ctaplot Documentation

Release 0.1

Thomas Vuillaume

May 06, 2019

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:

- 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-v1) for more details."

1.2 Install

The release 0.2 is available in pip:

```
pip install ctaplot
```

Requirements packages:

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

We recommend the use of anaconda

One can also clone the package and install locally:

```
git clone https://github.com/vuillaut/ctaplot.git
cd ctaplot
python setup.py install
```

1.3 License

1.4 Modules

1.4.1 plots.py

Functions to make IRF and other reconstruction quality-check plots

ctaplot.plots.**plot_angles_distribution**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *Outfile=None*)

Plot the distribution of reconstructed angles. Save figure to Outfile in png format.

Parameters

- **RecoAlt** (*numpy.ndarray*) –
- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float*) –
- **AzSource** (*float*) –
- **Outfile** (*string*) –

ctaplot.plots.**plot_angles_map_distri**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *E*, *Outfile=None*)

Plot the angles map distribution

Parameters

- **RecoAlt** (*numpy.ndarray*) –
- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float*) –
- **AzSource** (*float*) –
- **E** (*numpy.ndarray*) –
- **Outfile** (*str*) –

Returns fig

Return type *matplotlib.pyplot.figure*

ctaplot.plots.**plot_angular_res_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_performances*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.plot*) –

Returns ax

Return type *matplotlib.pyplot.axes*

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

Plot the CTA requirement for the angular resolution :param cta_site: :type cta_site: string, see *ana.cta_requirements* :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_res_per_energy**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *SimuE*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*, *ax=None*, ***kwargs*)

Plot the angular resolution as a function of the energy

Parameters

- **RecoAlt** (*numpy.ndarray*) –
- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float*) –
- **AzSource** (*float*) –
- **SimuE** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **kwargs** (args for *matplotlib.pyplot.errorbar*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_binned_stat**(*x*, *y*, *ax=None*, *errorbar=True*, *statistic='mean'*, *bins=20*, *percentile=68*, ***kwargs*)

