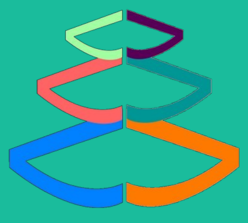


pynu analysis framework

(organizer shouldn't have put me in high-E)

Pablo – *DIPC*

2023/06/14 – local νACA

How, what and why?

The origin was when trying to make a combined analysis of all atmospheric neutrino experiments
→ First we wanted to combine SK and IceCube

Measuring Oscillations with A Million Atmospheric Neutrinos

C. A. Argüelles [1,*] P. Fernández [2,3,†] I. Martínez-Soler [1,‡] and M. Jin (靳淼辰) [1,§]

[1] *Department of Physics & Laboratory for Particle Physics and Cosmology, Harvard University, Cambridge, MA 02138, USA*

[2] *University of Liverpool, Department of Physics, Liverpool, United Kingdom*

[3] *Donostia International Physics Center DIPC, San Sebastián/Donostia, E-20018, Spain*

→ Then, we thought why don't we add ORCA and set our study to foresee the sensitvity by 2030?

→ But, why don't you add HyperK too, referee said...

The solution was to make an analysis framework that allows any number of experiments of any kind and takes into account and out-of-the-box all the correlations among common systematics

How, what and why?

As this was done with 50% theoreticians, the parametrization of systematic errors was done analytically and event-by-event

→ This is very time consuming
→ The number of systematics is of $O(10^2)$
→ Which meant really long times for each point we wanted to probe

The solution was to analytically differentiate the function parametrizing the systematics and compute the gradient of the log-likelihood (more latter).
This reduces the amount of time by several orders of magnitude (depends on the number of systematics)

In addition, the code was modified so any of these functions (which basically re-weight the simulation and called physics tunes from now on) can be not only systematics but parameters to fit

Now

Search or jump to... / Pull requests Issues Codespaces Marketplace Explore

pabloferm / Pynu Private

Code Issues Pull requests Actions Projects Security 1 Insights Settings

PyNu Framework

The aim of this software is to perform neutrino analysis in the most general and flexible way. There are three modules:

- PyNuFit: It is the core of the package handling simulations, data and fitting.
- Plot: A plotting toolkit to extract all the information from the analysis (UNDER CONSTRUCTION).
- Report: Automated module for preparing a report containing the detailed information of the analysis and the results (UNDER CONSTRUCTION).

For a more complete documentation, please open the `docs/pynu.html` folder with your browser.

Documentation is ongoing

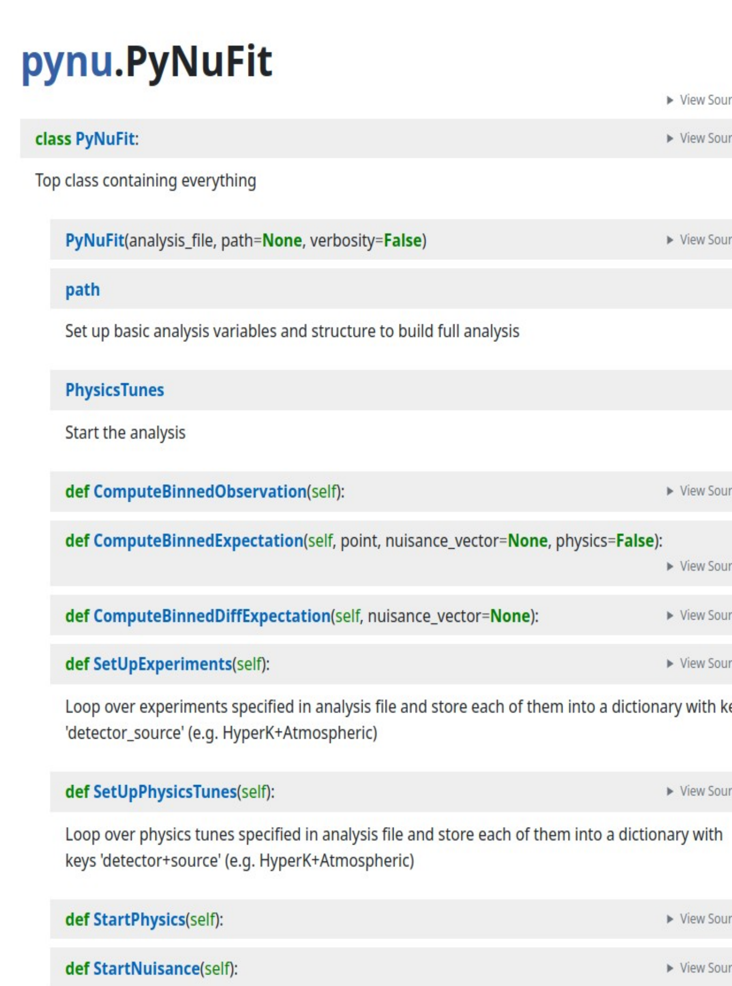
pynu

Search...

API Documentation

class PyNuFit

- PyNuFit()
- path
- PhysicsTunes
- ComputeBinnedObservation()
- ComputeBinnedExpectation()
- ComputeBinnedDiffExpectation()
- SetUpExperiments()
- SetUpPhysicsTunes()
- StartPhysics()
- StartNuisance()
- SetBinnedObservedEvents()
- SetExpectedWeights()
- SetBinnedExpectedEvents()
- SetBinnedDiffExpectedEvents()
- ApplyFixedWeights()
- ApplyNominalWeights()
- ApplyTrueWeights()
- ApplyPhysicsWeights()
- ApplyNuisanceWeights()
- ApplyOscillations()
- ApplyWeights()
- GetDiffLogWeights()
- set_likelihood()
- FitModel()



Introduction and main objectives

The project is becoming more robust and designed to accommodate all the features and more of all the current existing fitters in HK

- Event-by-event oscillations or binned following Magnus expansion
- Markov Chain Monte Carlo fit or grid of points
- Pre-binned and pre-computed systematics or event-by-event

…

- All the fitters use binned log-likelihood method, pynu also allows to fit using unbinned log-likelihood

→ *Basics are in place – also discussing with Thorsten, Federico and Mathias for ML-based KDE, https://arxiv.org/abs/2002.09436*

Introduction and main objectives

- **Neutrino analysis software (in Python) for neutrino oscillations, flux and cross-sections**

- **Focus on flexibility: easy to include any neutrino source, cross-section model and detector**
  Same framework, different analyses and any combination of them.

- **Made to accommodate any number of experiments accounting for their correlations**
  Joint analyses can be performed out of the box or with very little modifications.
  The plan for is to include as many experiments and features as possible as the project develops (already atmospherics for HK, all SK phases for official MCs, and DeepCore, IC, ORCA → next step accelerators)

Introduction and main objectives

- **Systematics, physics parameters are treated in the same way and generally called Physics Tunes**
  These are functions for each parameter of the flux, cross section, detector or oscillations
  Each of them can be treated as systematics (nuisance), physics or fixed in an analysis
- **Includes analytic calculations to improve fit convergence**
  These analyses usually require high CPU resources due to large simulation data-sets and large quantity of systematics.
  To overcome this, the code computes analytically (when possible) the gradient of the log-likelihood over the systematics space

Definitions

- **Experiment:** detector + source

- **Analysis is made of:**
  - Detectors – including simulations and data
  - Neutrino sources
  - Cross-sections
  - Oscillations

- **Types of parameters:**
  - Fixed: does not change in the analysis but allows to reweight the simulations to test different models
  - Nuisance: systematic paramters
  - Physics: free parameters to be fitted with the provided data or simulation

The definition

Everything is specified in a single input xml file

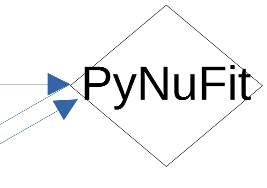
```
<Analysis>

<!--Anything common to a given neutrino source-->

<!-- Atmospheric neutrinos -->
    <NeutrinoSource name="Atmospheric">
        <status> 1 </status>
        <physics name='normalization_below1GeV'>
        </physics>
        <nuisance name='nunubar_ratio'>
        </nuisance>
        <nuisance name='flavor_ratio'>
        </nuisance>
        <nuisance name='tilt'>
        </nuisance>
    </NeutrinoSource>

<!-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


<!--Anything common to a given detector type-->

<!-- Water-Cherenkov -->
    <NeutrinoTarget name="Water">
        <status> 1 </status>
        <physics name='XSecNuTau'>
        </physics>
        <nuisance name='DIS'>
        </nuisance>
        <nuisance name='CCQE'>
        </nuisance>
        <fixed name='NCHad'>
        </fixed>
    </NeutrinoTarget>
<!-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
<!-- Hyper-Kamiokande -->
    <NeutrinoExperiment name="HyperK">
        <status> 1 </status>
        <target name="Water"> </target>
        <source name='Atmospheric'>
            <status> 1 </status>
            <MCFiles name='/home/pablofer/HyperK/HKOsc/SK/allSK4_mc.root.hdf5'>
            </MCFiles>
            <exposure> 10 </exposure> <!-- years-->
            <MCexposure> 500 </MCexposure> <!-- This is total exposure of all the specified MC files-->
            <DataFiles name='...'>
            </DataFiles>
        </source>
    </NeutrinoExperiment>


<!--Anything common to a SINGLE given neutrino physics scenario-->

<!-- Pick one: 3-Osc, 3+N-Osc, LV, NSI -->
<!-- Please, specify all non-zero parameters. If not specified, the missing parameters are assumed to be zero -->

    <NeutrinoOscillations name="3-Osc">
        <status> 1 </status>
        <flavors> 3 </flavors>
        <fixed name='Sin2Theta12'>
        </fixed>
        <fixed name='Sin2Theta13'>
        </fixed>
        <physics name='Sin2Theta23'>
        </physics>
        <fixed name='Dm221'>
        </fixed>
        <physics name='dCP'>
        </physics>
        <physics name='Dm231'>
        </physics>
        <physics name='Ordering'>
        </physics>
    </NeutrinoOscillations>

</Analysis>
```

