ctaplot Documentation

Release 0.4.0

Thomas Vuillaume

May 11, 2020

Contents

CHAPTER 1

Contents

1.1 ctaplot

ctaplot is a collection of functions to make instrument response functions (IRF) and reconstruction quality-checks plots for Imaging Atmospheric Cherenkov Telescopes such as CTA

Given a list of reconstructed and simulated quantities, compute and plot the Instrument Response Functions such as:

- angular resolution
- energy resolution
- effective surface
- impact point resolution

You may find examples in the documentation.

- Code : https://github.com/vuillaut/ctaplot
- Documentation : https://ctaplot.readthedocs.io/en/latest/
- Author contact: Thomas Vuillaume - thomas.vuillaume@lapp.in2p3.fr
- License: MIT

The CTA instrument response functions data used in ctaplot come from the CTA Consortium and Observatory and may be found on the cta-observatory website .

In cases for which the CTA instrument response functions are used in a research project, we ask to add the following acknowledgement in any resulting publication:

"This research has made use of the CTA instrument response functions provided by the CTA Consortium and Observatory, see http://www.cta-observatory.org/science/cta-performance/ (version prod3b-v2) for more details."

1.1.1 Install

Requirements packages:

- python > 3.6
- numpy
- scipy>=0.19
- matplotlib>=2.0
- astropy

We recommend the use of anaconda

The package is available through pip:

```
pip install ctaplot
```

```
export GAMMABOARD_DATA=path_to_the_data_directory
```

We recommend that you add this line to your bash source file (*$HOME/.bashrc* or *$HOME/.bash_profile*)

1.1.2 GammaBoard

A dashboard to show them all.

GammaBoard is a simple jupyter dashboard thought to display metrics assessing the reconstructions performances of Imaging Atmospheric Cherenkov Telescopes (IACTs). Deep learning is a lot about bookkeeping and trials and errors. GammaBoard ease this bookkeeping and allows quick comparison of the reconstruction performances of your machine learning experiments.

It is a working prototype used in CTA, especially by the [GammaLearn](https://gitlab.lapp.in2p3.fr/GammaLearn/) project.

Run GammaBoard

To launch the dashboard, you can simply try the command:

```
gammaboard
```

This will run a temporary copy of the dashboard (a jupyter notebook). Local changes that you make in the dashboard will be discarded afterwards.

GammaBoard is using data in a specific directory storing all your experiments files. This directory is known under *$GAMMABOARD_DATA* by default. However, you can change the path access at any time in the dashboard itself.

Demo

Here is a simple demo of GammaBoard:

- On top the plots (metrics) such as angular resolution and energy resolution.
- Below, the list of experiments in the user folder.

When an experiment is selected in the list, the data is automatically loaded, the metrics computed and displayed. A list of information provided during the training phase is also displayed. As many experiments results can be overlaid. When an experiment is deselected, it simply is removed from the plots.

1.2 License

MIT License

Copyright (c) 2018 ctaplot

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.3 Modules

1.3.1 plots.py

Functions to make IRF and other reconstruction quality-check plots

`ctaplot.plots.`**`plot_resolution`**(*bins*, *res*, *log=False*, *ax=None*, ***kwargs*)

Plot the passed resolution.

Parameters

- **`bins`** (1D *numpy.ndarray*) –
- **`res`** (2D *numpy.ndarray* - output from *ctpalot.ana.resolution*) – res[:,0]: resolution res[:,1]: lower confidence limit res[:,2]: upper confidence limit
- **`log`** (*bool*) – if true, x is logscaled
- **`ax`** (*matplotlib.pyplot.axes*) –
- **`kwargs`** (kwargs for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_resolution_difference**(*bins*, *reference_resolution*, *new_resolution*, *ax=None*, ***kwargs*)

Plot the algebric difference between a new resolution and reference resolution.

Parameters

- **bins** (*numpy.ndarray*) –
- **reference_resolution** (*numpy.ndarray*) – output from *ctaplot.ana.resolution*
- **new_resolution** (*numpy.ndarray*) – output from *ctaplot.ana.resolution*
- **ax** (*matplotlib.pyplot.axis*) –
- **kwargs** (args for *ctaplot.plots.plot_resolution*) –

Returns ax

Return type *matplotlib.pyplot.axis*

ctaplot.plots.**plot_energy_resolution**(*simu_energy*, *reco_energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *ax=None*, ***kwargs*)

Plot the enregy resolution as a function of the energy

Parameters

- **simu_energy** (*numpy.ndarray*) –
- **reco_energy** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **bias_correction** (*bool*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_binned_bias**(*simu*, *reco*, *x*, *relative_scaling_method=None*, *ax=None*, *bins=10*, *log=False*, ***kwargs*)

Plot the bias between *simu* and *reco* as a function of bins of *x*

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –
- **x** (*numpy.ndarray*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*
- **ax** (*matplotlib.pyplot.axis*) –
- **bins** (bins for *numpy.histogram*) –
- **log** (*bool*) – if True, logscale is applied to the x axis
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axis*

ctaplot.plots.**plot_energy_bias**(*simu_energy*, *reco_energy*, *ax=None*, ***kwargs*)
Plot the energy bias

Parameters

- **simu_energy** (*numpy.ndarray*) –
- **reco_energy** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_impact_parameter_error_per_bin**(*x*, *reco_x*, *reco_y*, *simu_x*, *simu_y*, *bins=10*, *ax=None*, ***kwargs*)
Plot the impact parameter error per bin

Parameters

- **x** (*numpy.ndarray*) –
- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **bins** (arg for *np.histogram*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *plot_resolution*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_layout_map**(*tel_x*, *tel_y*, *tel_type=None*, *ax=None*, ***kwargs*)
Plot the layout map of telescopes positions

Parameters

- **tel_x** (*numpy.ndarray*) –
- **tel_y** (*numpy.ndarray*) –
- **TelId** (*numpy.ndarray*) –
- **tel_type** (*numpy.ndarray*) –
- **LayoutId** (*numpy.ndarray*) –
- **Outfile** (*string*) –

ctaplot.plots.**plot_multiplicity_per_telescope_type**(*multiplicity*, *telescope_type*, *ax=None*, *outfile=None*, *quartils=False*, ***kwargs*)
Plot the multiplicity for each telescope type

Parameters

- **multiplicity** (*numpy.ndarray*) –
- **telescope_type** (*numpy.ndarray*) – same shape as *multiplicity*

- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*path*) –
- **quartils** (*bool - True to plot 50% and 90% quartil mark*) –
- **kwargs** (args for *matplotlib.pyplot.hist*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_multiplicity_hist**(*multiplicity*, *ax=None*, *outfile=None*, *quartils=False*, ***kwargs*)

Histogram of the telescopes multiplicity

Parameters

- **multiplicity** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string*) –
- ****kwargs** (args for *matplotlib.pyplot.bar*) –

ctaplot.plots.**plot_angles_distribution**(*reco_alt*, *reco_az*, *source_alt*, *source_az*, *outfile=None*)

Plot the distribution of reconstructed angles in two axes. Save figure to *outfile* in png format.

Parameters

- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **source_alt** (*float*) –
- **source_az** (*float*) –
- **outfile** (*string*) –

Returns

Return type *matplotlib.pyplot.figure*

ctaplot.plots.**plot_angles_map_distri**(*reco_alt*, *reco_az*, *source_alt*, *source_az*, *energies*, *outfile=None*)

Plot the angles map distribution

Parameters

- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **source_alt** (*float*) –
- **source_az** (*float*) –
- **energies** (*numpy.ndarray*) –
- **outfile** (*str*) –

Returns fig

Return type *matplotlib.pyplot.figure*

ctaplot.plots.**plot_angular_resolution_cta_performance**(*cta_site*, *ax=None*, ***kwargs*)

Plot the official CTA performances (June 2018) for the angular resolution

Parameters

- **cta_site** (string, see *ana.cta_performance*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_angular_resolution_cta_requirement**(*cta_site*, *ax=None*, ***kwargs*)

Plot the CTA requirement for the angular resolution :param cta_site: see *ctaplot.ana.cta_requirement* :type cta_site: string :param ax: :type ax: *matplotlib.pyplot.axes* :param kwargs: :type kwargs: args for *matplotlib.pyplot.plot*

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_angular_resolution_per_energy**(*reco_alt*, *reco_az*, *mc_alt*, *mc_az*, *energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *ax=None*, ***kwargs*)

Plot the angular resolution as a function of the energy

Parameters

- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **mc_alt** (*numpy.ndarray*) –
- **mc_az** (*numpy.ndarray*) –
- **energy** (*numpy.ndarray*) – energies in TeV
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_angular_resolution_per_off_pointing_angle**(*simu_alt*, *simu_az*, *reco_alt*, *reco_az*, *alt_pointing*, *az_pointing*, *res_degree=False*, *bins=10*, *ax=None*, ***kwargs*)

Plot the angular resolution as a function of the angular separation between events true position and the pointing direction. Angles must be given in radians.

Parameters

- **simu_alt** (*numpy.ndarray*) –

- **simu_az** (*numpy.ndarray*) –
- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **alt_pointing** (*numpy.ndarray*) –
- **az_pointing** (*numpy.ndarray*) –
- **res_degree** (*bool*) – if True, the angular resolution is computed in degrees.
- **bins** (int or *numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (kwargs for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_bias_per_energy**(*simu*, *reco*, *energy*, *relative_scaling_method=None*, *ax=None*, ***kwargs*)

Plot the bias per bins of energy

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –
- **energy** (*numpy.ndarray*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*
- **ax** (*matplotlib.pyplot.axis*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axis*

ctaplot.plots.**plot_binned_stat**(*x*, *y*, *statistic='mean'*, *bins=20*, *errorbar=False*, *percentile=68.27*, *ax=None*, ***kwargs*)

Plot statistics on the quantity y binned following the quantity x. The statistic can be given by a string ('mean', 'sum', 'max'...) or a function. See *scipy.stats.binned_statistic*. Errorbars may be added and represents the dispersion (given by the percentile option) of the y distribution around the measured value in a bin. These error bars might not make sense for some statistics, it is left to the user to use the function responsibly.