Plot binned statistic with errorbars corresponding to the given percentile

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)
```

ctaplot.plots.**plot_dispersion**(*X_true*, *X_exp*, *x_log=False*, *ax=None*, ***kwargs*)

Plot the dispersion around an expected value X_true

Parameters

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

Returns

Return type *maptlotlib.pyplot.axes*

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

Parameters

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

Returns ax

Return type *matplotlib.pyplot.axes*

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

Parameters

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

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_effective_area_per_energy**(*SimuE*, *RecoE*, *simuArea*, *ax=None*, ***kwargs*)
Plot the effective area as a function of the energy

Parameters

- **SimuE** (*numpy.ndarray* - all simulated event energies) –
- **RecoE** (*numpy.ndarray* - all reconstructed event energies) –
- **simuArea** (*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()
>>> simue = 10**(-2 + 4*np.random.rand(1000))
>>> recoe = 10**(-2 + 4*np.random.rand(100))
>>> ax = ctaplot.plots.plot_effective_area_per_energy(simue, recoe, 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_bias**(*SimuE*, *RecoE*, *ax=None*, ***kwargs*)

Plot the energy bias

Parameters

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

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_energy_distribution**(*SimuE*, *RecoE*, *ax=None*, *outfile=None*, *maskSimuDetected=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

- **SimuE** (*Numpy 1d array of simulated energies*) –
- **RecoE** (*Numpy 1d array of reconstructed energies*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **outfile** (*string - output file path*) –
- **- Numpy 1d array - mask of detected particles for the SimuE array** (*maskSimuDetected*) –

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

Plot the enregy resolution as a function of the energy

Parameters

- **SimuE** (*numpy.ndarray*) –

- **RecoE** (*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_energy_resolution_cta_performances**(*cta_site*, *ax=None*, ***kwargs*)

Plot the cta performances (June 2018) for the energy resolution

Parameters

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

Returns ax

Return type *matplotlib.pyplot.axes*

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

Plot the cta requirement for the energy resolution

Parameters

- **cta_site** (string, see *ana.cta_requirements*) –
- **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**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *E=None*, *ax=None*, *Outfile=None*)

Plot a map in angles [in degrees] of the photons seen by the telescope (after reconstruction)

Parameters

- **RecoAlt** (*numpy.ndarray*) –
- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float, source Altitude*) –

- **AzSource** (*float, source Azimuth*) –
- **E** (*numpy.ndarray*) –
- **Outfile** (*string*) –

ctaplot.plots.**plot_impact_map**(*impactX*, *impactY*, *telX*, *telY*, *telTypes=None*, *Outfile='ImpactMap.png'*)

Map of the site with telescopes positions and impact points heatmap

Parameters

- **impactX** (*numpy.ndarray*) –
- **impactY** (*numpy.ndarray*) –
- **telX** (*numpy.ndarray*) –
- **telY** (*numpy.ndarray*) –
- **telTypes** (*numpy.ndarray*) –
- **Outfile** (*string - name of the output file*) –

ctaplot.plots.**plot_impact_parameter_error**(*RecoX*, *RecoY*, *SimuX*, *SimuY*, *ax=None*, ***kwargs*)

plot impact parameter error distribution and save it under Outfile :param RecoX: :type RecoX: *numpy.ndarray* :param RecoY: :type RecoY: *numpy.ndarray* :param SimuX: :type SimuX: *numpy.ndarray* :param SimuY: :type SimuY: *numpy.ndarray* :param Outfile: :type Outfile: string

ctaplot.plots.**plot_impact_parameter_error_per_energy**(*RecoX*, *RecoY*, *SimuX*, *SimuY*, *SimuE*, *ax=None*, ***kwargs*)

plot the impact parameter error distance as a function of energy and save the plot as Outfile :param RecoX: :type RecoX: *numpy.ndarray* :param RecoY: :type RecoY: *numpy.ndarray* :param SimuX: :type SimuX: *numpy.ndarray* :param SimuY: :type SimuY: *numpy.ndarray* :param SimuE: :type SimuE: *numpy.ndarray* :param ax: :type ax: *matplotlib.pyplot.axes* :param kwargs: :type kwargs: args for *matplotlib.pyplot.errorbar*

Returns E, err_mean

Return type numpy arrays

ctaplot.plots.**plot_impact_parameter_error_per_multiplicity**(*RecoX*, *RecoY*, *SimuX*, *SimuY*, *Multiplicity*, *max_mult=None*, *ax=None*, ***kwargs*)

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

Parameters

- **RecoX** (*numpy.ndarray*) –
- **RecoY** (*numpy.ndarray*) –
- **SimuX** (*numpy.ndarray*) –
- **SimuY** (*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. :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 ax: :type ax: *matplotlib.pyplot.axes* :param kwargs: :type kwargs: kwargs for *matplotlib.pyplot.hist2d*

Returns

Return type ax

ctaplot.plots.**plot_impact_point_heatmap**(*RecoX*, *RecoY*, *ax=None*, *Outfile=None*)

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

Parameters

- **RecoX** (*numpy.ndarray*) –
- **RecoY** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **Outfile** (*string*) –

