient

CalkitStandard.**reflection_coefficient**(*freq: Quantity*) → Quantity

Obtain the combined reflection coefficient of the standard.

See Eq. 18 of M16.

edges_cal.reflection_coefficient.CalkitStandard.termination_gamma

CalkitStandard.**termination_gamma**(*freq: Quantity*) → Quantity

Reflection coefficient of the termination.

Eq. 19 of M16.

edges_cal.reflection_coefficient.CalkitStandard.termination_impedance

CalkitStandard.**termination_impedance**(*freq: Quantity*) → Quantity

The impedance of the termination of the standard.

See Eq. 22-25 of M16 for open and short standards. The match standard uses the input measured resistance as the impedance.

Attributes

intrinsic_gamma	The intrinsic reflection coefficient of the idealized standard.
name	The name of the standard.
resistance	
offset_impedance	
offset_delay	
offset_loss	
capacitance_model	
inductance_model	

edges_cal.reflection_coefficient.CalkitStandard.intrinsic_gamma

property CalkitStandard.**intrinsic_gamma: float**

The intrinsic reflection coefficient of the idealized standard.

edges_cal.reflection_coefficient.CalkitStandard.name

property CalkitStandard.**name: str**

The name of the standard. Inferred from the resistance.

edges_cal.reflection_coefficient.CalkitStandard.resistance

CalkitStandard.**resistance: float | Quantity**

edges_cal.reflection_coefficient.CalkitStandard.offset_impedance

CalkitStandard.**offset_impedance: float | Quantity**

edges_cal.reflection_coefficient.CalkitStandard.offset_delay

CalkitStandard.**offset_delay: float | Quantity**

edges_cal.reflection_coefficient.CalkitStandard.offset_loss

CalkitStandard.**offset_loss**: float | Quantity

edges_cal.reflection_coefficient.CalkitStandard.capacitance_model

CalkitStandard.**capacitance_model**: *Polynomial* | None

edges_cal.reflection_coefficient.CalkitStandard.inductance_model

CalkitStandard.**inductance_model**: *Polynomial* | None

edges_cal.receiver_calibration_func

Functions for calibrating the receiver.

Functions

calibrated_antenna_temperature(temp_raw, ...)	Use M17 Eq.
get_F(gamma_rec, gamma_ant)	Get the F parameter for a given receiver and antenna.
get_K(gamma_rec, gamma_ant[, f_ratio, ...])	Determine the S11-dependent factors for each term in Eq.
get_alpha(gamma_rec, gamma_ant)	Get the alpha parameter for a given receiver and antenna.
get_calibration_quantities_iterative(f_norm, ...)	Derive calibration parameters using the scheme laid out in Monsalve (2017).
get_linear_coefficients(gamma_ant, ...[, t_load])	Use Monsalve (2017) Eq.
get_linear_coefficients_from_K(k, sca, off, ...)	Calculate linear coefficients a and b from noise-wave parameters K0-4.
noise_wave_param_fit(f_norm, gamma_rec, ...)	Fit noise-wave polynomial parameters.
power_ratio(temp_ant, gamma_ant, gamma_rec, ...)	Compute the ratio of raw powers from the three-position switch.
temperature_thermistor(resistance[, coeffs, ...])	Convert resistance of a thermistor to temperature.
uncalibrated_antenna_temperature(temp, ...)	Use M17 Eq.

edges_cal.receiver_calibration_func.calibrated_antenna_temperature

edges_cal.receiver_calibration_func.**calibrated_antenna_temperature**(*temp_raw*, *gamma_ant*, *gamma_rec*, *sca*, *off*, *t_unc*, *t_cos*, *t_sin*, *t_load=300*)

Use M17 Eq. 7 to determine calibrated temperature from an uncalibrated temperature.

Parameters

- **temp_raw** (*array_like*) – The raw (uncalibrated) temperature spectrum, T*.
- **gamma_ant** (*array_like*) – S11 of the antenna/load.
- **gamma_rec** (*array_like*) – S11 of the receiver.

- **sca,off** (*array_like*) – Scale and offset calibration parameters (i.e. C1 and C2). These are in the form of arrays over frequency (i.e. it is not the polynomial coefficients).
- **t_unc, t_cos, t_sin** (*array_like*) – Noise-wave calibration parameters (uncorrelated, cosine, sine). These are in the form of arrays over frequency (i.e. not the polynomial coefficients).
- **t_load** (*float, optional*) – The nominal temperature of the internal ambient load. This *must match* the value used to derive the calibration parameters in the first place.

edges_cal.receiver_calibration_func.get_F

edges_cal.receiver_calibration_func.**get_F**(*gamma_rec: ndarray*, *gamma_ant: ndarray*) → ndarray

Get the F parameter for a given receiver and antenna.

Parameters

- **gamma_rec** (*np.ndarray*) – The reflection coefficient (S11) of the receiver.
- **gamma_ant** (*np.ndarray*) – The reflection coefficient (S11) of the antenna

Returns

F (*np.ndarray*) – The F parameter (see M17)

edges_cal.receiver_calibration_func.get_K

edges_cal.receiver_calibration_func.**get_K**(*gamma_rec*, *gamma_ant*, *f_ratio=None*, *alpha=None*, *gain=None*)

Determine the S11-dependent factors for each term in Eq. 7 (Monsalve 2017).

Parameters

- **gamma_rec** (*array_like*) – Receiver S11
- **gamma_ant** (*array_like*) – Antenna (or load) S11.
- **f_ratio** (*array_like, optional*) – The F factor (Eq. 3 of Monsalve 2017). Computed if not given.
- **alpha** (*array_like, optional*) – The alpha factor (Eq. 4 of Monsalve, 2017). Computed if not given.
- **gain** (*array_like, optional*) – The transmission function, (1 - Gamma_rec^2). Computed if not given.

Returns

K0, K1, K2, K3 (*array_like*) – Factors corresponding to T_ant, T_unc, T_cos, T_sin respectively.

edges_cal.receiver_calibration_func.get_alpha

edges_cal.receiver_calibration_func.**get_alpha**(*gamma_rec: ndarray*, *gamma_ant: ndarray*) → ndarray

Get the alpha parameter for a given receiver and antenna.

Parameters

- **gamma_rec** (*np.ndarray*) – The reflection coefficient of the receiver.
- **gamma_ant** (*np.ndarray*) – The reflection coefficient fo the antenna.

edges_cal.receiver_calibration_func.get_calibration_quantities_iterative

edges_cal.receiver_calibration_func.**get_calibration_quantities_iterative**(*f_norm: ndarray*, *temp_raw: dict*, *gamma_rec: ndarray*, *gamma_ant: dict*, *temp_ant: dict*, *cterms: int*, *wterms: int*, *temp_amb_internal: float = 300*)

Derive calibration parameters using the scheme laid out in Monsalve (2017).

All equation numbers and symbol names come from M17 (arxiv:1602.08065).

Parameters

- **f_norm** (*array_like*) – Normalized frequencies (arbitrarily normalised, but standard assumption is that the centre is zero, and the scale is such that the range is (-1, 1))
- **temp_raw** (*dict*) – Dictionary of antenna uncalibrated temperatures, with keys 'ambient', 'hot_load, 'short' and 'open'. Each value is an array with the same length as f_norm.
- **gamma_rec** (*float array*) – Receiver S11 as a function of frequency.
- **gamma_ant** (*dict*) – Dictionary of antenna S11, with keys 'ambient', 'hot_load, 'short' and 'open'. Each value is an array with the same length as f_norm.
- **temp_ant** (*dict*) – Dictionary like *gamma_ant*, except that the values are modelled/smoothed thermistor temperatures for each source load.
- **cterms** (*int*) – Number of polynomial terms for the C_i
- **wterms** (*int*) – Number of polynonmial temrs for the T_i
- **temp_amb_internal** (*float*) – The ambient internal temperature, interpreted as T_L. Note: this must be the same as the T_L used to generate T*.

Returns

sca, off, tu, tc, ts (*np.poly1d*) – 1D polynomial fits for each of the Scale (C_1), Offset (C_2), and noise-wave temperatures for uncorrelated, cos and sin components.

edges_cal.receiver_calibration_func.get_linear_coefficients

edges_cal.receiver_calibration_func.**get_linear_coefficients**(*gamma_ant*, *gamma_rec*, *sca*, *off*, *t_unc*, *t_cos*, *t_sin*, *t_load=300*)

Use Monsalve (2017) Eq. 7 to determine a and b, such that T = aT* + b.

Parameters

- **gamma_ant** (*array_like*) – S11 of the antenna/load.
- **gamma_rec** (*array_like*) – S11 of the receiver.
- **sca,off** (*array_like*) – Scale and offset calibration parameters (i.e. C1 and C2). These are in the form of arrays over frequency (i.e. it is not the polynomial coefficients).
- **t_unc, t_cos, t_sin** (*array_like*) – Noise-wave calibration parameters (uncorrelated, cosine, sine). These are in the form of arrays over frequency (i.e. not the polynomial coefficients).

- **t_load** (*float, optional*) – The nominal temperature of the internal ambient load. This *must match* the value used to derive the calibration parameters in the first place.

edges_cal.receiver_calibration_func.get_linear_coefficients_from_K

edges_cal.receiver_calibration_func.**get_linear_coefficients_from_K**(*k*, *sca*, *off*, *t_unc*, *t_cos*, *t_sin*, *t_load=300*)

Calculate linear coefficients a and b from noise-wave parameters K0-4.

Parameters

- **k** (*np.ndarray*) – Shape (4, nfreq) array with each of the K-coefficients.
- **sca,off** (*array_like*) – Scale and offset calibration parameters (i.e. C1 and C2). These are in the form of arrays over frequency (i.e. it is not the polynomial coefficients).
- **t_unc, t_cos, t_sin** (*array_like*) – Noise-wave calibration parameters (uncorrelated, cosine, sine). These are in the form of arrays over frequency (i.e. not the polynomial coefficients).
- **t_load** (*float, optional*) – The nominal temperature of the internal ambient load. This *must match* the value used to derive the calibration parameters in the first place.

edges_cal.receiver_calibration_func.noise_wave_param_fit

edges_cal.receiver_calibration_func.**noise_wave_param_fit**(*f_norm: ndarray*, *gamma_rec: ndarray*, *gamma_open: ndarray*, *gamma_short: ndarray*, *temp_raw_open: ndarray*, *temp_raw_short: ndarray*, *temp_thermistor_open: ndarray*, *temp_thermistor_short: ndarray*, *wterms: int*)

Fit noise-wave polynomial parameters.

Parameters

- **f_norm** (*array_like*) – Normalized frequencies (arbitrarily normalised, but standard assumption is that the centre is zero, and the scale is such that the range is (-1, 1))
- **gamma_rec** (*array-like*) – Reflection coefficient, as function of frequency, of the receiver.
- **gamma_open** (*array-like*) – Reflection coefficient, as function of frequency, of the open load.
- **gamma_short** (*array-like*) – Reflection coefficient, as function of frequency, of the shorted load.
- **temp_raw_open** (*array-like*) – Raw measured spectrum temperature of open load.
- **temp_raw_short** (*array-like*) – Raw measured spectrum temperature of shorted load.
- **temp_thermistor_open** (*array-like*) – Measured (known) temperature of open load.
- **temp_thermistor_short** (*array-like*) – Measured (known) temperature of shorted load.
- **wterms** (*int*) – The number of polynomial terms to use for each of the noise-wave functions.

Returns

Tunc, Tcos, Tsin (*array_like*) – The solutions to each of T_unc, T_cos and T_sin as functions of frequency.

edges_cal.receiver_calibration_func.power_ratio

edges_cal.receiver_calibration_func.**power_ratio**(*temp_ant*, *gamma_ant*, *gamma_rec*, *scale*, *offset*, *temp_unc*, *temp_cos*, *temp_sin*, *temp_noise_source*, *temp_load*, *return_terms=False*)

Compute the ratio of raw powers from the three-position switch.