Overview

xml analysis file

contains all the relevant information entering the analysis, i.e. experiments, source, nuisance, models

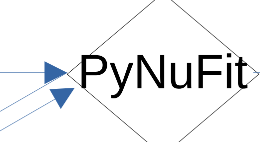


xml reader

sets all the analysis items

Overview

xml analysis file
contains all the relevant information entering the analysis, i.e. experiments, source, nuisance, models



Physics Tunes
functions to compute weights for each item specified in the analysis
- flux, detector, cross-section, oscillations
- fixed, nuisance, physics
It also contains the analytic derivative of (most of) these functions

xml reader
sets all the analysis items

Overview

xml analysis file

contains all the relevant information entering the analysis, i.e. experiments, source, nuisance, models

xml reader

sets all the analysis items

PyNuFit

Computes observed and expected events and the gradient of the latter

Physics Tunes

functions to compute weights for each item specified in the analysis

- flux, detector, cross-section, oscillations

- fixed, nuisance, physics

It also contains the analytic derivative of (most of) these functions

Overview

xml analysis file
contains all the relevant information entering the analysis, i.e. experiments, source, nuisance, models

xml reader
sets all the analysis items

PyNuFit
Computes observed and expected events and the gradient of the latter

Physics Tunes
functions to compute weights for each item specified in the analysis
- flux, detector, cross-section, oscillations
- fixed, nuisance, physics
It also contains the analytic derivative of (most of) these functions

Fitter
currently binned $\chi 2$ assuming Poisson statistics
uses the derivatives of PT to compute
- the gradient of $\chi 2$
- intial guesses of the nuisance which minimize the $\chi 2$

Overview

xml analysis file

contains all the relevant information entering the analysis, i.e. experiments, source, nuisance, models

xml reader

sets all the analysis items

PyNuFit

Computes observed and expected events and the gradient of the latter

Write output

Physics Tunes

functions to compute weights for each item specified in the analysis

- flux, detector, cross-section, oscillations

- fixed, nuisance, physics

It also contains the analytic derivative of (most of) these functions

Fitter

currently binned $\chi 2$ assuming Poisson statistics

uses the derivatives of PT to compute

- the gradient of $\chi 2$

- intial guesses of the nuisance which minimize the $\chi 2$

An example to guide the explanation

HK atmospheric neutrinos using official SK-IV MC

From the previous xml analysis file

Physics			
Block	**Name**	**Truth**	**No. points**
Atmospheric Flux	Norm < 1 GeV	1	11
Cross section	σ (ντ)	1	9
Oscillations	$\sin^2\theta_{23}$	0.517	13
	Δm^2_{31}	0.0025	9

→ *11583 points*

Fixed		
Block	**Name**	**Value**
Cross section	NC hadron prod.	1.0
Oscillations, 3f	$\sin^2\theta_{12}$	0.304
	$\sin^2\theta_{13}$	0.022
	Δm^2_{21}	0.0000742
	δCP	4.14
	Ordering	normal

Systematics		
Block	**Name**	**Prior**
Atmospheric Flux	ν/anti-ν	5%
	e/μ	2%
	Spectral index	20%
Cross section	DIS	5.00%
	CCQE	10.00%

A test example

```
-----------------------------------
Neutrino sources considered:
 +  Atmospheric
===================================
-----------------------------------
Neutrino targets considered:
 +  Water
===================================
-----------------------------------
Detectors considered:
 +  HyperK
===================================
Notice: Parameter dCP has been moved to fixed.
Notice: Parameter Ordering has been moved to fixed.
-----------------------------------
Oscillation scenario:
 +  3-Osc
===================================
List of Nuisance
 + From Atmospheric: ['nunubar_ratio', 'flavor_ratio', 'tilt']
 + From Water: ['DIS', 'CCQE']
 + From HyperK: []
 + From 3-Osc: []
===================================
List of Physics/Fit
 + From Atmospheric: ['normalization_below1GeV']
 + From Water: ['XSecNuTau']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta23', 'Dm231']
===================================
List of Fixed
 + From Atmospheric: []
 + From Water: ['NCHad']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta12', 'Sin2Theta13', 'Dm221', 'dCP', 'Ordering']
===================================
You have specified the following files for each experiment and source:
 - MC files for Atmospheric in HyperK
   + ['/home/pablofer/HyperK/HKOsc/SK/allSK4_mc.root.hdf5']
 - Data files for Atmospheric in HyperK
   + []
```

Atmospheric neutrino sensitivity analysis assuming 10 years of HyperK using SK-IV official MC with H-neutron tagging.

The MC is scaled by 8.2 accounting for the different volumen of HK w.r.t. SK.

Reduced number of systematics:

A test example

```
-------------------------------------
Neutrino sources considered:
 +  Atmospheric
=====================================
-------------------------------------
Neutrino targets considered:
 +  Water
=====================================
-------------------------------------
Detectors considered:
 +  HyperK
=====================================
Notice: Parameter dCP has been moved to fixed.
Notice: Parameter Ordering has been moved to fixed.
-------------------------------------
Oscillation scenario:
 +  3-Osc
=====================================
List of Nuisance
 + From Atmospheric: ['nunubar_ratio', 'flavor_ratio', 'tilt']
 + From Water: ['DIS', 'CCQE']
 + From HyperK: []
 + From 3-Osc: []
=====================================
List of Physics/Fit
 + From Atmospheric: ['normalization_below1GeV']
 + From Water: ['XSecNuTau']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta23', 'Dm231']
=====================================
List of Fixed
 + From Atmospheric: []
 + From Water: ['NCHad']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta12', 'Sin2Theta13', 'Dm221', 'dCP', 'Ordering']
=====================================
You have specified the following files for each experiment and source:
 - MC files for Atmospheric in HyperK
   + ['/home/pablofer/HyperK/HKOsc/SK/allSK4_mc.root.hdf5']
 - Data files for Atmospheric in HyperK
   + []
```

Atmospheric neutrino sensitivity analysis assuming 10 years of HyperK using SK-IV official MC with H-neutron tagging.

The MC is scaled by 8.2 accounting for the different volumen of HK w.r.t. SK.

Reduced number of systematics:

A test example

```
-------------------------------------
Neutrino sources considered:
 +  Atmospheric
=====================================
-------------------------------------
Neutrino targets considered:
 +  Water
=====================================
-------------------------------------
Detectors considered:
 +  HyperK
=====================================
Notice: Parameter dCP has been moved to fixed.
Notice: Parameter Ordering has been moved to fixed.
-------------------------------------
Oscillation scenario:
 +  3-Osc
=====================================
List of Nuisance
 + From Atmospheric: ['nunubar_ratio', 'flavor_ratio', 'tilt']
 + From Water: ['DIS', 'CCQE']
 + From HyperK: []
 + From 3-Osc: []
=====================================
List of Physics/Fit
 + From Atmospheric: ['normalization_below1GeV']
 + From Water: ['XSecNuTau']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta23', 'Dm231']
=====================================
List of Fixed
 + From Atmospheric: []
 + From Water: ['NCHad']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta12', 'Sin2Theta13', 'Dm221', 'dCP', 'Ordering']
=====================================
You have specified the following files for each experiment and source:
 - MC files for Atmospheric in HyperK
   + ['/home/pablofer/HyperK/HKOsc/SK/allSK4_mc.root.hdf5']
 - Data files for Atmospheric in HyperK
   + []
```

Atmospheric neutrino sensitivity analysis assuming 10 years of HyperK using SK-IV official MC with H-neutron tagging.

The MC is scaled by 8.2 accounting for the different volumen of HK w.r.t. SK.

Reduced number of systematics:

Arbitrary physics parameters to fit

A test example

```
-------------------------------------
Neutrino sources considered:
 +  Atmospheric
=====================================
-------------------------------------
Neutrino targets considered:
 +  Water
=====================================
-------------------------------------
Detectors considered:
 +  HyperK
=====================================
Notice: Parameter dCP has been moved to fixed.
Notice: Parameter Ordering has been moved to fixed.
-------------------------------------
Oscillation scenario:
 +  3-Osc
=====================================
List of Nuisance
 + From Atmospheric: ['nunubar_ratio', 'flavor_ratio', 'tilt']
 + From Water: ['DIS', 'CCQE']
 + From HyperK: []
 + From 3-Osc: []
=====================================
List of Physics/Fit
 + From Atmospheric: ['normalization_below1GeV']
 + From Water: ['XSecNuTau']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta23', 'Dm231']
=====================================
List of Fixed
 + From Atmospheric: []
 + From Water: ['NCHad']
 + From HyperK: []
 + From 3-Osc: ['Sin2Theta12', 'Sin2Theta13', 'Dm221', 'dCP', 'Ordering']
=====================================
You have specified the following files for each experiment and source:
 - MC files for Atmospheric in HyperK
   + ['/home/pablofer/HyperK/HKOsc/SK/allSK4_mc.root.hdf5']
 - Data files for Atmospheric in HyperK
   + []
```

Atmospheric neutrino sensitivity analysis assuming 10 years of HyperK using SK-IV official MC with H-neutron tagging.

