Unbinned likelihood analysis in gammapy

Giacomo D'Amico

University of Bergen, Norway

in collaboration with **Julia Djuvsland**

cherenkov telescope array

14 February 2022

VERY IMPORTANT DISCLAIMER

I am not an expert neither of ***python*** nor of ***gammapy***

The chance that I did something **redundant, wrong** or that could have been **coded** better is high!

The Unbinned Likelihood

$$L(\theta) = \frac{(s(\theta)+b)^{N_{on}}}{N_{on}!} e^{-s(\theta)-b} \prod_{i=1}^{N_{on}} \left(\frac{s(\theta)}{s(\theta)+b} \cdot f_s(E_i|\theta) + \frac{b}{s(\theta)+b} \cdot f_b(E_i) \right)$$

N_{on} = Total number of events observed

E_i = Energy of i-th event

$s(\theta)$ = Expected signal events = $\int_{\Delta E} \frac{dN_s}{dE}(E|\theta) dE$

b = Expected background events = $\int_{\Delta E} \frac{dN_b}{dE}(E) dE$

$f_s(E|\theta)$ = PDF of detecting a signal event with a given energy = $\frac{1}{s(\theta)} \frac{dN_s}{dE}(E)$

$f_b(E)$ = PDF of detecting a background event with a given energy = $\frac{1}{b} \frac{dN_b}{dE}(E)$

Apart from the counts, we need to keep the information on:

- the **single events variables** (like the energy)
- the **PDF** of these variables for a **signal** and **background** population respectively

$$\log L(\theta) = -s(\theta) - b + \sum_{i=1}^{N_{on}} \log \left(\frac{dN_s}{dE}(E_i|\theta) + \frac{dN_b}{dE}(E_i) \right)$$

The UnbinnedDataset class

```
class UnbinnedDataset(MapDataset):
    """Perform unbinned model likelihood fit on maps.

    Parameters
    ----------
    events : `~gammapy.data.EventList`
        Event list
    models : `~gammapy.modeling.models.Models`
        Source sky models.
    counts : `~gammapy.maps.WcsNDMap` or `~gammapy.utils.fits.HDULocation`
        Counts cube
    exposure : `~gammapy.maps.WcsNDMap` or `~gammapy.utils.fits.HDULocation`
        Exposure cube
    background : `~gammapy.maps.WcsNDMap` or `~gammapy.utils.fits.HDULocation`
        Background cube
    mask_fit : `~gammapy.maps.WcsNDMap` or `~gammapy.utils.fits.HDULocation`
        Mask to apply to the likelihood for fitting.
    psf : `~gammapy.irf.PSFMap` or `~gammapy.utils.fits.HDULocation`
        PSF kernel
    edisp : `~gammapy.irf.EDispKernel` or `~gammapy.irf.EDispMap` or `~gammapy.utils.fits.HDULocation`
        Energy dispersion kernel
    mask_safe : `~gammapy.maps.WcsNDMap` or `~gammapy.utils.fits.HDULocation`
        Mask defining the safe data range.
    gti : `~gammapy.data.GTI`
        GTI of the observation or union of GTI if it is a stacked observation
    meta_table : `~astropy.table.Table`
        Table listing information on observations used to create the dataset.
        One line per observation for stacked datasets.

    See Also
    --------
    MapDataset, SpectrumDataset, FluxPointsDataset
    """

    stat_type = "unbinned"
    tag = "UnbinnedDataset"

    def __init__(
        self,
        events=None,
        models=None,
        counts=None,
        exposure=None,
        background=None,
        psf=None,
        edisp=None,
        mask_safe=None,
        mask_fit=None,
        gti=None,
        meta_table=None,
        name=None,
    ):
        super().__init__(models,
            counts,
            exposure,
            background,
            psf,
            edisp,
            mask_safe,
            mask_fit,
            gti,
            meta_table,
            name,
            )

        self.events = events
```

data.EventList

this is the novel parameter!

We need to define a **NEW *stat_sum()*** that performs the **unbinned likelihood** computation

```
def stat_sum(self,**kwargs):
    """Unbinned likelihood given the current model parameters."""
    self.update_flux()

    if self.events is None:
        stat     = -np.inf
    else:
        energies = self.events.energy
        s, b     = self.tot_signal_obs,  self.tot_bkg
        marks    = self.predicted_dnde(energies, obs=True).value
        # CHECK IF ALL MARKS ARE BIGGER THAN ZERO
        if np.sum( marks>0) == len(marks):
            logmarks = np.sum(np.log(marks))
        else:
            logmarks = -np.inf
        logL = -s  -b + logmarks
        stat = -2*logL

    return stat
```

dataset.predicted_dnde() gives $\frac{dN_s}{dE}(E)$ and/or $\frac{dN_b}{dE}(E)$

```
dataset.predicted_dnde( 13*u.TeV)
```

$[55.678075]\ \frac{1}{\mathrm{TeV}}$

Right now this is done by performing a linear interpolation

Some extra features - *fake_events()*

Like ***dataset.fake()*** but for the events simulation

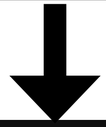
Mainly inspired by ***datasets.MapDatasetEventSampler()***

```
def fake_events(self, signal=True, bkg=True, obs=True,random_state="random-seed"):
    """Simulate fake events for the current model and reduced IRFs.

    This method overwrites the events defined on the dataset object
    and counts are also updated accordingly.

    Parameters
    ----------
    random_state : {int, 'random-seed', 'global-rng', `~numpy.random.RandomState`}
            Defines random number generator initialisation.
            Passed to `~gammapy.utils.random.get_random_state`.
    obs          : Bool
            True if dN/dE in observed  energy,
            i.e. by incluidng edisp effects
            By default is True
    signal       : Bool
            True if you want to include in the
            simulation the signal of gamma
            By default is True
    bkg          : Bool
            True if you want to include in the
            simulation the background
            By default is True

    """
```

```
dataset.events.table
```

Table length=862

ENERGY	RA	DEC
TeV	deg	deg
float64	float64	float64
11.541427454134913	150.58	-13.260000000000005
16.08937458575912	150.58	-13.260000000000005
11.154471130915814	150.58	-13.260000000000005
11.789510999644488	150.58	-13.260000000000005



```
dataset.fake_events()
dataset.events.table
```