Parameters

- **temp_ant** (*array_like, shape (NFREQS,)*) – Temperature of the antenna, or simulator.
- **gamma_ant** (*array_like, shape (NFREQS,)*) – S11 of the antenna (or simulator)
- **gamma_rec** (*array_like, shape (NFREQS,)*) – S11 of the receiver.
- **scale** (`np.poly1d`) – A polynomial representing the C_1 term.
- **offset** (`np.poly1d`) – A polynomial representing the C_2 term
- **temp_unc** (`np.poly1d`) – A polynomial representing the uncorrelated noise-wave parameter
- **temp_cos** (`np.poly1d`) – A polynomial representing the cosine noise-wave parameter
- **temp_sin** (`np.poly1d`) – A polynomial representing the sine noise-wave parameter
- **temp_noise_source** (*array_like, shape (NFREQS,)*) – Temperature of the internal noise source.
- **temp_load** (*array_like, shape (NFREQS,)*) – Temperature of the internal load
- **return_terms** (*bool, optional*) – If True, return the terms of Qp, rather than the sum of them._

Returns

array_like (*the quantity Q_P as a function of frequency.*)

Notes

Computes (as a model)

$$Q_P = (P_ant - P_L)/(P_N S - P_L)$$

edges_cal.receiver_calibration_func.temperature_thermistor

edges_cal.receiver_calibration_func.**temperature_thermistor**(*resistance: float | ndarray*, *coeffs: str | Sequence = 'oven_industries_TR136_170'*, *kelvin: bool = True*)

Convert resistance of a thermistor to temperature.

Uses a pre-defined set of standard coefficients.

Parameters

- **resistance** (*float or array_like*) – The measured resistance (Ohms).
- **coeffs** (*str or len-3 iterable of floats, optional*) – If str, should be an identifier of a standard set of coefficients, otherwise, should specify the coefficients.

- **kelvin** (*bool, optional*) – Whether to return the temperature in K or C.

Returns

float or array_like – The temperature for each *resistance* given.

edges_cal.receiver_calibration_func.uncalibrated_antenna_temperature

edges_cal.receiver_calibration_func.**uncalibrated_antenna_temperature**(*temp*, *gamma_ant*, *gamma_rec*, *sca*, *off*, *t_unc*, *t_cos*, *t_sin*, *t_load=300*)

Use M17 Eq. 7 to determine uncalibrated temperature from a calibrated temperature.

Parameters

- **temp** (*array_like*) – The true (or calibrated) temperature spectrum.
- **gamma_ant** (*array_like*) – S11 of the antenna/load.
- **gamma_rec** (*array_like*) – S11 of the receiver.
- **sca,off** (*array_like*) – Scale and offset calibration parameters (i.e. C1 and C2). These are in the form of arrays over frequency (i.e. it is not the polynomial coefficients).
- **t_unc, t_cos, t_sin** (*array_like*) – Noise-wave calibration parameters (uncorrelated, cosine, sine). These are in the form of arrays over frequency (i.e. not the polynomial coefficients).
- **t_load** (*float, optional*) – The nominal temperature of the internal ambient load. This *must match* the value used to derive the calibration parameters in the first place.

edges_cal.modelling

Functions for generating least-squares model fits for linear models.

Functions

LogPoly(**kwargs)	A factory function for a LogPoly model.
get_mdl(model)	Get a linear model class from a string input.
get_mdl_inst(model, **kwargs)	Get a model instance from given string input.
tuple_converter(x)	Convert input to tuple of floats.

edges_cal.modelling.LogPoly

edges_cal.modelling.**LogPoly**(***kwargs*)

A factory function for a LogPoly model.

edges_cal.modelling.get_mdl

edges_cal.modelling.**get_mdl**(*model: str | type[*edges_cal.modelling.Model*]*) → type[*edges_cal.modelling.Model*]

Get a linear model class from a string input.

edges_cal.modelling.get_mdl_inst

edges_cal.modelling.**get_mdl_inst**(*model: str | Model | type[*edges_cal.modelling.Model*]*, ***kwargs*) → *Model*

Get a model instance from given string input.

edges_cal.modelling.tuple_converter

edges_cal.modelling.**tuple_converter**(*x*)

Convert input to tuple of floats.

Classes

CentreTransform(*, range[, centre])	
ComplexMagPhaseModel(mag, phs)	A composite model that is specifically for complex functions in mag/phase.
ComplexRealImagModel(real, imag)	A composite model that is specifically for complex functions in real/imag.
CompositeModel(*, models[, extra_basis])	
EdgesPoly(*[, parameters, n_terms, ...])	Polynomial with an offset corresponding to approximate galaxy spectral index.
FixedLinearModel(*, model, x[, init_basis])	A base class for a linear model fixed at a certain set of co-ordinates.
Foreground(*[, parameters, n_terms, ...])	Base class for Foreground models.
Fourier(*[, parameters, n_terms, transform, ...])	A Fourier-basis model.
FourierDay(*[, parameters, n_terms, transform])	A Fourier-basis model with period of 24 (hours).
IdentityTransform()	
LinLog(*[, parameters, n_terms, with_cmb, ...])	
Log10Transform(*[, scale])	A transform that takes the logarithm of the input.
LogTransform(*[, scale])	A transform that takes the logarithm of the input.
Model(*[, parameters, n_terms, transform])	A base class for a linear model.
ModelFit(model, ydata[, weights])	A class representing a fit of model to data.
ModelTransform()	
NoiseWaves(*, freq, gamma_src, gamma_rec[, ...])	
PhysicalLin(*[, parameters, n_terms, ...])	Foreground model using a linearized physical model of the foregrounds.
Polynomial(*[, parameters, n_terms, ...])	A polynomial foreground model.
ScaleTransform(*, scale)	
ShiftTransform(*[, shift])	
UnitTransform(*, range)	A transform that takes the input range down to -1 to 1.
ZerotooneTransform(*, range)	A transform that takes an input range down to (0,1).

edges_cal.modelling.CentreTransform

class edges_cal.modelling.**CentreTransform**(*, *range*, *centre=0.0*)

Methods

__init__(*, range[, centre])	Method generated by attrs for class CentreTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.CentreTransform.__init__

CentreTransform.**__init__**(*, *range*, *centre=0.0*) → None

Method generated by attrs for class CentreTransform.

edges_cal.modelling.CentreTransform.get

classmethod CentreTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.CentreTransform.transform

CentreTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

range
centre

edges_cal.modelling.CentreTransform.range

CentreTransform.**range: tuple[float, float]**

edges_cal.modelling.CentreTransform.centre

CentreTransform.**centre: float**

edges_cal.modelling.ComplexMagPhaseModel

class edges_cal.modelling.**ComplexMagPhaseModel**(*mag:* Model | FixedLinearModel, *phs:* Model | FixedLinearModel)

A composite model that is specifically for complex functions in mag/phase.

Methods

__init__(mag, phs)	Method generated by attrs for class ComplexMagPhaseModel.
at(**kwargs)	Get an evaluated linear model.
fit(ydata[, weights, xdata])	Create a linear-regression fit object.
from_yaml(loader, node)	Convert a representation node to a Python object.
to_yaml(dumper, data)	Convert a Python object to a representation node.

edges_cal.modelling.ComplexMagPhaseModel.__init__

ComplexMagPhaseModel.**__init__**(*mag:* Model | FixedLinearModel, *phs:* Model | FixedLinearModel) → None

Method generated by attrs for class ComplexMagPhaseModel.

edges_cal.modelling.ComplexMagPhaseModel.at

ComplexMagPhaseModel.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.ComplexMagPhaseModel.fit

ComplexMagPhaseModel.**fit**(*ydata:* *ndarray*, *weights:* *ndarray* | *float* = *1.0*, *xdata:* *ndarray* | *None* = *None*)

Create a linear-regression fit object.

edges_cal.modelling.ComplexMagPhaseModel.from_yaml

classmethod ComplexMagPhaseModel.**from_yaml**(*loader*, *node*)

Convert a representation node to a Python object.

edges_cal.modelling.ComplexMagPhaseModel.to_yaml

classmethod ComplexMagPhaseModel.**to_yaml**(*dumper*, *data*)

Convert a Python object to a representation node.

Attributes

yaml_flow_style
yaml_loader
yaml_tag
mag
phs

edges_cal.modelling.ComplexMagPhaseModel.yaml_flow_style

ComplexMagPhaseModel.**yaml_flow_style = None**

edges_cal.modelling.ComplexMagPhaseModel.yaml_loader

ComplexMagPhaseModel.**yaml_loader = [<class 'yaml.loader.Loader'>, <class 'yaml.loader.FullLoader'>, <class 'yaml.loader.UnsafeLoader'>]**

edges_cal.modelling.ComplexMagPhaseModel.yaml_tag

ComplexMagPhaseModel.**yaml_tag = 'ComplexMagPhaseModel'**

edges_cal.modelling.ComplexMagPhaseModel.mag

ComplexMagPhaseModel.**mag**: *Model* | *FixedLinearModel*

edges_cal.modelling.ComplexMagPhaseModel.phs

ComplexMagPhaseModel.**phs**: *Model* | *FixedLinearModel*

edges_cal.modelling.ComplexRealImagModel

class edges_cal.modelling.**ComplexRealImagModel**(*real:* Model | FixedLinearModel, *imag:* Model | FixedLinearModel)

A composite model that is specifically for complex functions in real/imag.

Methods

__init__(real, imag)	Method generated by attrs for class ComplexRealImagModel.
at(**kwargs)	Get an evaluated linear model.
fit(ydata[, weights, xdata])	Create a linear-regression fit object.
from_yaml(loader, node)	Convert a representation node to a Python object.
to_yaml(dumper, data)	Convert a Python object to a representation node.

edges_cal.modelling.ComplexRealImagModel.__init__

ComplexRealImagModel.**__init__**(*real:* Model | FixedLinearModel, *imag:* Model | FixedLinearModel) → None

Method generated by attrs for class ComplexRealImagModel.

edges_cal.modelling.ComplexRealImagModel.at

ComplexRealImagModel.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.ComplexRealImagModel.fit

ComplexRealImagModel.**fit**(*ydata:* *ndarray*, *weights:* *ndarray* | *float* = *1.0*, *xdata:* *ndarray* | *None* = *None*)

Create a linear-regression fit object.

edges_cal.modelling.ComplexRealImagModel.from_yaml

classmethod ComplexRealImagModel.**from_yaml**(*loader*, *node*)

Convert a representation node to a Python object.

edges_cal.modelling.ComplexRealImagModel.to_yaml

classmethod ComplexRealImagModel.**to_yaml**(*dumper*, *data*)

Convert a Python object to a representation node.

Attributes

yaml_flow_style
yaml_loader
yaml_tag
real
imag

edges_cal.modelling.ComplexRealImagModel.yaml_flow_style

ComplexRealImagModel.**yaml_flow_style = None**

edges_cal.modelling.ComplexRealImagModel.yaml_loader

ComplexRealImagModel.**yaml_loader = [<class 'yaml.loader.Loader'>, <class 'yaml.loader.FullLoader'>, <class 'yaml.loader.UnsafeLoader'>]**

edges_cal.modelling.ComplexRealImagModel.yaml_tag

ComplexRealImagModel.**yaml_tag = 'ComplexRealImagModel'**

edges_cal.modelling.ComplexRealImagModel.real

ComplexRealImagModel.**real:** *Model* | *FixedLinearModel*

edges_cal.modelling.ComplexRealImagModel.imag

ComplexRealImagModel.**imag:** *Model* | *FixedLinearModel*

edges_cal.modelling.CompositeModel

class edges_cal.modelling.**CompositeModel**(*, *models: dict[str, edges_cal.modelling.Model]*, *extra_basis: dict[str, numpy.ndarray] = _Nothing.NOTHING*)

Methods

__init__(*, models[, extra_basis])	Method generated by attrs for class CompositeModel.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
get_basis_term(indx, x)	Define the basis terms for the model.
get_basis_term_transformed(indx, x)	Get the basis function term after coordinate tranformation.
get_basis_terms(x)	Get a 2D array of all basis terms at `x`.
get_extra_basis(model[, x])	Get the extra model-dependent basis function for a given model.
get_model(model[, parameters, x, with_extra])	Calculate a sub-model.
with_nterms(model[, n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get a new model with specified parameters.

edges_cal.modelling.CompositeModel.__init__

CompositeModel.**__init__**(*, *models: dict[str, edges_cal.modelling.Model]*, *extra_basis: dict[str, numpy.ndarray] = _Nothing.NOTHING*) → None

Method generated by attrs for class CompositeModel.

edges_cal.modelling.CompositeModel.at

CompositeModel.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.CompositeModel.fit

CompositeModel.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.CompositeModel.get_basis_term

CompositeModel.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis terms for the model.

edges_cal.modelling.CompositeModel.get_basis_term_transformed

CompositeModel.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis function term after coordinate tranformation.

edges_cal.modelling.CompositeModel.get_basis_terms

CompositeModel.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.CompositeModel.get_extra_basis

CompositeModel.**get_extra_basis**(*model: str*, *x: ndarray | None = None*)

Get the extra model-dependent basis function for a given model.

edges_cal.modelling.CompositeModel.get_model

CompositeModel.**get_model**(*model: str*, *parameters: ndarray = None*, *x: ndarray | None = None*, *with_extra: bool = False*)

Calculate a sub-model.

edges_cal.modelling.CompositeModel.with_nterms

CompositeModel.**with_nterms**(*model: str*, *n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.CompositeModel.with_params

CompositeModel.**with_params**(*parameters: Sequence*)

Get a new model with specified parameters.