Parameters

- **x** (*numpy.ndarray*) –
- **y** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **errorbar** (*bool*) –
- **statistic** (string or callable - see *scipy.stats.binned_statistic*) –
- **bins** (bins for *scipy.stats.binned_statistic*) –
- **kwargs** (if errorbar: kwargs for *matplotlib.pyplot.hlines* else: kwargs for *matplotlib.pyplot.plot*) –

Returns

Return type *matplotlib.pyplot.axes*

Examples

```
>>> from ctaplot.plots import plot_binned_stat
>>> import numpy as np
>>> x = np.random.rand(1000)
>>> y = x**2
>>> plot_binned_stat(x, y, statistic='median', bins=40, percentile=95, marker='o',
↪ linestyle='')
```

ctaplot.plots.**plot_dispersion**(*simu_x*, *reco_x*, *x_log=False*, *ax=None*, ***kwargs*)
Plot the dispersion around an expected value X_true

Parameters

- **simu_x** (*numpy.ndarray*) –
- **reco_x** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.hist2d*) –

Returns

Return type *maptlotlib.pyplot.axes*

ctaplot.plots.**plot_effective_area_cta_performance**(*cta_site*, *ax=None*, ***kwargs*)
Plot the CTA performances for the effective area

Parameters

- **cta_site** (*string*) – see *ctaplot.ana.cta_requirement*
- **ax** (*matplotlib.pyplot.axes*) –

Returns **ax**

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_effective_area_cta_requirement**(*cta_site*, *ax=None*, ***kwargs*)
Plot the CTA requirement for the effective area

Parameters

- **cta_site** (*string*) – see *ctaplot.ana.cta_requirement*
- **ax** (*matplotlib.pyplot.axes*) –

Returns **ax**

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_effective_area_per_energy**(*simu_energy*, *reco_energy*, *simulated_area*, *ax=None*, ***kwargs*)
Plot the effective area as a function of the energy

Parameters

- **simu_energy** (*numpy.ndarray* - all simulated event energies) –
- **reco_energy** (*numpy.ndarray* - all reconstructed event energies) –
- **simulated_area** (*float*) –

- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (options for *maplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

Example

```
>>> import numpy as np
>>> import ctaplot
>>> irf = ctaplot.ana.irf_cta()
>>> simu_e = 10**(-2 + 4*np.random.rand(1000))
>>> reco_e = 10**(-2 + 4*np.random.rand(100))
>>> ax = ctaplot.plots.plot_effective_area_per_energy(simu_e, reco_e, irf.
↪LaPalmaArea_prod3)
```

ctaplot.plots.**plot_effective_area_per_energy_power_law**(*emin*, *emax*, *total_number_events*, *spectral_index*, *reco_energy*, *simu_area*, *ax=None*, ***kwargs*)

Plot the effective area as a function of the energy. The effective area is computed using the *ctaplot.ana.effective_area_per_energy_power_law*.

Parameters

- **emin** (*float*) – min simulated energy
- **emax** (*float*) – max simulated energy
- **total_number_events** (*int*) – total number of simulated events
- **spectral_index** (*float*) – spectral index of the simulated power-law
- **reco_energy** (*numpy.ndarray*) – reconstructed energies
- **simu_area** (*float*) – simulated core area
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_energy_distribution**(*mc_energy*, *reco_energy*, *ax=None*, *outfile=None*, *mask_mc_detected=True*)

Plot the energy distribution of the simulated particles, detected particles and reconstructed particles The plot might be saved automatically if *outfile* is provided.

Parameters

- **mc_energy** (*Numpy 1d array of simulated energies*) –
- **reco_energy** (*Numpy 1d array of reconstructed energies*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string - output file path*) –

- **mask_mc_detected** (*Numpy 1d array - mask of detected particles for the SimuE array*) –

ctaplot.plots.**plot_energy_resolution_cta_performance**(*cta_site*, *ax=None*, ***kwargs*)
Plot the cta performances (June 2018) for the energy resolution

Parameters

- **cta_site** (*string*) – see *ctaplot.ana.cta_performance*
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_energy_resolution_cta_requirement**(*cta_site*, *ax=None*, ***kwargs*)
Plot the cta requirement for the energy resolution

Parameters

- **cta_site** (*string*) – see *ana.cta_requirement*
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_feature_importance**(*feature_keys*, *feature_importances*, *ax=None*)
Plot features importance after model training (typically from scikit-learn)

Parameters

- **feature_keys** (*list of string*) –
- **feature_importances** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –

Returns

Return type ax

ctaplot.plots.**plot_field_of_view_map**(*reco_alt*, *reco_az*, *source_alt*, *source_az*, *energies=None*, *ax=None*, *outfile=None*)
Plot a map in angles [in degrees] of the photons seen by the telescope (after reconstruction)

Parameters

- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **source_alt** (*float, source Altitude*) –
- **source_az** (*float, source Azimuth*) –
- **energies** (*numpy.ndarray* - if given, set the colorbar) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string - if None, the plot is not saved*) –

Returns ax

Return type *matplitlib.pyplot.axes*

ctaplot.plots.**plot_impact_map**(*impact_x*, *impact_y*, *tel_x*, *tel_y*, *tel_types=None*, *ax=None*, *Outfile='ImpactMap.png'*, *hist_kwargs={}*, *scatter_kwargs={}*)

Map of the site with telescopes positions and impact points heatmap

Parameters

- **impact_x** (*numpy.ndarray*) –
- **impact_y** (*numpy.ndarray*) –
- **tel_x** (*numpy.ndarray*) –
- **tel_y** (*numpy.ndarray*) –
- **tel_types** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **hist_kwargs** (*kwargs* for *matplotlib.pyplot.hist*) –
- **scatter_kwargs** (*kwargs* for *matplotlib.pyplot.scatter*) –
- **Outfile** (*string - name of the output file*) –

ctaplot.plots.**plot_impact_parameter_error_per_energy**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *energy*, *ax=None*, ***kwargs*)

Deprecated since version 18/08/2019: *plot_impact_parameter_error_per_energy* will be removed in a future release.Use *plot_impact_parameter_resolution_per_energy* instead

plot the impact parameter error distance as a function of energy and save the plot as Outfile :param reco_x: :type reco_x: *numpy.ndarray* :param reco_y: :type reco_y: *numpy.ndarray* :param simu_x: :type simu_x: *numpy.ndarray* :param simu_y: :type simu_y: *numpy.ndarray* :param energy: :type energy: *numpy.ndarray* :param ax: :type ax: *matplotlib.pyplot.axes* :param kwargs: :type kwargs: args for *matplotlib.pyplot.errorbar*

Returns **energy, err_mean**

Return type numpy arrays

ctaplot.plots.**plot_impact_parameter_error_per_multiplicity**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *multiplicity*, *max_mult=None*, *ax=None*, ***kwargs*)

Plot the impact parameter error as a function of multiplicity TODO: refactor and clean code

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **multiplicity** (*numpy.ndarray*) –
- **max_mult** (*optional, max multiplicity - float*) –
- **ax** (*matplotlib.pyplot.axes*) –

ctaplot.plots.**plot_impact_parameter_error_site_center**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *ax=None*, ***kwargs*)

Plot the impact parameter error as a function of the distance to the site center.

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (kwargs for *matplotlib.pyplot.hist2d*) –

Returns

Return type ax

ctaplot.plots.**plot_impact_parameter_resolution_per_energy**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *energy*, *ax=None*, ***kwargs*)

Parameters

- **reco_x** –
- **reco_y** –
- **simu_x** –
- **simu_y** –
- **energy** –
- **ax** –
- **kwargs** –

ctaplot.plots.**plot_impact_point_heatmap**(*reco_x*, *reco_y*, *ax=None*, *outfile=None*)

Plot the heatmap of the impact points on the site ground and save it under Outfile

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string*) –

ctaplot.plots.**plot_impact_point_map_distri**(*reco_x*, *reco_y*, *tel_x*, *tel_y*, *fit=False*, *outfile=None*, ***kwargs*)

Map and distributions of the reconstructed impact points.

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **tel_x** (*numpy.ndarray*) – X positions of the telescopes
- **tel_y** (*numpy.ndarray*) – Y positions of the telescopes
- **kde** (*bool - if True, makes a gaussian fit of the point density*) –
- **outfile** (*'str' - save a png image of the plot under 'string.png'*) –

Returns fig

Return type *matplotlib.pyplot.figure*

ctaplot.plots.**plot_impact_resolution_per_energy**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *simu_energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *ax=None*, ***kwargs*)

Plot the angular resolution as a function of the energy

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*float*) –
- **simu_y** (*float*) –
- **simu_energy** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_migration_matrix**(*x*, *y*, *ax=None*, *colorbar=False*, *xy_line=False*, *hist2d_args={}*, *line_args={}*)

Make a simple plot of a migration matrix

Parameters

- **x** (list or *numpy.ndarray*) –
- **y** (list or *numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **colorbar** (*matplotlib.colorbar*) –
- **hist2d_args** (dict, args for *matplotlib.pyplot.hist2d*) –
- **line_args** (dict, args for *matplotlib.pyplot.plot*) –

Returns

Return type *matplotlib.pyplot.axes*

Examples

```
>>> from ctaplot.plots import plot_migration_matrix
>>> import matplotlib
>>> x = np.random.rand(10000)
>>> y = x**2
>>> plot_migration_matrix(x, y, colorbar=True, hist2d_args=dict(norm=matplotlib.
↪colors.LogNorm()))
In this example, the colorbar will be log normed
```

ctaplot.plots.**plot_multiplicity_per_energy**(*multiplicity*, *energies*, *ax=None*, *outfile=None*)

Plot the telescope multiplicity as a function of the energy The plot might be saved automatically if *outfile* is provided.

Parameters

- **multiplicity** (*numpy.ndarray*) –
- **energies** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string*) –

ctaplot.plots.**plot_resolution_per_energy**(*reco*, *simu*, *energy*, *ax=None*, ***kwargs*)

Plot a variable resolution as a function of the energy

Parameters

- **reco** (*numpy.ndarray*) –
- **simu** (*numpy.ndarray*) –
- **energy** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_sensitivity_cta_performance**(*cta_site*, *ax=None*, ***kwargs*)

Plot the CTA performances for the sensitivity

Parameters

- **cta_site** (string - see *ctaplot.ana.cta_requirement*) –
- **ax** (*matplotlib.pyplot.axes*, optional) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_sensitivity_cta_requirement**(*cta_site*, *ax=None*, ***kwargs*)

Plot the CTA requirement for the sensitivity :param cta_site: :type cta_site: string - see *ctaplot.ana.cta_requirement* :param ax: :type ax: *matplotlib.pyplot.axes*, optional

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_theta2**(*reco_alt*, *reco_az*, *mc_alt*, *mc_az*, *ax=None*, ***kwargs*)

Plot the theta2 distribution and display the corresponding angular resolution in degrees. The input must be given in radians.