Table length=862

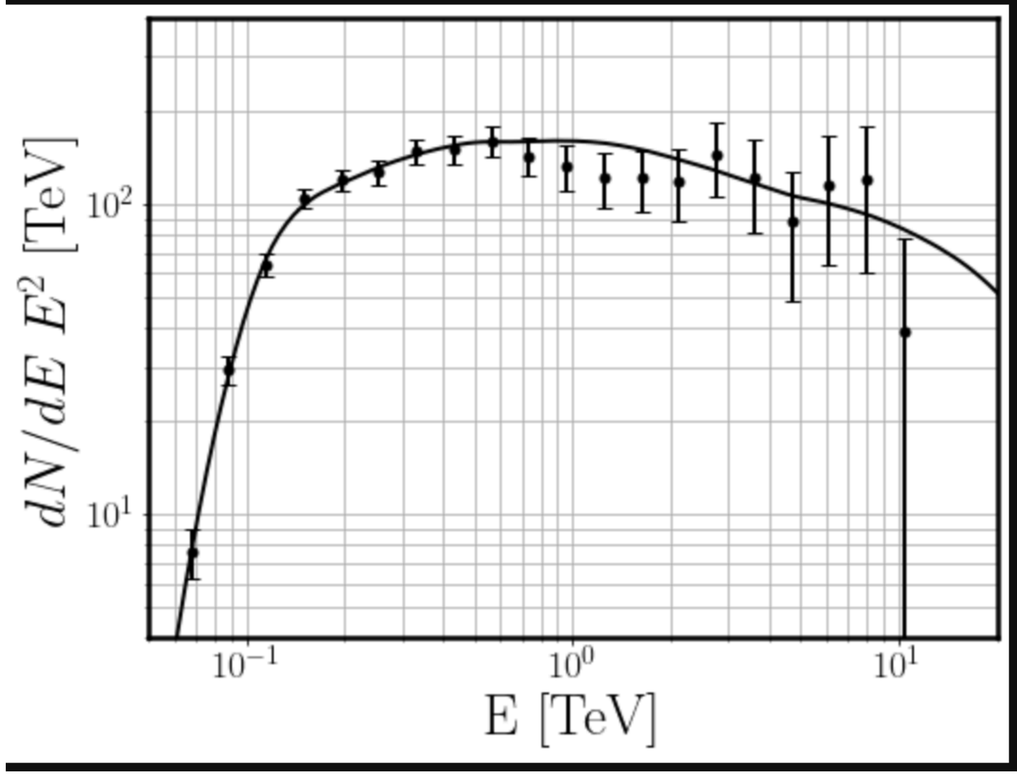
ENERGY	RA	DEC
TeV	deg	deg
float64	float64	float64
9.660178387300016	150.58	-13.260000000000005
8.444854455961387	150.58	-13.260000000000005
11.253832764582006	150.58	-13.260000000000005
8.028920247115588	150.58	-13.260000000000005

Some extra features - *plot_predicted/observed_counts()*

```
def plot_predicted_dnde(self, ax=None, fig=None, obs=True,bkg=False, signal=True, line=True,**kwargs):
    """Plot the predicted differential counts dN/dE * E**2

    Parameters
    ----------
    energy       : Quantity
           must have energy dimension
    obs          : Bool
           True if dN/dE in observed  energy,
           i.e. by incluidng edisp effects
           By default is True
    signal        : Bool
           True if you want to include in the
           simulation the signal of gamma
           By default is True
    bkg          : Bool
           True if you want to include in the
           simulation the background
           By default is True
    """
```

```
def plot_observed_dnde(self, ax=None, fig=None, emin=None,emax=None,en_bins=10,**kwargs):
    """Plot the observed differential counts dN/dE * E**2

    Parameters
    ----------
    energy       : Quantity
           must have energy dimension
    obs          : Bool
           True if dN/dE in observed  energy,
           i.e. by incluidng edisp effects
           By default is True
    signal        : Bool
           True if you want to include in the
           simulation the signal of gamma
           By default is True
    bkg          : Bool
           True if you want to include in the
           simulation the background
           By default is True
    """
```

```
fig, ax = plt.subplots(figsize=(8,6),nrows=1, ncols=1)
ax.set_ylim(bottom=4e0,top=4e2)
ax.set_xlim([5e-2,2e1])

#Plot Model SED
plot_dict = dict(color='black',linewidth=2)
ax, _     = dataset.plot_predicted_dnde(ax=ax,fig=fig, **plot_dict)
#Plot OBSERVED SED
plot_dict = dict(color='black')
ax, _     = dataset.plot_observed_dnde(ax=ax,fig=fig,en_bins=22,**plot_dict)
```



Fit with MapDataset and UnbinnedDataset

First we define an *UnbinnedDataset* and *MapDataset*

```
energy_axis      = MapAxis.from_energy_bounds( "0.01 TeV", "100 TeV", nbin=45, per_decade=True, name=
energy_axis_true = MapAxis.from_energy_bounds( "0.01 TeV", "100 TeV", nbin=45, per_decade=True, name=
migra_axis       = MapAxis.from_bounds(0.5, 2, nbin=150, node_type="edges", name="migra")
geom    = WcsGeom.create(frame="icrs", skydir=pointing, width=(2, 2), binsz=0.02, axes=[energy_axis])
maker   = MapDatasetMaker(selection=["exposure","edisp",  ])

#  UnbinnedDatase
d_empty                 = UnbinnedDataset.create( geom,
                            energy_axis_true=energy_axis_true,
                            migra_axis=migra_axis, name="my-dataset")
unbinned_dataset        = maker.run(d_empty, observation)
unbinned_dataset.models = models

#  MapDatase
d_empty                 = MapDataset.create( geom,
                            energy_axis_true=energy_axis_true,
                            migra_axis=migra_axis, name="my-dataset")
dataset                 = maker.run(d_empty, observation)
dataset.models          = models
```

1

Then we simulate events using *.fake_events()*

```
###################################
# SIMULATING EVENTS
###################################
n_obs = 100
unbinned_datasets = Datasets()
datasets          = Datasets()

for idx in range(n_obs):
    #UNBINNED
    unbinned_dataset_fake                       = unbinned_dataset.copy(name=f"obs-{idx}")
    unbinned_dataset_fake.meta_table["OBS_ID"] = [idx]
    unbinned_dataset_fake.models                = models.copy()
    unbinned_dataset_fake.fake_events(bkg=False,random_state=idx)   Events simulation
    unbinned_datasets.append(unbinned_dataset_fake)

    #BINNED
    dataset_fake                                = dataset.copy(name=f"obs-{idx}")
    dataset_fake.meta_table["OBS_ID"]           = [idx]
    dataset_fake.models                         = models.copy()
    # counts from the simulated events
    fake_counts                                 = Map.from_geom(dataset_fake.geoms["geom"])
    fake_counts.fill_events(unbinned_dataset_fake.events)
    dataset_fake.counts                         = fake_counts
    datasets.append(dataset_fake)
```

2

Lastly, we fit the paramters using the *Fit* class

```
###################################
# PERFORM THE FIT
###################################

for unbinned_idataset, idataset in zip(unbinned_datasets, datasets):

    # Unbinned
    fit     = Fit()
    result  = fit.optimize(unbinned_idataset)
    # Binned
    fit     = Fit()
    result  = fit.optimize(idataset)
```