Attributes

model_idx	Dictionary of parameter indices correponding to each model.
n_terms	The number of terms in the full composite model.
parameters	The read-only list of parameters of all sub-models.
models	
extra_basis	

edges_cal.modelling.CompositeModel.model_idx

CompositeModel.**model_idx**

Dictionary of parameter indices correponding to each model.

edges_cal.modelling.CompositeModel.n_terms

CompositeModel.**n_terms**

The number of terms in the full composite model.

edges_cal.modelling.CompositeModel.parameters

CompositeModel.**parameters**

The read-only list of parameters of all sub-models.

edges_cal.modelling.CompositeModel.models

CompositeModel.**models**: dict[str, *edges_cal.modelling.Model*]

edges_cal.modelling.CompositeModel.extra_basis

CompositeModel.**extra_basis**: dict[str, numpy.ndarray]

edges_cal.modelling.EdgesPoly

class edges_cal.modelling.**EdgesPoly**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *offset=-2.5*)

Polynomial with an offset corresponding to approximate galaxy spectral index.

Parameters

- **offset** (*float*) – The offset to use. Default is close to the Galactic spectral index.
- **kwargs** – All other arguments are passed through to *Polynomial*.

Methods

__init__(*[, parameters, n_terms, ...])	Method generated by attrs for class EdgesPoly.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis functions of the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at `x`.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.EdgesPoly.__init__

EdgesPoly.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *offset=-2.5*) → None

Method generated by attrs for class EdgesPoly.

edges_cal.modelling.EdgesPoly.at

EdgesPoly.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.EdgesPoly.fit

EdgesPoly.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.EdgesPoly.from_str

static EdgesPoly.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.EdgesPoly.get_basis_term

EdgesPoly.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.EdgesPoly.get_basis_term_transformed

EdgesPoly.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.EdgesPoly.get_basis_terms

EdgesPoly.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.EdgesPoly.with_nterms

EdgesPoly.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.EdgesPoly.with_params

EdgesPoly.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
offset
parameters
n_terms
transform

edges_cal.modelling.EdgesPoly.default_n_terms

EdgesPoly.**default_n_terms**: int | None = None

edges_cal.modelling.EdgesPoly.n_terms_max

EdgesPoly.**n_terms_max**: int = 1000000

edges_cal.modelling.EdgesPoly.n_terms_min

EdgesPoly.**n_terms_min**: int = 1

edges_cal.modelling.EdgesPoly.offset

EdgesPoly.**offset**: float

edges_cal.modelling.EdgesPoly.parameters

EdgesPoly.**parameters**: Sequence | None

edges_cal.modelling.EdgesPoly.n_terms

EdgesPoly.**n_terms**: int

edges_cal.modelling.EdgesPoly.transform

EdgesPoly.**transform**: *ModelTransform*

edges_cal.modelling.FixedLinearModel

class edges_cal.modelling.**FixedLinearModel**(*, *model:* Model, *x*, *init_basis=None*)

A base class for a linear model fixed at a certain set of co-ordinates.

Using this class caches the basis functions at the particular coordinates, and thus speeds up the fitting of multiple sets of data at those co-ordinates.

Parameters

- **model** (*edges_cal.modelling.Model*) – The linear model to evaluate at the co-ordinates
- **x** (*numpy.ndarray*) – A set of co-ordinates at which to evaluate the model.
- **init_basis** – If the basis functions of the model, evaluated at x, are known already, they can be input directly to save computation time.

Methods

__init__(*, model, x[, init_basis])	Method generated by attrs for class FixedLinearModel.
at_x(x)	Return a new *FixedLinearModel* at given coordinates.
fit(ydata[, weights, xdata])	Create a linear-regression fit object.
from_yaml(loader, node)	Convert a representation node to a Python object.
to_yaml(dumper, data)	Method to convert to YAML format.
with_nterms(n_terms[, parameters])	Return a new *FixedLinearModel* with given nterms and parameters.
with_params(parameters)	Return a new *FixedLinearModel* with givne parameters.

edges_cal.modelling.FixedLinearModel.__init__

FixedLinearModel.**__init__**(*, *model:* Model, *x*, *init_basis=None*) → None

Method generated by attrs for class FixedLinearModel.

edges_cal.modelling.FixedLinearModel.at_x

FixedLinearModel.**at_x**(*x: ndarray*) → *FixedLinearModel*

Return a new *FixedLinearModel* at given co-ordinates.

edges_cal.modelling.FixedLinearModel.fit

FixedLinearModel.**fit**(*ydata: ndarray*, *weights: ndarray | float = 1.0*, *xdata: ndarray | None = None*)

Create a linear-regression fit object.

edges_cal.modelling.FixedLinearModel.from_yaml

classmethod FixedLinearModel.**from_yaml**(*loader*, *node*)

Convert a representation node to a Python object.

edges_cal.modelling.FixedLinearModel.to_yaml

classmethod FixedLinearModel.**to_yaml**(*dumper*, *data*)

Method to convert to YAML format.

edges_cal.modelling.FixedLinearModel.with_nterms

FixedLinearModel.**with_nterms**(*n_terms: int*, *parameters: Sequence | None = None*) → *FixedLinearModel*

Return a new *FixedLinearModel* with given nterms and parameters.

edges_cal.modelling.FixedLinearModel.with_params

FixedLinearModel.**with_params**(*parameters: Sequence*) → *FixedLinearModel*

Return a new *FixedLinearModel* with givne parameters.

Attributes

basis	The (cached) basis functions at default_x.
n_terms	The number of terms/parameters in the model.
parameters	The parameters of the model, if set.
yaml_flow_style	
yaml_loader	
yaml_tag	
model	
x	

edges_cal.modelling.FixedLinearModel.basis

FixedLinearModel.**basis**

The (cached) basis functions at default_x.

Shape (n_terms, x).

edges_cal.modelling.FixedLinearModel.n_terms

property FixedLinearModel.**n_terms**

The number of terms/parameters in the model.

edges_cal.modelling.FixedLinearModel.parameters

property FixedLinearModel.**parameters: ndarray | None**

The parameters of the model, if set.

edges_cal.modelling.FixedLinearModel.yaml_flow_style

FixedLinearModel.**yaml_flow_style = None**

edges_cal.modelling.FixedLinearModel.yaml_loader

FixedLinearModel.**yaml_loader = [<class 'yaml.loader.Loader'>, <class 'yaml.loader.FullLoader'>, <class 'yaml.loader.UnsafeLoader'>]**

edges_cal.modelling.FixedLinearModel.yaml_tag

FixedLinearModel.**yaml_tag = '!Model'**

edges_cal.modelling.FixedLinearModel.model

FixedLinearModel.**model:** *Model*

edges_cal.modelling.FixedLinearModel.x

FixedLinearModel.**x:** ndarray

edges_cal.modelling.Foreground

class edges_cal.modelling.**Foreground**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*)

Base class for Foreground models.

Parameters

- **f_center** (*float*) – A "center" or "reference" frequency. Typically models will have their co-ordindates divided by this frequency before solving for the co-efficients.
- **with_cmb** (*bool*) – Whether to add a simple CMB component to the foreground.

Methods

__init__(*[, parameters, n_terms, with_cmb, ...])	Method generated by attrs for class Foreground.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis terms for the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at **x**.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.Foreground.__init__

Foreground.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*) → None

Method generated by attrs for class Foreground.

edges_cal.modelling.Foreground.at

Foreground.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.Foreground.fit

Foreground.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.Foreground.from_str

static Foreground.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.Foreground.get_basis_term

abstract Foreground.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis terms for the model.

edges_cal.modelling.Foreground.get_basis_term_transformed

Foreground.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.Foreground.get_basis_terms

Foreground.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.Foreground.with_nterms

Foreground.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.Foreground.with_params

Foreground.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
with_cmb
f_center
transform
parameters
n_terms

edges_cal.modelling.Foreground.default_n_terms

Foreground.**default_n_terms**: int | None = None

edges_cal.modelling.Foreground.n_terms_max

Foreground.**n_terms_max**: int = 1000000

edges_cal.modelling.Foreground.n_terms_min

Foreground.**n_terms_min**: int = 1

edges_cal.modelling.Foreground.with_cmb

Foreground.**with_cmb**: bool

edges_cal.modelling.Foreground.f_center

Foreground.**f_center**: float

edges_cal.modelling.Foreground.transform

Foreground.**transform**: *ModelTransform*

edges_cal.modelling.Foreground.parameters

Foreground.**parameters**: Sequence | None

edges_cal.modelling.Foreground.n_terms

Foreground.**n_terms**: int

edges_cal.modelling.Fourier

class edges_cal.modelling.**Fourier**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *period=6.283185307179586*)

A Fourier-basis model.

Methods

__init__(*[, parameters, n_terms, ...])	Method generated by attrs for class Fourier.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis functions of the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at **x**.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.Fourier.__init__

Fourier.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *period=6.283185307179586*) → None

Method generated by attrs for class Fourier.

edges_cal.modelling.Fourier.at

Fourier.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.Fourier.fit

Fourier.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.Fourier.from_str

static Fourier.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.Fourier.get_basis_term

Fourier.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.Fourier.get_basis_term_transformed

Fourier.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.Fourier.get_basis_terms

Fourier.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at **x**.

edges_cal.modelling.Fourier.with_nterms

Fourier.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.Fourier.with_params

Fourier.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
period
parameters
n_terms
transform

edges_cal.modelling.Fourier.default_n_terms

Fourier.**default_n_terms**: int | None = None

edges_cal.modelling.Fourier.n_terms_max

Fourier.**n_terms_max**: int = 1000000

edges_cal.modelling.Fourier.n_terms_min

Fourier.**n_terms_min**: int = 1

edges_cal.modelling.Fourier.period

Fourier.**period**: float

edges_cal.modelling.Fourier.parameters

Fourier.**parameters**: Sequence | None

edges_cal.modelling.Fourier.n_terms

Fourier.**n_terms**: int

edges_cal.modelling.Fourier.transform

Fourier.**transform**: *ModelTransform*

edges_cal.modelling.FourierDay

class edges_cal.modelling.**FourierDay**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*)

A Fourier-basis model with period of 24 (hours).