ctaplot.plots.**plot_impact_point_map_distri**(*RecoX*, *RecoY*, *telX*, *telY*, ***options*)

Map and distributions of the reconstructed impact points.

Parameters

- **RecoX** (*numpy.ndarray*) –
- **RecoY** (*numpy.ndarray*) –
- **telX** (*numpy.ndarray*) –
- **telY** (*numpy.ndarray*) –
- **options** – kde=True : make a gaussian fit of the point density Outfile='string' : 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_layout_map**(*TelX*, *TelY*, *TelId*, *TelType*, *LayoutId*, *Outfile='LayoutMap'*)

Plot the layout map of telescopes positions - depreciated

Parameters

- **TelX** (*numpy.ndarray*) –
- **TelY** (*numpy.ndarray*) –
- **TelId** (*numpy.ndarray*) –
- **TelType** (*numpy.ndarray*) –
- **LayoutId** (*numpy.ndarray*) –
- **Outfile** (*string*) –

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_hist**(*multiplicity*, *ax=None*, *Outfile=None*, *xmin=0*, *xmax=100*)

Histogram of the telescopes multiplicity

Parameters

- **multiplicity** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **Outfile** (*string*) –
- **xmin** (*float*) –
- **xmax** (*float*) –

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_multiplicity_per_telescope_type**(*EventTup*, *Outfile=None*)
Plot the multiplicity for each telescope type

Parameters

- **EventTup** –
- **Outfile** –

ctaplot.plots.**plot_reco_histo**(*y_true*, *y_reco*)
plot the histogram of a reconstructed feature after prediction from a machine learning algorithm plt.show() to display :param y_true: :type y_true: real values of the feature to predict :param y_reco: :type y_reco: predicted values by the ML algo

ctaplot.plots.**plot_resolution_per_energy**(*reco*, *simu*, *SimuE*, *ax=None*, ***kwargs*)
Plot a variable resolution as a function of the energy

Parameters

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

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_sensitivity_cta_performances**(*cta_site*, *ax=None*, ***kwargs*)
Plot the CTA performances for the sensitivity :param cta_site: :type cta_site: string - see *ctaplot.ana.cta_requirements* :param ax: :type ax: *matplotlib.pyplot.axes*, optional

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_sensitivity_cta_requirements**(*cta_site*, *ax=None*, ***kwargs*)
Plot the CTA requirement for the sensitivity :param cta_site: :type cta_site: string - see *ctaplot.ana.cta_requirements* :param ax: :type ax: *matplotlib.pyplot.axes*, optional

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**plot_site**(*tel_x*, *tel_y*, *ax=None*, ***kwargs*)
Plot the telescopes positions :param tel_x: :type tel_x: 1D numpy array :param tel_y: :type tel_y: 1D numpy array :param ax: :type ax: *~matplotlib.axes.Axes* or None :param **kwargs: :type **kwargs: Extra keyword arguments are passed to *matplotlib.pyplot.scatter*

Returns ax

Return type *~matplotlib.axes.Axes*

ctaplot.plots.**plot_site_map**(*telX*, *telY*, *telTypes=None*, *Outfile='SiteMap.png'*)

Map of the site with telescopes positions

Parameters

- **telX** (*numpy.ndarray*) –
- **telY** (*numpy.ndarray*) –
- **telTypes** (*numpy.ndarray*) –
- **Outfile** (*string - name of the output file*) –

ctaplot.plots.**plot_theta2**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *ax=None*, ***kwargs*)

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

Parameters

- **RecoAlt** (*numpy.ndarray* - reconstructed altitude angle in radians) –
- **RecoAz** (*numpy.ndarray* - reconstructed azimuth angle in radians) –
- **AltSource** (*numpy.ndarray* - true altitude angle in radians) –
- **AzSource** (*numpy.ndarray* - true azimuth angle in radians) –
- **ax** (*matplotlib.pyplot.axes*) –
- ****kwargs** (options for *matplotlib.pyplot.hist*) –

ctaplot.plots.**saveplot_angular_res_per_energy**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*, *SimuE*, *ax=None*, *Outfile='AngRes'*, *cta_site=None*, ***kwargs*)

Plot the angular resolution as a function of the energy and save the plot in png format

Parameters

- **RecoAlt** (*numpy.ndarray*) –
- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float*) –
- **AzSource** (*float*) –
- **SimuE** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **Outfile** (*string*) –
- **cta_site** (*string*) –
- **kwargs** (args for *hipectaold.plots.plot_angular_res_per_energy*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**saveplot_effective_area_per_energy**(*SimuE*, *RecoE*, *simuArea*, *ax=None*, *Outfile='AngRes'*, *cta_site=None*, ***kwargs*)

Plot the angular resolution as a function of the energy and save the plot in png format

Parameters

- **RecoAlt** (*numpy.ndarray*) –

- **RecoAz** (*numpy.ndarray*) –
- **AltSource** (*float*) –
- **AzSource** (*float*) –
- **SimuE** (*numpy.ndarray*) –
- **ax** (*matplotlib.pyplot.axes*) –
- **Outfile** (*string*) –
- **cta_site** (*string*) –
- **kwargs** (args for *ctaplot.plots.plot_angular_res_per_energy*) –

Returns ax

Return type *matplotlib.pyplot.axes*

ctaplot.plots.**saveplot_energy_resolution**(*SimuE*, *RecoE*, *Outfile='EnergyResolution.png'*, *cta_site=None*)

plot the energy resolution of the reconstruction :param SimuE: :type SimuE: *numpy.ndarray* :param RecoE: :type RecoE: *numpy.ndarray* :param cta_goal: :type cta_goal: boolean - If True CTA energy resolution requirement is plotted

Returns ax

Return type *matplotlib.pyplot.axes*

1.4.2 ana.py

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