3

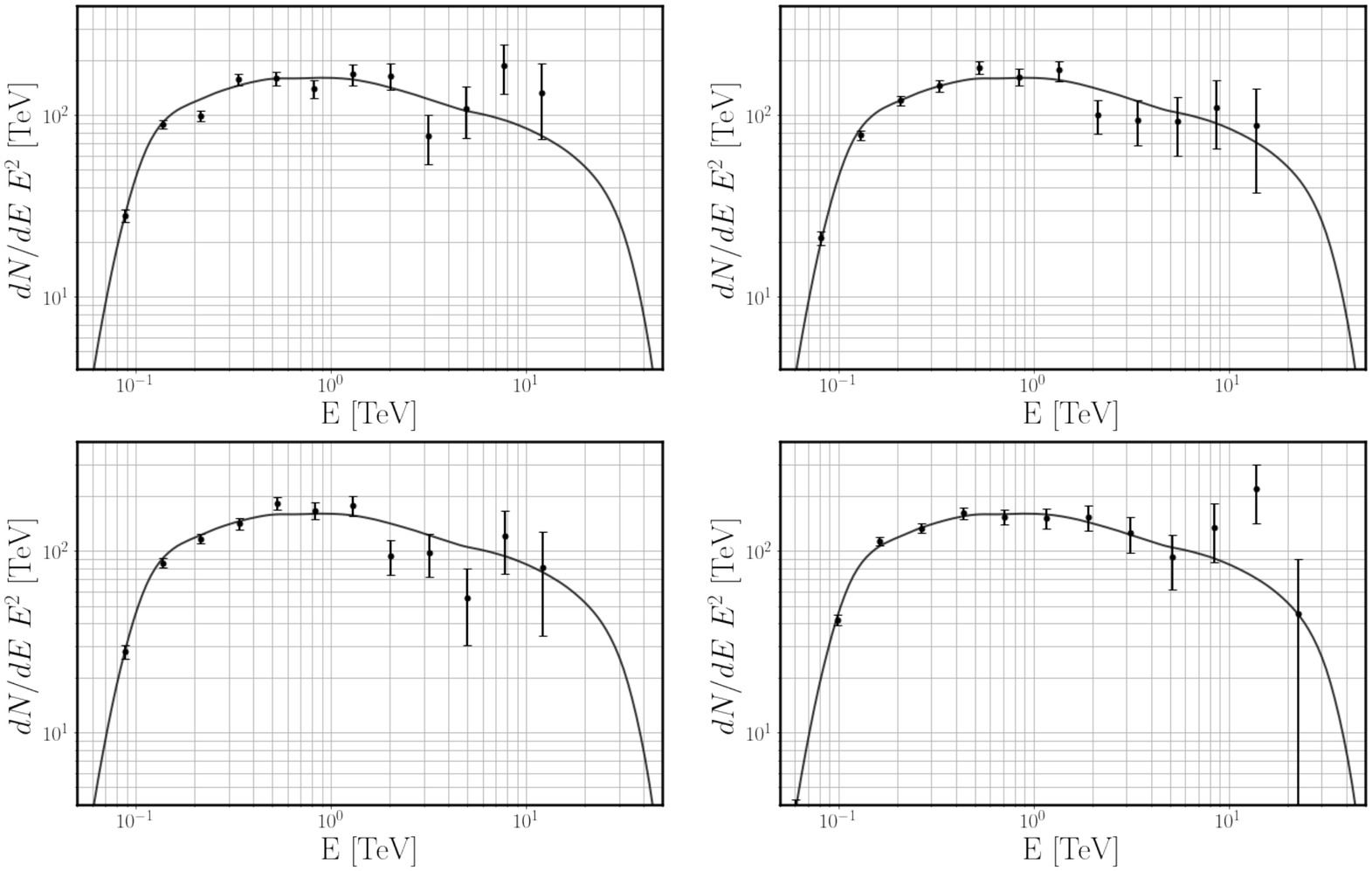
Tests of performance - a Power-Law fit

$$\Phi_0 \cdot \left(\frac{E}{1\ \mathrm{TeV}} \right)^{-\gamma}$$

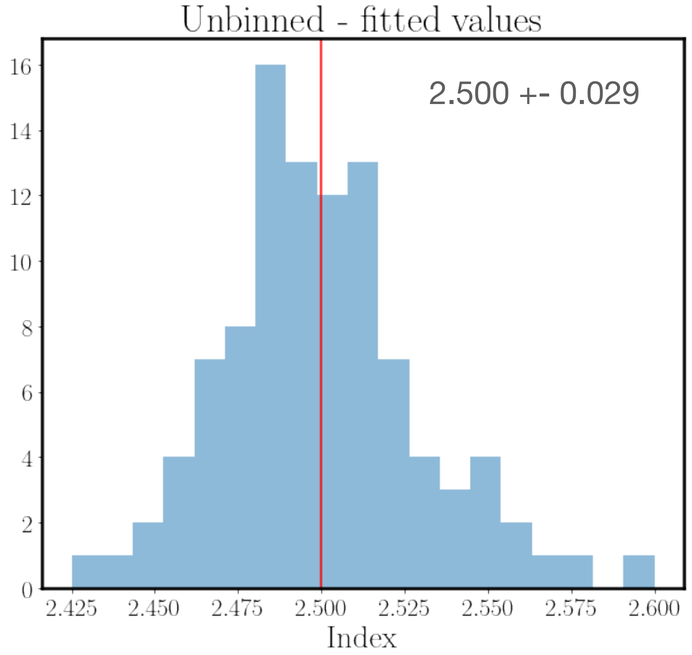
$$\Phi_0 = 10^{-12}/(cm^2\ s\ TeV)$$

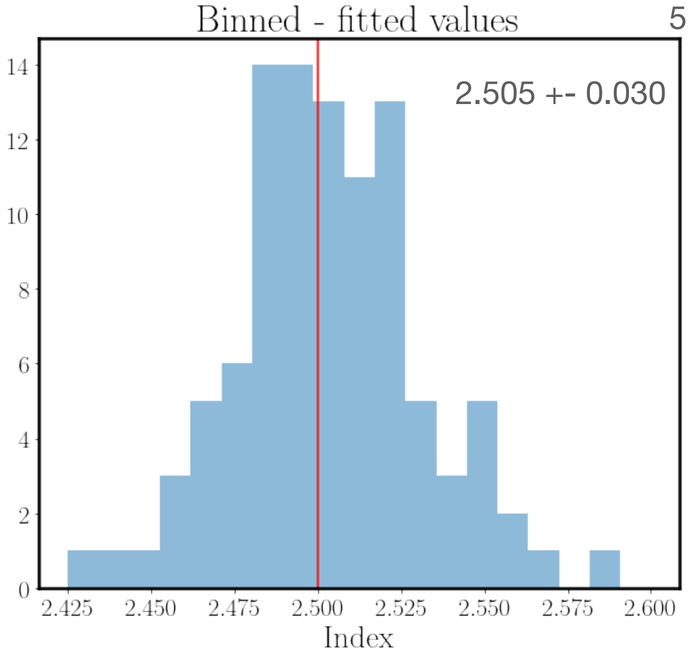
$$\gamma = 2.5$$

Assuming this model, we perform **100 simulations** and we fit them using the "standard" ***MapDataset*** or the new ***UnbinnedDataset*** class