Methods

__init__(*[, parameters, n_terms, transform])	Method generated by attrs for class FourierDay.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis functions of the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at **x**.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.FourierDay.__init__

FourierDay.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*) → None

Method generated by attrs for class FourierDay.

edges_cal.modelling.FourierDay.at

FourierDay.**at**(**kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.FourierDay.fit

FourierDay.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.FourierDay.from_str

static FourierDay.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a `Model` given a string name.

edges_cal.modelling.FourierDay.get_basis_term

FourierDay.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.FourierDay.get_basis_term_transformed

FourierDay.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.FourierDay.get_basis_terms

FourierDay.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.FourierDay.with_nterms

FourierDay.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new `Model` with given nterms and parameters.

edges_cal.modelling.FourierDay.with_params

FourierDay.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
parameters
n_terms
transform

edges_cal.modelling.FourierDay.default_n_terms

FourierDay.**default_n_terms**: int | None = None

edges_cal.modelling.FourierDay.n_terms_max

FourierDay.**n_terms_max**: int = 1000000

edges_cal.modelling.FourierDay.n_terms_min

FourierDay.**n_terms_min**: int = 1

edges_cal.modelling.FourierDay.parameters

FourierDay.**parameters**: Sequence | None

edges_cal.modelling.FourierDay.n_terms

FourierDay.**n_terms**: int

edges_cal.modelling.FourierDay.transform

FourierDay.**transform**: *ModelTransform*

edges_cal.modelling.IdentityTransform

class edges_cal.modelling.**IdentityTransform**

Methods

__init__()	Method generated by attrs for class IdentityTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.IdentityTransform.__init__

IdentityTransform.**__init__**() → None

Method generated by attrs for class IdentityTransform.

edges_cal.modelling.IdentityTransform.get

classmethod IdentityTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.IdentityTransform.transform

IdentityTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

edges_cal.modelling.LinLog

class edges_cal.modelling.**LinLog**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*, *beta=-2.5*)

Methods

__init__(*[, parameters, n_terms, with_cmb, ...])	Method generated by attrs for class LinLog.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis functions of the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at `x`.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.LinLog.__init__

LinLog.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*, *beta=-2.5*) → None

Method generated by attrs for class LinLog.

edges_cal.modelling.LinLog.at

LinLog.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.LinLog.fit

LinLog.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.LinLog.from_str

static LinLog.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.LinLog.get_basis_term

LinLog.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.LinLog.get_basis_term_transformed

LinLog.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.LinLog.get_basis_terms

LinLog.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.LinLog.with_nterms

LinLog.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.LinLog.with_params

LinLog.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

beta
default_n_terms
n_terms_max
n_terms_min
with_cmb
f_center
transform
parameters
n_terms

edges_cal.modelling.LinLog.beta

LinLog.**beta**: float

edges_cal.modelling.LinLog.default_n_terms

LinLog.**default_n_terms**: int | None = None

edges_cal.modelling.LinLog.n_terms_max

LinLog.**n_terms_max**: int = 1000000

edges_cal.modelling.LinLog.n_terms_min

LinLog.**n_terms_min**: int = 1

edges_cal.modelling.LinLog.with_cmb

LinLog.**with_cmb**: bool

edges_cal.modelling.LinLog.f_center

LinLog.**f_center**: float

edges_cal.modelling.LinLog.transform

LinLog.**transform**: *ModelTransform*

edges_cal.modelling.LinLog.parameters

LinLog.**parameters**: Sequence | None

edges_cal.modelling.LinLog.n_terms

LinLog.**n_terms**: int

edges_cal.modelling.Log10Transform

class edges_cal.modelling.**Log10Transform**(*, *scale: float = 1.0*)

A transform that takes the logarithm of the input.

Methods

__init__(*[, scale])	Method generated by attrs for class Log10Transform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.Log10Transform.__init__

Log10Transform.**__init__**(*, *scale: float = 1.0*) → None

Method generated by attrs for class Log10Transform.

edges_cal.modelling.Log10Transform.get

classmethod Log10Transform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.Log10Transform.transform

Log10Transform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

scale

edges_cal.modelling.Log10Transform.scale

Log10Transform.**scale: float**

edges_cal.modelling.LogTransform

class edges_cal.modelling.**LogTransform**(*, *scale: float = 1.0*)

A transform that takes the logarithm of the input.

Methods

__init__(*[, scale])	Method generated by attrs for class LogTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.LogTransform.__init__

LogTransform.**__init__**(*, *scale: float = 1.0*) → None

Method generated by attrs for class LogTransform.

edges_cal.modelling.LogTransform.get

classmethod LogTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.LogTransform.transform

LogTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

scale

edges_cal.modelling.LogTransform.scale

LogTransform.**scale: float**

edges_cal.modelling.Model

class edges_cal.modelling.**Model**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform: ModelTransform = IdentityTransform()*)

A base class for a linear model.

Methods

__init__(*[, parameters, n_terms, transform])	Method generated by attrs for class Model.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis terms for the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at `x`.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.Model.__init__

Model.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*) → None

Method generated by attrs for class Model.

edges_cal.modelling.Model.at

Model.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.Model.fit

Model.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.Model.from_str

static Model.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.Model.get_basis_term

abstract Model.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis terms for the model.

edges_cal.modelling.Model.get_basis_term_transformed

Model.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.Model.get_basis_terms

Model.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at **x**.

edges_cal.modelling.Model.with_nterms

Model.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.Model.with_params

Model.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
parameters
n_terms
transform

edges_cal.modelling.Model.default_n_terms

Model.**default_n_terms**: int | None = None

edges_cal.modelling.Model.n_terms_max

Model.**n_terms_max**: int = 1000000

edges_cal.modelling.Model.n_terms_min

Model.**n_terms_min**: int = 1

edges_cal.modelling.Model.parameters

Model.**parameters**: Sequence | None

edges_cal.modelling.Model.n_terms

Model.**n_terms**: int

edges_cal.modelling.Model.transform

Model.**transform**: *ModelTransform*

edges_cal.modelling.ModelFit

class edges_cal.modelling.**ModelFit**(*model:* FixedLinearModel, *ydata:* *ndarray*, *weights:* *ndarray* | *float* = *1.0*)

A class representing a fit of model to data.

Parameters

- **model** (*edges_cal.modelling.FixedLinearModel*) – The evaluatable model to fit to the data.
- **ydata** (*numpy.ndarray*) – The values of the measured data.
- **weights** (*numpy.ndarray* | *float*) – The weight of the measured data at each point. This corresponds to the *variance* of the measurement (not the standard deviation). This is appropriate if the weights represent the number of measurements going into each piece of data.

Raises

ValueError – If model_type is not str, or a subclass of *Model*.

Methods

__init__(model, ydata[, weights])	Method generated by attrs for class ModelFit.
evaluate([x])	Evaluate the best-fit model.
get_sample([size])	Generate a random sample from the posterior distribution.
reduced_weighted_chi2()	The weighted chi^2 divided by the degrees of freedom.
weighted_rms()	The weighted root-mean-square of the residuals.

edges_cal.modelling.ModelFit.__init__

ModelFit.**__init__**(*model:* FixedLinearModel, *ydata:* *ndarray*, *weights:* *ndarray* | *float* = *1.0*) → None

Method generated by attrs for class ModelFit.

edges_cal.modelling.ModelFit.evaluate

ModelFit.**evaluate**(*x:* *ndarray* | *None* = *None*) → ndarray

Evaluate the best-fit model.

Parameters

x (*np.ndarray, optional*) – The co-ordinates at which to evaluate the model. By default, use the input data co-ordinates.

Returns

y (*np.ndarray*) – The best-fit model evaluated at **x**.

edges_cal.modelling.ModelFit.get_sample

ModelFit.**get_sample**(*size: int | tuple[int] = 1*)

Generate a random sample from the posterior distribution.

edges_cal.modelling.ModelFit.reduced_weighted_chi2

ModelFit.**reduced_weighted_chi2**() → float

The weighted chi^2 divided by the degrees of freedom.

edges_cal.modelling.ModelFit.weighted_rms

ModelFit.**weighted_rms**() → float

The weighted root-mean-square of the residuals.

Attributes

degrees_of_freedom	The number of degrees of freedom of the fit.
fit	A model that has parameters set based on the best fit to this data.
hessian	The Hessian matrix of the linear parameters.
model_parameters	The best-fit model parameters.
parameter_covariance	The Covariance matrix of the parameters.
residual	Residuals of data to model.
weighted_chi2	The chi^2 of the weighted fit.
model	
ydata	
weights	

edges_cal.modelling.ModelFit.degrees_of_freedom

ModelFit.**degrees_of_freedom**

The number of degrees of freedom of the fit.

edges_cal.modelling.ModelFit.fit

ModelFit.**fit**

A model that has parameters set based on the best fit to this data.

edges_cal.modelling.ModelFit.hessian

ModelFit.**hessian**
: The Hessian matrix of the linear parameters.

edges_cal.modelling.ModelFit.model_parameters

ModelFit.**model_parameters**
: The best-fit model parameters.

edges_cal.modelling.ModelFit.parameter_covariance

ModelFit.**parameter_covariance**
: The Covariance matrix of the parameters.

edges_cal.modelling.ModelFit.residual

ModelFit.**residual**
: Residuals of data to model.

edges_cal.modelling.ModelFit.weighted_chi2

ModelFit.**weighted_chi2**
: The chi^2 of the weighted fit.

edges_cal.modelling.ModelFit.model

ModelFit.**model**: *FixedLinearModel*

edges_cal.modelling.ModelFit.ydata

ModelFit.**ydata**: ndarray

edges_cal.modelling.ModelFit.weights

ModelFit.**weights**: ndarray | float

edges_cal.modelling.ModelTransform

class edges_cal.modelling.**ModelTransform**

Methods

__init__()	Method generated by attrs for class ModelTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.ModelTransform.__init__

ModelTransform.**__init__**() → None

Method generated by attrs for class ModelTransform.

edges_cal.modelling.ModelTransform.get

classmethod ModelTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.ModelTransform.transform

abstract ModelTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

edges_cal.modelling.NoiseWaves

class edges_cal.modelling.**NoiseWaves**(**, freq: ndarray, gamma_src: dict[str, numpy.ndarray], gamma_rec: ndarray, c_terms: int = 5, w_terms: int = 6, parameters: Sequence | None = None, with_tload: bool = True*)

Methods

__init__(*, freq, gamma_src, gamma_rec[, ...])	Method generated by attrs for class NoiseWaves.
from_calobs(calobs[, cterms, wterms, ...])	Initialize a noise wave model from a calibration observation.
get_data_from_calobs(calobs[, tns, sim, loads])	Generate input data to fit from a calibration observation.
get_fitted(data[, weights])	Get a new noise wave model with fitted parameters.
get_full_model(src[, parameters])	Get the full model (all noise-waves) for a particular input source.
get_linear_model([with_k])	Define and return a Model.
get_noise_wave(noise_wave[, parameters, src])	Get the model for a particular noise-wave term.
with_params_from_calobs(calobs[, cterms, wterms])	Get a new noise wave model with parameters fitted using standard methods.

edges_cal.modelling.NoiseWaves.__init__

NoiseWaves.**__init__**(*, *freq: ndarray*, *gamma_src: dict[str, numpy.ndarray]*, *gamma_rec: ndarray*, *c_terms: int = 5*, *w_terms: int = 6*, *parameters: Sequence | None = None*, *with_tload: bool = True*) → None

Method generated by attrs for class NoiseWaves.

edges_cal.modelling.NoiseWaves.from_calobs

classmethod NoiseWaves.**from_calobs**(*calobs*, *cterms=None*, *wterms=None*, *sources=None*, *with_tload: bool = True*, *loads: dict | None = None*) → *NoiseWaves*

Initialize a noise wave model from a calibration observation.

edges_cal.modelling.NoiseWaves.get_data_from_calobs

NoiseWaves.**get_data_from_calobs**(*calobs*, *tns: Model | None = None*, *sim: bool = False*, *loads: dict | None = None*) → ndarray

Generate input data to fit from a calibration observation.

edges_cal.modelling.NoiseWaves.get_fitted

NoiseWaves.**get_fitted**(*data: ndarray*, *weights: ndarray | None = None*) → *NoiseWaves*

Get a new noise wave model with fitted parameters.

edges_cal.modelling.NoiseWaves.get_full_model

NoiseWaves.**get_full_model**(*src: str*, *parameters: Sequence | None = None*) → ndarray

Get the full model (all noise-waves) for a particular input source.

edges_cal.modelling.NoiseWaves.get_linear_model

NoiseWaves.**get_linear_model**(*with_k: bool = True*) → *CompositeModel*

Define and return a Model.

Parameters

with_k – Whether to use the K matrix as an "extra basis" in the linear model.

edges_cal.modelling.NoiseWaves.get_noise_wave

NoiseWaves.**get_noise_wave**(*noise_wave: str*, *parameters: Sequence | None = None*, *src: str | None = None*) → ndarray

Get the model for a particular noise-wave term.

edges_cal.modelling.NoiseWaves.with_params_from_calobs

NoiseWaves.**with_params_from_calobs**(*calobs*, *cterms=None*, *wterms=None*) → *NoiseWaves*

Get a new noise wave model with parameters fitted using standard methods.