Parameters

- **reco_alt** (*numpy.ndarray* - reconstructed altitude angle in radians) –
- **reco_az** (*numpy.ndarray* - reconstructed azimuth angle in radians) –
- **mc_alt** (*numpy.ndarray* - true altitude angle in radians) –
- **mc_az** (*numpy.ndarray* - true azimuth angle in radians) –
- **ax** (*matplotlib.pyplot.axes*) –
- ****kwargs** (options for *matplotlib.pyplot.hist*) –

Returns ax

Return type *matplotlib.pyplot.axes*

1.3.2 ana.py

Contain mathematical functions to make results analysis (compute angular resolution, effective surface, energy resolution...)

class ctaplot.ana.**irf_cta**
Bases: object

Class to handle Instrument Response Function data

set_E_bin(*E_bin*)

class ctaplot.ana.**cta_performance**(*site*)
Bases: object

get_angular_resolution()

get_effective_area(*observation_time=50*)
Return the effective area at the given observation time in hours. NB: Only 50h supported Returns the energy array and the effective area array :param observation_time: :type observation_time: optional

Returns

Return type *numpy.ndarray*, *numpy.ndarray*

get_energy_resolution()

get_sensitivity(*observation_time=50*)

class ctaplot.ana.**cta_requirement**(*site*)
Bases: object

get_angular_resolution()

get_effective_area(*observation_time=50*)
Return the effective area at the given observation time in hours. NB: Only 0.5h supported Returns the energy array and the effective area array :param observation_time: :type observation_time: optional

Returns

Return type *numpy.ndarray*, *numpy.ndarray*

get_energy_resolution()

get_sensitivity(*observation_time=50*)

ctaplot.ana.**stat_per_energy**(*energy*, *y*, *statistic='mean'*)
Return statistic for the given quantity per energy bins. The binning is given by irf_cta

Parameters

- **energy** (*numpy.ndarray* (1d)) – event energies
- **y** (*numpy.ndarray* (1d)) –
- **statistic** (*string*) – see *scipy.stat.binned_statistic*

Returns bin_stat, bin_edges, binnumber

Return type *numpy.ndarray, numpy.ndarray, numpy.ndarray*

ctaplot.ana.**bias**(*simu*, *reco*)
Compute the bias of a reconstructed variable as *median(reco-simu)*

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –

Returns

Return type float

ctaplot.ana.**relative_bias**(*simu*, *reco*, *relative_scaling_method='s1'*)

Compute the relative bias of a reconstructed variable as *median(reco-simu)/relative_scaling(simu, reco)*

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*

ctaplot.ana.**relative_scaling**(*simu*, *reco*, *method='s0'*)

Define the relative scaling for the relative error calculation. There are different ways to calculate this scaling factor. The easiest and most spread one is simply *np.abs(simu)*. However this is possible only when *simu != 0*. Possible methods:

- None or 's0': scale = 1
- 's1': *scale = np.abs(simu)*
- 's2': *scale = np.abs(reco)*
- 's3': *scale = (np.abs(simu) + np.abs(reco))/2.*
- 's4': *scale = np.max([np.abs(reco), np.abs(simu)], axis=0)*

This method is not exposed but kept for tests and future reference. The *s1* method is used in all *ctaplot* functions.

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –

Returns

Return type *numpy.ndarray*

ctaplot.ana.**angular_resolution**(*reco_alt*, *reco_az*, *simu_alt*, *simu_az*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Compute the angular resolution as the Qth (standard being 68) containment radius of theta2 with lower and upper limits on this value corresponding to the confidence value required (1.645 for 95% confidence)

Parameters

- **reco_alt** (*numpy.ndarray* - reconstructed altitude angle in radians) –
- **reco_az** (*numpy.ndarray* - reconstructed azimuth angle in radians) –
- **simu_alt** (*numpy.ndarray* - true altitude angle in radians) –
- **simu_az** (*numpy.ndarray* - true azimuth angle in radians) –
- **percentile** (*float - percentile, 68 corresponds to one sigma*) –
- **confidence_level** (*float*) –

Returns

Return type *numpy.array* [angular_resolution, lower limit, upper limit]

ctaplot.ana.**angular_separation_altaz**(*alt1*, *az1*, *alt2*, *az2*, *unit='rad'*)

Compute the angular separation in radians or degrees between two pointing direction given with alt-az

Parameters

- **alt1** (1d *numpy.ndarray*, altitude of the first pointing direction) –
- **az1** (1d *numpy.ndarray* azimuth of the first pointing direction) –
- **alt2** (1d *numpy.ndarray*, altitude of the second pointing direction) –
- **az2** (1d *numpy.ndarray*, azimuth of the second pointing direction) –
- **unit** (*'deg' or 'rad'*) –

Returns

Return type 1d *numpy.ndarray* or float, angular separation

ctaplot.ana.**angular_resolution_per_bin**(*simu_alt*, *simu_az*, *reco_alt*, *reco_az*, *x*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *bins=10*)

Compute the angular resolution per binning of x

Parameters

- **simu_alt** (*numpy.ndarray*) –
- **simu_az** (*numpy.ndarray*) –
- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **x** (*numpy.ndarray*) –
- **percentile** (*float*) – 0 < percentile < 100
- **confidence_level** (*float*) – 0 < confidence_level < 1
- **bias_correction** (*bool*) –
- **bins** (int or *numpy.ndarray*) –

Returns bins: 1D *numpy.ndarray* ang_res: 2D *numpy.ndarray*

Return type bins, ang_res

ctaplot.ana.**angular_resolution_per_energy**(*reco_alt*, *reco_az*, *simu_alt*, *simu_az*, *energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Plot the angular resolution as a function of the event simulated energy

Parameters

- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **simu_alt** (*numpy.ndarray*) –
- **simu_az** (*numpy.ndarray*) –
- **energy** (*numpy.ndarray*) –
- ****kwargs** (args for *angular_resolution*) –

Returns (E, RES)

Return type (1d numpy array, 1d numpy array) = Energies, Resolution

ctaplot.ana.**angular_resolution_per_off_pointing_angle**(*simu_alt*, *simu_az*, *reco_alt*, *reco_az*, *alt_pointing*, *az_pointing*, *bins=10*)

Compute the angular resolution as a function of separation angle for the pointing direction

Parameters

- **simu_alt** (*numpy.ndarray*) –
- **simu_az** (*numpy.ndarray*) –
- **reco_alt** (*numpy.ndarray*) –
- **reco_az** (*numpy.ndarray*) –
- **alt_pointing** (*numpy.ndarray*) –
- **az_pointing** (*numpy.ndarray*) –
- **bins** (float or *numpy.ndarray*) –

Returns bins: 1D *numpy.ndarray* res: 2D *numpy.ndarray* - resolutions with confidence intervals (output from *ctaplot.ana.resolution*)

Return type (bins, res)

ctaplot.ana.**energy_resolution**(*true_energy*, *reco_energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Compute the energy resolution of reco_energy as the percentile (68 as standard) containment radius of ‘true_energy-reco_energy)/simu_energy with the lower and upper confidence limits defined by the given confidence level

Parameters

- **true_energy** (*1d numpy array of simulated energies*) –
- **reco_energy** (*1d numpy array of reconstructed energies*) –
- **percentile** (*float*) – <= 100

Returns

Return type *numpy.array* - [energy_resolution, lower_confidence_limit, upper_confidence_limit]

ctaplot.ana.**energy_bias**(*simu_energy*, *reco_energy*)

Compute the energy relative bias per energy bin.

Parameters

- **simu_energy** (*1d numpy array of simulated energies*) –
- **reco_energy** (*1d numpy array of reconstructed energies*) –

Returns (energy_bins, bias)

Return type tuple of 1d numpy arrays - energy, energy bias

ctaplot.ana.**energy_resolution_per_energy**(*simu_energy*, *reco_energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Parameters

- **simu_energy** (*1d numpy array of simulated energies*) –
- **reco_energy** (*1d numpy array of reconstructed energies*) –

Returns (e, e_res)

Return type tuple of 1d numpy arrays - energy, resolution in energy

ctaplot.ana.**bias_per_energy**(*simu*, *reco*, *energy*, *relative_scaling_method=None*)

Bias between *simu* and *reco* per bins of energy

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –
- **energy** (: *numpy.ndarray*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*

Returns **bins, bias**

Return type *numpy.ndarray, numpy.ndarray*

ctaplot.ana.**resolution_per_bin**(*x*, *y_true*, *y_reco*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method=None*, *bins=10*)

Resolution of y as a function of binned x.

Parameters

- **x** (*numpy.ndarray*) –
- **y_true** (*numpy.ndarray*) –
- **y_reco** (*numpy.ndarray*) –
- **percentile** (*float*) –
- **confidence_level** (*float*) –
- **bias_correction** (*bool*) –
- **relative_scaling_method** (see *ctaplot.ana.relative_scaling*) –
- **bins** (int or *numpy.ndarray* (see *numpy.histogram*)) –

Returns **(x_bins, res)** – x_bins: bins for x res: resolutions with confidence level intervals for each bin

Return type (*numpy.ndarray*, *numpy.ndarray*)

ctaplot.ana.**resolution**(*simu*, *reco*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method='s1'*)

Compute the resolution of reco as the Qth (68.27 as standard = 1 sigma) containment radius of *(simu-reco)/relative_scaling* with the lower and upper confidence limits defined the values inside

the error_percentile

Parameters

- **simu** (*numpy.ndarray* (1d)) – simulated quantity
- **reco** (*numpy.ndarray* (1d)) – reconstructed quantity
- **percentile** (*float*) – percentile for the resolution containment radius
- **error_percentile** (*float*) – percentile for the confidence limits
- **bias_correction** (*bool*) – if True, the resolution is corrected with the bias computed on simu and reco
- **relative_scaling** (*str*) – see *ctaplot.ana.relative_scaling*

Returns

Return type *numpy.ndarray* - [resolution, lower_confidence_limit, upper_confidence_limit]

ctaplot.ana.**resolution_per_energy**(*simu*, *reco*, *simu_energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Parameters

- **simu** (1d *numpy.ndarray* of simulated energies) –
- **reco** (1d *numpy.ndarray* of reconstructed energies) –

Returns energy_bins - 1D *numpy.ndarray* resolution: - 3D *numpy.ndarray* see *ctaplot.ana.resolution*

Return type (energy_bins, resolution)

ctaplot.ana.**impact_resolution_per_energy**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *energy*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method=None*)