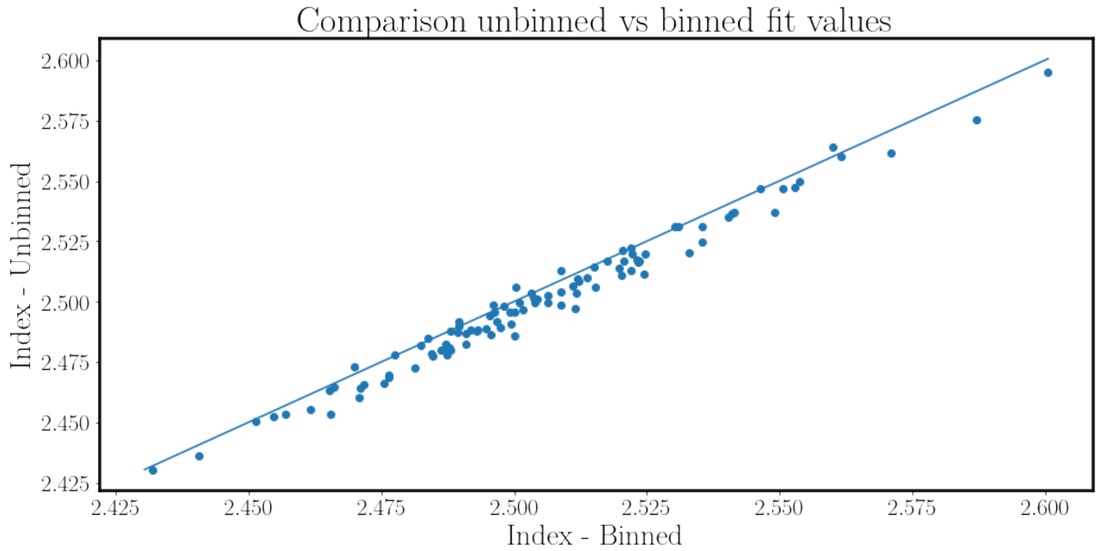


Tests of performance - a Power-Law fit

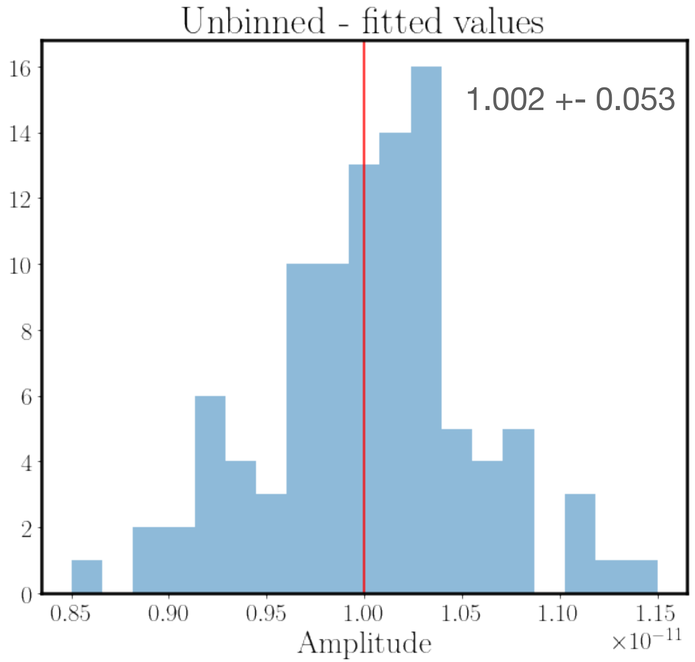


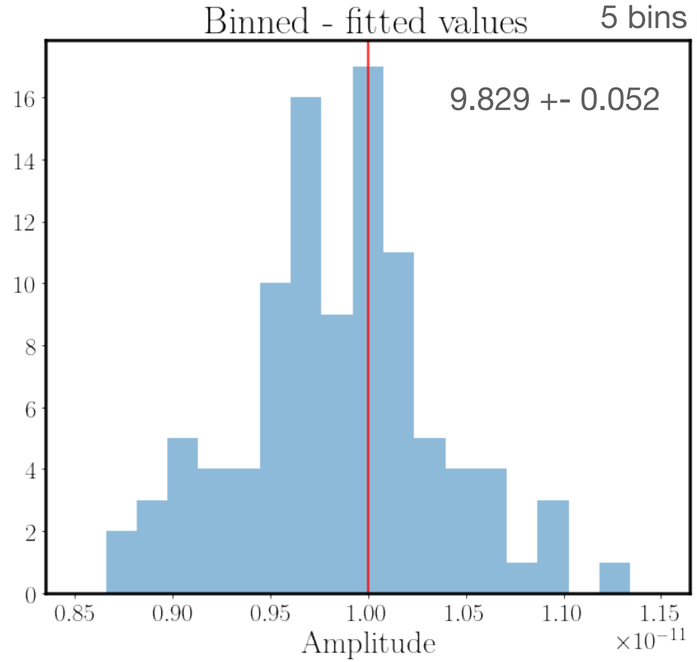


5 bins per decade

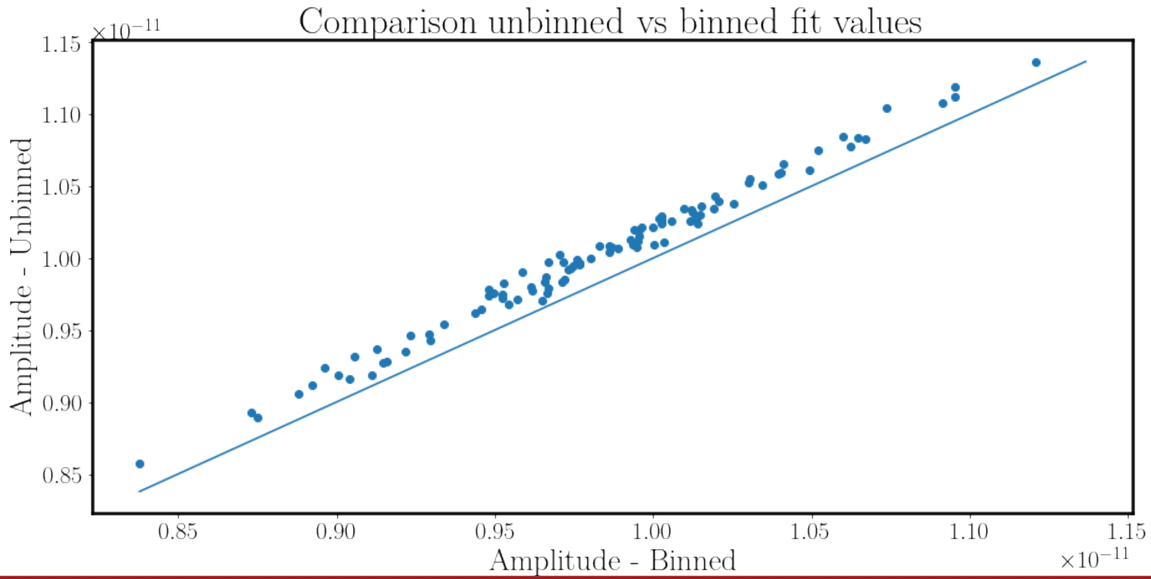


Tests of performance - a Power-Law fit



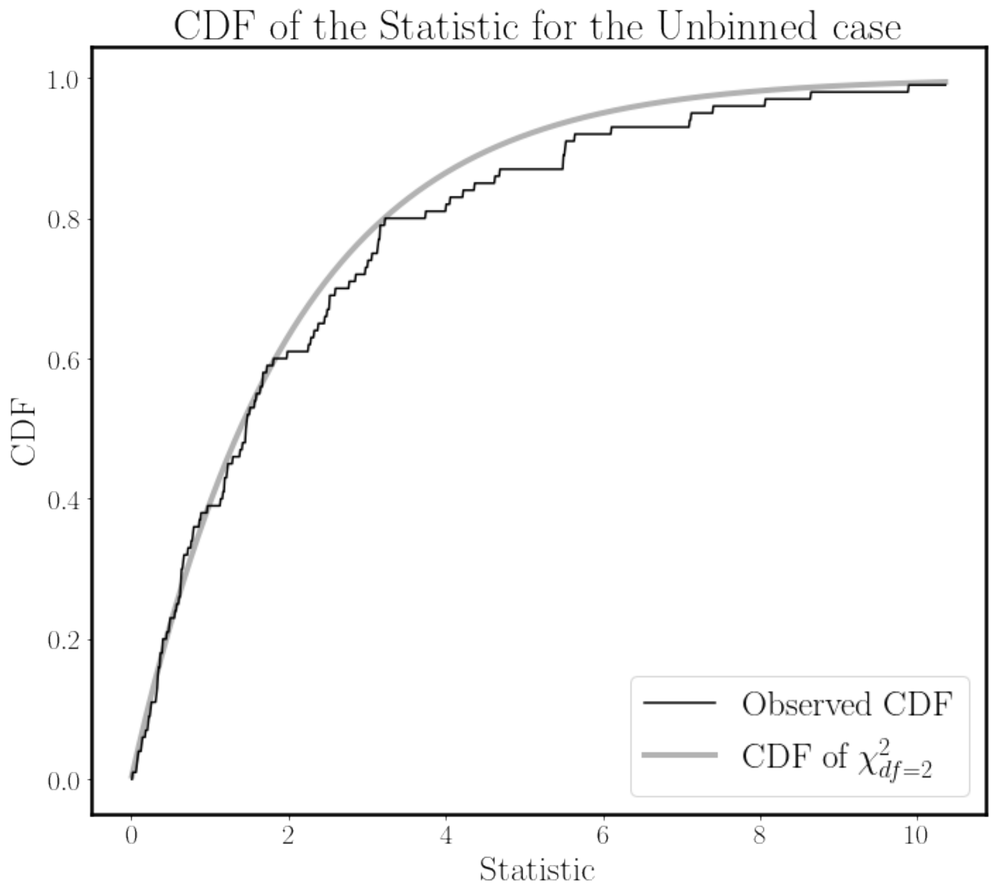


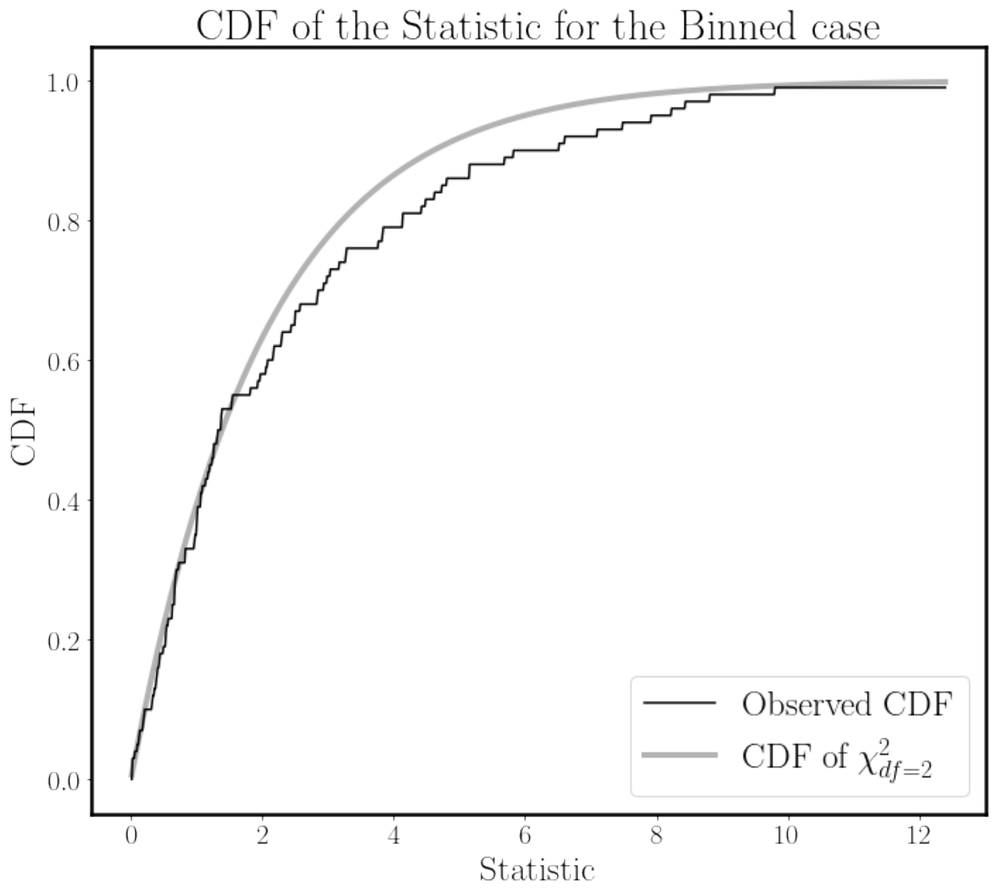