Attributes

linear_model	The actual composite linear model object associated with the noise waves.
src_names	List of names of inputs sources (eg.
freq	
gamma_src	
gamma_rec	
c_terms	
w_terms	
parameters	
with_tload	

edges_cal.modelling.NoiseWaves.linear_model

NoiseWaves.**linear_model**

The actual composite linear model object associated with the noise waves.

edges_cal.modelling.NoiseWaves.src_names

NoiseWaves.**src_names**

List of names of inputs sources (eg. ambient, hot_load, open, short).

edges_cal.modelling.NoiseWaves.freq

NoiseWaves.**freq**: ndarray

edges_cal.modelling.NoiseWaves.gamma_src

NoiseWaves.**gamma_src**: dict[str, numpy.ndarray]

edges_cal.modelling.NoiseWaves.gamma_rec

NoiseWaves.**gamma_rec**: ndarray

edges_cal.modelling.NoiseWaves.c_terms

NoiseWaves.**c_terms**: int

edges_cal.modelling.NoiseWaves.w_terms

NoiseWaves.**w_terms**: int

edges_cal.modelling.NoiseWaves.parameters

NoiseWaves.**parameters**: Sequence | None

edges_cal.modelling.NoiseWaves.with_tload

NoiseWaves.**with_tload**: bool

edges_cal.modelling.PhysicalLin

class edges_cal.modelling.**PhysicalLin**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*)

Foreground model using a linearized physical model of the foregrounds.

Methods

__init__(*[, parameters, n_terms, with_cmb, ...])	Method generated by attrs for class PhysicalLin.
at(**kwargs)	Get an evaluated linear model.
fit(xdata, ydata[, weights])	Create a linear-regression fit object.
from_str(model, **kwargs)	Obtain a *Model* given a string name.
get_basis_term(indx, x)	Define the basis functions of the model.
get_basis_term_transformed(indx, x)	Get the basis term after coordinate transformation.
get_basis_terms(x)	Get a 2D array of all basis terms at `x`.
with_nterms([n_terms, parameters])	Return a new *Model* with given nterms and parameters.
with_params(parameters)	Get new model with different parameters.

edges_cal.modelling.PhysicalLin.__init__

PhysicalLin.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *with_cmb=False*, *f_center=75.0*, *transform:* ModelTransform = *_Nothing.NOTHING*) → None

Method generated by attrs for class PhysicalLin.

edges_cal.modelling.PhysicalLin.at

PhysicalLin.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.PhysicalLin.fit

PhysicalLin.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.PhysicalLin.from_str

static PhysicalLin.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a *Model* given a string name.

edges_cal.modelling.PhysicalLin.get_basis_term

PhysicalLin.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.PhysicalLin.get_basis_term_transformed

PhysicalLin.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.PhysicalLin.get_basis_terms

PhysicalLin.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.PhysicalLin.with_nterms

PhysicalLin.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.PhysicalLin.with_params

PhysicalLin.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
with_cmb
f_center
transform
parameters
n_terms

edges_cal.modelling.PhysicalLin.default_n_terms

PhysicalLin.**default_n_terms**: int = 5

edges_cal.modelling.PhysicalLin.n_terms_max

PhysicalLin.**n_terms_max**: int = 5

edges_cal.modelling.PhysicalLin.n_terms_min

PhysicalLin.**n_terms_min**: int = 1

edges_cal.modelling.PhysicalLin.with_cmb

PhysicalLin.**with_cmb**: bool

edges_cal.modelling.PhysicalLin.f_center

PhysicalLin.**f_center**: float

edges_cal.modelling.PhysicalLin.transform

PhysicalLin.**transform**: *ModelTransform*

edges_cal.modelling.PhysicalLin.parameters

PhysicalLin.**parameters**: Sequence | None

edges_cal.modelling.PhysicalLin.n_terms

PhysicalLin.**n_terms**: int

edges_cal.modelling.Polynomial

class edges_cal.modelling.**Polynomial**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *offset=0*)

A polynomial foreground model.

Parameters

- **log_x** (*bool*) – Whether to fit the poly coefficients with log-space co-ordinates.
- **offset** (*float*) – An offset to use for each index in the polynomial model.
- **kwargs** – All other arguments passed through to *Foreground*.

Notes

The polynomial model can be written

$$\sum_{i=0}^{n} c_i y^{i+offset},$$

where `y` is `log(x)` if `log_x=True` and simply `x` otherwise.

Methods

`__init__`(*[, parameters, n_terms, ...])	Method generated by attrs for class Polynomial.
`at`(**kwargs)	Get an evaluated linear model.
`fit`(xdata, ydata[, weights])	Create a linear-regression fit object.
`from_str`(model, **kwargs)	Obtain a `Model` given a string name.
`get_basis_term`(indx, x)	Define the basis functions of the model.
`get_basis_term_transformed`(indx, x)	Get the basis term after coordinate transformation.
`get_basis_terms`(x)	Get a 2D array of all basis terms at `x`.
`with_nterms`([n_terms, parameters])	Return a new `Model` with given nterms and parameters.
`with_params`(parameters)	Get new model with different parameters.

edges_cal.modelling.Polynomial.__init__

Polynomial.**__init__**(*, *parameters=None*, *n_terms=_Nothing.NOTHING*, *transform:* ModelTransform = *IdentityTransform()*, *offset=0*) → None

Method generated by attrs for class Polynomial.

edges_cal.modelling.Polynomial.at

Polynomial.**at**(***kwargs*) → *FixedLinearModel*

Get an evaluated linear model.

edges_cal.modelling.Polynomial.fit

Polynomial.**fit**(*xdata: ndarray*, *ydata: ndarray*, *weights: ndarray | float = 1.0*) → *ModelFit*

Create a linear-regression fit object.

edges_cal.modelling.Polynomial.from_str

static Polynomial.**from_str**(*model: str*, ***kwargs*) → *Model*

Obtain a `Model` given a string name.

edges_cal.modelling.Polynomial.get_basis_term

Polynomial.**get_basis_term**(*indx: int*, *x: ndarray*) → ndarray

Define the basis functions of the model.

edges_cal.modelling.Polynomial.get_basis_term_transformed

Polynomial.**get_basis_term_transformed**(*indx: int*, *x: ndarray*) → ndarray

Get the basis term after coordinate transformation.

edges_cal.modelling.Polynomial.get_basis_terms

Polynomial.**get_basis_terms**(*x: ndarray*) → ndarray

Get a 2D array of all basis terms at `x`.

edges_cal.modelling.Polynomial.with_nterms

Polynomial.**with_nterms**(*n_terms: int | None = None*, *parameters: Sequence | None = None*) → *Model*

Return a new *Model* with given nterms and parameters.

edges_cal.modelling.Polynomial.with_params

Polynomial.**with_params**(*parameters: Sequence | None*)

Get new model with different parameters.

Attributes

default_n_terms
n_terms_max
n_terms_min
offset
parameters
n_terms
transform

edges_cal.modelling.Polynomial.default_n_terms

Polynomial.**default_n_terms: int | None = None**

edges_cal.modelling.Polynomial.n_terms_max

Polynomial.**n_terms_max**: int = 1000000

edges_cal.modelling.Polynomial.n_terms_min

Polynomial.**n_terms_min**: int = 1

edges_cal.modelling.Polynomial.offset

Polynomial.**offset**: float

edges_cal.modelling.Polynomial.parameters

Polynomial.**parameters**: Sequence | None

edges_cal.modelling.Polynomial.n_terms

Polynomial.**n_terms**: int

edges_cal.modelling.Polynomial.transform

Polynomial.**transform**: *ModelTransform*

edges_cal.modelling.ScaleTransform

class edges_cal.modelling.**ScaleTransform**(*, *scale*)

Methods

__init__(*, scale)	Method generated by attrs for class ScaleTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.ScaleTransform.__init__

ScaleTransform.**__init__**(*, *scale*) → None

Method generated by attrs for class ScaleTransform.

edges_cal.modelling.ScaleTransform.get

classmethod ScaleTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.ScaleTransform.transform

ScaleTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

scale

edges_cal.modelling.ScaleTransform.scale

ScaleTransform.**scale: float**

edges_cal.modelling.ShiftTransform

class edges_cal.modelling.**ShiftTransform**(*, *shift=0.0*)

Methods

__init__(*[, shift])	Method generated by attrs for class ShiftTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.ShiftTransform.__init__

ShiftTransform.**__init__**(*, *shift=0.0*) → None

Method generated by attrs for class ShiftTransform.

edges_cal.modelling.ShiftTransform.get

classmethod ShiftTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.ShiftTransform.transform

ShiftTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

shift

edges_cal.modelling.ShiftTransform.shift

ShiftTransform.**shift: float**

edges_cal.modelling.UnitTransform

class edges_cal.modelling.**UnitTransform**(*, *range*)

A transform that takes the input range down to -1 to 1.

Methods

__init__(*, range)	Method generated by attrs for class UnitTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.UnitTransform.__init__

UnitTransform.**__init__**(*, *range*) → None

Method generated by attrs for class UnitTransform.

edges_cal.modelling.UnitTransform.get

classmethod UnitTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.UnitTransform.transform

UnitTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

range

edges_cal.modelling.UnitTransform.range

UnitTransform.**range: tuple[float, float]**

edges_cal.modelling.ZerotooneTransform

class edges_cal.modelling.**ZerotooneTransform**(*, *range*)

A transform that takes an input range down to (0,1).

Methods

__init__(*, range)	Method generated by attrs for class ZerotooneTransform.
get(model)	Get a ModelTransform class.
transform(x)	Transform the coordinates.

edges_cal.modelling.ZerotooneTransform.__init__

ZerotooneTransform.**__init__**(*, *range*) → None

Method generated by attrs for class ZerotooneTransform.

edges_cal.modelling.ZerotooneTransform.get

classmethod ZerotooneTransform.**get**(*model: str*) → type[*edges_cal.modelling.ModelTransform*]

Get a ModelTransform class.

edges_cal.modelling.ZerotooneTransform.transform

ZerotooneTransform.**transform**(*x: ndarray*) → ndarray

Transform the coordinates.

Attributes

range

edges_cal.modelling.ZerotooneTransform.range

ZerotooneTransform.**range: tuple[float, float]**

edges_cal.xrfi

Functions for excising RFI.

Functions

detrend_meanfilt(data[, flags, half_size])	Detrend array using a mean filter.
detrend_medfilt(data[, flags, half_size])	Detrend array using a median filter.
flagged_filter(data, size[, kind, flags, ...])	Perform an n-dimensional filter operation on optionally flagged data.
model_filter(x, data, *[, model, ...])	Flag data by subtracting a smooth model and iteratively removing outliers.
robust_divide(num, den)	Prevent division by zero.
visualise_model_info(info[, n])	Make a nice visualisation of the info output from *xrfi_model()*.
xrfi_explicit([spectrum, flags, rfi_file, ...])	Excise RFI from given data using an explicitly set list of flag ranges.
xrfi_medfilt(spectrum[, threshold, flags, ...])	Generate RFI flags for a given spectrum using a median filter.
xrfi_model(spectrum, *, freq[, inplace, ...])	Flag RFI by subtracting a smooth model and iteratively removing outliers.
xrfi_model_sweep(spectrum, *[, freq, flags, ...])	Flag RFI by using a moving window and a low-order polynomial to detrend.
xrfi_watershed([spectrum, freq, flags, ...])	Apply a watershed over frequencies and times for flags.

edges_cal.xrfi.detrend_meanfilt

edges_cal.xrfi.**detrend_meanfilt**(*data: ndarray*, *flags: ndarray | None = None*, *half_size: tuple[int | None] | None = None*)

Detrend array using a mean filter.

Parameters

- **data** (*array*) – Data to detrend. Can be an array of any number of dimensions.
- **flags** (*boolean array, optional*) – Flags specifying data to ignore in the detrend. If not given, don't ignore anything.
- **half_size** (*tuple of int/None*) – The half-size of the kernel to convolve (kernel size will be 2*half_size+1). Value of zero (for any dimension) omits that axis from the kernel, effectively applying the detrending for each subarray along that axis. Value of None will effectively (but slowly) perform a median along the entire axis before running the kernel over the other axis.

Returns

out (*array*) – An array containing the outlier significance metric. Same type and size as *data*.