ctaplot.ana.**angles_modulo_degrees**(*RecoAlt*, *RecoAz*, *SimuAlt*, *SimuAz*)

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_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_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.**bias**(*simu*, *reco*)

Compute the bias of a reconstructed variable.

Parameters

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

Returns

Return type float

class ctaplot.ana.**cta_performances**

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_requirements**

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.**effective_area**(*SimuE*, *RecoE*, *simuArea*)
Compute the effective area from a list of simulated energies and reconstructed energies :param SimuE: :type SimuE: 1d numpy array :param RecoE: :type RecoE: 1d numpy array :param simuArea: :type simuArea: float - area on which events are simulated

Returns

Return type float = effective area

ctaplot.ana.**effective_area_per_energy**(*SimuE*, *RecoE*, *simuArea*)
Compute the effective area per energy bins from a list of simulated energies and reconstructed energies

Parameters

- **SimuE**(*1d numpy array*) –
- **RecoE**(*1d numpy array*) –
- **simuArea**(*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*, *RecoE*, *simuArea*)
Compute the effective area per energy bins from a list of simulated energies and reconstructed energies

Parameters

- **SimuE**(*1d numpy array*) –
- **RecoE**(*1d numpy array*) –
- **simuArea**(*float - area on which events are simulated*) –

Returns (E, Seff)

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

ctaplot.ana.**energy_bias**(*SimuE*, *RecoE*)
Compute the energy bias per energy bin. :param SimuE: :type SimuE: 1d numpy array of simulated energies :param RecoE: :type RecoE: 1d numpy array of reconstructed energies

Returns (e, biasE)

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

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 DeltaE/E 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_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.**get_angles_0pi**(*angles*)
return angles modulo between 0 and +pi

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

Returns

Return type *numpy.ndarray*

ctaplot.ana.**get_angles_pipi**(*angles*)
return angles modulo between -pi and +pi

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

Returns

Return type *numpy.ndarray*

ctaplot.ana.**impact_parameter_error**(*RecoX*, *RecoY*, *SimuX*, *SimuY*)
compute the error distance between simulated and reconstructed impact parameters :param RecoX: :type RecoX: 1d numpy array :param RecoY: :param SimuX: :param SimuY:

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*)
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

- **RecoX** (*numpy.ndarray*) –
- **RecoY** (*numpy.ndarray*) –
- **SimuX** (*numpy.ndarray*) –
- **SimuY** (*numpy.ndarray*) –
- **confidence_level** (*float*) –

Returns

Return type *numpy.array* - [impact_resolution, lower_limit, upper_limit]

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*)

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*) –

Returns (E, RES)

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

class ctaplot.ana.**irf_cta**

Bases: object

Class to handle Instrument Response Function data

set_E_bin(*E_bin*)

ctaplot.ana.**logbin_mean**(*E_bin*)

Function that gives back the mean of each bin in logscale

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

Returns

Return type *numpy.ndarray*

ctaplot.ana.**logspace_decades_nbin**(*Xmin*, *Xmax*, *n=5*)

return an array with logspace and n bins / decade :param Xmin: :type Xmin: float :param Xmax: :type Xmax: float :param n: :type n: int - number of bins per decade

Returns

Return type 1D Numpy array

ctaplot.ana.**mask_range**(*X*, *Xmin=0*, *Xmax=inf*)

create a mask for X to get values between Xmin and Xmax :param X: :type X: 1d numpy array :param Xmin: :type Xmin: float :param Xmax: :type Xmax: float

Returns

Return type 1d numpy array of boolean

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.**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.**resolution**(*simu*, *reco*, *percentile=68.27*, *confidence_level=0.95*, *bias_correction=False*)

Compute the resolution of reco as the Qth (68.27 as standard = 1 sigma) containment radius of (simu-reco)/reco 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

Returns

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

ctaplot.ana.**resolution_per_energy**(*simu*, *reco*, *simu_energy*, *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.**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.`**`theta2`**(*RecoAlt*, *RecoAz*, *AltSource*, *AzSource*)
Compute the theta2 in radians

Parameters

- **`RecoAlt`** (1d *numpy.ndarray* - reconstructed Altitude in radians) –
- **`RecoAz`** (1d *numpy.ndarray* - reconstructed Azimuth in radians) –
- **`AltSource`** (1d *numpy.ndarray* - true Altitude in radians) –
- **`AzSource`** (1d *numpy.ndarray* - true Azimuth in radians) –

Returns

Return type 1d *numpy.ndarray*

1.5 Examples

1.5.1 How to easily plot CTA IRF requirements and performances

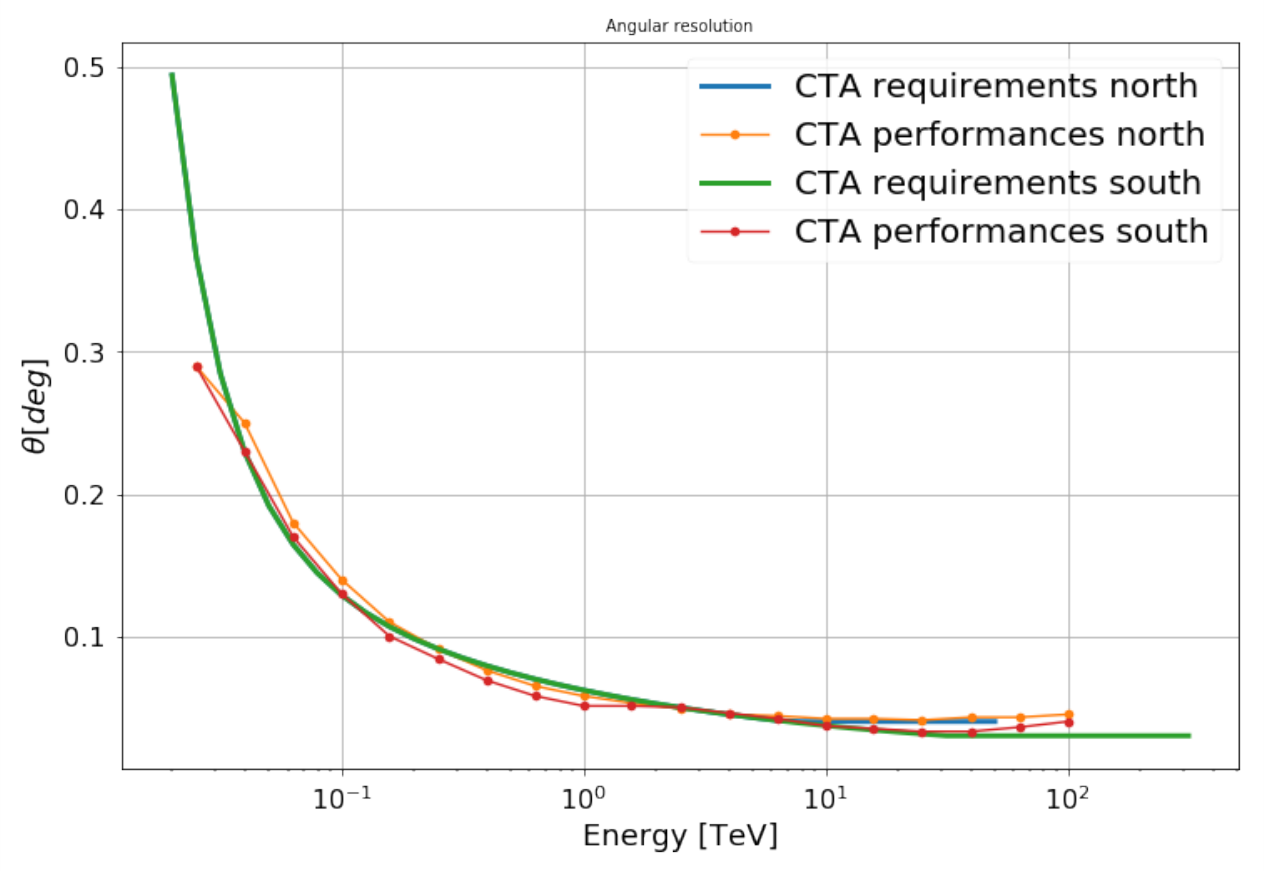
CTA performances are up-to-date and public and can be found on the cta-observatory website

[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)
```