Plot the angular resolution as a function of the event simulated energy

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **energy** (*numpy.ndarray*) –
- **percentile** (*float*) – see *ctaplot.ana.resolution*
- **confidence_level** (*float*) – see *ctaplot.ana.resolution*
- **bias_correction** (*bool*) – see *ctaplot.ana.resolution*
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*

Returns (energy, resolution)

Return type (1d numpy array, 1d numpy array)

ctaplot.ana.**impact_parameter_error**(*reco_x*, *reco_y*, *simu_x*, *simu_y*)

compute the error distance between simulated and reconstructed impact parameters :param reco_x: :type reco_x: 1d numpy array :param reco_y: :type reco_y: 1d numpy array :param simu_x: :type simu_x: 1d numpy array :param simu_y: :type simu_y: 1d numpy array

Returns 1d numpy array

Return type distances

ctaplot.ana.**impact_resolution**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method=None*)

Compute the shower impact parameter resolution as the Qth (68 as standard) containment radius of the square distance to the simulated one with the lower and upper limits corresponding to the required confidence level

Parameters

- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –

- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **percentile** (*float*) – see *ctaplot.ana.resolution*
- **confidence_level** (*float*) – see *ctaplot.ana.resolution*
- **bias_correction** (*bool*) – see *ctaplot.ana.resolution*
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*

Returns (impact_resolution, lower_confidence_level, upper_confidence_level)

Return type (*numpy.array*, *numpy.array*, *numpy.array*)

ctaplot.ana.**distance2d_resolution**(*reco_x*, *reco_y*, *simu_x*, *simu_y*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method=None*)

Compute the 2D distance resolution as the Qth (standard being 68) containment radius of the relative distance with lower and upper limits on this value corresponding to the confidence value required (1.645 for 95% confidence)

Parameters

- **reco_x** (*numpy.ndarray* - reconstructed x position) –
- **reco_y** (*numpy.ndarray* - reconstructed y position) –
- **simu_x** (*numpy.ndarray* - true x position) –
- **simu_y** (*numpy.ndarray* - true y position) –
- **percentile** (*float - percentile, 68.27 corresponds to one sigma*) –
- **confidence_level** (*float*) –
- **bias_correction** (*bool*) –
- **relative_scaling_method** (*str*) –
  - see *ctaplot.ana.relative_scaling*

Returns

Return type *numpy.array* [angular_resolution, lower limit, upper limit]

ctaplot.ana.**distance2d_resolution_per_bin**(*x*, *reco_x*, *reco_y*, *simu_x*, *simu_y*, *bins=10*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *relative_scaling_method=None*)

Compute the 2D distance per bin of x

Parameters

- **x** (*numpy.ndarray*) –
- **reco_x** (*numpy.ndarray*) –
- **reco_y** (*numpy.ndarray*) –
- **simu_x** (*numpy.ndarray*) –
- **simu_y** (*numpy.ndarray*) –
- **bins** (bins args of *np.histogram*) –

- **percentile** (*float - percentile, 68.27 corresponds to one sigma*) –
- **confidence_level** (*float*) –
- **bias_correction** (*bool*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*

Returns

Return type x_bins, distance_res

ctaplot.ana.**power_law_integrated_distribution**(*xmin*, *xmax*, *total_number_events*, *spectral_index*, *bins*)

For each bin, return the expected number of events for a power-law distribution. bins: *numpy.ndarray*, e.g. *np.logspace(np.log10(emin), np.logspace(xmax))*

Parameters

- **xmin** (*float*, min of the simulated power-law) –
- **xmax** (*float*, max of the simulated power-law) –
- **total_number_events** (*int*) –
- **spectral_index** (*float*) –
- **bins** (*numpy.ndarray*) –

Returns y

Return type *numpy.ndarray*, len(y) = len(bins) - 1

ctaplot.ana.**effective_area**(*simu_energy*, *reco_energy*, *simu_area*)

Compute the effective area from a list of simulated energies and reconstructed energies :param simu_energy: :type simu_energy: 1d numpy array :param reco_energy: :type reco_energy: 1d numpy array :param simu_area: :type simu_area: float - area on which events are simulated

Returns

Return type float = effective area

ctaplot.ana.**effective_area_per_energy**(*simu_energy*, *reco_energy*, *simu_area*)

Compute the effective area per energy bins from a list of simulated energies and reconstructed energies

Parameters

- **simu_energy** (*1d numpy array*) –
- **reco_energy** (*1d numpy array*) –
- **simu_area** (*float - area on which events are simulated*) –

Returns (E, Seff)

Return type (1d numpy array, 1d numpy array)

ctaplot.ana.**effective_area_per_energy_power_law**(*emin*, *emax*, *total_number_events*, *spectral_index*, *reco_energy*, *simu_area*)

Compute the effective area per energy bins from a list of simulated energies and reconstructed energies

Parameters

- **emin** (*float*) –
- **emax** (*float*) –

- **total_number_events** (*int*) –
- **spectral_index** (*float*) –
- **reco_energy** (*1d numpy array*) –
- **simu_area** (*float - area on which events are simulated*) –

Returns (energy, effective_area)

Return type (1d numpy array, 1d numpy array)

ctaplot.ana.**bias_per_bin**(*simu*, *reco*, *x*, *relative_scaling_method=None*, *bins=10*)
Bias between *simu* and *reco* per bin of *x*.

Parameters

- **simu** (*numpy.ndarray*) –
- **reco** (*numpy.ndarray*) –
- **x** (: *numpy.ndarray*) –
- **relative_scaling_method** (*str*) – see *ctaplot.ana.relative_scaling*
- **bins** (bins for *numpy.histogram*) –

Returns bins, bias

Return type *numpy.ndarray, numpy.ndarray*

ctaplot.ana.**percentile_confidence_interval**(*x*, *percentile=68*, *confidence_level=0.95*)
Return the confidence interval for the qth percentile of x for a given confidence level

REF: http://people.stat.sfu.ca/~cschwarz/Stat-650/Notes/PDF/ChapterPercentiles.pdf S. Chakraborti and J. Li, Confidence Interval Estimation of a Normal Percentile, doi:10.1198/000313007X244457

Parameters

- **x** (*numpy.ndarray*) –
- **percentile** (*float*) – 0 < percentile < 100
- **confidence_level** (*float*) – 0 < confidence level (by default 95%) < 1

ctaplot.ana.**logbin_mean**(*x_bin*)
Function that gives back the mean of each bin in logscale

Parameters **x_bin** (*numpy.ndarray*) –

Returns

Return type *numpy.ndarray*

ctaplot.ana.**binned_statistic**(*x*, *values*, *statistic='mean'*, *bins=10*, *range=None*)
Compute a binned statistic for one or more sets of data.

This is a generalization of a histogram function. A histogram divides the space into bins, and returns the count of the number of points in each bin. This function allows the computation of the sum, mean, median, or other statistic of the values (or set of values) within each bin.

Parameters

- **x** (*(N,) array_like*) – A sequence of values to be binned.
- **values** (*(N,) array_like or list of (N,) array_like*) – The data on which the statistic will be computed. This must be the same shape as *x*, or a set of sequences

- each the same shape as *x*. If *values* is a set of sequences, the statistic will be computed on each independently.

- **statistic** (*string or callable, optional*) – The statistic to compute (default is 'mean'). The following statistics are available:
  - 'mean' : compute the mean of values for points within each bin. Empty bins will be represented by NaN.
  - 'std' : compute the standard deviation within each bin. This is implicitly calculated with ddof=0.
  - 'median' : compute the median of values for points within each bin. Empty bins will be represented by NaN.
  - 'count' : compute the count of points within each bin. This is identical to an unweighted histogram. *values* array is not referenced.
  - 'sum' : compute the sum of values for points within each bin. This is identical to a weighted histogram.
  - 'min' : compute the minimum of values for points within each bin. Empty bins will be represented by NaN.
  - 'max' : compute the maximum of values for point within each bin. Empty bins will be represented by NaN.
  - function : a user-defined function which takes a 1D array of values, and outputs a single numerical statistic. This function will be called on the values in each bin. Empty bins will be represented by function([]), or NaN if this returns an error.
- **bins** (*int or sequence of scalars, optional*) – If *bins* is an int, it defines the number of equal-width bins in the given range (10 by default). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths. Values in *x* that are smaller than lowest bin edge are assigned to bin number 0, values beyond the highest bin are assigned to `bins[-1]`. If the bin edges are specified, the number of bins will be, (nx = len(bins)-1).
- **range** (*(float, float) or [(float, float)], optional*) – The lower and upper range of the bins. If not provided, range is simply `(x.min(), x.max())`. Values outside the range are ignored.

Returns

- **statistic** (*array*) – The values of the selected statistic in each bin.
- **bin_edges** (*array of dtype float*) – Return the bin edges `(length(statistic)+1)`.
- **binnumber** (*1-D ndarray of ints*) – Indices of the bins (corresponding to *bin_edges*) in which each value of *x* belongs. Same length as *values*. A binnumber of *i* means the corresponding value is between (bin_edges[i-1], bin_edges[i]).

See also:

`numpy.digitize()`, `numpy.histogram()`, `binned_statistic_2d()`, `binned_statistic_dd()`

Notes

All but the last (righthand-most) bin is half-open. In other words, if *bins* is `[1, 2, 3, 4]`, then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

New in version 0.11.0.