Notes

This detrending is very good for data that has most of the RFI flagged already, but will perform very poorly when un-flagged RFI still exists. It is often useful to precede this with a median filter.

edges_cal.xrfi.detrend_medfilt

edges_cal.xrfi.**detrend_medfilt**(*data: ndarray*, *flags: ndarray | None = None*, *half_size: tuple[int | None] | None = None*)

Detrend array using a median filter.

Note: ripped from here: https://github.com/HERA-Team/hera_qm/blob/master/hera_qm/xrfi.py

---

Parameters

- **data** (*array*) – Data to detrend. Can be an array of any number of dimensions.
- **flags** (*boolean array, optional*) – Flags specifying data to ignore in the detrend. If not given, don't ignore anything.
- **half_size** (*tuple of int/None*) – The half-size of the kernel to convolve (kernel size will be 2*half_size+1). Value of zero (for any dimension) omits that axis from the kernel, effectively applying the detrending for each subarray along that axis. Value of None will effectively (but slowly) perform a median along the entire axis before running the kernel over the other axis.

Returns

out (*array*) – An array containing the outlier significance metric. Same type and size as *data*.

Notes

This detrending is very good for data with large RFI compared to the noise, but also reasonably large noise compared to the spectrum steepness. If the noise is small compared to the steepness of the spectrum, individual windows can become *almost always* monotonic, in which case the randomly non-monotonic bins "stick out" and get wrongly flagged. This can be helped three ways:

1) Use a smaller bin width. This helps by reducing the probability that a bin will be randomly non-monotonic. However it also loses signal-to-noise on the RFI.

2) Pre-fit a smooth model that "flattens" the spectrum. This helps by reducing the probability that bins will be monotonic (higher noise level wrt steepness). It has the disadvantage that fitted models can be wrong when there's RFI there.

3) Follow the medfilt with a meanfilt: if the medfilt is able to flag most/all of the RFI, then a following meanfilt will tend to "unfilter" the wrongly flagged parts.

edges_cal.xrfi.flagged_filter

edges_cal.xrfi.**flagged_filter**(*data: ndarray*, *size: int | tuple[int]*, *kind: str = 'median'*, *flags: ndarray | None = None*, *mode: str | None = None*, *interp_flagged: bool = True*, ***kwargs*)

Perform an n-dimensional filter operation on optionally flagged data.

Parameters

- **data** (*np.ndarray*) – The data to filter. Can be of arbitrary dimension.
- **size** (*int or tuple*) – The size of the filtering convolution kernel. If tuple, one entry per dimension in *data*.
- **kind** (*str, optional*) – The function to apply in each window. Typical options are *mean* and *median*. For this function to work, the function kind chosen here must have a corresponding *nan<function>* implementation in numpy.
- **flags** (*np.ndarray, optional*) – A boolean array specifying data to omit from the filtering.
- **mode** (*str, optional*) – The mode of the filter. See `scipy.ndimage.generic_filter` for details. By default, 'nearest' if size < data.size otherwise 'reflect'.
- **interp_flagged** (*bool, optional*) – Whether to fill in flagged entries with its filtered value. Otherwise, flagged entries are set to their original value.
- **kwargs** – Other options to pass to the generic filter function.

Returns

np.ndarray – The filtered array, of the same shape and type as `data`.

Notes

This function can typically be used to implement a flagged median filter. It does have some limitations in this regard, which we will now describe.

It would be expected that a perfectly smooth monotonic function, after median filtering, should remain identical to the input. This is only the case for the default 'nearest' mode. For the alternative 'reflect' mode, the edge-data will be corrupted from the input. On the other hand, it may be expected that if the kernel width is equal to or larger than the data size, that the operation is merely to perform a full collapse over that dimension. This is the

case only for mode 'reflect', while again mode 'nearest' will continue to yield (a very slow) identity operation. By default, the mode will be set to 'reflect' if the size is >= the data size, with an emitted warning.

Furthermore, a median filter is *not* an identity operation, even on monotonic functions, for an even-sized kernel (in this case it's the average of the two central values).

Also, even for an odd-sized kernel, if using flags, some of the windows will contain an odd number of useable data, in which case the data surrounding the flag will not be identical to the input.

Finally, flags near the edges can have strange behaviour, depending on the mode.

edges_cal.xrfi.model_filter

edges_cal.xrfi.**model_filter**(*x: ndarray*, *data: ndarray*, ***, *model:* Model = *Polynomial(parameters=None, n_terms=3, transform=IdentityTransform(), offset=0.0)*, *resid_model:* Model = *Polynomial(parameters=None, n_terms=5, transform=IdentityTransform(), offset=0.0)*, *flags: ndarray | None = None*, *weights: ndarray | None = None*, *n_resid: int = -1*, *threshold: float | None = None*, *max_iter: int = 20*, *increase_order: bool = True*, *min_terms: int = 0*, *max_terms: int = 10*, *min_resid_terms: int = 3*, *decrement_threshold: float = 0*, *min_threshold: float = 5*, *watershed: int | dict[float, int] | None = None*, *flag_if_broken: bool = True*, *init_flags: ndarray | None = None*, *std_estimator: Literal['model', 'medfilt', 'std', 'mad', 'sliding_rms'] = 'model'*, *medfilt_width: int = 100*, *sliding_rms_width: int = 100*)

Flag data by subtracting a smooth model and iteratively removing outliers.

On each iteration, a model is fit to the unflagged data, and another model is fit to the absolute residuals. Bins with absolute residuals greater than `threshold` are flagged, and the process is repeated until no new flags are found.

Parameters

- **x** – The coordinates of the data.
- **data** – The data (same shape as `x`).
- **model** – A model to fit to the data.
- **resid_model** – The model to fit to the absolute residuals.
- **flags** (*array-like, optional*) – The flags associated with the data (same shape as `spectrum`).
- **weights** (*array-like,, optional*) – The weights associated with the data (same shape as `spectrum`).
- **n_resid** (*int, optional*) – The number of polynomial terms to use to fit the residuals.
- **threshold** (*float, optional*) – The factor by which the absolute residual model is multiplied to determine outliers.
- **max_iter** (*int, optional*) – The maximum number of iterations to perform.
- **accumulate** (*bool, optional*) – Whether to accumulate flags on each iteration.
- **increase_order** (*bool, optional*) – Whether to increase the order of the polynomial on each iteration.
- **decrement_threshold** (*float, optional*) – An amount to decrement the threshold by every iteration. Threshold will never go below `min_threshold`.
- **min_threshold** (*float, optional*) – The minimum threshold to decrement to.

- **watershed** – How many data points *on each side* of a flagged point that should be flagged. If a dictionary, you can give keys as the threshold above which z-scores will be flagged, and as values, the number of bins flagged beside it. Use 0.0 threshold to indicate the base threshold.
- **init_flags** – Initial flags that are not remembered after the first iteration. These can help with getting the initial model. If a tuple, should be a min and max frequency of a range to flag.
- **std_estimator** – The estimator to use to get the standard deviation of each sample.
- **medfilt_width** – Only used if *std_estimator='medfilt'*. The width (in number of bins) to use for the median filter.

Returns

flags – Boolean array of the same shape as `data`.

edges_cal.xrfi.robust_divide

edges_cal.xrfi.**robust_divide**(*num*, *den*)

Prevent division by zero.

This function will compute division between two array-like objects by setting values to infinity when the denominator is small for the given data type. This avoids floating point exception warnings that may hide genuine problems in the data.

Parameters

- **num** (*array*) – The numerator.
- **den** (*array*) – The denominator.

Returns

out (*array*) – The result of dividing num / den. Elements where b is small (or zero) are set to infinity.

edges_cal.xrfi.visualise_model_info

edges_cal.xrfi.**visualise_model_info**(*info:* ModelFilterInfo | ModelFilterInfoContainer, *n: int = 0*)

Make a nice visualisation of the info output from *xrfi_model()*.

Parameters

- **info** – The output `info` from *xrfi_model()*.
- **n** – The number of iterations to plot. Default is to plot them all. Negative numbers will plot the last n, and positive will plot the first n.

edges_cal.xrfi.xrfi_explicit

edges_cal.xrfi.**xrfi_explicit**(*spectrum: ndarray | None = None*, ***, *freq: ndarray*, *flags: ndarray | None = None*, *rfi_file=None*, *extra_rfi=None*) → ndarray[bool]

Excise RFI from given data using an explicitly set list of flag ranges.

Parameters

- **spectrum** – This parameter is unused in this function.
- **freq** – Frequencies, in MHz, of the data.

- **flags** – Known flags.
- **rfi_file** (*str, optional*) – A YAML file containing the key 'rfi_ranges', which should be a list of 2-tuples giving the (min, max) frequency range of known RFI channels (in MHz). By default, uses a file included in *edges-analysis* with known RFI channels from the MRO.
- **extra_rfi** (*list, optional*) – A list of extra RFI channels (in the format of the *rfi_ranges* from the *rfi_file*).

Returns

flags (*array-like*) – Boolean array of the same shape as `spectrum` indicated which channels/times have flagged RFI.

edges_cal.xrfi.xrfi_medfilt

edges_cal.xrfi.**xrfi_medfilt**(*spectrum: ndarray*, *threshold: float = 6*, *flags: ndarray | None = None*, *kf: int = 8*, *kt: int = 8*, *inplace: bool = True*, *max_iter: int = 1*, *poly_order: int = 0*, *accumulate: bool = False*, *use_meanfilt: bool = True*) → tuple[numpy.ndarray, dict[str, Any]]

Generate RFI flags for a given spectrum using a median filter.

Parameters

- **spectrum** (*array-like*) – Either a 1D array of shape `(NFREQS,)` or a 2D array of shape `(NTIMES, NFREQS)` defining the measured raw spectrum. If 2D, a 2D filter in freq*time will be applied by default. One can perform the filter just over frequency (in the case that *NTIMES > 1*) by setting *kt=0*.
- **threshold** (*float, optional*) – Number of effective sigma at which to clip RFI.
- **flags** (*array-like, optional*) – Boolean array of pre-existing flagged data to ignore in the filtering.
- **kt, kf** (*tuple of int/None*) – The half-size of the kernel to convolve (eg. kernel size over frequency will be `2*kt+1`). Value of zero (for any dimension) omits that axis from the kernel, effectively applying the detrending for each subarray along that axis. Value of None will effectively (but slowly) perform a median along the entire axis before running the kernel over the other axis.
- **inplace** (*bool, optional*) – If True, and flags are given, update the flags in-place instead of creating a new array.
- **max_iter** (*int, optional*) – Maximum number of iterations to perform. Each iteration uses the flags of the previous iteration to achieve a more robust estimate of the flags. Multiple iterations are more useful if `poly_order > 0`.
- **poly_order** (*int, optional*) – If greater than 0, fits a polynomial to the spectrum before performing the median filter. Only allowed if spectrum is 1D. This is useful for getting the number of false positives down. If max_iter>1, the polynomial will be refit on each iteration (using new flags).
- **accumulate** (*bool,optional*) – If True, on each iteration, accumulate flags. Otherwise, use only flags from the previous iteration and then forget about them. Recommended to be False.
- **use_meanfilt** (*bool, optional*) – Whether to apply a mean filter *after* the median filter. The median filter is good at getting RFI, but can also pick up non-RFI if the spectrum is steep compared to the noise. The mean filter is better at only getting RFI if the RFI has already been flagged.

Returns

flags (*array-like*) – Boolean array of the same shape as `spectrum` indicated which channels/times have flagged RFI.

Notes

The default combination of using a median filter followed by a mean filter works quite well. The median filter works quite well at picking up large RFI (wrt to the noise level), but can also create false positives if the noise level is small wrt the steepness of the slope. Following by a flagged mean filter tends to remove these false positives (as it doesn't get pinned to zero when the function is monotonic).

It is unclear whether performing an iterative filtering is very useful unless using a polynomial subtraction. With polynomial subtraction, one should likely use at least a few iterations, without accumulation, so that the polynomial is not skewed by the as-yet-unflagged RFI.