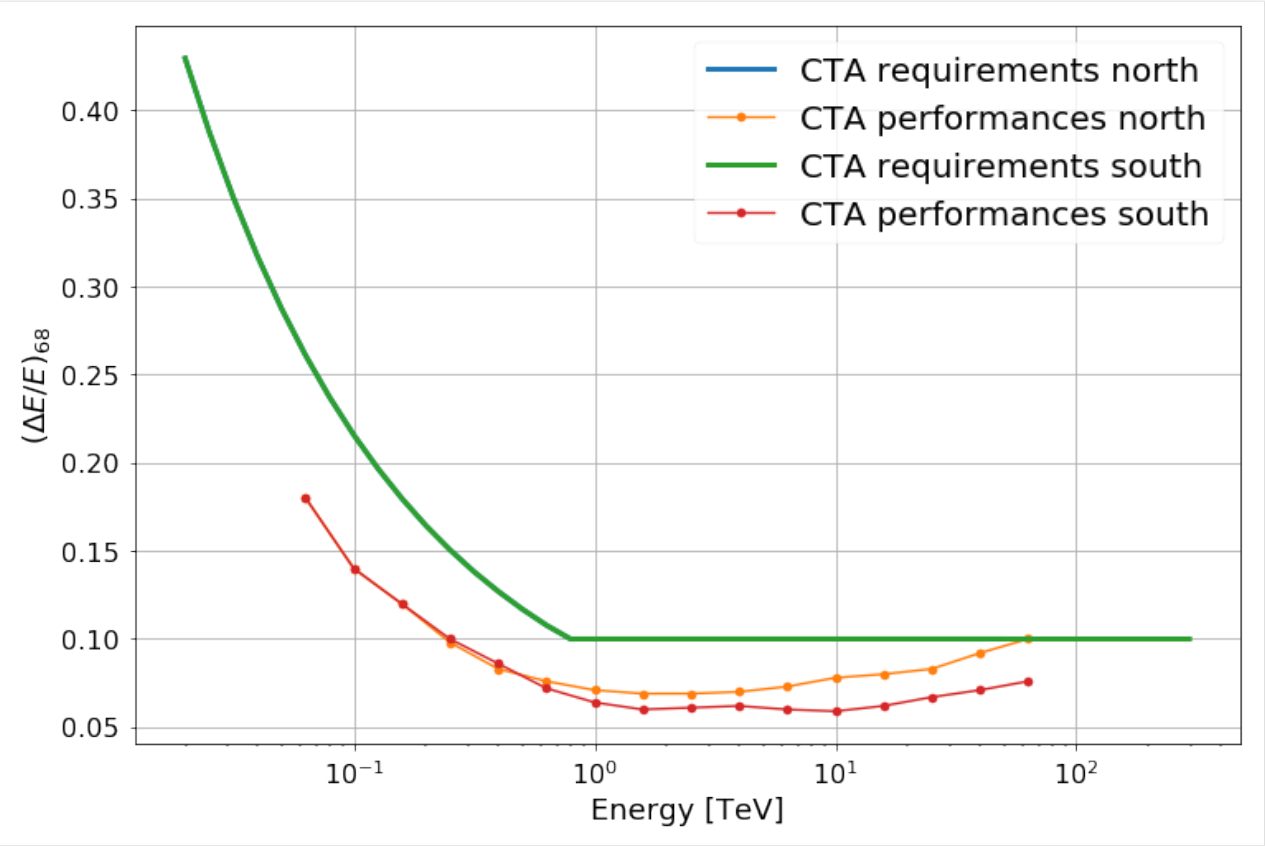
Angular resolution

[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);
```



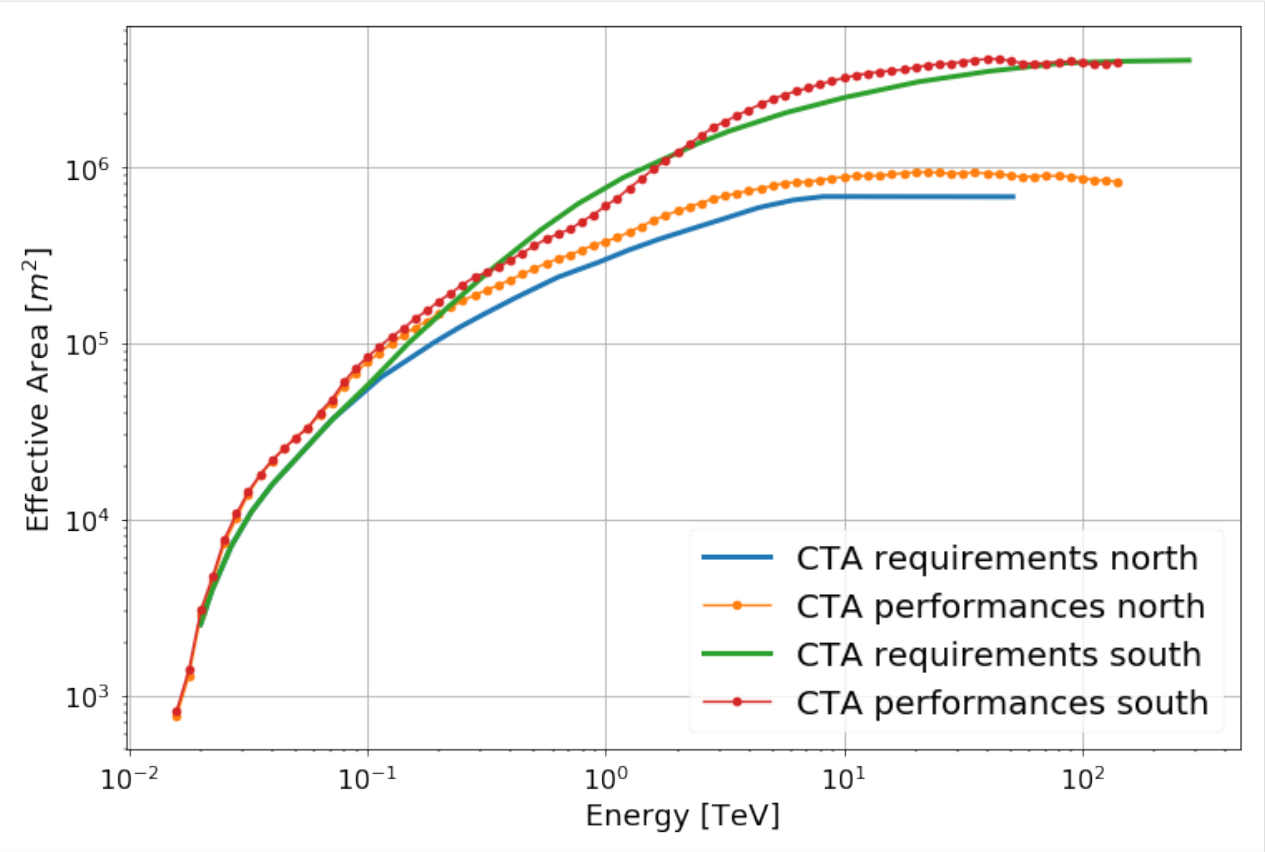
Energy resolution

[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);
```