5 bins per decade



Tests of performance - a Power-Law fit

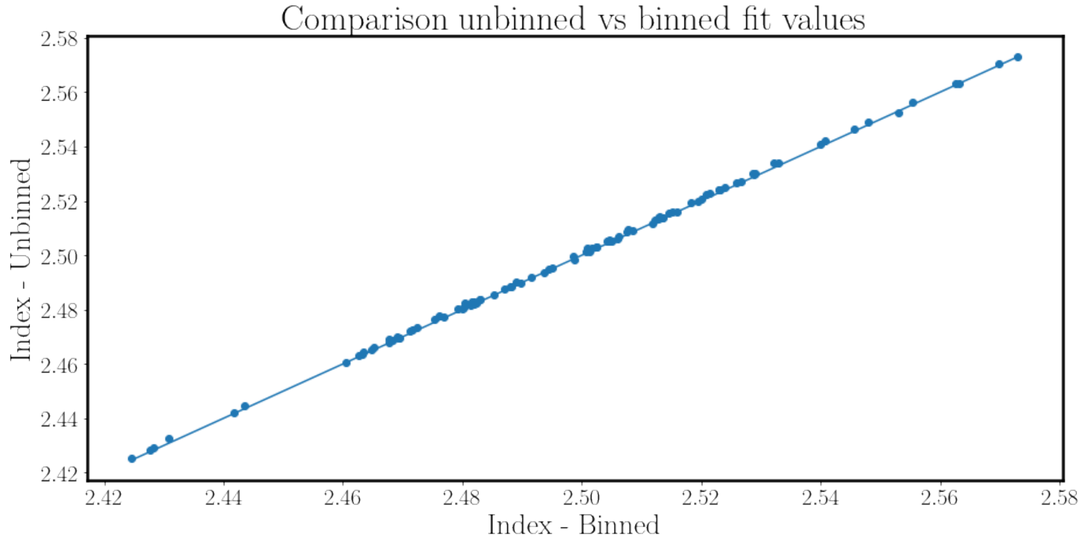
The new **Unbinned** Likelihood statistic is behaving like a **statistic**

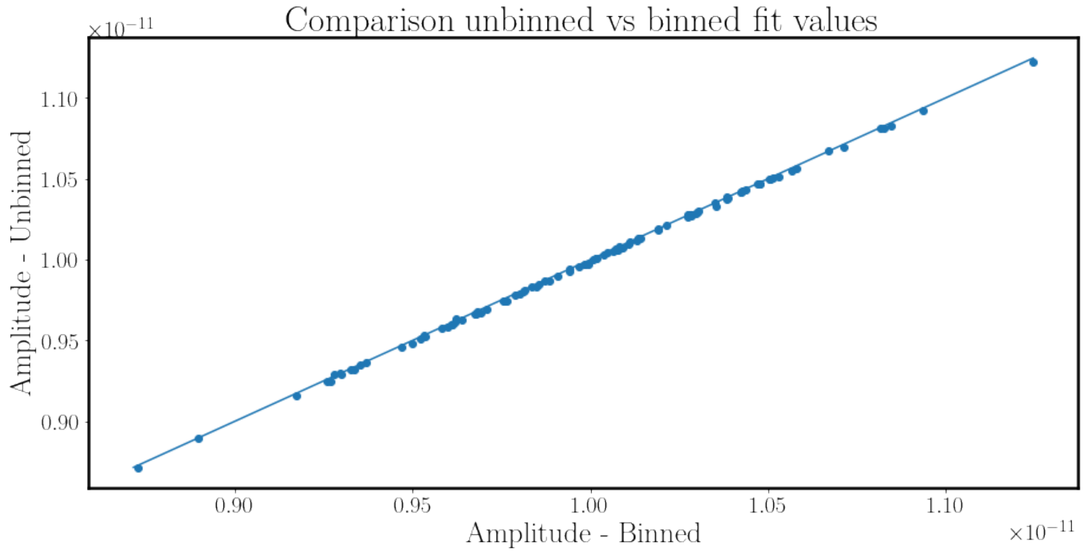




Tests of performance - a Power-Law fit

What if we **increase** the number of bins for the binned approach?