The MC is scaled by 8.2 accounting for the different volumen of HK w.r.t. SK.

Reduced number of systematics:

Arbitrary physics parameters to fit

(code for converting root files into hdf5 format)

Physics Tunes: Neutrino Oscillations

- **Using nuSQuIDS (https://arxiv.org/pdf/2112.13804.pdf) package to handle almost any kind of neutrino oscillations and scenarios:**
  - 3-flavor oscillation scenario
  - Sterile ν
  - NSI
  - Lorentz violation

  Quantum decoherence and neutrino decay being included

pynu.PhysicsTunes

This module contains the PhysicsTunes class and all the submodules for each of the blocks. It also contains a set uilities which are still under construction.

- **Calculation is done event-by-event or on a grid**

Physics Tunes: Atm. Neutrino Flux

pynu.PhysicsTunes.Flux.AtmoFlux

▸ View Source

class AtmosphericFlux(PhysicsTunes.PhysicsTunes.Tune): ▸ View Source

Class containing the tunes for the atmospheric neutrino flux.

def normalization(self, experiment, x): ▸ View Source

Method for modifying the atmospheric flux normalization.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the weights from this tune.

def diff_normalization(self, experiment, x): ▸ View Source

Method for computing the derivative of the weights of the atm. flux normalization w.r.t. the tuning parameter.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the derivative of the `normalization` weights.

def tilt(self, experiment, x): ▸ View Source

Method for modifying the power-law of the atmospheric flux normalization taking as reference $E_\nu^0 = 10\ GeV$. That is $\Phi(E_\nu) \sim \left(\frac{E_\nu}{E_\nu^0}\right)^x$.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the weights from this tune.

def diff_tilt(self, experiment, x): ▸ View Source

Method for computing the derivative of the weights of the flux tilt w.r.t. the tuning parameter, i.e. $\frac{\partial\Phi(E_\nu)}{\partial x} \sim \left(\frac{E_\nu}{E_\nu^0}\right)^x \ln\left(\frac{E_\nu}{E_\nu^0}\right)$.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the derivative of the `tilt` weights.

def zenith_up(self, experiment, x): ▸ View Source

Method for modifying the zenith angle dependence of the up-going (negative $\cos\theta_{zen}$) fraction of the atmospheric flux assuming the relative uncertainty is parametrized as, $\eta(\cos\theta_{zen}) = 1 - x * \tanh^2(\cos\theta_{zen})$.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the weights from this tune.

def diff_zenith_up(self, experiment, x): ▸ View Source

Method for computing the derivative of the weights of zenith-dependence variation of up-going neutrinos w.r.t. the tuning parameter, i.e. $\frac{d\eta(\cos\theta_{zen})}{dx} = -\tanh^2(\cos\theta_{zen})$.

Arguments:

- **x (float):** Value of the tuning parameter.
- **experiment (** `pynu.Experiments.Experiment` **class):** Class containing the information of the experiment,
- of special interest are the Monte Carlos simulations.

Returns:

Numpy.array or float with the derivative of the `zenith_up` weights.

Physics Tunes: Neutrino-water cross section

Function for tau neutrino cross section tune

Its derivative

Physics Tunes

Everything is done in runtime and on an event-by event basis (by default), that means:

- Binning is done during runtime
- Systematics are implemented as functions and their effect is computed during runtime (no systematics pre-computation required)
- Allows any parametrization of systematics as function of any MC variable

We will focus on binned $\chi 2$ (~ $-2log(H/H_0)$) fit and minimizing over systematics

χ^2 calculation

- **Usual binned $\chi2$ calculation**

$$\chi^2 = 2\ln\left(\frac{\mathcal{L}(Exp(\vec{\theta})|Obs)}{\mathcal{L}(Obs|Obs)}\right)$$

Poisson stats. →

$$\chi^2 = 2\sum_i \left(E_i - O_i + O_i \ln\left(\frac{O_i}{E_i}\right)\right) + 2\sum_j \ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$$

O_i: Observed number of events in ith bin (data or simulation with assumed true values)
E_i: Expected number of events in ith bin at a give physics point and with nominal nuisance
E'_i: Expected number of events in ith bin modified by the values of nuisance parameters

μ_j: nominal value of jth nuisance parameter
P_j: Prior distribution for j^{th} nuisance parameter
x_j: current value of jth nuisance parameter

χ^2 calculation

pynu.fitter.BinnedLogLikelihoodRatio

▸ View Source

class BinnedLogLikelihoodRatio: ▸ View Source

Class containing all the information needed to perform an analysis and the methods for computing the log likelihood ratio ($-2\ln\left(\frac{L(Exp.)}{L(Obs.)}\right) \sim \chi^2$) given a set of binned observed data, binned expected events at a given physics point and nuisance parameters, and assuming Poisson statistics.

BinnedLogLikelihoodRatio(obervation, nominal_nuisance, sigma_nuisance, dist_nuisance) ▸ View Source

Initiates the class by storing the non-changing items of the χ^2 calculation.

Arguments:

- **obervation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array).
- **nominal_nuisance (list of float):** Produced from the xml analysis file, it contains the nominal values assumed for the nuisance parameters.
- **sigma_nuisance (list of float):** Produced from the xml analysis file, it contains the standard deviation values assumed for the nuisance parameters.
- **dist_nuisance (list of str):** Produced from the xml analysis file, it contains the type of distribution which is assumed for each nuisance.

def stats_only(self, expectation): ▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i\left(E_i - O_i + O_i\ln\left(\frac{O_i}{E_i}\right)\right)$, given the dictionary of binned expected number of events for each experiment of the analysis.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array): similarly to obervation, but for a given physics and nuisance values.

Returns:

Float with the value of χ^2.

def stats_and_systematics(self, expectation, nuisance): ▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i\left(E_i - O_i + O_i\ln\left(\frac{O_i}{E_i}\right)\right) + 2\sum_j\ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$, given the dictionary of binned expected number of events for each experiment of the analysis and taking into account the nuisance penalty terms.

Arguments:

- **expectation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Float with the value of χ^2 with nuisance.

def gradient(self, expectation, diff_expectation, nuisance): ▸ View Source

Returns the gradient of binned χ^2 computed analytically, given the dictionary of binned expected number of events for each experiment of the analysis and its derivative with respect to every nuisance parameter.

$$\nabla_j\chi^2 = 2\sum_i\left(1 - \frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j} + \frac{2}{P_j^{nuis}(x)}\frac{d\,P_j^{nuis}(x)}{dx_j}$$

LIMITATION: Currently, this is only done for nuisance following normal distributions. Other distributions like Beta will come soon.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Numpy array with each component of $\nabla\chi^2$.

def nuisance_penalty(self, nuisance): ▸ View Source

Returns the penalty term associated to nuisance parameters for the χ^2 computation.

Arguments:

- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by
- `pynu.analysis_reader.ParseXML` class.

Returns:

Float with $\sum_j\ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$.

def analytic_priors_bounds(self, expectation, diff_expectation): ▸ View Source

Returns the first-order values of the nuisance parameters which minimize the χ^2 at a given physics points. Here, first-order means we assume that the binned expected number of events is not modified by nuisance parameters, i.e. nuisance parameters are assumed to take the default value in this approximation.

$\nabla_j\chi^2 = 0$, and at first order, $E_i' \approx E_i + \frac{\partial E_i}{\partial x_j}(x_j - \mu_j)$, where E_i is the number of expected events with nuisance at their nomnial values.

$$\widetilde{x_j} = \mu_j + \frac{\sum\left(1-\frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}}{\sum\frac{O_i}{{E_i}^2}\left(\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}\right)^2 - \frac{1}{\sigma_j^2}}$$

Further, bounds for the final values of the nuisance parameters as follows.

$x_j \in [\widetilde{x_j} - \delta_j, \widetilde{x_j} + \delta_j]$, where $\delta_j = \min(2\cdot|\widetilde{x_j} - \mu_j|, \sigma)$

All this information is very useful for the minimizer to find faster the values of nuisance parameters minimizing the χ^2.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).

Returns:

Numpy array with the estimate for the nuisance parameters. Tuple with the lower and upper bounds for the nuisance parameters. Tuple(Tuple(lower,upper)).

χ^2 minimization

- **To be minimized over systematic parameters**
  - This is done numerically over the physically allowed nuisance parameters (very CPU consuming)

- **Then, the Jacobian over all nuisance *j* must be zero**

$$\nabla_j \chi^2 = 2 \sum_{Expmnt.} \sum_{i \in Bins} \left(1 - \frac{O_i}{E'_i}\right) \frac{\partial E'_i}{\partial x_j} + 2\,\frac{x_j - \mu_j}{\sigma_j^2} = 0$$

Note: this assumes Gaussian nuisance only for the time being

χ^2 calculation

pynu.fitter.BinnedLogLikelihoodRatio

▸ View Source

class BinnedLogLikelihoodRatio: ▸ View Source

Class containing all the information needed to perform an analysis and the methods for computing the log likelihood ratio ($-2\ln\left(\frac{L(Exp.)}{L(Obs.)}\right) \sim \chi^2$) given a set of binned observed data, binned expected events at a given physics point and nuisance parameters, and assuming Poisson statistics.