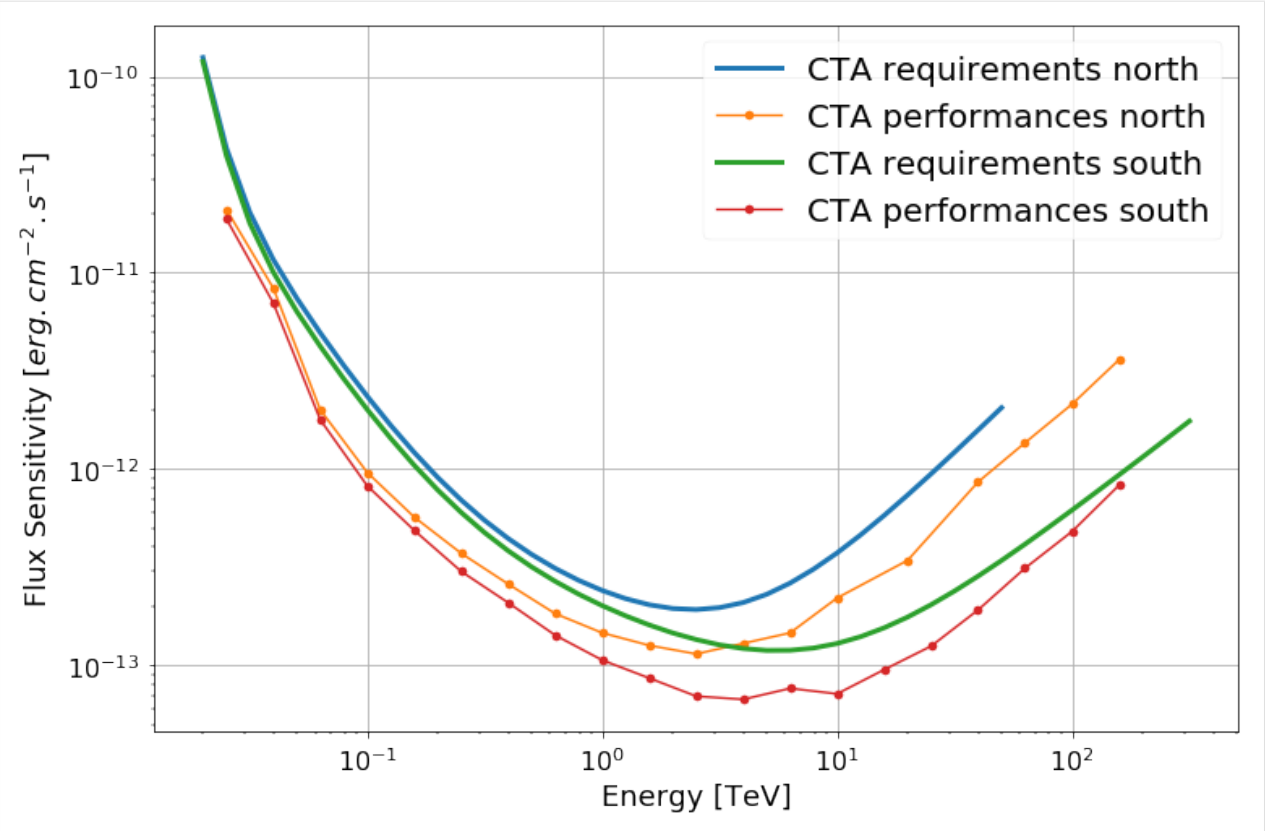
Effective Area

[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);
```



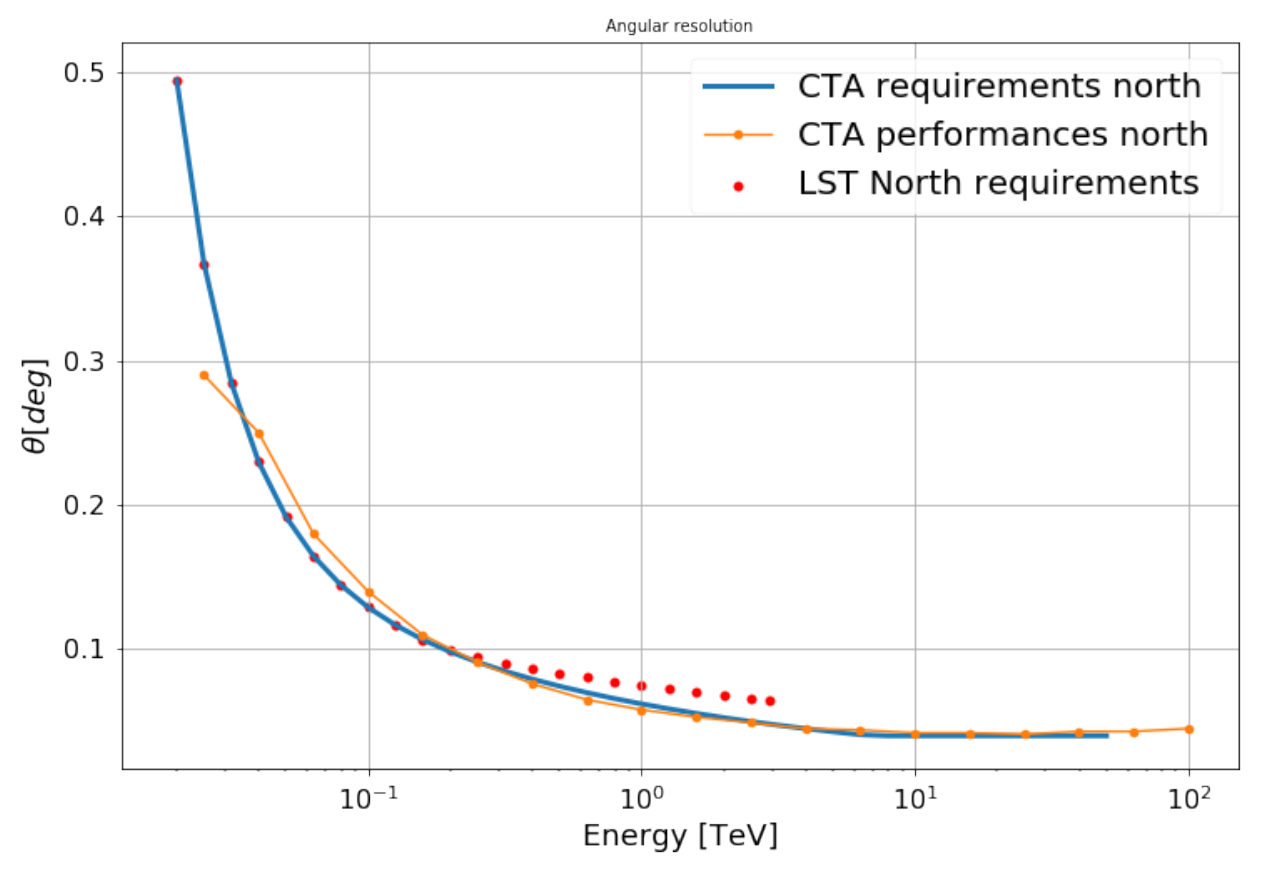
[ ]:

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);
```