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

First some basic examples:

Create two evenly spaced bins in the range of the given sample, and sum the corresponding values in each of those bins:

```
>>> values = [1.0, 1.0, 2.0, 1.5, 3.0]
>>> stats.binned_statistic([1, 1, 2, 5, 7], values, 'sum', bins=2)
BinnedStatisticResult(statistic=array([4. , 4.5]), bin_edges=array([1., 4., 7.]),␣
↪binnumber=array([1, 1, 1, 2, 2]))
```

Multiple arrays of values can also be passed. The statistic is calculated on each set independently:

```
>>> values = [[1.0, 1.0, 2.0, 1.5, 3.0], [2.0, 2.0, 4.0, 3.0, 6.0]]
>>> stats.binned_statistic([1, 1, 2, 5, 7], values, 'sum', bins=2)
BinnedStatisticResult(statistic=array([[4. , 4.5],
       [8. , 9. ]]), bin_edges=array([1., 4., 7.]), binnumber=array([1, 1, 1, 2,␣
↪2]))
```

```
>>> stats.binned_statistic([1, 2, 1, 2, 4], np.arange(5), statistic='mean',
...                        bins=3)
BinnedStatisticResult(statistic=array([1., 2., 4.]), bin_edges=array([1., 2., 3.,␣
↪4.]), binnumber=array([1, 2, 1, 2, 3]))
```

As a second example, we now generate some random data of sailing boat speed as a function of wind speed, and then determine how fast our boat is for certain wind speeds:

```
>>> windspeed = 8 * np.random.rand(500)
>>> boatspeed = .3 * windspeed**.5 + .2 * np.random.rand(500)
>>> bin_means, bin_edges, binnumber = stats.binned_statistic(windspeed,
...                 boatspeed, statistic='median', bins=[1,2,3,4,5,6,7])
>>> plt.figure()
>>> plt.plot(windspeed, boatspeed, 'b.', label='raw data')
>>> plt.hlines(bin_means, bin_edges[:-1], bin_edges[1:], colors='g', lw=5,
...            label='binned statistic of data')
>>> plt.legend()
```

Now we can use `binnumber` to select all datapoints with a windspeed below 1:

```
>>> low_boatspeed = boatspeed[binnumber == 0]
```

As a final example, we will use `bin_edges` and `binnumber` to make a plot of a distribution that shows the mean and distribution around that mean per bin, on top of a regular histogram and the probability distribution function:

```
>>> x = np.linspace(0, 5, num=500)
>>> x_pdf = stats.maxwell.pdf(x)
>>> samples = stats.maxwell.rvs(size=10000)
```

```
>>> bin_means, bin_edges, binnumber = stats.binned_statistic(x, x_pdf,
...         statistic='mean', bins=25)
>>> bin_width = (bin_edges[1] - bin_edges[0])
>>> bin_centers = bin_edges[1:] - bin_width/2
```

```
>>> plt.figure()
>>> plt.hist(samples, bins=50, density=True, histtype='stepfilled',
...          alpha=0.2, label='histogram of data')
>>> plt.plot(x, x_pdf, 'r-', label='analytical pdf')
>>> plt.hlines(bin_means, bin_edges[:-1], bin_edges[1:], colors='g', lw=2,
...            label='binned statistic of data')
>>> plt.plot((binnumber - 0.5) * bin_width, x_pdf, 'g.', alpha=0.5)
>>> plt.legend(fontsize=10)
>>> plt.show()
```

1.4 Examples

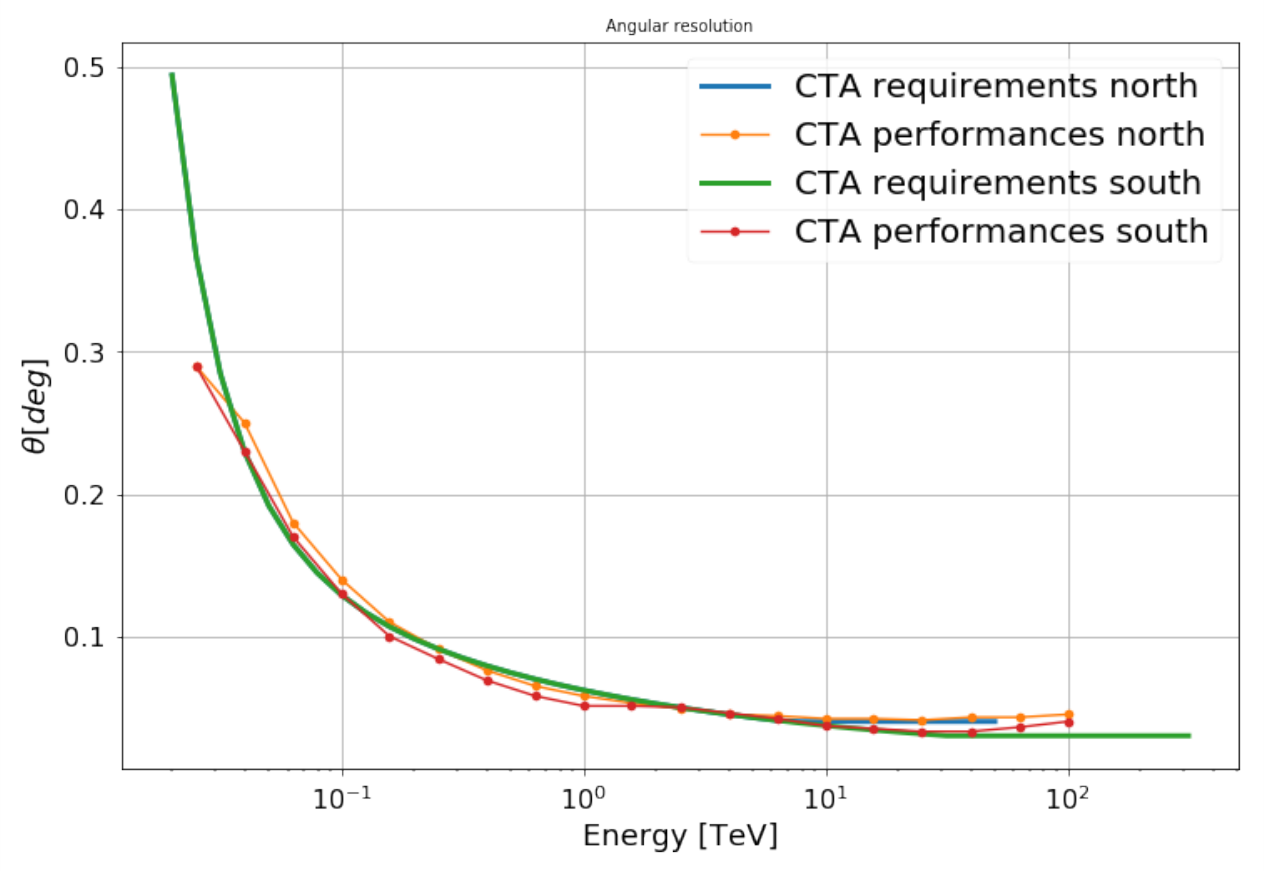
1.4.1 How to easily plot CTA IRF requirements and performances

CTA performances are up-to-date and public and can be found on the cta-observatory website

[1]:
```
import ctaplot
from ctaplot.dataset import get
import numpy as np
import matplotlib.pyplot as plt
import matplotlib

%matplotlib inline
font = {'size'   : 20}
matplotlib.rc('font', **font)
```

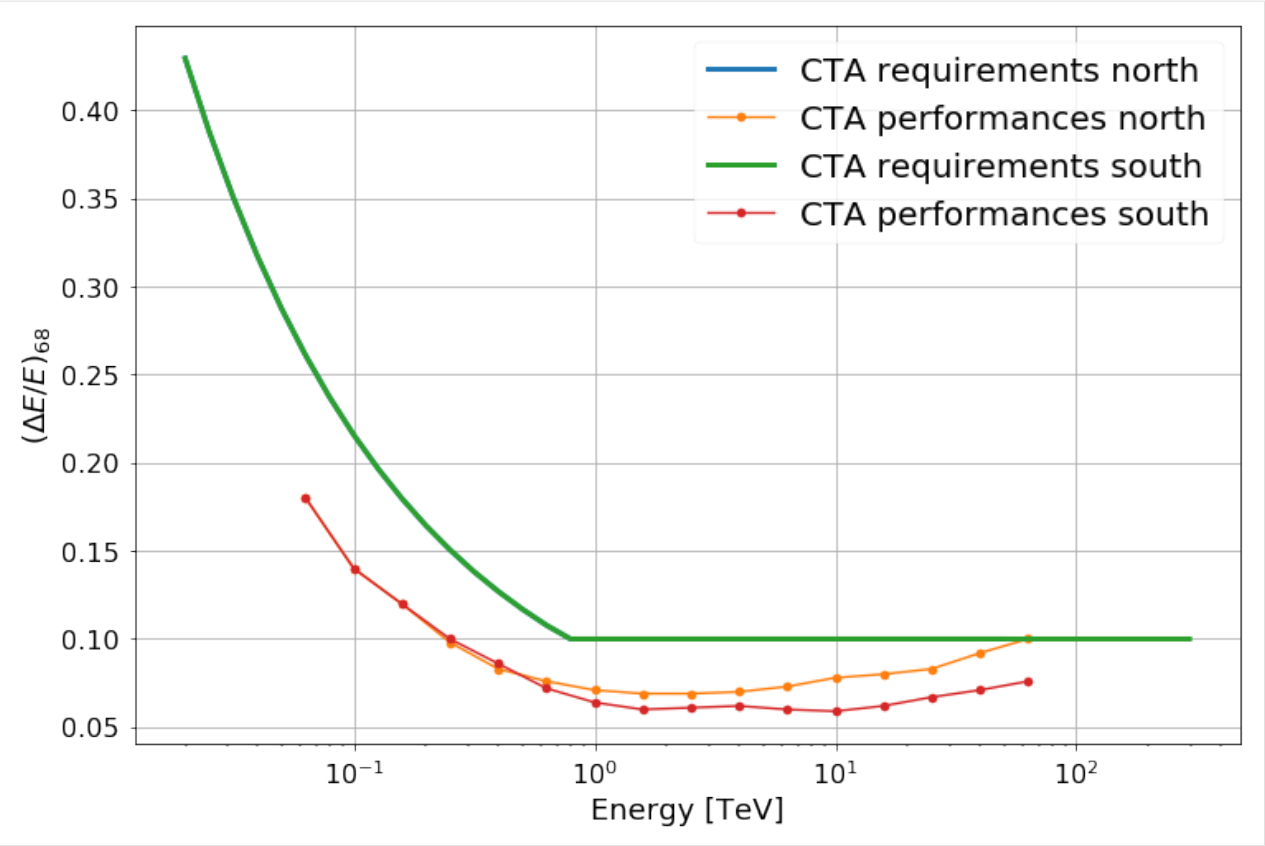
Angular resolution

[3]:
```
fig, ax = plt.subplots(figsize=(12,8))
ax = ctaplot.plot_angular_res_requirement('north', ax=ax, linewidth=3)
ax = ctaplot.plot_angular_res_cta_performance('north', ax=ax, marker='o')
ax = ctaplot.plot_angular_res_requirement('south', ax=ax,  linewidth=3)
ax = ctaplot.plot_angular_res_cta_performance('south', ax=ax, marker='o')
ax.grid()
plt.legend(prop = font);
```