BinnedLogLikelihoodRatio(obervation, nominal_nuisance, sigma_nuisance, dist_nuisance) ▸ View Source

Initiates the class by storing the non-changing items of the χ^2 calculation.

Arguments:

- **obervation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array).
- **nominal_nuisance (list of float):** Produced from the xml analysis file, it contains the nominal values assumed for the nuisance parameters.
- **sigma_nuisance (list of float):** Produced from the xml analysis file, it contains the standard deviation values assumed for the nuisance parameters.
- **dist_nuisance (list of str):** Produced from the xml analysis file, it contains the type of distribution which is assumed for each nuisance.

def stats_only(self, expectation): ▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i \left(E_i - O_i + O_i \ln\left(\frac{O_i}{E_i}\right)\right)$, given the dictionary of binned expected number of events for each experiment of the analysis.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array): similarly to obervation, but for a given physics and nuisance values.

Returns:

Float with the value of χ^2.

def stats_and_systematics(self, expectation, nuisance): ▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i \left(E_i - O_i + O_i \ln\left(\frac{O_i}{E_i}\right)\right) + 2\sum_j \ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$, given the dictionary of binned expected number of events for each experiment of the analysis and taking into account the nuisance penalty terms.

Arguments:

- **expectation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Float with the value of χ^2 with nuisance.

def gradient(self, expectation, diff_expectation, nuisance): ▸ View Source

Returns the gradient of binned χ^2 computed analytically, given the dictionary of binned expected number of events for each experiment of the analysis and its derivative with respect to every nuisance parameter.

$$\nabla_j \chi^2 = 2\sum_i \left(1 - \frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j} + \frac{2}{P_j^{nuis}(x)}\frac{d\,P_j^{nuis}(x)}{dx_j}$$

LIMITATION: Currently, this is only done for nuisance following normal distributions. Other distributions like Beta will come soon.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Numpy array with each component of $\nabla\chi^2$.

def nuisance_penalty(self, nuisance): ▸ View Source

Returns the penalty term associated to nuisance parameters for the χ^2 computation.

Arguments:

- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by
- `pynu.analysis_reader.ParseXML` class.

Returns:

Float with $\sum_j \ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$.

def analytic_priors_bounds(self, expectation, diff_expectation): ▸ View Source

Returns the first-order values of the nuisance parameters which minimize the χ^2 at a given physics points. Here, first-order means we assume that the binned expected number of events is not modified by nuisance parameters, i.e. nuisance parameters are assumed to take the default value in this approximation.

$\nabla_j\chi^2 = 0$, and at first order, $E_i' \approx E_i + \frac{\partial E_i}{\partial x_j}(x_j - \mu_j)$, where E_i is the number of expected events with nuisance at their nomnial values.

$$\tilde{x}_j = \mu_j + \frac{\sum\left(1-\frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}}{\sum\frac{O_i}{E_i^2}\left(\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}\right)^2 - \frac{1}{\sigma_j^2}}$$

Further, bounds for the final values of the nuisance parameters as follows.

$x_j \in [\tilde{x}_j - \delta_j, \tilde{x}_j + \delta_j]$, where $\delta_j = \min(2\cdot|\tilde{x}_j - \mu_j|, \sigma)$

All this information is very useful for the minimizer to find faster the values of nuisance parameters minimizing the χ^2.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).

Returns:

Numpy array with the estimate for the nuisance parameters. Tuple with the lower and upper bounds for the nuisance parameters. Tuple(Tuple(lower,upper)).

χ^2 minimization

- **Further, in first approximation:**

$$\nabla_j \chi^2 = 0 \xrightarrow{E_i' \approx E_i + \frac{\partial E_i}{\partial x_j}(x_j - \mu_j)} \nabla_j \chi^2 \approx 2 \sum \left(1 - \frac{O_i}{E_i}\left(1 - \frac{1}{E_i}\frac{\partial E_i}{x_j}(x_j - \mu_j)\right)\right) \frac{\partial\ E_i}{\partial x_j}\bigg|_{x_j=\mu_j} + 2\,\frac{x_j - \mu_j}{\sigma_j^2} \approx 0$$

- **We can analytically estimate the value of each x_j**

$$\widetilde{x}_j = \mu_j + \frac{\sum \left(1 - \frac{O_i}{E_i}\right) \frac{\partial\ E_i}{\partial x_j}\Big|_{x_j=\mu_j}}{\sum \frac{O_i}{E_i{}^2}\left(\frac{\partial\ E_i}{\partial x_j}\Big|_{x_j=\mu_j}\right)^2 - \frac{1}{\sigma_j^2}}$$

χ^2 calculation

pynu.fitter.BinnedLogLikelihoodRatio

▸ View Source

```
class BinnedLogLikelihoodRatio:
```

▸ View Source

Class containing all the information needed to perform an analysis and the methods for computing the log likelihood ratio ($-2\ln\left(\frac{L(Exp.)}{L(Obs.)}\right) \sim \chi^2$) given a set of binned observed data, binned expected events at a given physics point and nuisance parameters, and assuming Poisson statistics.

```
BinnedLogLikelihoodRatio(obervation, nominal_nuisance, sigma_nuisance, dist_nuisance)
```

▸ View Source

Initiates the class by storing the non-changing items of the χ^2 calculation.

Arguments:

- **obervation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array).
- **nominal_nuisance (list of float):** Produced from the xml analysis file, it contains the nominal values assumed for the nuisance parameters.
- **sigma_nuisance (list of float):** Produced from the xml analysis file, it contains the standard deviation values assumed for the nuisance parameters.
- **dist_nuisance (list of str):** Produced from the xml analysis file, it contains the type of distribution which is assumed for each nuisance.

```
def stats_only(self, expectation):
```

▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i \left(E_i - O_i + O_i \ln\left(\frac{O_i}{E_i}\right)\right)$, given the dictionary of binned expected number of events for each experiment of the analysis.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array): similarly to obervation, but for a given physics and nuisance values.

Returns:

Float with the value of χ^2.

```
def stats_and_systematics(self, expectation, nuisance):
```

▸ View Source

Returns the value of binned $\chi^2 = 2\sum_i \left(E_i - O_i + O_i \ln\left(\frac{O_i}{E_i}\right)\right) + 2\sum_j \ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$, given the dictionary of binned expected number of events for each experiment of the analysis and taking into account the nuisance penalty terms.

Arguments:

- **expectation (dict):** Produced by PyNuFit and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Float with the value of χ^2 with nuisance.

```
def gradient(self, expectation, diff_expectation, nuisance):
```

▸ View Source

Returns the gradient of binned χ^2 computed analytically, given the dictionary of binned expected number of events for each experiment of the analysis and its derivative with respect to every nuisance parameter.

$$\nabla_j \chi^2 = 2\sum_i \left(1 - \frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j} + \frac{2}{P_j^{nuis}(x)}\frac{d\,P_j^{nuis}(x)}{dx_j}$$

LIMITATION: Currently, this is only done for nuisance following normal distributions. Other distributions like Beta will come soon.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).
- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by ParseXML class.

Returns:

Numpy array with each component of $\nabla\chi^2$.

```
def nuisance_penalty(self, nuisance):
```

▸ View Source

Returns the penalty term associated to nuisance parameters for the χ^2 computation.

Arguments:

- **nuisance (list of float):** Values for the nuisance parameters ordered as provided by
- `pynu.analysis_reader.ParseXML` class.

Returns:

Float with $\sum_j \ln\left(\frac{P_j^{nuis}(x)}{P_j^{nuis}(x=\mu)}\right)$.

```
def analytic_priors_bounds(self, expectation, diff_expectation):
```

▸ View Source

Returns the first-order values of the nuisance parameters which minimize the χ^2 at a given physics points. Here, first-order means we assume that the binned expected number of events is not modified by nuisance parameters, i.e. nuisance parameters are assumed to take the default value in this approximation.

$\nabla_j\chi^2 = 0$, and at first order, $E_i' \approx E_i + \frac{\partial E_i}{\partial x_j}(x_j - \mu_j)$, where E_i is the number of expected events with nuisance at their nomnial values.

$$\widetilde{x_j} = \mu_j + \frac{\sum\left(1-\frac{O_i}{E_i}\right)\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}}{\sum\frac{O_i}{E_i^2}\left(\frac{\partial E_i}{\partial x_j}\Big|_{x_j=\mu_j}\right)^2 - \frac{1}{\sigma_j^2}}$$

Further, bounds for the final values of the nuisance parameters as follows.

$x_j \in [\widetilde{x_j} - \delta_j, \widetilde{x_j} + \delta_j]$, where $\delta_j = \min(2\cdot|\widetilde{x_j} - \mu_j|, \sigma)$

All this information is very useful for the minimizer to find faster the values of nuisance parameters minimizing the χ^2.

Arguments:

- **expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (Experiment(str): binned events (numpy.array) similarly to obervation, but for a given physics and nuisance values.
- **diff_expectation (dict):** Produced by `pynu.PyNuFit` and follows the structue (nuisance parameter (str): (Experiment(str): binned events (numpy.array)).

Returns:

Numpy array with the estimate for the nuisance parameters. Tuple with the lower and upper bounds for the nuisance parameters. Tuple(Tuple(lower,upper)).