[ ]:

1.5.2 How is resolution computed

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

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)
```

(continues on next page)

(continued from previous page)

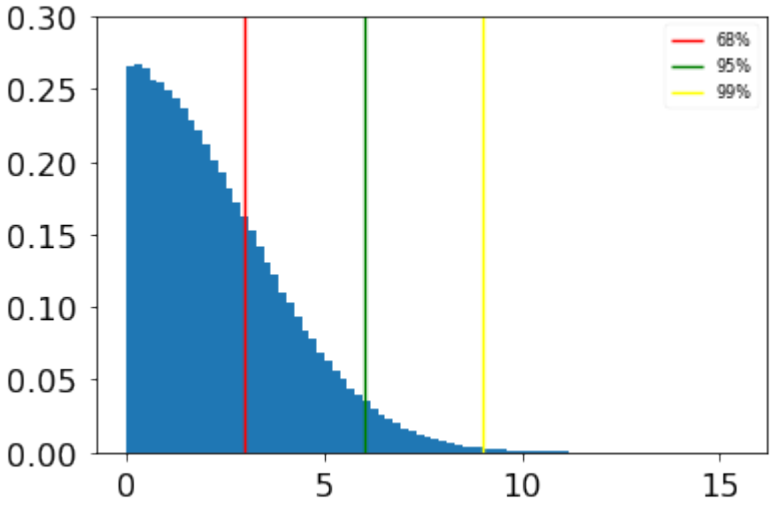
```
sig_99 = np.percentile(np.abs(X - loc), 99.73)

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)
```

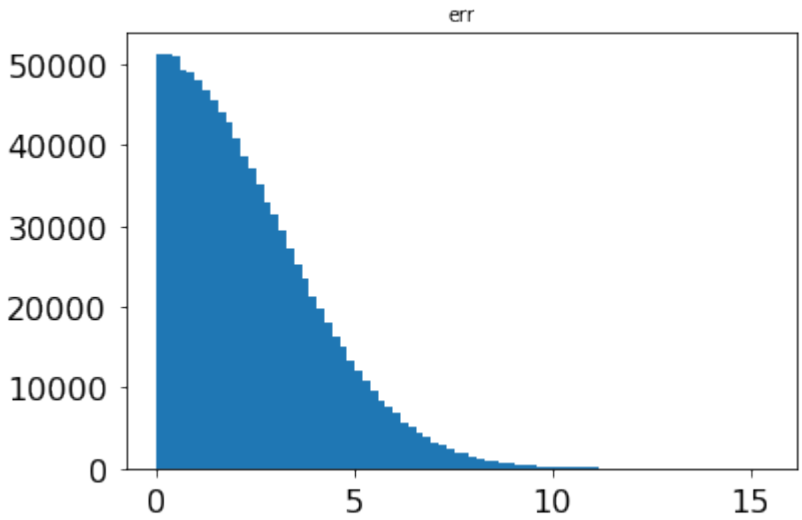
```
68th percentile = 3.0010
95th percentile = 5.9976
99th percentile = 8.9889
```



Resolution

The resolution is defined as the 68th percentile of the relative error `err = (reco - simu)/reco`
Hence, if the relative error follows a normal distribution, the resolution is equal to the sigma of the distribution

[3]:
```
err = np.abs(X - loc)
simu = loc * np.ones(X.shape[0])
plt.hist(err, bins=80)
plt.title('err')
plt.show()
```



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]:
```
reco = simu / (1 - (X - loc))
res = resolution(simu, reco)
print(res)
assert np.isclose(res[0], scale, rtol=1e-2)
```

```
[3.00101448 2.99622303 3.0056059 ]
```

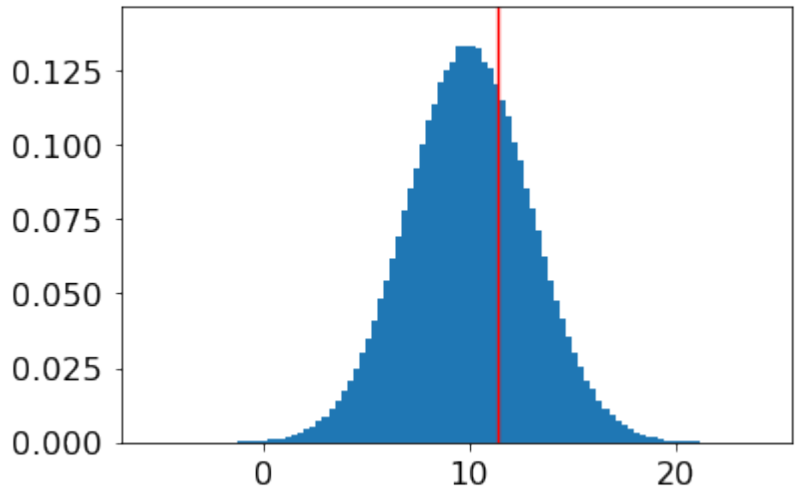
Error bars

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:

[5]:
```
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
```

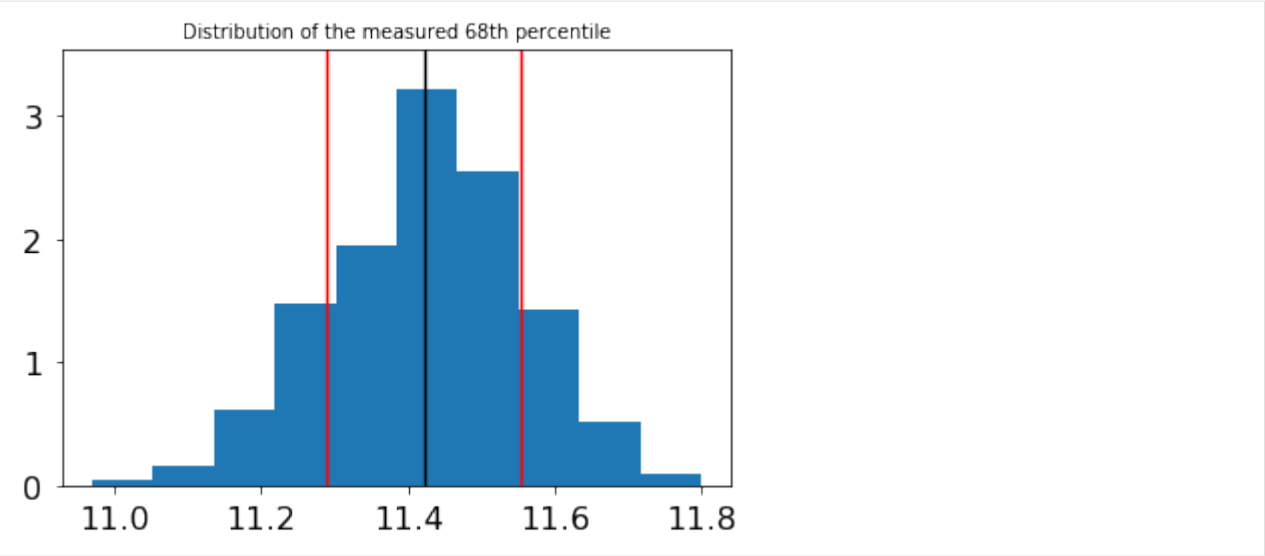
We can consider this as our underlying (infinite) distribution.
If we take a random sample in this ditribution, the measurement of a percentile will come with a measurement error.
We can assess this error by taking multiple random samples and taking the distribution of measured percentile values.

[6]:
```
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 = 0.13226
```



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%

[7]:
```
# 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]))
print("To be compared with: ({:.4f}, {:.4f})".format(all_68.mean()-all_68.std()*3,␣
↪all_68.mean()+all_68.std()*3))
```