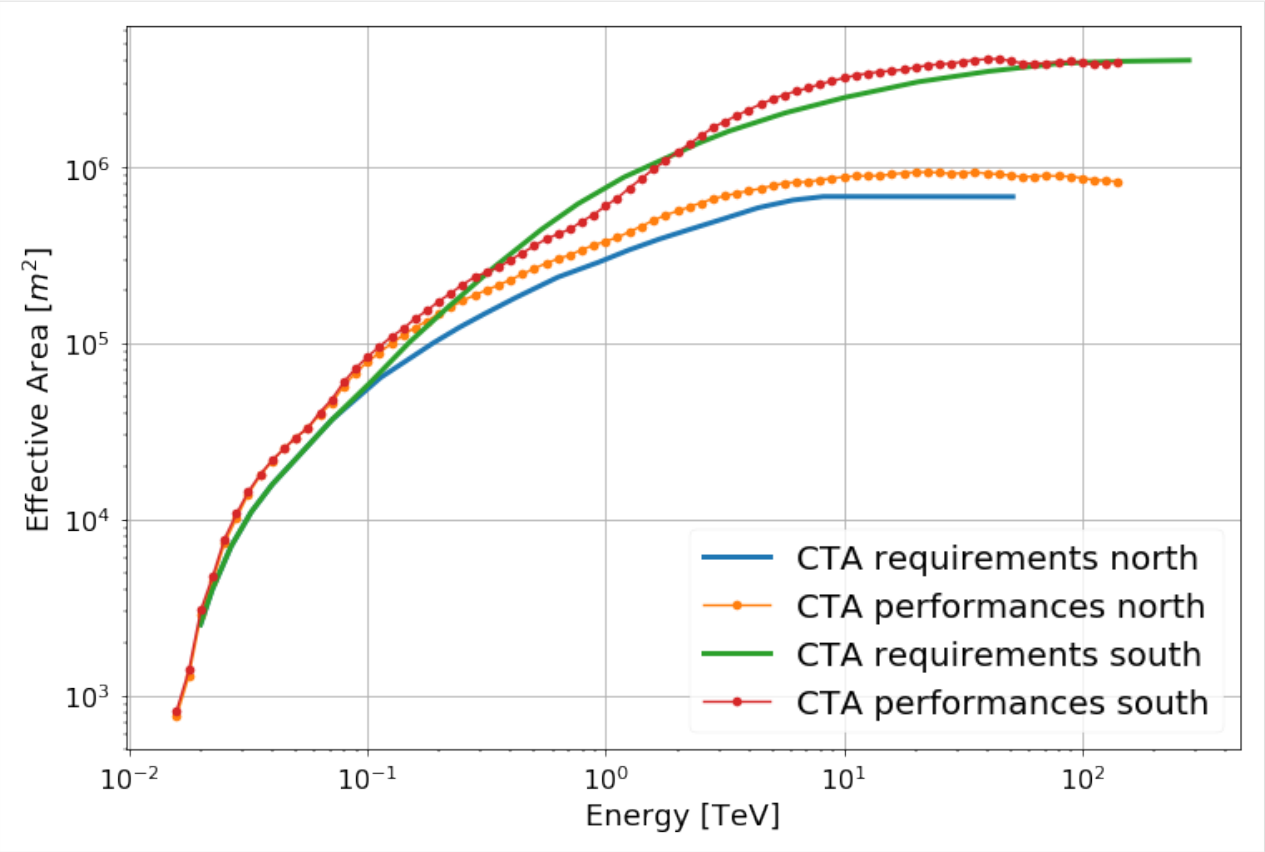
Energy resolution

[5]:
```
fig, ax = plt.subplots(figsize=(12,8))
ax = ctaplot.plot_energy_resolution_requirements('north', ax=ax, linewidth=3)
ax = ctaplot.plot_energy_resolution_cta_performances('north', ax=ax, marker='o')
ax = ctaplot.plot_energy_resolution_requirements('south', ax=ax,  linewidth=3)
ax = ctaplot.plot_energy_resolution_cta_performances('south', ax=ax, marker='o')
ax.grid()
plt.legend(prop = font);
```



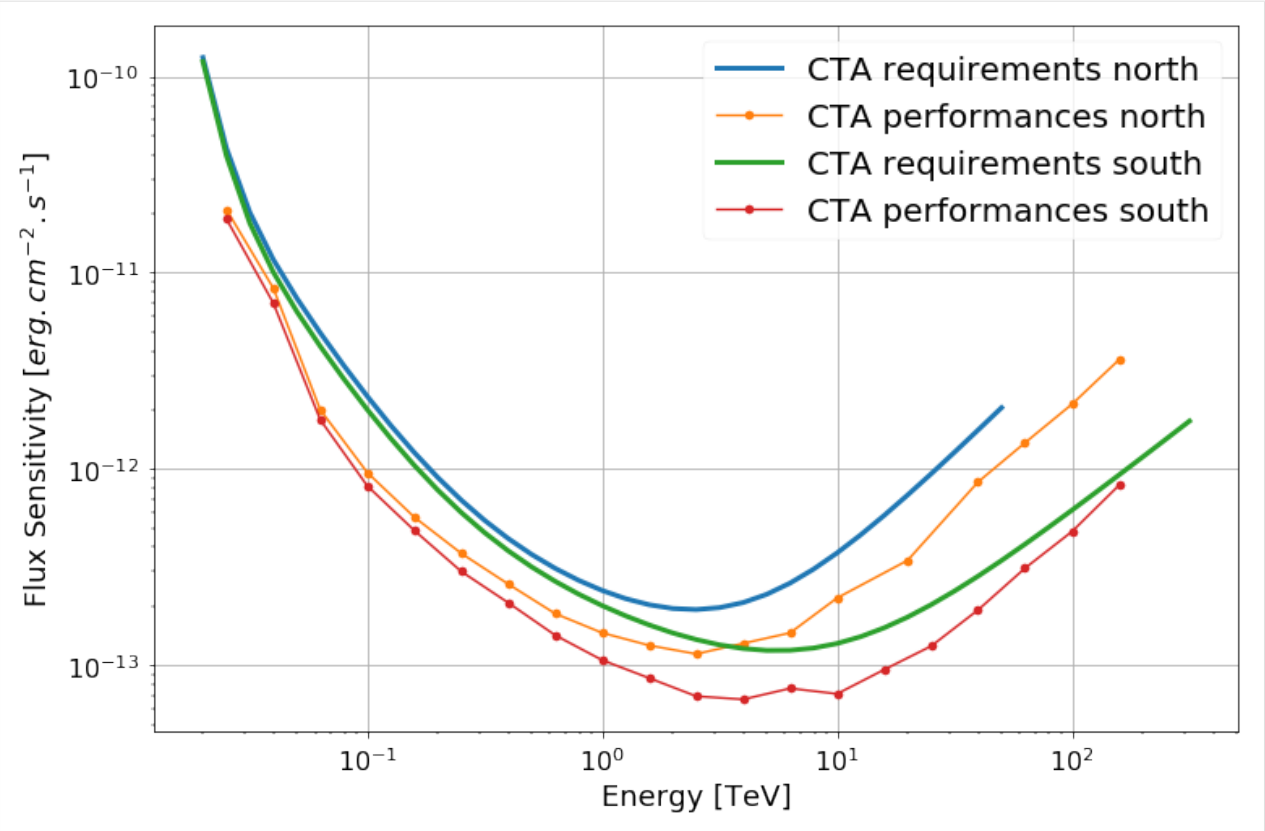
Effective Area

[7]:
```
fig, ax = plt.subplots(figsize=(12,8))
ax = ctaplot.plot_effective_area_requirement('north', ax=ax, linewidth=3)
ax = ctaplot.plot_effective_area_performances('north', ax=ax, marker='o')
ax = ctaplot.plot_effective_area_requirement('south', ax=ax,  linewidth=3)
ax = ctaplot.plot_effective_area_performances('south', ax=ax, marker='o')
ax.grid()
plt.legend(prop = font);
```



Sensitivity

[8]:
```
fig, ax = plt.subplots(figsize=(12,8))
ax = ctaplot.plot_sensitivity_requirement('north', ax=ax, linewidth=3)
ax = ctaplot.plot_sensitivity_performances('north', ax=ax, marker='o')
ax = ctaplot.plot_sensitivity_requirement('south', ax=ax,  linewidth=3)
ax = ctaplot.plot_sensitivity_performances('south', ax=ax, marker='o')
ax.set_ylabel(r'Flux Sensitivity $[erg.cm^{-2}.s^{-1}]$')
ax.grid()
plt.legend(prop = font);
```

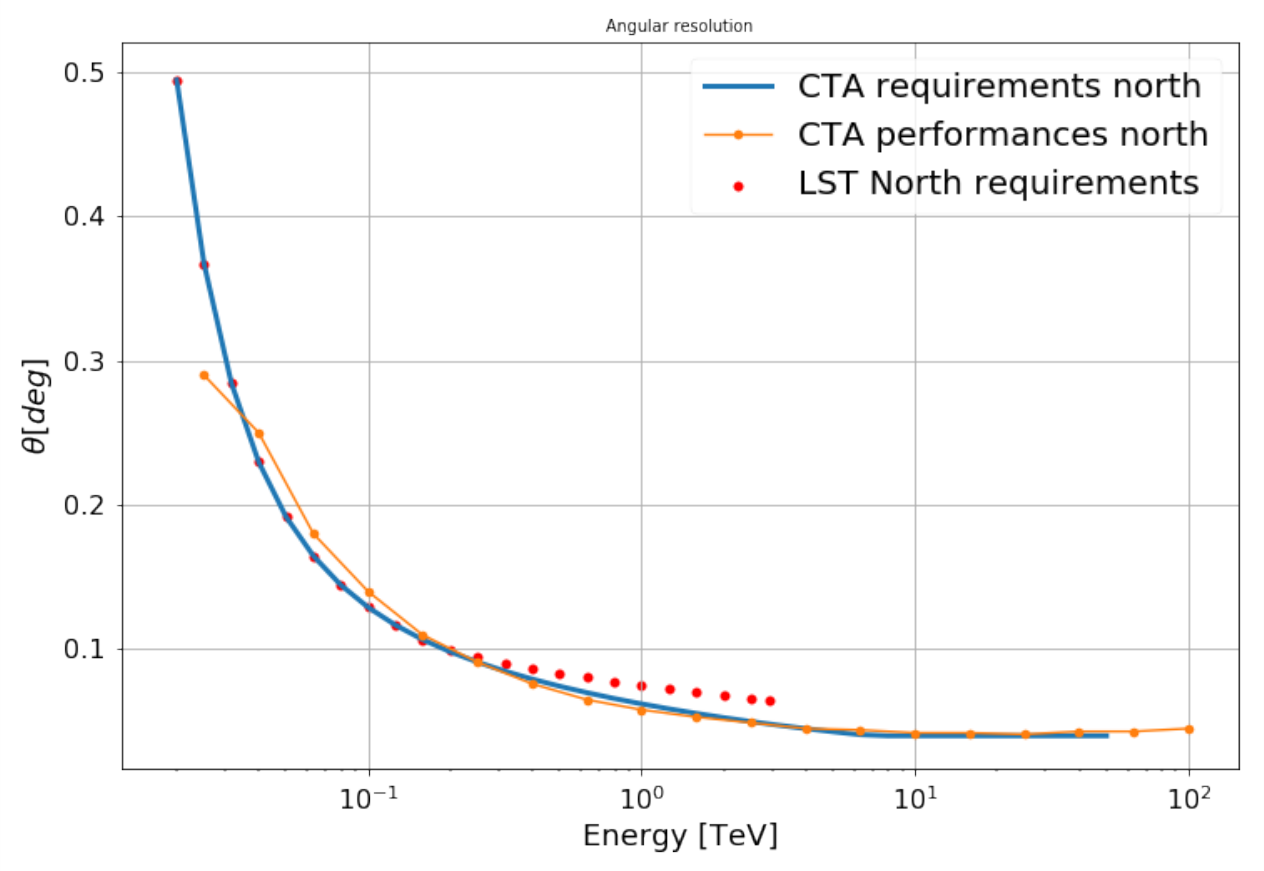


[ ]:

Sub-arrays

[10]:
```
lst_north_angres_requirements = np.loadtxt(get('cta_requirements_North-50h-LST-AngRes.
↪dat'))
```

[11]:
```
fig, ax = plt.subplots(figsize=(12,8))
ax = ctaplot.plot_angular_res_requirement('north', ax=ax, linewidth=3)
ax = ctaplot.plot_angular_res_cta_performance('north', ax=ax, marker='o')
ax.scatter(lst_north_angres_requirements[:,0], lst_north_angres_requirements[:,1],
           label="LST North requirements",
           color='red')

ax.grid()
plt.legend(prop = font);
```



[ ]:

1.4.2 How is resolution computed

Index

- *Normal distribution*
- *Resolution*
- *Relative scaling*
- *Error bars*

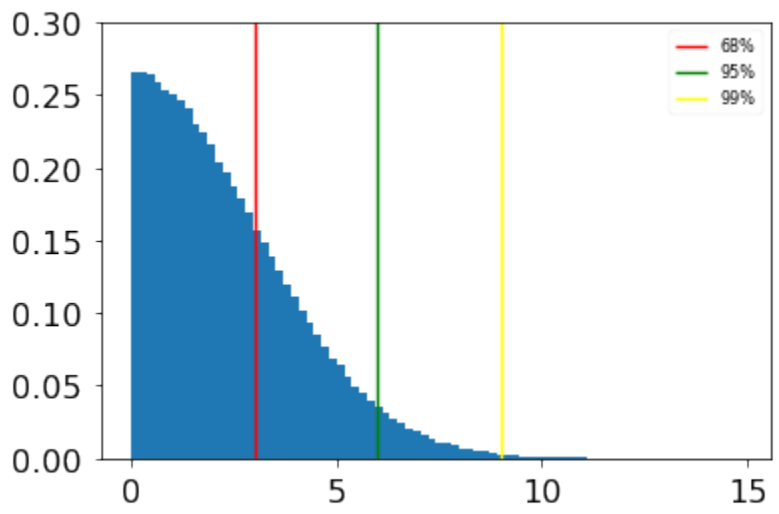
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from ctaplot.ana import resolution, relative_scaling
```

Normal distribution

For a nomal distribution, σ corresponds to the 68 percentile of the distribution
See the 68–95–99.7 rule

[2]:
```
loc = 10
scale = 3

X = np.random.normal(size=1000000, scale=scale, loc=loc)
plt.hist(np.abs(X - loc), bins=80, density=True)

sig_68 = np.percentile(np.abs(X - loc), 68.27)
sig_95 = np.percentile(np.abs(X - loc), 95.45)
sig_99 = np.percentile(np.abs(X - loc), 99.73)

plt.vlines(sig_68, 0, 0.3, label='68%', color='red')
plt.vlines(sig_95, 0, 0.3, label='95%', color='green')
plt.vlines(sig_99, 0, 0.3, label='99%', color='yellow')
plt.ylim(0,0.3)
plt.legend()


print("68th percentile = {:.4f}".format(sig_68))
print("95th percentile = {:.4f}".format(sig_95))
print("99th percentile = {:.4f}".format(sig_99))

assert np.isclose(sig_68, scale, rtol=1e-2)
assert np.isclose(sig_95, 2 * scale, rtol=1e-2)
assert np.isclose(sig_99, 3 * scale, rtol=1e-2)
```

```
68th percentile = 3.0016
95th percentile = 5.9941
99th percentile = 8.9812
```

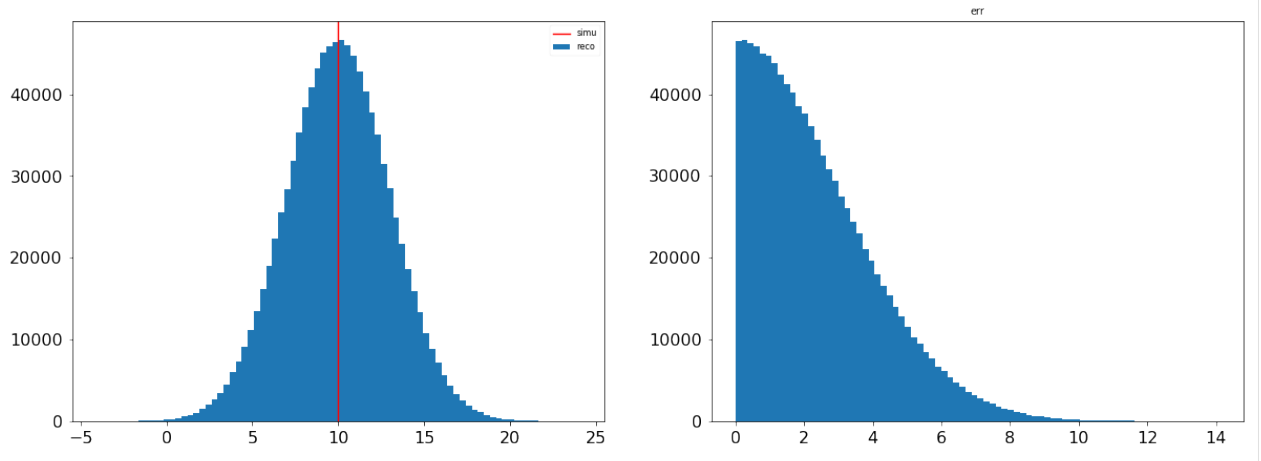


Resolution

The resolution is defined as the 68th percentile of the relative error `err = (reco - simu)/scaling`.
(see the *relative scaling* section for more info on it).
Hence, if the relative error follows a normal distribution, the resolution is equal to the sigma of the distribution

[3]:
```
simu = loc * np.ones(X.shape[0])
reco = np.random.normal(loc=loc, scale=scale, size=X.shape[0])

err = np.abs(simu - reco)

fig, axes = plt.subplots(1, 2, figsize=(20,7))
axes[0].hist(reco, bins=80, label='reco')
axes[0].axvline(loc, 0, 1, color='red', label='simu')
axes[0].legend()
axes[1].hist(err, bins=80)
axes[1].set_title('err')
```

[3]:
```
Text(0.5, 1.0, 'err')
```



Let's define `reco` in order to have a relative error equals to `err`.
Its resolution is equals to the sigma of the distribution

[4]:
```
res = resolution(simu, reco, relative_scaling_method=None)
print(res)
assert np.isclose(res[0], scale, rtol=1e-2)
```

```
[3.00210261 2.99740973 3.0068767 ]
```

Relative scaling

The resolution can be measured on the absolute error or on the relative one.

$$err = (reco - simu)/scaling$$

There is no absolute definition of the relative scaling and several are proposed in `ctaplot`.
The choice can be passed through the `relative_scaling_method`:

Methods:

- `s0` : no scaling (scaling = 1)
- `s1` : $ scaling = |simul| $
- `s2` : $ scaling = |recol| $
- `s3` : $ scaling = (|simul| + |recol|)/2 $
- `s4` : $ scaling = max(|recol, |simul|) $

The default one for the resolution is `s1`.

Note that methods involving `reco` are more subject to deviation from the expected result:

[5]:
```
for method in ['s1', 's2', 's3', 's4']:
    res = resolution(simu, reco, relative_scaling_method=method)
    print("Method {} gives a resolution = {:.3f} to be compared with the expected␣
↪value = {}".format(method, res[0], scale/loc))
```