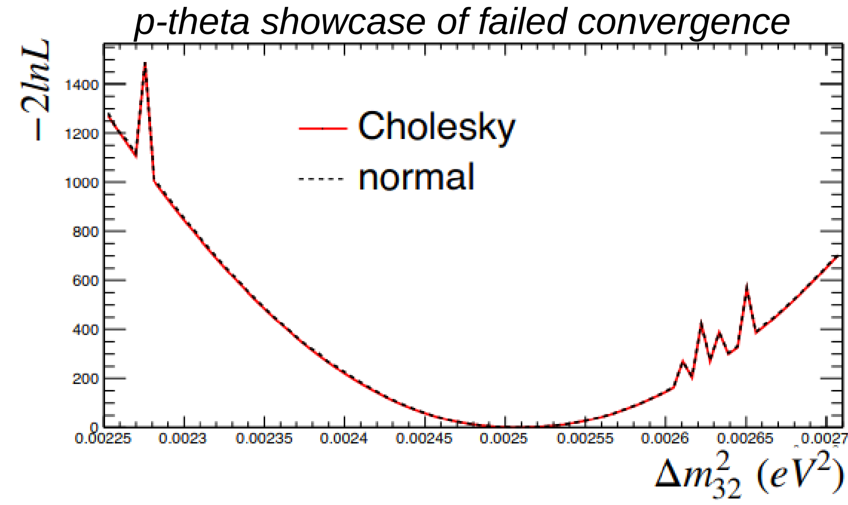
χ^2 calculation

The performance:
- Without any analytic implementations, each point of the (11853 in the) analysis takes on average **5min 31s (timescale of official fitters)**
- With analytic derivatives and estimation, each point of the (11853 in the) analysis takes on average **15s**

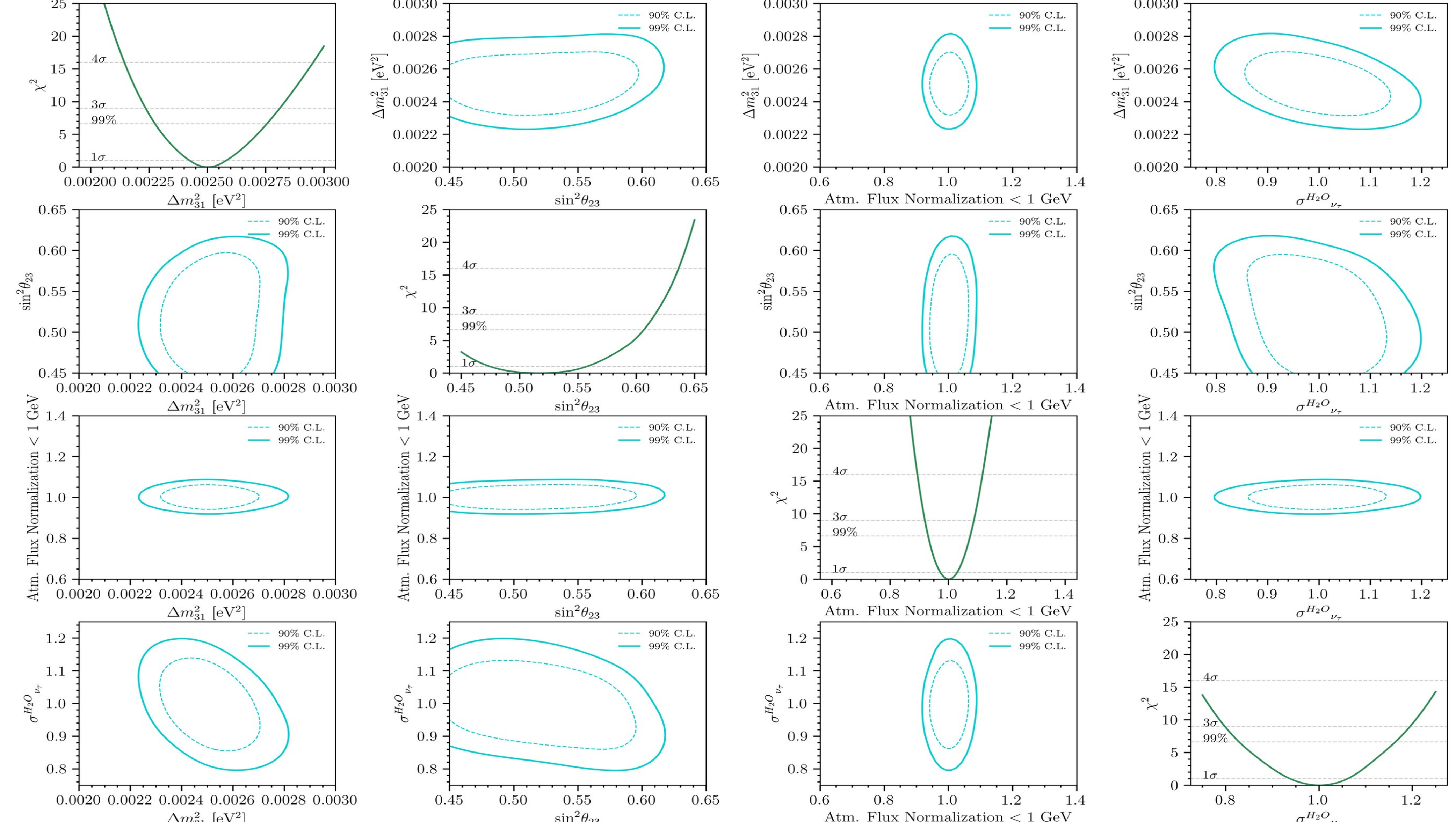
a factor 22 improvement in time, the results are almost identical

→ *with analytic additions the χ2 is lower, which means that the method and minimization is more robust*

p-theta showcase of failed convergence



(Data set is 2M events)



90% C.L.
99% C.L.
χ^2
Δm^2_{31} [eV^2]
$\sin^2\theta_{23}$
Atm. Flux Normalization < 1 GeV
$\sigma^{H_2O}{}_{\nu_\tau}$
1σ
99%
3σ
4σ

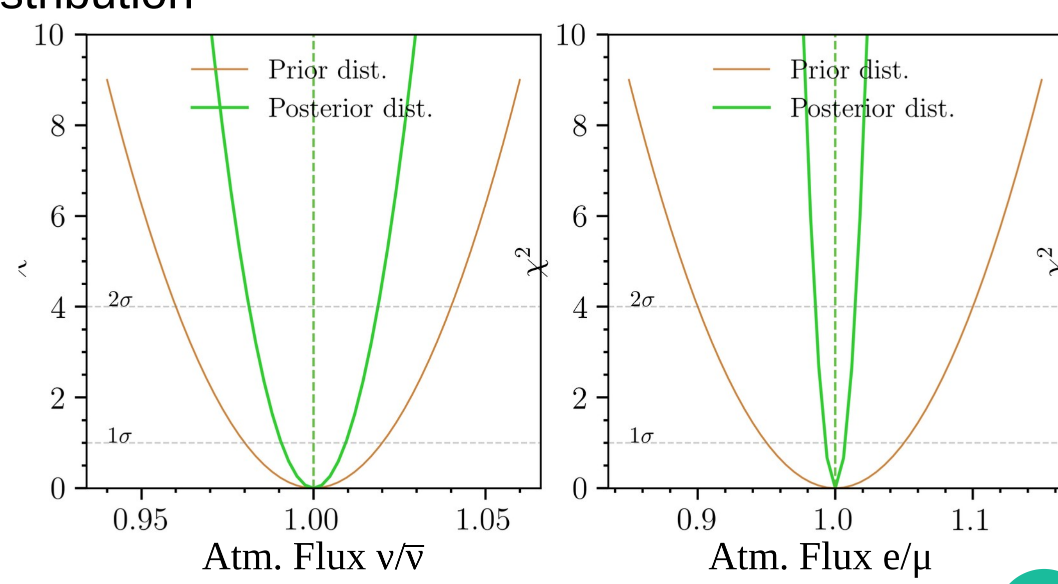
Bonus goodness

Computing the posterior probabilities of the nuisance parameters has been only implemented for T2K fitters

With the derivative information of the nuisance parameters we can estimate the Fisher information and, therefore, the posterior distribution

$$\frac{\partial \chi^2}{\partial x_j}(\vec{x}_{\mathrm{bf}}) \approx \frac{2}{\sigma_j^{post}}(x_j - x_j^{\mathrm{bf}})$$

$$\chi^2(x_j^{post}) \approx -2\log\left(\frac{P(x_j, \sigma_j^{post})}{P_0}\right)$$