```
Measured percentile: 11.3698
Confidence interval: (11.1170, 11.6991)
To be compared with: (11.0254, 11.8190)
```

Because we are dealing with normal distributions here, we can verify that this corresponds (within statistical margins) to the interval given by twice the standard deviations:

[8]:
```
err68 = np.abs(all_68 - all_68.mean())
all_68.mean() - 3 * err68.std(), all_68.mean() + 3 * err68.std()
```

[8]:
```
(11.183070932430383, 11.661320517902158)
```

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

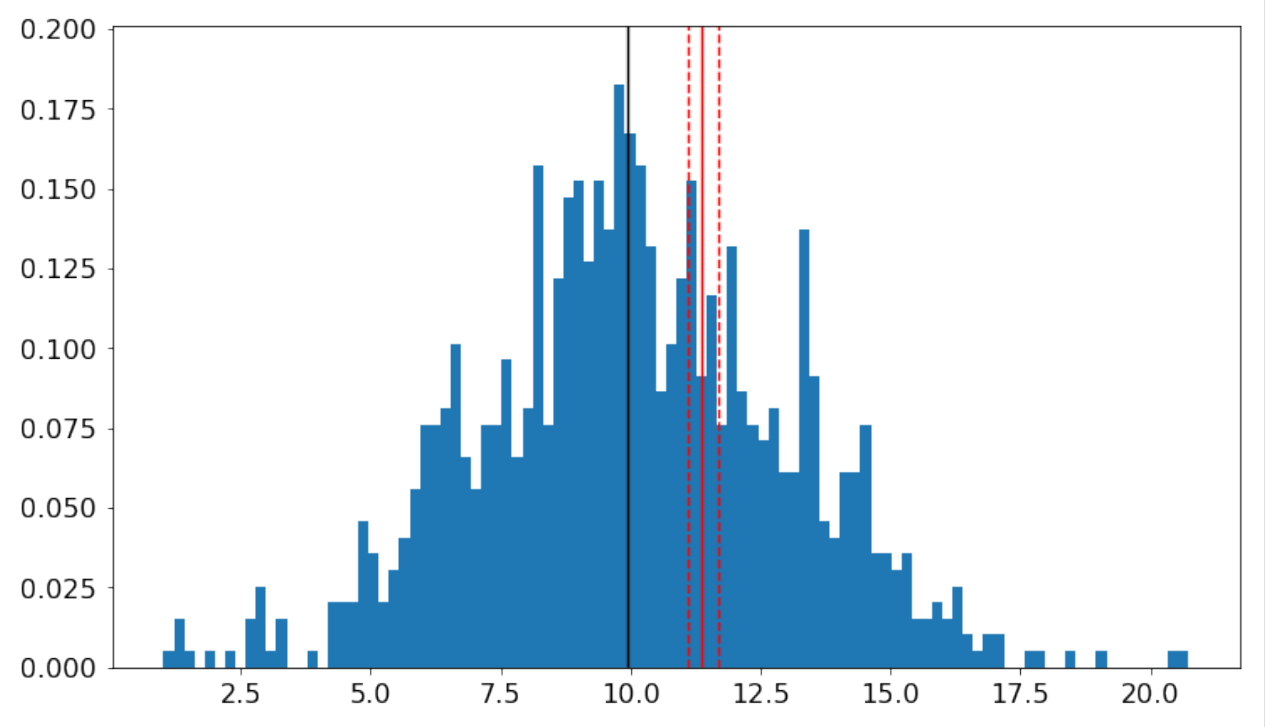
[9]:
```
from ctaplot.ana import percentile_confidence_interval

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: 11.370
Interval with a confidence level of 99.0%: (11.117, 11.699)
```



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: 9.99688
Confidence interval at 99%: (9.988429037280191, 10.005916820660993)
To be compared with: (9.987875456368489, 10.00587545636849)
```

Python Module Index

C

Index

R

S

T