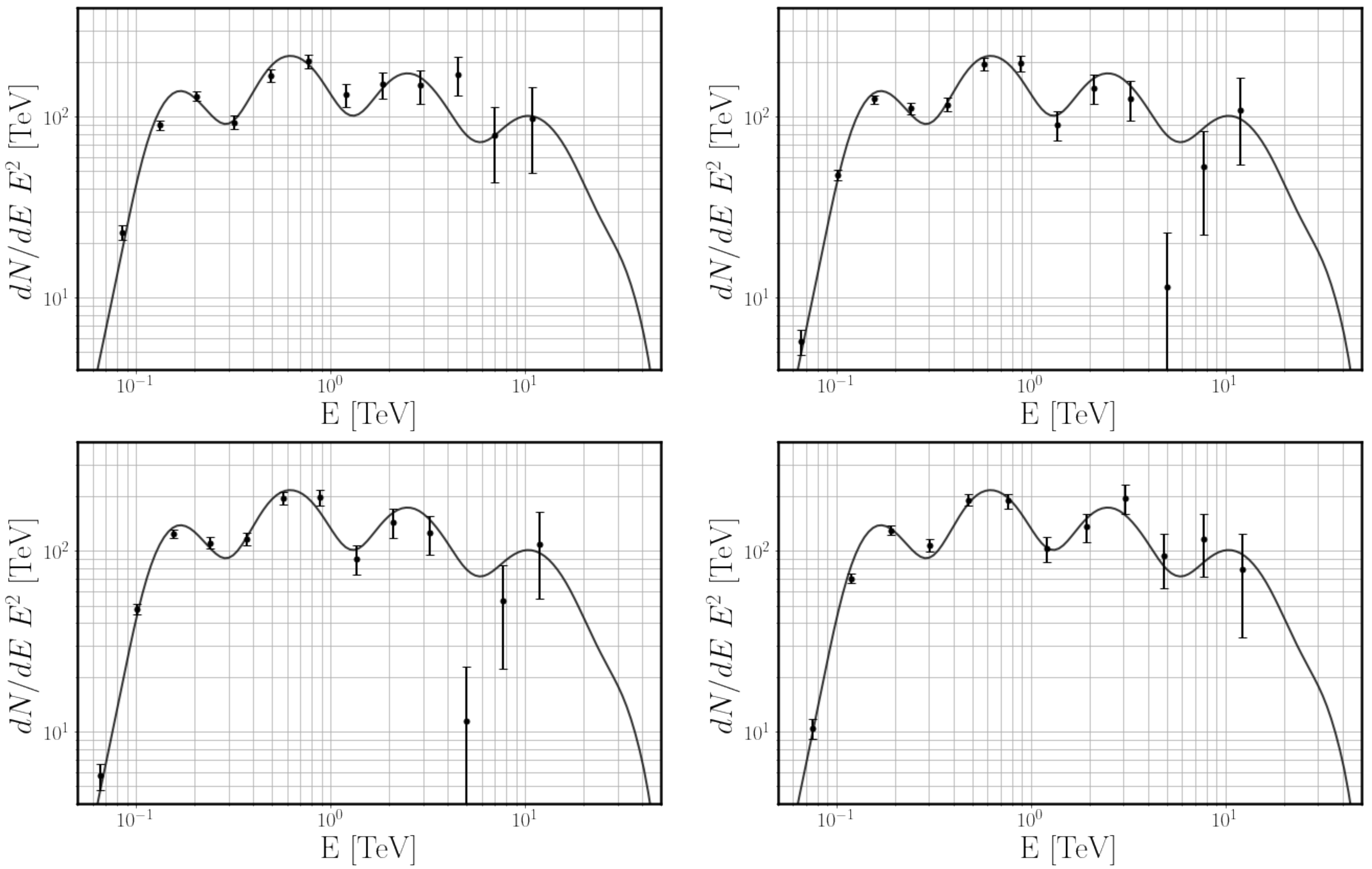




The two classes (***MapDataset*** and ***UnbinnedDataset***) provide the same **fitted parameters** when the number of bins is very large (45 per decade)

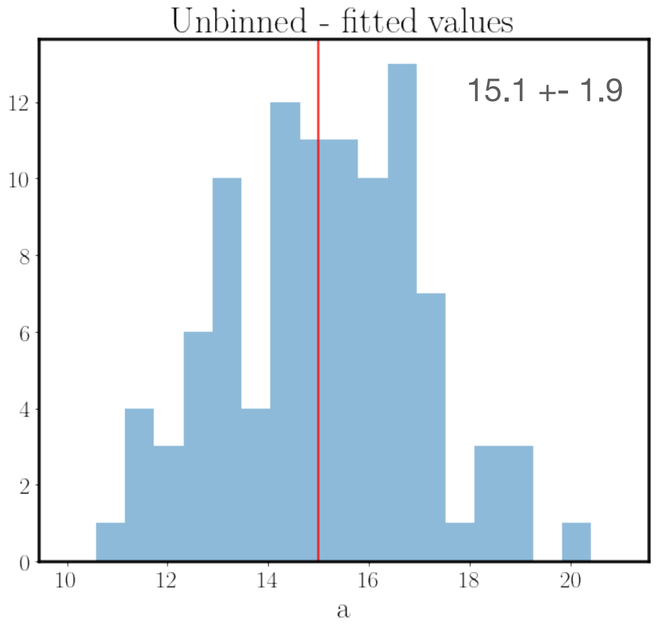
This confirms that the new class is working good!

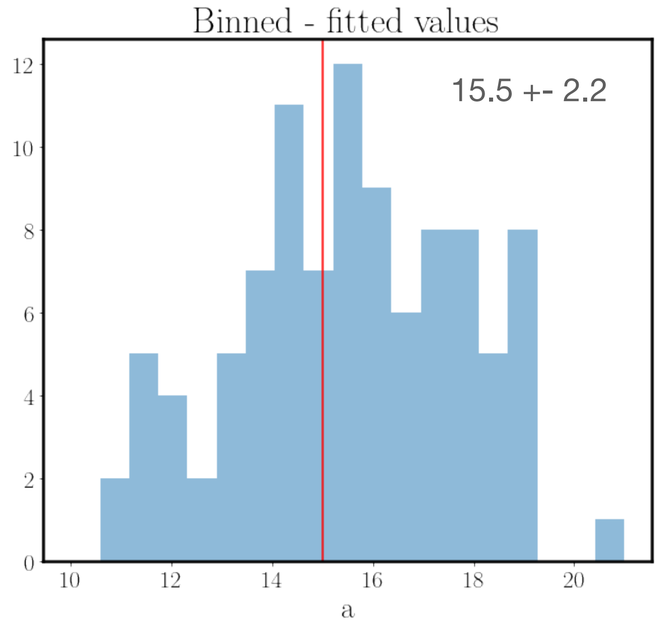
Tests of performance - a custom ToyModel