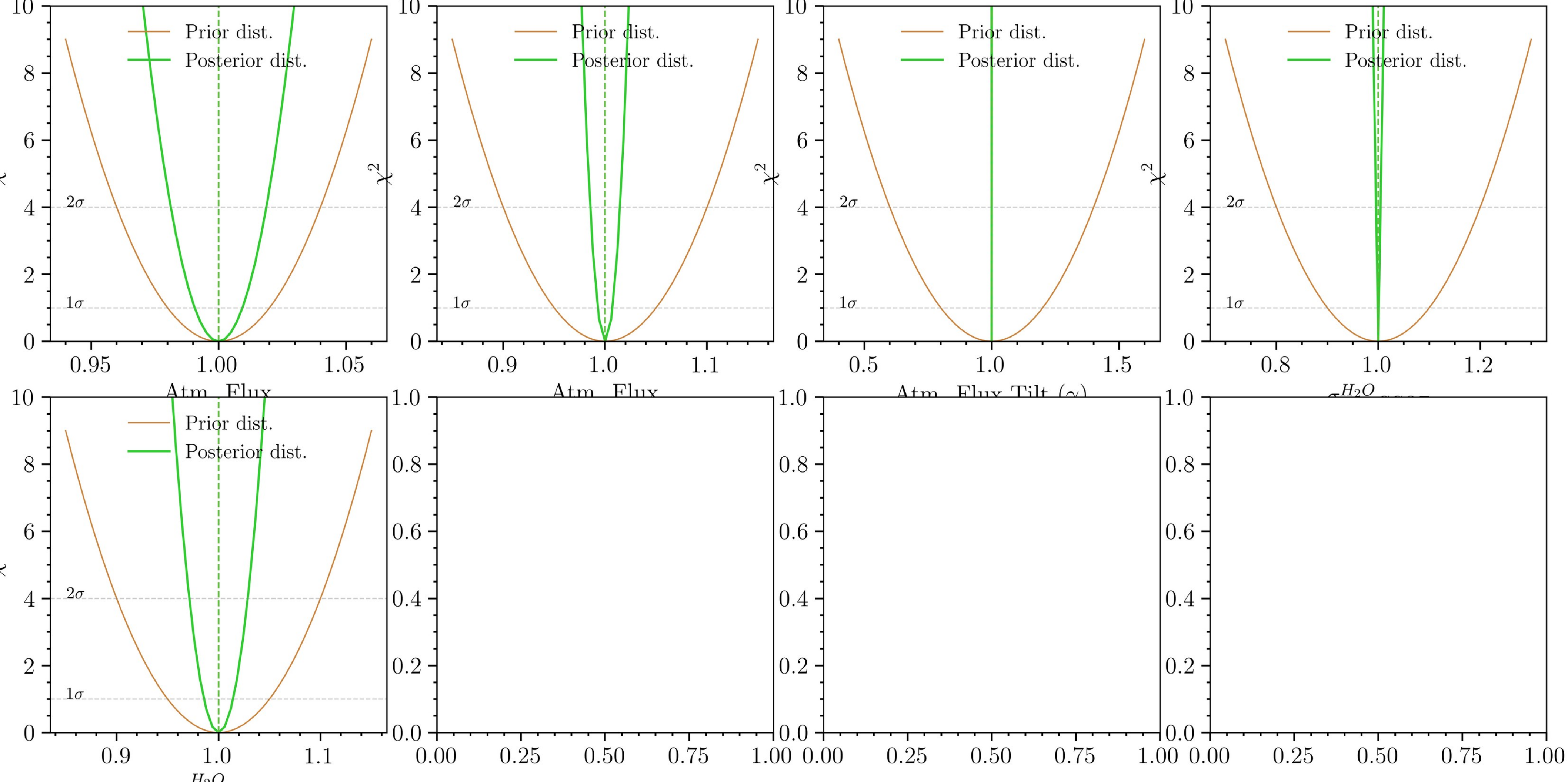


Next steps

- Make official oscillation analysis for HyperK or SuperK and compare it with Osc3++ (almost there)
- Make sterile neutrino and NSI sensitivty studies for HyperK (trivial once the previous)
- Keep developing (do you?)
- All SuperK phases are implemented, try to develop for data analysis in SK? Already proposed

Back up!

Preliminary attempt of posterior nuisance with derivatives information



χ^2 considerations

- **This is assuming nuisance are Gaussian distributed, but this is not always the case!**
  → Beta distribution case is implemented everywhere except for the derivative (ongoing)

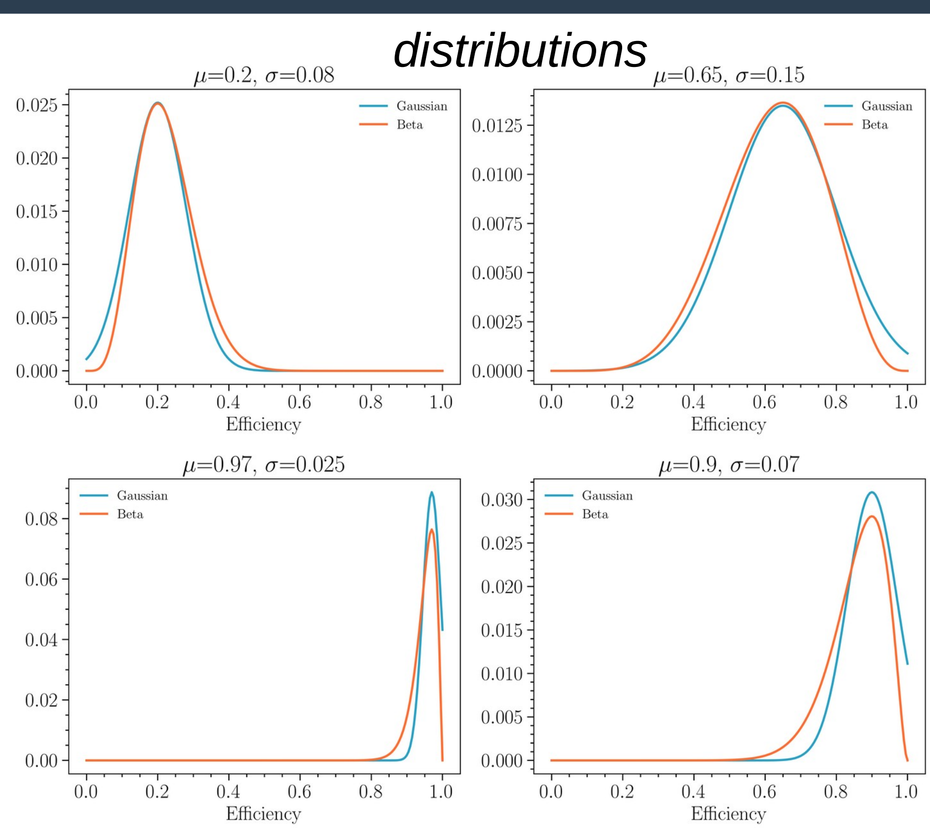
χ^2 considerations

- The case of nuisance distributed as a Beta function between 0 and 1(ideal for efficiencies)

$$B(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$$

$$\mathrm{B}(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

distributions

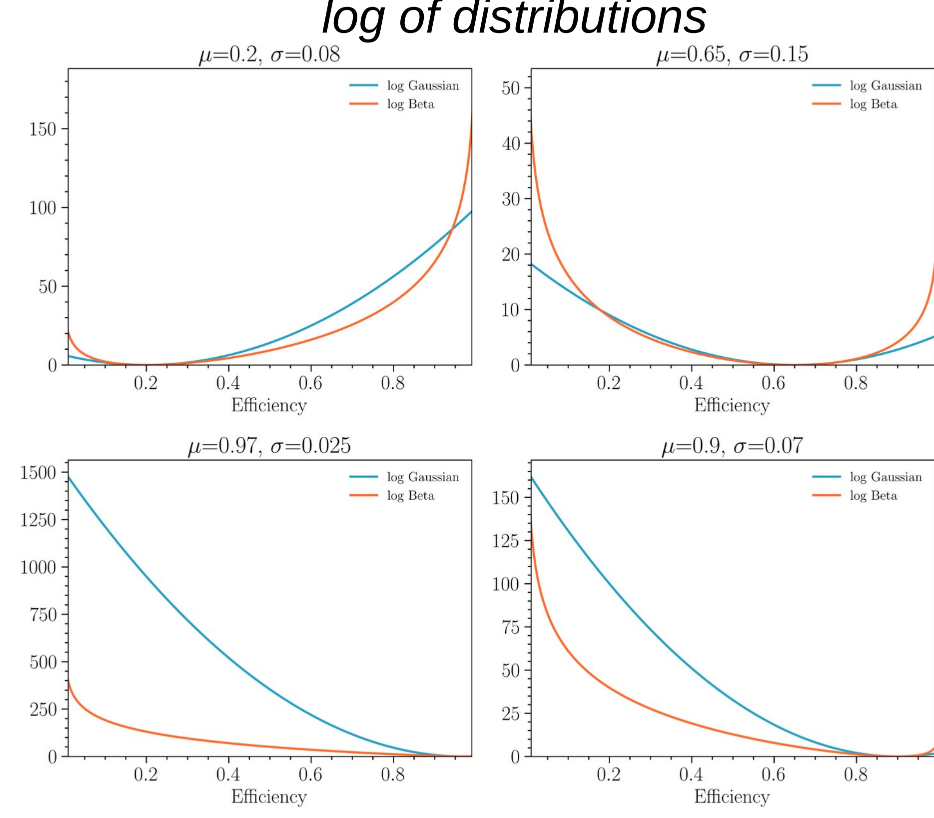


χ^2 considerations

- The case of nuisance distributed as a Beta function between 0 and 1(ideal for efficiencies)

$$B(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$$

$$\mathrm{B}(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$



A working example

Definition of experiment

```xml
<!-- IceCube Upgrade -->
    <NeutrinoExperiment name="IceCube-Upgrade">
        <status> 1 </status>
        <target name="Water"> </target>
        <source name='Atmospheric'>
            <status> 1 </status>
            <MCFiles name='/home/pablofer/Pheno/Atmospherics/MC/IceCubeUpgrade/data/
            neutrino_mc.csv'>
                <status> 1 </status>
            </MCFiles>
            <exposure> 5 </exposure> <!-- years-->
            <MCexposure> 5 </MCexposure> <!-- This is total exposure of all the specified MC
            files-->
            <DataFiles name='../Simulation/HyperK/data/output/SK/combined.hdf5'>
                <status> 0 </status>
            </DataFiles>

        </source>
```