```
Method s1 gives a resolution = 0.300 to be compared with the expected value = 0.3
Method s2 gives a resolution = 0.288 to be compared with the expected value = 0.3
Method s3 gives a resolution = 0.297 to be compared with the expected value = 0.3
Method s4 gives a resolution = 0.258 to be compared with the expected value = 0.3
```

NB:
- the angular resolution uses no scaling
- the energy resolution uses scaling `s1`
- the impact resolution uses no scaling by default

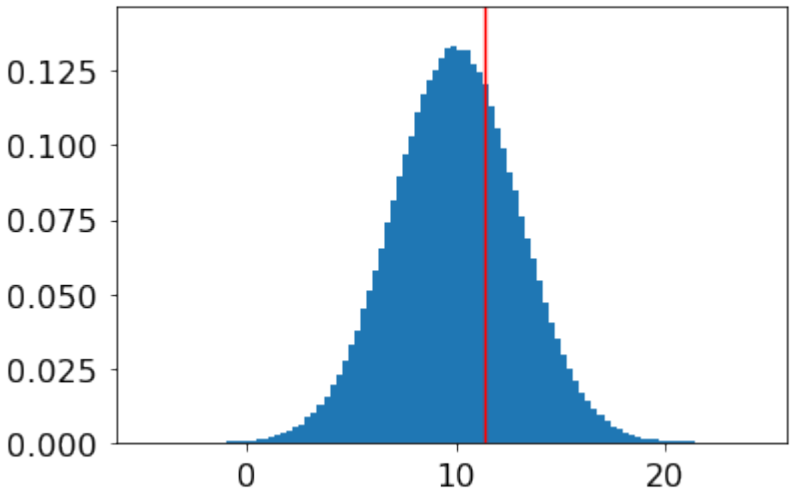
Error bars

Errors bars in resolution plots are given by the confidence interval (by default at 95%, equivalent to 2 sigmas for a normal distribution).
This means that we can be confident at 95% that the resolution values are within the range given by the error bars.

The implementation for percentile confidence interval follows: - http://people.stat.sfu.ca/~cschwarz/Stat-650/Notes/PDF/ChapterPercentiles.pdf

Example with a normal distribution:

[6]:
```
nbins, bins, patches = plt.hist(X, bins=100, density=True)
ymax = 1.1 * nbins.max()
plt.ylim(0, ymax)
plt.vlines(np.percentile(X, 68.27), 0, ymax, color='red')
plt.show()
print("The true 68th percentile of this distribution is: {:.4f}".format(np.
↪percentile(X, 68.27)))
```

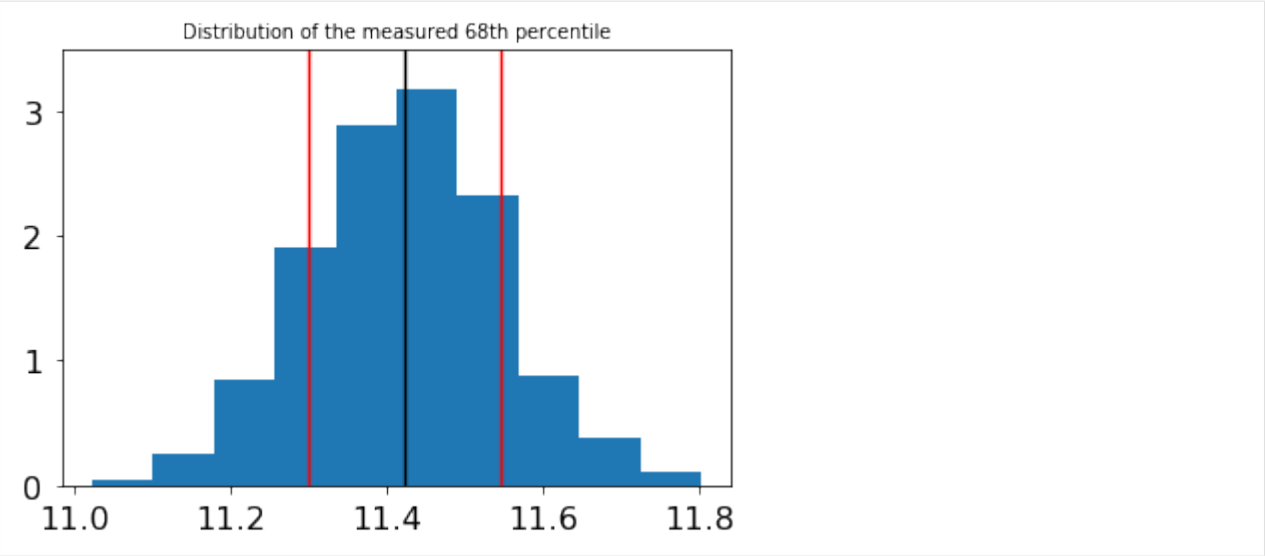



```
The true 68th percentile of this distribution is: 11.4245
```

We can consider this as our underlying (infinite) distribution.
If we take a random sample in this ditribution, the measurement of a percentile will come with a measurement error.
We can assess this error by taking multiple random samples and taking the distribution of measured percentile values.

[7]:
```
n = 1000
p = 0.6827
all_68 = []
for i in range(int(len(X)/n)):
    all_68.append(np.percentile(X[i*n:(i+1)*n], p*100))

all_68 = np.array(all_68)
nbins, bins, patches = plt.hist(all_68, density=True);
ymax = 1.1 * nbins.max()
plt.vlines(all_68.mean(), 0, ymax)
plt.vlines(all_68.mean() + all_68.std(), 0, ymax, color='red')
plt.vlines(all_68.mean() - all_68.std(), 0, ymax, color='red')
plt.ylim(0, ymax)
plt.title("Distribution of the measured 68th percentile")

print("Standard deviation = {:.5f}".format(all_68.std()))
```

```
Standard deviation = 0.12287
```



To evaluate directly the confidence interval from a sub-sample of the distribution, one can use the following formulae:

$$R_{low} = n * pz * \sqrt{n * p * (1p)}$$

$$R_{up} = n * p + z * \sqrt{n * p * (1p)}$$

with p the percentile and z the confidence level desired.

And the confidence interval given by: $(X[R_{low}], X[R_{up}])$

The confidence level is given by the cumulative distribution function (`scipy.stats.norm.ppf`).
Some useful values:

- z = 0.47 for a confidence level of 68%
- z = 1.645 for a confidence level of 95%
- z = 2.33 for a confidence level of 99%

[8]:
```
# confidence level:
z = 2.33

# sub-sample:
x = X[:n]

rl = int(n * p - z * np.sqrt(n * p * (1-p)))
ru = int(n * p + z * np.sqrt(n * p * (1-p)))
print("Measured percentile: {:.4f}".format(np.percentile(x, p*100)))
print("Confidence interval: ({:.4f}, {:.4f})".format(np.sort(x)[rl], np.sort(x)[ru]))
```

(continues on next page)

(continued from previous page)

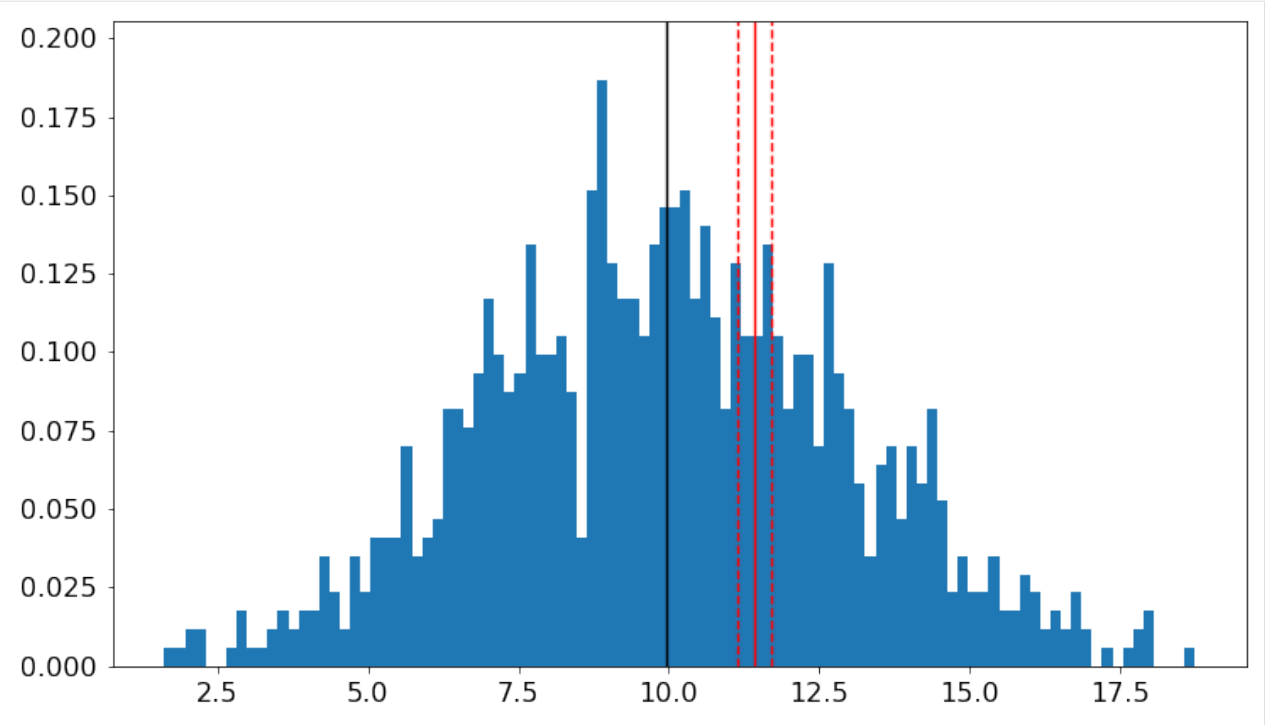
```
print("To be compared with: ({:.4f}, {:.4f})".format(all_68.mean()-all_68.std()*3,␣
↪all_68.mean()+all_68.std()*3))
print("which is the corresponding confidence interval given by the normal␣
↪distribution of the measured percentiles")
```

```
Measured percentile: 11.4301
Confidence interval: (11.1279, 11.7104)
To be compared with: (11.0539, 11.7911)
which is the corresponding confidence interval given by the normal distribution of␣
↪the measured percentiles
```

In ctaplot, this is computed by the function `percentile_confidence interval`:

[9]:
```
from ctaplot.ana import percentile_confidence_interval

p = 68.27
confidence_level = 0.99
pci = percentile_confidence_interval(x, percentile=p, confidence_level=0.99)
print("68th percentile: {:.3f}".format(np.percentile(x, p)))
print("Interval with a confidence level of {}%: ({:.3f}, {:.3f})".format(confidence_
↪level*100, pci[0], pci[1]))


plt.figure(figsize=(12,7))
nbins, bins, patches = plt.hist(x, bins=100, density=True);
ymax = 1.1 * nbins.max()

plt.vlines(np.percentile(x, 50), 0, ymax, color='black')
plt.vlines(np.percentile(x, p), 0, ymax, color='red')
plt.vlines(pci[0], 0, ymax, linestyles='--', color='red')
plt.vlines(pci[1], 0, ymax, linestyles='--', color='red',)
plt.ylim(0, ymax);
```

```
68th percentile: 11.430
Interval with a confidence level of 99.0%: (11.128, 11.710)
```



Note: The same method could be applied around the median.
In this case, the confidence interval is also given by $-\sigma/\sqrt{n}$ for a normal distribution.

[10]:
```
pci = percentile_confidence_interval(X, percentile=50, confidence_level=0.99)
print("Median: {:.5f}".format(np.median(X)))
print("Confidence interval at 99%: {}".format(pci))
print("To be compared with: ({}, {})".format(np.median(X)-3*scale/np.sqrt(len(X)), np.
↪median(X)+3*scale/np.sqrt(len(X))))
```

```
Median: 10.00197
Confidence interval at 99%: (9.992948666358563, 10.01069362511663)
To be compared with: (9.992972062483608, 10.010972062483608)
```

Python Module Index

C

Index

R

S