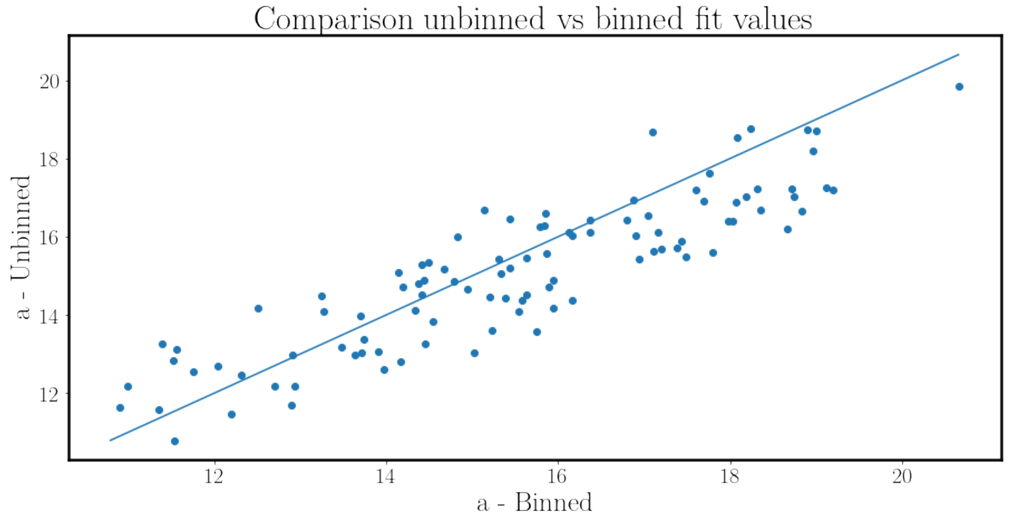


The only free parameter is "***a***" which is a dimensionless variable that is proportional to the intensity of the *wiggling*

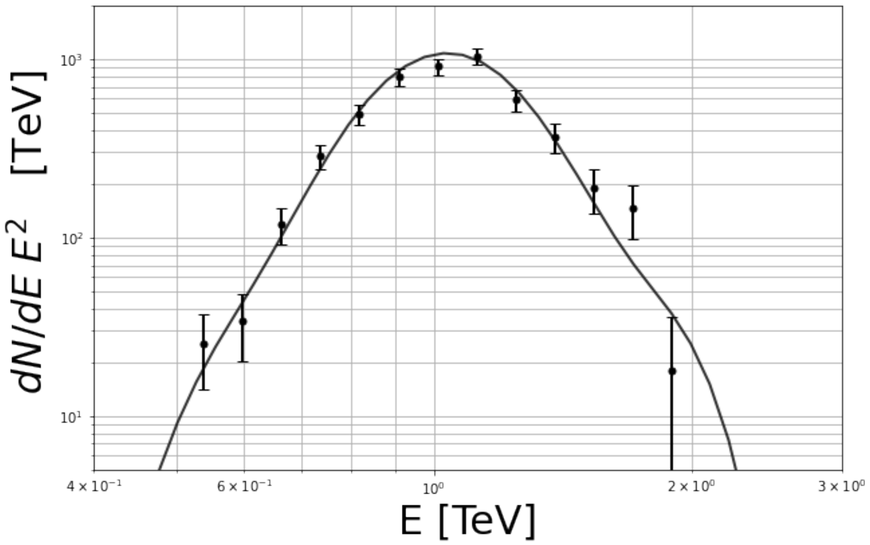
Tests of performance - a custom ToyModel