A working example

Definition of the experiment class

```python
class ICUp_Atm(Experiment):
    def __init__(self, dict_of_details, scenario):
        super(ICUp_Atm, self).__init__(dict_of_details)

        self.Detector = 'IceCube-Upgrade'
        self.Target = 'Water'
        self.Source = 'Atmospheric'
        self.Scenario = scenario

        self.Definition()

        self.MCVariables()

        self.Binning()
        self.SetBinner_2D()

        if self.DataFit:
            self.DataVariables()
            self.BinData()
```

Each experiment has a dedicated class which:

- Inherits the structure of a general *Experiment* class
  -
```python
    def Definition(self):
        self.Definition = {
            self.Detector: 'Detector',
            self.Target: 'XSection',
            self.Source: 'Flux',
            self.Scenario: 'Osc'}
```
  - General *Binning* functions

A working example

Definition of the experiment class

```python
class ICUp_Atm(Experiment):
    def __init__(self, dict_of_details, scenario):
        super(ICUp_Atm, self).__init__(dict_of_details)

        self.Detector = 'IceCube-Upgrade'
        self.Target = 'Water'
        self.Source = 'Atmospheric'
        self.Scenario = scenario

        self.Definition()

        self.MCVariables()

        self.Binning()
        self.SetBinner_2D()

        if self.DataFit:
            self.DataVariables()
            self.BinData()
```

Each experiment has a dedicated class which:

- Inherits the structure of a general *Experiment* class
  - Functions which compute the weights based on input PhysicsTunes

```python
def StartPhysicsWeights(self):  # Starts expected weights with fixed values
    self.PhysicsWeight = 1

def UpdatePhysicsWeights(self, w):
    self.PhysicsWeight = w * self.PhysicsWeight

# Contains all non-changing weights of the analysis, i.e. fixed
def UpdateBaseWeights(self, w):
    self.BaseWeight = w * self.BaseWeight

# Contains all weights of the analysis except for those relative to
# nuisance parameters
def StartNuisanceWeights(self):  # Starts expected weights with fixed values
    self.NuisanceWeight = 1

def UpdateNuisanceWeights(self, w):
    self.NuisanceWeight = w * self.NuisanceWeight

# Contains all non-changing weights of the analysis, i.e. fixed
def UpdateNominalWeights(self, w):
    self.NominalWeight = w * self.NominalWeight

def SetExpectedWeight(self):
    self.ExpectedWeight = self.PhysicsWeight * self.NuisanceWeight

def SetExpectedBinned(self):
    self.ExpectedBinned = self.BinMC(self.ExpectedWeight)
    self.RemoveFewEntries('Expected')

def SetObservedBinned(self):
    if self.DataFit:
        self.ObservedBinned = self.BinData()
    else:
        self.ObservedBinned = self.BinMC(self.NominalWeight)
    self.FewEntries = self.ObservedBinned > 4
    self.RemoveFewEntries('Observed')
```

A working example

Definition of the experiment class

```
class ICUp_Atm(Experiment):
    def __init__(self, dict_of_details, scenario):
        super(ICUp_Atm, self).__init__(dict_of_details)

        self.Detector = 'IceCube-Upgrade'
        self.Target = 'Water'
        self.Source = 'Atmospheric'
        self.Scenario = scenario

        self.Definition()

        self.MCVariables()

        self.Binning()
        self.SetBinner_2D()

        if self.DataFit:
            self.DataVariables()
            self.BinData()
```

Each experiment has a dedicated class which:

- Implements specific items of the experiment
  - MCVariables

```
def MCVariables(self):
    d_itype = self.MC['pid']
    d_Etrue = self.MC['true_energy']
    self.EReco = self.MC['reco_energy'][condition]
    self.CosZReco = np.cos(self.MC['reco_zenith'])
    self.CosZTrue = np.cos(self.MC['true_zenith'])
    self.AziTrue = self.MC['true_azimuth']
    self.CC = self.MC['current_type']
    self.nuPDG = np.int_(self.MC['pdg'])
    self.ETrue = self.MC['true_energy']
    self.Weight = self.MC['weight']
    self.Sample = self.MC['pid']  # Sample of each event
    self.Mode = self.NEUTMode()

    self.NumberOfEvents = self.Sample.size
    self.Samples = np.unique(self.Sample)  # Samples in the analysis
    # Number of samples in the analysis
    self.NumberOfSamples = 1 + np.amax(self.Samples)
    self.Erec_min = 1
    self.Erec_max = 1e3
    self.Etrue_min = 1
    self.Etrue_max = 1e3
    self.E_edges = [self.Erec_min, self.Erec_max]
    self.Z_edges = [-1, 1]

    self.Norm *= 365 * 24 * 60 * 60 * 1e4
```

  - Which binning

```
def BinMC(self, array, shift_E=1, bias_E=0):
    self.CosThetaReco = self.CosZReco
    return self.BinIt_MC_2D(array, shift_E=1, bias_E=0)

def BinData(self):
    self.dCosThetaReco = self.dCosZReco
    return self.BinIt_Data_2D()

def Binning(self):
    NErec = 40
    erec = np.logspace(
        np.log10(
            self.Erec_min), np.log10(
            self.Erec_max), NErec + 1, endpoint=True)
    z10bins = np.array([-1, -0.8, -0.6, -0.4, -0.2,
                        0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
    self.EnergyBins = {0: erec, 1: erec}
    self.CTBins = {0: z10bins, 1: z10bins}
```

A working example

Definition of atm. neutrino flux PhysicsTunes

```xml
<!-- Atmospheric neutrinos -->
    <NeutrinoSource name="Atmospheric">
        <status> 1 </status>
        <nuisance name='FluxNormalization_Above1GeV'>
            <status> 1 </status>
            <sigma> 0.1 </sigma>
            <nominal> 1.0 </nominal>
            <distribution> normal </distribution>
        </nuisance>
        <nuisance name='NuNuBarRatio'>
            <status> 1 </status>
            <sigma> 0.05 </sigma>
            <nominal> 1.0 </nominal>
            <distribution> normal </distribution>
        </nuisance>
        <nuisance name='FlavorRatio'>
            <status> 1 </status>
            <sigma> 0.02 </sigma>
            <nominal> 1.0 </nominal>
            <distribution> normal </distribution>
        </nuisance>
        <physics name='FluxTilt'>
            <points> 9 </points>
            <min> 0.9 </min>
            <max> 1.1 </max>
            <true> 1.0 </true>
        </physics>
    </NeutrinoSource>
```

PhysicsTunes → AtmosphericFlux

```python
class AtmosphericFlux(Tune):

    def FluxNormalization(self, experiment, x):
        return x

    def Diff_FluxNormalization(self, experiment, x):
        return 1

    def FluxNormalization_Below1GeV(self, experiment, x):
        nev = np.ones(experiment.NumberOfEvents)
        nev[experiment.ETrue < 1] = x
        return nev

    def Diff_FluxNormalization_Below1GeV(self, experiment, x):
        nev = np.zeros(experiment.NumberOfEvents)
        nev[experiment.ETrue < 1] = 1
        return nev

    def FluxNormalization_Above1GeV(self, experiment, x):
        nev = np.ones(experiment.NumberOfEvents)
        nev[experiment.ETrue > 1] = x
        return nev

    def Diff_FluxNormalization_Above1GeV(self, experiment, x):
        nev = np.zeros(experiment.NumberOfEvents)
        nev[experiment.ETrue > 1] = 1
        return nev

    def FluxTilt(self, experiment, x):
        E0Gam = 10  # GeV
        nev = (experiment.ETrue / E0Gam)**x
        return nev

    def Diff_FluxTilt(self, experiment, x):
        E0Gam = 10  # GeV
        nev = (experiment.ETrue / E0Gam)**x * np.log(experiment.ETrue / E0Gam)
        return nev
```

A working example

Definition of water cross section parameters
PhysicsTunes

```
<!-- Water-Cherenkov -->
    <NeutrinoTarget name="Water">
        <status> 1 </status>
        <physics name='XSecNuTau'>
            <points> 9 </points>
            <min> 0.75 </min>
            <max> 1.25 </max>
            <true> 1.0 </true>
        </physics>
        <nuisance name='DIS'>
            <status> 1 </status>
            <sigma> 0.05 </sigma>
            <nominal> 1.0 </nominal>
            <distribution> normal </distribution>
        </nuisance>
        <fixed name='NCHad'>
            <status> 1 </status>
            <value> 1.0 </value>
        </fixed>
    </NeutrinoTarget>
```