Choice of kernel size can be important. The wider the kernel, the more "signal-to-noise" one will get on the RFI. Also, if there is a bunch of RFI all clumped together, it will definitely be missed by a kernel window of order double the size of the clump or less. By increasing the kernel size, these clumps are picked up, but edge-effects become more prevalent in this case. One option here would be to iterate over kernel sizes (getting smaller), such that very large blobs are first flagged out, then progressively finer detail is added. Use `xrfi_iterative_medfilt` for that.

edges_cal.xrfi.xrfi_model

`edges_cal.xrfi.`**`xrfi_model`**(*spectrum: ndarray*, ***, *freq: ndarray*, *inplace: bool = False*, *init_flags: ndarray | tuple[float, float] | None = None*, *flags: ndarray | None = None*, ***kwargs*)

Flag RFI by subtracting a smooth model and iteratively removing outliers.

On each iteration, a model is fit to the unflagged data, and another model is fit to the absolute residuals. Bins with absolute residuals greater than `n_abs_resid_threshold` are flagged, and the process is repeated until no new flags are found.

Parameters

- **spectrum** (*array-like*) – A 1D spectrum. Note that instead of a spectrum, model residuals can be passed. The function does *not* assume the input is positive.
- **freq** – The frequencies associated with the spectrum.
- **inplace** (*bool, optional*) – Whether to fill up given flags array with the updated flags.
- **init_flags** – Initial flags that are not remembered after the first iteration. These can help with getting the initial model. If a tuple, should be a min and max frequency of a range to flag.
- ****kwargs** – All other parameters passed to *model_filter()*

Returns

flags (*array-like*) – Boolean array of the same shape as `spectrum` indicated which channels/times have flagged RFI.

edges_cal.xrfi.xrfi_model_sweep

edges_cal.xrfi.**xrfi_model_sweep**(*spectrum: ndarray*, **, freq: ndarray | None = None*, *flags: ndarray | None = None*, *weights: ndarray | None = None*, *model: Model = Polynomial(parameters=None, n_terms=3, transform=IdentityTransform(), offset=0.0)*, *window_width: int = 100*, *use_median: bool = True*, *n_bootstrap: int = 20*, *threshold: float | None = 3.0*, *which_bin: str = 'last'*, *watershed: int = 0*, *max_iter: int = 1*) → tuple[numpy.ndarray, dict]

Flag RFI by using a moving window and a low-order polynomial to detrend.

This is similar to *xrfi_medfilt()*, except that within each sliding window, a low-order polynomial is fit, and the std dev of the residuals is used as the underlying distribution width at which to clip RFI.

Parameters

- **spectrum** (*array-like*) – A 1D or 2D array, where the last axis corresponds to frequency. The data measured at those frequencies.
- **flags** (*array-like*) – The boolean array of flags.
- **weights** (*array-like*) – The weights associated with the data (same shape as *spectrum*).
- **model_type** – The kind of model to use to fit each window. If a string, it must be the name of a `Model`.
- **window_width** (*int, optional*) – The width of the moving window in number of channels.
- **use_median** (*bool, optional*) – Instead of using bootstrap for the initial window, use Median Absolute Deviation. If True, `n_bootstrap` is not used. Note that this is typically more robust than bootstrap.
- **n_bootstrap** (*int, optional*) – Number of bootstrap samples to take to estimate the standard deviation of the data without RFI.
- **n_terms** – The number of terms in the model (if applicable).
- **threshold** – The number of sigma away from the fitted model must be before it is flagged. Higher numbers get less false positives, but may miss some true flags.
- **which_bin** – Which bin to flag in each window. May be "last" (default), "all". In each window, only this bin will be flagged (or all bins will be if "all").
- **watershed** – The number of bins beside each flagged RFI that are assumed to also be RFI.
- **max_iter** – The maximum number of iterations to use before determining the flags in a particular window.

Returns

- **flags** (*array-like*) – Boolean array of the same shape as `spectrum` indicated which channels/times have flagged RFI.
- **info** (*dict*) – A dictionary of info about the fit, that can be used to inspect what happened.

Notes

Some notes on this algorithm. The basic idea is that a window of a given width is used, and within that window, a model is fit to the spectrum data. The residuals of that fit are used to calculate the standard deviation (or the 'noise-level'), which gives an indication of outliers. This standard deviation may be found either by bootstrap sampling, or by using the Median Absolute Deviation (MAD). Both of these to *some extent* account for RFI that's still in the residuals, but the MAD is typically a bit more robust. **NOTE:** getting the estimate of the standard deviation wrong is one of the easiest ways for this algorithm to fail. It relies on a few assumptions. Firstly, the window can't be too large, or else the residuals within the window aren't stationary. Secondly, while previously-defined flags are used to flag out what might be RFI, os that those data are NOT used in getting the standard deviation, any remaining RFI will severely bias the std. Obviously, if RFI remains in the data, the model itself might not be very accurate either.

Note that for each window, at first the RFI in that window will likely be unflagged, and the std will be computed with all the channels, RFI included. This is why using the MAD or bootstrapping is required. Even if the std is predicted robustly via this method (i.e. there are more good bins than bad in the window), the model itself may not be very good, and so the resulting flags may not be very good. This is where using the option of `max_iter>1` is useful – in this case, the model is fit to the same window repeatedly until the flags in the window don't change between iterations (note this is NOT cumulative).

In the end, by default, only a single channel is actually flagged per-window. While inside the iterative loop, any number of flags can be set (in order to make a better prediction of the model and std), only the first, last or central pixel is actually flagged and used for the next window. This can be changed by setting `which_bin='all'`.

edges_cal.xrfi.xrfi_watershed

edges_cal.xrfi.**xrfi_watershed**(*spectrum: ndarray | None = None*, ***, *freq: ndarray | None = None*, *flags: ndarray | None = None*, *weights: ndarray | None = None*, *tol: float | tuple[float] = 0.5*, *inplace=False*)

Apply a watershed over frequencies and times for flags.

Make sure that times/freqs with many flags are all flagged.

Parameters

- **spectrum** – Not used in this routine.
- **flags** (*ndarray of bool*) – The existing flags.
- **tol** (*float or tuple*) – The tolerance – i.e. the fraction of entries that must be flagged before flagging the whole axis. If a tuple, the first element is for the frequency axis, and the second for the time axis.
- **inplace** (*bool, optional*) – Whether to update the flags in-place.

Returns

- *ndarray* – Boolean array of flags.
- *dict* – Information about the flagging procedure (empty for this function)

Classes

ModelFilterInfo(n_flags_changed, ...)	A simple object representing the information returned by *model_filter()*.
ModelFilterInfoContainer(models)	A container of *ModelFilterInfo* objects.

edges_cal.xrfi.ModelFilterInfo

class edges_cal.xrfi.**ModelFilterInfo**(*n_flags_changed: list[int]*, *total_flags: list[int]*, *models: list[*edges_cal.modelling.Model*]*, *res_models: list[*edges_cal.modelling.Model*]* | *None*, *n_iters: int*, *thresholds: list[float]*, *stds: list[numpy.ndarray[float]]*, *x: ndarray*, *data: ndarray*, *flags: list[numpy.ndarray[bool]]*)

A simple object representing the information returned by *model_filter()*.

Methods

__init__(n_flags_changed, total_flags, ...)	
from_file(fname[, group])	Create the object by reading from a HDF5 file.
get_absres_model([indx])	Get the *model* of the absolute residuals.
get_model([indx])	Get the model values.
get_residual([indx])	Get the residuals.
write(fname[, group])	Write the object to a HDF5 file.

edges_cal.xrfi.ModelFilterInfo.__init__

ModelFilterInfo.**__init__**(*n_flags_changed: list[int]*, *total_flags: list[int]*, *models: list[*edges_cal.modelling.Model*]*, *res_models: list[*edges_cal.modelling.Model*]* | *None*, *n_iters: int*, *thresholds: list[float]*, *stds: list[numpy.ndarray[float]]*, *x: ndarray*, *data: ndarray*, *flags: list[numpy.ndarray[bool]]*) → None

edges_cal.xrfi.ModelFilterInfo.from_file

classmethod ModelFilterInfo.**from_file**(*fname: str* | *Path*, *group: str = '/'*)

Create the object by reading from a HDF5 file.

edges_cal.xrfi.ModelFilterInfo.get_absres_model

ModelFilterInfo.**get_absres_model**(*indx: int = -1*)

Get the *model* of the absolute residuals.

edges_cal.xrfi.ModelFilterInfo.get_model

ModelFilterInfo.**get_model**(*indx: int = -1*)

Get the model values.

edges_cal.xrfi.ModelFilterInfo.get_residual

ModelFilterInfo.**get_residual**(*indx: int = -1*)

Get the residuals.

edges_cal.xrfi.ModelFilterInfo.write

ModelFilterInfo.**write**(*fname: str | Path*, *group: str = '/'*)

Write the object to a HDF5 file.

Attributes

n_flags_changed
total_flags
models
res_models
n_iters
thresholds
stds
x
data
flags

edges_cal.xrfi.ModelFilterInfo.n_flags_changed

ModelFilterInfo.**n_flags_changed**: list[int]

edges_cal.xrfi.ModelFilterInfo.total_flags

ModelFilterInfo.**total_flags**: list[int]

edges_cal.xrfi.ModelFilterInfo.models

ModelFilterInfo.**models**: list[*edges_cal.modelling.Model*]

edges_cal.xrfi.ModelFilterInfo.res_models

ModelFilterInfo.**res_models**: list[*edges_cal.modelling.Model*] | None

edges_cal.xrfi.ModelFilterInfo.n_iters

ModelFilterInfo.**n_iters**: int

edges_cal.xrfi.ModelFilterInfo.thresholds

ModelFilterInfo.**thresholds**: list[float]

edges_cal.xrfi.ModelFilterInfo.stds

ModelFilterInfo.**stds**: list[numpy.ndarray[float]]

edges_cal.xrfi.ModelFilterInfo.x

ModelFilterInfo.**x**: ndarray

edges_cal.xrfi.ModelFilterInfo.data

ModelFilterInfo.**data**: ndarray

edges_cal.xrfi.ModelFilterInfo.flags

ModelFilterInfo.**flags**: list[numpy.ndarray[bool]]

edges_cal.xrfi.ModelFilterInfoContainer

class edges_cal.xrfi.**ModelFilterInfoContainer**(*models: list[edges_cal.xrfi.ModelFilterInfo] = <factory>*)

A container of *ModelFilterInfo* objects.

This is almost a perfect drop-in replacement for a singular *ModelFilterInfo* instance, but combines a number of them together seamlessly. This can be useful if several sub-models were fit to one long stream of data.

Methods

__init__([models])	
append(model)	Create a new object by appending a set of info to the existing.
from_file(fname)	Create an object from a given file.
get_absres_model([indx])	Get the *model* of the absolute residuals.
get_model([indx])	Get the model values.
get_residual([indx])	Get the residual values.
write(fname)	Write the object to a file.

edges_cal.xrfi.ModelFilterInfoContainer.__init__

ModelFilterInfoContainer.**__init__**(*models: list[edges_cal.xrfi.ModelFilterInfo] = <factory>*) → None

edges_cal.xrfi.ModelFilterInfoContainer.append

ModelFilterInfoContainer.**append**(*model:* ModelFilterInfo) → *ModelFilterInfoContainer*

Create a new object by appending a set of info to the existing.

edges_cal.xrfi.ModelFilterInfoContainer.from_file

classmethod ModelFilterInfoContainer.**from_file**(*fname: str*)

Create an object from a given file.

edges_cal.xrfi.ModelFilterInfoContainer.get_absres_model

ModelFilterInfoContainer.**get_absres_model**(*indx: int = -1*)

Get the *model* of the absolute residuals.

edges_cal.xrfi.ModelFilterInfoContainer.get_model

ModelFilterInfoContainer.**get_model**(*indx: int = -1*)

Get the model values.

edges_cal.xrfi.ModelFilterInfoContainer.get_residual

ModelFilterInfoContainer.**get_residual**(*indx: int = -1*)

Get the residual values.

edges_cal.xrfi.ModelFilterInfoContainer.write

ModelFilterInfoContainer.**write**(*fname: str*)

Write the object to a file.