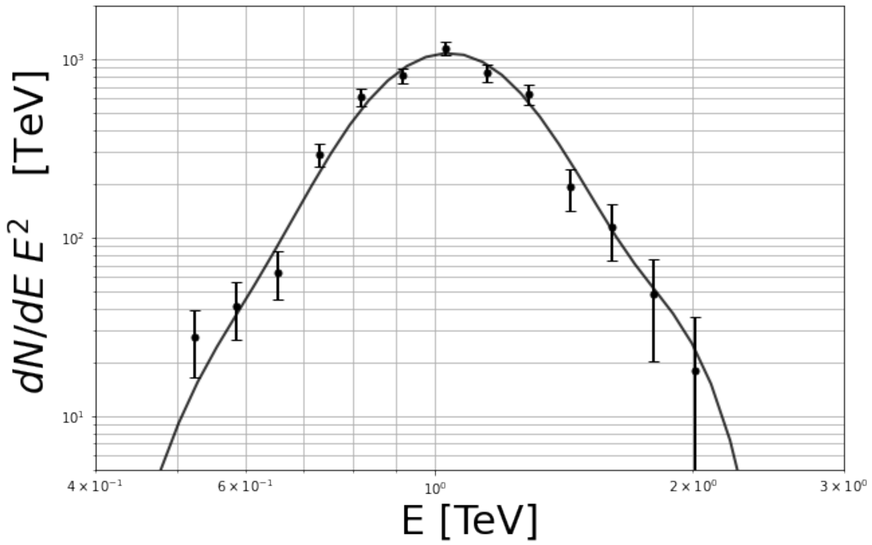


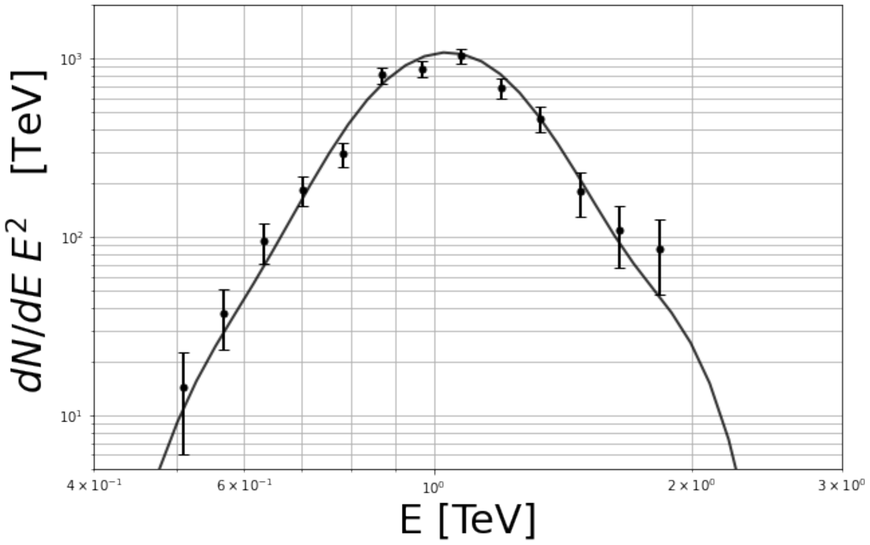


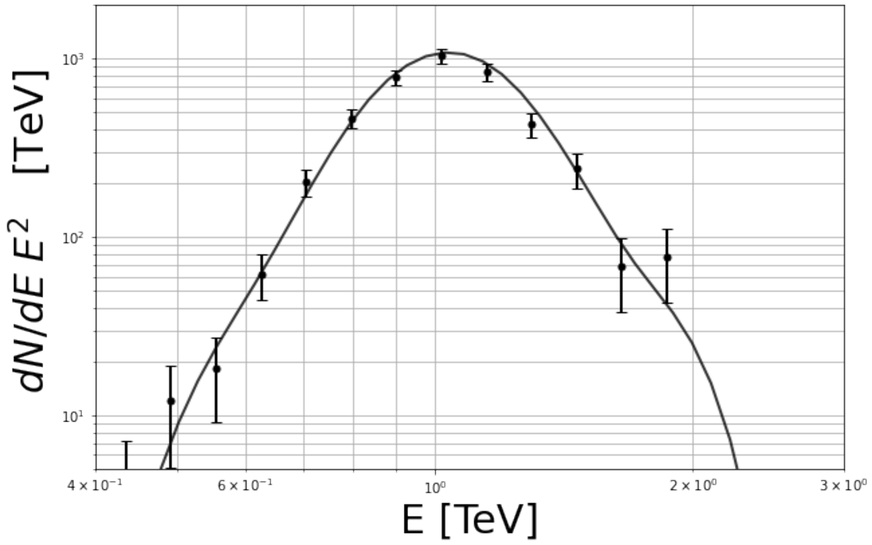


Tests of performance - a Gaussian model



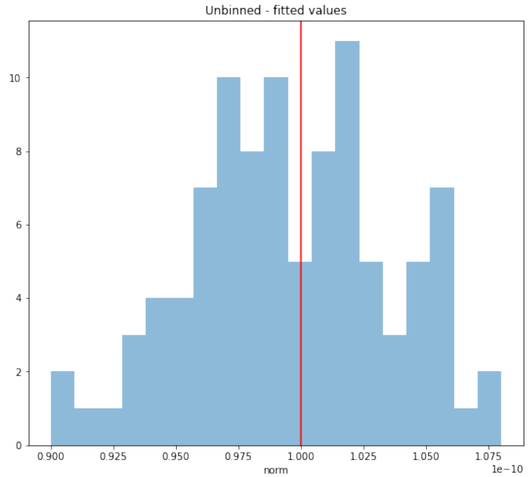


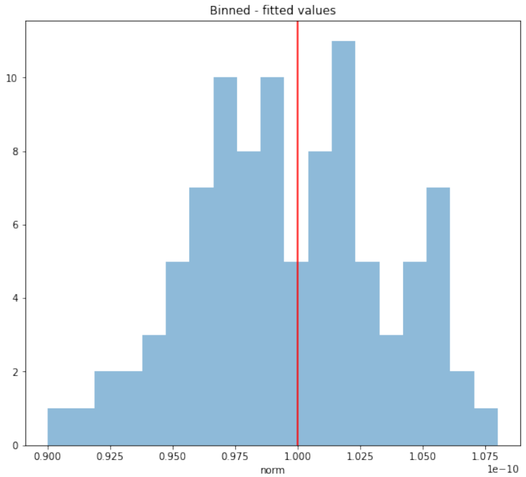




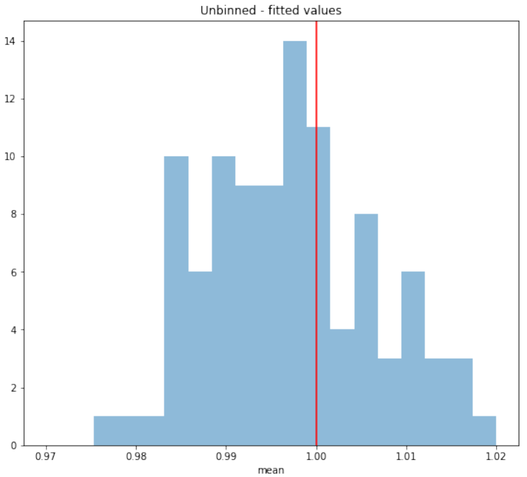
Tests of performance - a Gaussian model

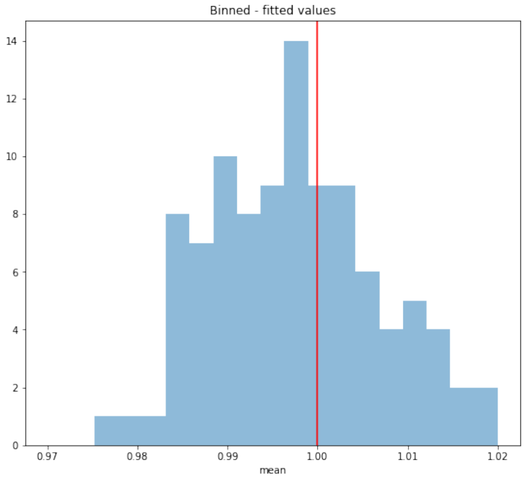
NORM



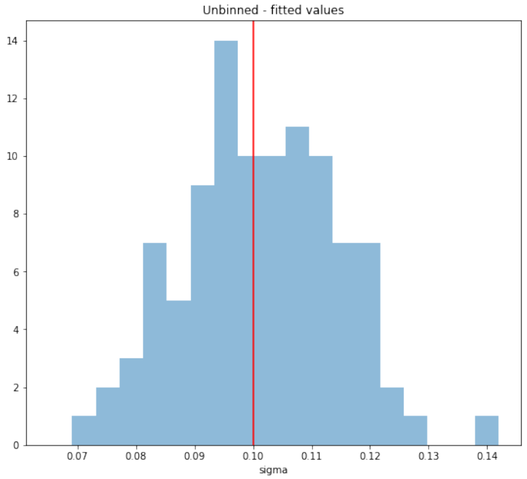


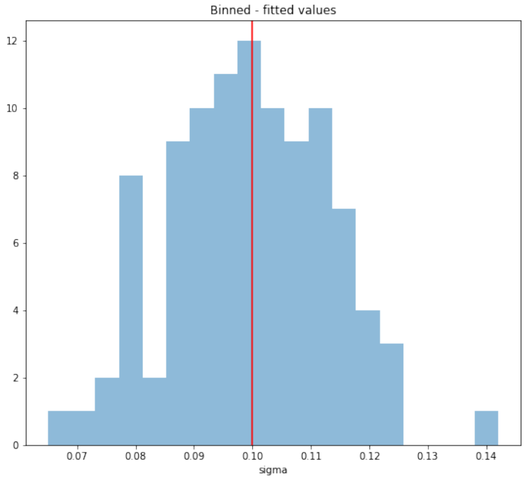
MEAN



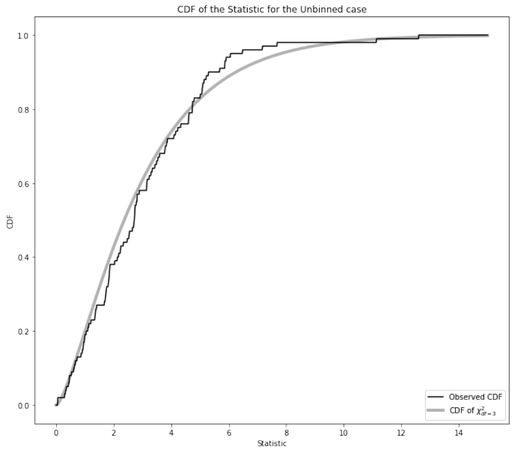


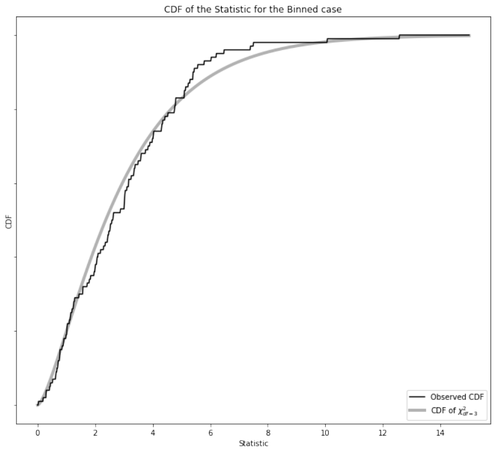
SIGMA





Statistic test CDF





Outlook

- This preliminary “**unbinned**” version of the class *MapDataset* looks doing its job well:
  - its **usage** is the same of a “standard” Dataset class
  - its **performance** has been tested for different scenarios and it is in agreement with the binned approach in the limit of infinite bins

- **Next?**
  - Extend the Unbinned likelihood from 1D to 3D by including also the **spatial coordinates**
  - Extend the Unbinned likelihood to an **On/Off measurement**
  - Make an “unbinned” version **not only** for the *MapDataset* class, but also for the other Dataset classes