Attributes

data	The raw data that was filtered.
flags	The returned flags on each iteration.
n_flags_changed	The number of flags changed on each filtering iteration.
n_iters	The number of iterations of the filtering.
stds	The standard deviations at each datum for each iteration.
thresholds	The threshold at each iteration.
total_flags	The total number of flags after each iteration.
x	The data coordinates.
models	

edges_cal.xrfi.ModelFilterInfoContainer.data

ModelFilterInfoContainer.**data**

The raw data that was filtered.

edges_cal.xrfi.ModelFilterInfoContainer.flags

ModelFilterInfoContainer.**flags**
The returned flags on each iteration.

edges_cal.xrfi.ModelFilterInfoContainer.n_flags_changed

ModelFilterInfoContainer.**n_flags_changed**
The number of flags changed on each filtering iteration.

edges_cal.xrfi.ModelFilterInfoContainer.n_iters

ModelFilterInfoContainer.**n_iters**
The number of iterations of the filtering.

edges_cal.xrfi.ModelFilterInfoContainer.stds

ModelFilterInfoContainer.**stds**
The standard deviations at each datum for each iteration.

edges_cal.xrfi.ModelFilterInfoContainer.thresholds

ModelFilterInfoContainer.**thresholds**
The threshold at each iteration.

edges_cal.xrfi.ModelFilterInfoContainer.total_flags

ModelFilterInfoContainer.**total_flags**
The total number of flags after each iteration.

edges_cal.xrfi.ModelFilterInfoContainer.x

ModelFilterInfoContainer.**x**
The data coordinates.

edges_cal.xrfi.ModelFilterInfoContainer.models

ModelFilterInfoContainer.**models**: list[*edges_cal.xrfi.ModelFilterInfo*]

Exceptions

NoDataError

edges_cal.tools

Tools to use in other modules.

Functions

as_readonly(x)	Get a read-only view into an array without copying.
bin_array(x[, size])	Simple unweighted mean-binning of an array.
dct_of_list_to_list_of_dct(dct)	Take a dict of key: list pairs and turn it into a list of all combos of dicts.
gauss_smooth(x, size[, decimate_at])	Smooth x with a Gaussian function, and reduces the size of the array.
get_data_path(pth)	Impute the global data path to a given input in place of a colon.
is_unit(unit)	Whether the given input is a recognized unit.
unit_convert_or_apply(x, unit[, in_place, warn])	Safely convert a given value to a quantity.
unit_converter(unit)	Return a function that will convert values to a given quantity.
vld_unit(unit[, equivalencies])	Attr validator to check physical type.

edges_cal.tools.as_readonly

edges_cal.tools.**as_readonly**(*x: ndarray*) → ndarray

Get a read-only view into an array without copying.

edges_cal.tools.bin_array

edges_cal.tools.**bin_array**(*x: ndarray*, *size: int = 1*) → ndarray

Simple unweighted mean-binning of an array.

Parameters

- **x** – The array to be binned. Only the last axis will be binned.
- **size** – The size of the bins.

Notes

The last axis of *x* is binned. It is assumed that the coordinates corresponding to *x* are regularly spaced, so the final average just takes *size* values and averages them together.

If the array is not divisible by *size*, the last values are left out.

Examples

Simple 1D example:

```
>>> x = np.array([1, 1, 2, 2, 3, 3])
>>> bin_array(x, size=2)
[1, 2, 3]
```

The last remaining values are left out:

```
>>> x = np.array([1, 1, 2, 2, 3, 3, 4])
>>> bin_array(x, size=2)
[1, 2, 3]
```

edges_cal.tools.dct_of_list_to_list_of_dct

edges_cal.tools.**dct_of_list_to_list_of_dct**(*dct: dict[str, Sequence]*) → list[dict]

Take a dict of key: list pairs and turn it into a list of all combos of dicts.

Parameters

dct – A dictionary for which each value is an iterable.

Returns

list – A list of dictionaries, each having the same keys as the input `dct`, but in which the values are the elements of the original iterables.

Examples

```
>>> dct_of_list_to_list_of_dct(
>>>     { 'a': [1, 2], 'b': [3, 4]}
[
    {"a": 1, "b": 3},
    {"a": 1, "b": 4},
    {"a": 2, "b": 3},
    {"a": 2, "b": 4},
]
```

edges_cal.tools.gauss_smooth

edges_cal.tools.**gauss_smooth**(*x: ndarray*, *size: int*, *decimate_at: int | None = None*) → ndarray

Smooth x with a Gaussian function, and reduces the size of the array.

edges_cal.tools.get_data_path

edges_cal.tools.**get_data_path**(*pth: str | Path*) → Path

Impute the global data path to a given input in place of a colon.

edges_cal.tools.is_unit

edges_cal.tools.**is_unit**(*unit: str*) → bool

Whether the given input is a recognized unit.

edges_cal.tools.unit_convert_or_apply

edges_cal.tools.**unit_convert_or_apply**(*x: float | Quantity*, *unit: str | Unit*, *in_place: bool = False*, *warn: bool = False*) → Quantity

Safely convert a given value to a quantity.

edges_cal.tools.unit_converter

edges_cal.tools.**unit_converter**(*unit: str | Unit*) → Callable[[float | Quantity], Quantity]

Return a function that will convert values to a given quantity.

edges_cal.tools.vld_unit

edges_cal.tools.**vld_unit**(*unit: str | Unit*, *equivalencies=()*) → Callable[[Any, Attribute, Any], None]

Attr validator to check physical type.

Classes

FrequencyRange(f, *, f_low, f_high[, ...])	Class defining a set of frequencies.

edges_cal.tools.FrequencyRange

class edges_cal.tools.**FrequencyRange**(*f: tp.FreqType*, *, *f_low: tp.FreqType = <Quantity 0. MHz>*, *f_high: float = <Quantity inf MHz>*, *bin_size=1*, *alan_mode=False*, *post_bin_f_low: tp.Freqtype = <Quantity 0. MHz>*, *post_bin_f_high: float = <Quantity inf MHz>*)

Class defining a set of frequencies.

A given frequency range can be cut on either end, and be made more sparse.

Parameters

- **f** – An array of frequencies defining a given spectrum.
- **f_low** – A minimum frequency to keep in the array. Default is min(f).
- **f_high** – A minimum frequency to keep in the array. Default is min(f).
- **bin_size** (*int*) – Bin input frequencies into bins of this size.
- **alan_mode** (*bool*) – Only applicable if bin_size > 1. If True, then take every bin_size frequency, starting from the 0th channel. Otherwise take the mean of each bin as the resulting frequency.

Methods

__init__(f, *[, f_low, f_high, bin_size, ...])	Method generated by attrs for class FrequencyRange.
clone(**kwargs)	Make a new frequency range object with updated parameters.
denormalize(f)	De-normalise a set of frequencies.
from_edges([n_channels, max_freq, ...])	Construct a *FrequencyRange* object with underlying EDGES freqs.
normalize(f)	Normalise a set of frequencies.
with_new_mask(**kwargs)	Make a new read-only frequency range object with the same freqs.

edges_cal.tools.FrequencyRange.__init__

FrequencyRange.**__init__**(*f: tp.FreqType*, *, *f_low: tp.FreqType = <Quantity 0. MHz>*, *f_high: float = <Quantity inf MHz>*, *bin_size=1*, *alan_mode=False*, *post_bin_f_low: tp.Freqtype = <Quantity 0. MHz>*, *post_bin_f_high: float = <Quantity inf MHz>*) → None

Method generated by attrs for class FrequencyRange.

edges_cal.tools.FrequencyRange.clone

FrequencyRange.**clone**(***kwargs*)

Make a new frequency range object with updated parameters.

edges_cal.tools.FrequencyRange.denormalize

FrequencyRange.**denormalize**(*f*)

De-normalise a set of frequencies.

Normalizes such that -1 aligns with `min` and +1 aligns with `max`.

Parameters
f (*array_like*) – Frequencies to de-normalize

Returns
array_like, shape [f,] – The de-normalized frequencies.

edges_cal.tools.FrequencyRange.from_edges

classmethod FrequencyRange.**from_edges**(*n_channels: int = 32768*, *max_freq: float = <Quantity 200. MHz>*, *keep_full: bool = True*, *f_low=<Quantity 0. MHz>*, *f_high=<Quantity inf MHz>*, ***kwargs*) → *FrequencyRange*

Construct a *FrequencyRange* object with underlying EDGES freqs.

Parameters

- **n_channels** (*int*) – Number of channels
- **max_freq** (*float*) – Maximum frequency in original measurement.
- **keep_full** – Whether to keep the full underlying frequency array, or just the part of the array inside the mask.
- **f_low, f_high** – A frequency range to keep.
- **kwargs** – All other arguments passed through to *FrequencyRange*.

Returns

FrequencyRange – The FrequencyRange object with the correct underlying frequencies.

Notes

This is correct. The channel width is the important thing. The channel width is given by the FFT. We actually take 32678*2 samples of data at 400 Mega-samples per second. We only use the first half of the samples (since it's real input). Regardless, the frequency channel width is thus 400 MHz / (32678*2) == 200 MHz / 32678 ~ 6.103 kHz

edges_cal.tools.FrequencyRange.normalize

FrequencyRange.**normalize**(*f*) → ndarray

Normalise a set of frequencies.

Normalizes such that -1 aligns with `min` and +1 aligns with `max`.

Parameters

f (*array_like*) – Frequencies to normalize

Returns

array_like, shape [f,] – The normalized frequencies.

edges_cal.tools.FrequencyRange.with_new_mask

FrequencyRange.**with_new_mask**(***kwargs*)

Make a new read-only frequency range object with the same freqs.

Attributes

center	The center of the frequency array.
df	Resolution of the frequencies.
freq	The frequency array.
freq_full	Alias for *f*.
freq_recentred	The frequency array re-centred so that it extends from -1 to 1.
mask	Mask used to take input frequencies to output frequencies.
max	Maximum frequency in the array.
min	Minimum frequency in the array.
n	The number of frequencies in the (masked) array.
range	Full range of the frequencies.
bin_size	
alan_mode	
post_bin_f_low	
post_bin_f_high	

edges_cal.tools.FrequencyRange.center

FrequencyRange.**center**

The center of the frequency array.

edges_cal.tools.FrequencyRange.df

FrequencyRange.**df**

Resolution of the frequencies.

edges_cal.tools.FrequencyRange.freq

FrequencyRange.**freq**

The frequency array.

edges_cal.tools.FrequencyRange.freq_full

property FrequencyRange.**freq_full**

Alias for *f*.

edges_cal.tools.FrequencyRange.freq_recentred

FrequencyRange.**freq_recentred**

The frequency array re-centred so that it extends from -1 to 1.

edges_cal.tools.FrequencyRange.mask

FrequencyRange.**mask**

Mask used to take input frequencies to output frequencies.

edges_cal.tools.FrequencyRange.max

FrequencyRange.**max**

Maximum frequency in the array.

edges_cal.tools.FrequencyRange.min

FrequencyRange.**min**

Minimum frequency in the array.

edges_cal.tools.FrequencyRange.n

property FrequencyRange.**n: int**

The number of frequencies in the (masked) array.

edges_cal.tools.FrequencyRange.range

FrequencyRange.**range**

Full range of the frequencies.

edges_cal.tools.FrequencyRange.bin_size

FrequencyRange.**bin_size: int**

edges_cal.tools.FrequencyRange.alan_mode

FrequencyRange.**alan_mode: bool**

edges_cal.tools.FrequencyRange.post_bin_f_low

FrequencyRange.**post_bin_f_low: tp.Freqtype**

edges_cal.tools.FrequencyRange.post_bin_f_high

FrequencyRange.**post_bin_f_high: float**

Package Config

edges_cal.types	Simple type definitions for use internally.
edges_cal.cached_property	A wrapper of cached_property that also saves which things are cached.

edges_cal.types

Simple type definitions for use internally.

edges_cal.cached_property

A wrapper of cached_property that also saves which things are cached.

Functions

safe_property(f)	An alternative property decorator that substitates AttributeError for RunTime.

edges_cal.cached_property.safe_property

edges_cal.cached_property.**safe_property**(*f*)

An alternative property decorator that substitates AttributeError for RunTime.

Classes

cached_property(func)	

edges_cal.cached_property.cached_property

class edges_cal.cached_property.**cached_property**(*func*)

Methods

__init__(func)

edges_cal.cached_property.cached_property.__init__

cached_property.**__init__**(*func*)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z