PhysicsTunes → WaterXSection

```
class WaterXSection(Tune):

    def XSecNuTau(self, experiment, x):
        tau = np.ones(experiment.NumberOfEvents)
        tau[np.abs(experiment.nuPDG) == 16] = x
        return tau

    def Diff_XSecNuTau(self, experiment, x):
        tau = np.zeros(experiment.NumberOfEvents)
        tau[np.abs(experiment.nuPDG) == 16] = 1
        return tau

    def NCoverCC(self, experiment, x):
        nc = np.ones(experiment.NumberOfEvents)
        nc[experiment.CC == 0] = x
        return nc

    def Diff_NCoverCC(self, experiment, x):
        nc = np.zeros(experiment.NumberOfEvents)
        nc[experiment.CC == 0] = 1
        return nc

    def AxialMass(self, experiment, x):
        cc = np.ones(experiment.NumberOfEvents)
        cc[experiment.CC == 1] = 1 + 0.042 * \
            (x - 1) * 1.05 * np.log10(experiment.ETrue[experiment.CC == 1])
        return cc

    def Diff_AxialMass(self, experiment, x):
        cc = np.zeros(experiment.NumberOfEvents)
        cc[experiment.CC == 1] = 0.042 * 1.05 * \
            np.log10(experiment.ETrue[experiment.CC == 1])
        return cc

    def NCHad(self, experiment, x):
        nc = np.ones(experiment.NumberOfEvents)
        nc[experiment.CC == 0] = x
        return nc

    def Diff_NCHad(self, experiment, x):
        nc = np.zeros(experiment.NumberOfEvents)
        nc[experiment.CC == 0] = 1
        return nc
```

A working example

Definition of oscillations PhysicsTunes

```xml
<NeutrinoOscillations name="3-Osc">
    <status> 1 </status>
    <flavors> 3 </flavors>
    <fixed name='Sin2Theta12'>
        <status> 1 </status>
        <value> 0.304 </value>
    </fixed>
    <fixed name='Sin2Theta13'>
        <status> 1 </status>
        <value> 0.022 </value>
    </fixed>
    <physics name='Sin2Theta23'>
        <points> 13 </points>
        <min> 0.45 </min>
        <max> 0.65 </max>
        <true> 0.517 </true>
    </physics>
    <fixed name='Dm221'>
        <status> 1 </status>
        <value> 7.42e-5 </value>
    </fixed>
```

```xml
    <physics name='dCP'>
        <points> 1 </points>
        <min> 4.14 </min>
        <max> 4.14 </max>
        <true> 4.14 </true>
    </physics>
    <physics name='Dm231'>
        <points> 9 </points>
        <min> 2.0e-3 </min>
        <max> 3.0e-3 </max>
        <true> 2.5e-3 </true>
    </physics>
    <physics name='Ordering'>
        <points> 1 </points>
        <min> normal </min>
        <max> normal </max>
        <true> normal </true>
    </physics>
</NeutrinoOscillations>
```

PhysicsTunes → Oscillations

```python
# General oscillator

class Oscillator(Tune):
    def __init__(self, scenario, neutrino_flavors, source=None):
        super().__init__()

        ''' Support for SM and NSI scenarios '''
        self.Scenario = scenario
        self.Source = source
        self.NSI = False
        if 'NSI' in self.Scenario or 'nsi' in self.Scenario:
            self.NSI = True

        ''' Support for 3 active neutrinos and any number of sterile neutrinos '''
        self.NeutrinoFlavors = neutrino_flavors
```

```python
    def Sin2Theta13(self, experiment, x):
        self.Osc.Set_MixingAngle(0, 2, asin(sqrt(x)))
        self.Parameters['Sin2Theta13'] = x
        return self.GetOscillations()

    def Diff_Sin2Theta13(self, experiment, x):  # Numerical derivation
        h0 = x * (1 + self.eps)
        h1 = x * (1 - self.eps)
        w0 = self.Sin2Theta13(experiment, h0)
        w1 = self.Sin2Theta13(experiment, h1)
        dw = ((w0 - w1) / (h0 - h1))
        self.Parameters['Sin2Theta13'] = x
        # print(dw)
        return dw
```

The xml file

- **Provides all information about the analysis**
  - Sources and experiments and their simulation files
  - Physics scenario
  - Systematic uncertainties
  - Physics parameters
- **All parametrizations are contained in *Physics Tunes*, the xml file labels them as physics and systematics parameters for the analysis (core of flexibility)**
  - For example, no difference between the implementation of detector systematics and oscillation calculation

Custom analysis from xml file:

- **Specify any number of detectors**
- **Associate any number of neutrino sources to those detectors (e.g. HK-FD atm. and accel.)**
- **And all the related *Physics Tunes*, physics or systematics**
  - This eases (basically trivial for most cases) the implementation of joint analyses and their correlation (e.g. atm. and accel. neutrinos in HK-FD and accel. neutrinos in ND-280 and IWCD)

Neutrino Oscillations

- **Using nuSQuIDS (https://arxiv.org/pdf/2112.13804.pdf) package to handle almost any kind of neutrino oscillations and scenarios:**
  - 3-flavor oscillation scenario
  - Sterile ν searches
  - NSI
  - Lorentz violation
- **Calculation is done event-by-event or on a grid**

Systematics handling

- **Systematics are specified in the xml file and classified depending to their source:**
  - Flux
  - Cross-sections
  - Detector
  - Oscillations
- **Flexible implementation allows:**
  - Event-by-event weighting (compute effect of syst. on runtime)
  - Binned 1σ fractional error matrix/vector
    - with interpolation to avoid rigidity between other and this analysis binning

Systematics handling

```
-----------------------------------
Neutrino sources considered:
 +  Atmospheric
===================================
-----------------------------------
Experiments considered:
 +  HyperK , at  ../Simulation/HyperK/data/output/HK/combined.hdf5
===================================
-----------------------------------
Neutrino detectors considered:
 +  Water
===================================
-----------------------------------
Neutrino physics considered:
 +  Three Flavour
===================================
List of Systematics
 + From Atmospheric: ['FluxNormalization_Below1GeV', 'FluxNormalization_Above1GeV', 'FluxTilt',
'NuNuBarRatio', 'FlavorRatio', 'ZenithFluxUp', 'ZenithFluxDown']
 + From HyperK_Htag: ['SKIV_SKEnergyScale', 'SKIV_FCPCSeparation', 'SKIV_FiducialVolume',
'SKIV_FCReduction', 'SKIV_PCReduction', 'SKIV_SubGeV2ringPi0', 'SKIV_SubGeV1ringPi0',
'SKIV_MultiRing_NuNuBarSeparation', 'SKIV_MultiRing_EOtherSeparation',
'SKIV_PC_StopThruSeparation', 'SKIV_Pi0_RingSeparation', 'SKIV_E_RingSeparation',
'SKIV_Mu_RingSeparation', 'SKIV_SingleRing_PID', 'SKIV_MultiRing_PID', 'SKIV_DecayETagging']
 + From Water: ['XSecNuTau', 'NCoverCC', 'NCHad', 'DIS', 'CCQE', 'CCQENuBarNu', 'CCQEMuE',
'CC1Pi_Pi0Pi', 'CC1Pi_NuBarNuE', 'CC1Pi_NuBarNuMu', 'CC1PiProduction', 'CohPiProduction',
'AxialMass']
 + From Three Flavour: []
===================================
Processing simulation of HyperK experiment with a exposure of 5.0 years.
Your simulation file has 50 years
============================================================
=================== Starting analysis =====================
============================================================
0.304 0.02 0.57 7.42e-05 0.00245 3.9 normal process started
```

- **Systematics are split by labels into:**
  - Common among experiments
  - Experiment independent

  , with no restrictions in combination

- **Joint analyses are assumed by default**
  - Correlations between common and independent systematics across the given experiments

Performance

χ^2 calculation

- **Usual binned χ2 calculation**

$$\chi^2 = 2\ln\left(\frac{\mathcal{L}(Exp(\vec{\theta})|Obs)}{\mathcal{L}(Obs|Obs)}\right) \xrightarrow{\text{Poisson stats.}} \chi^2 = \sum_{Expmnt.}\sum_{i\in Bins}\left(E_i' - O_i + O_i\cdot\log\left(\frac{O_i}{E_i'}\right)\right) + \sum_{j\in Syst.}\left(\frac{\mu_j - x_j}{\sigma_j}\right)^2$$

χ^2 calculation

- **Usual binned χ2 calculation**

$$\chi^2 = 2\ln\left(\frac{\mathcal{L}(Exp(\vec{\theta})|Obs)}{\mathcal{L}(Obs|Obs)}\right) \xrightarrow{\text{Poisson stats.}} \chi^2 = \sum_{Expmnt.}\sum_{i\in Bins}\left(E_i' - O_i + O_i\cdot\log\left(\frac{O_i}{E_i'}\right)\right) + \sum_{j\in Syst.}\left(\frac{\mu_j - x_j}{\sigma_j}\right)^2$$

  - If f_{ij} are given as inputs, the re-weighting is linear

$$E_i' = E_i\left(1 + \sum_{j\in Syst.} f_i(x_j)\right), \quad f_i(x_j) = x_j\cdot f_{ij}$$

χ^2 calculation

- **Usual binned χ2 calculation**

$$\chi^2 = 2\ln\left(\frac{P_{model}}{P_{data/truth}}\right)$$

Poisson stats.

$$\chi^2 = \sum_{Expmnt.}\sum_{i\in Bins}\left(E'_i - O_i + O_i\cdot\log\left(\frac{O_i}{E'_i}\right)\right) + \sum_{j\in Syst.}\left(\frac{\mu_j - x_j}{\sigma_j}\right)^2$$

  - If f_{ij} are given as inputs, the re-weighting is linear

$$E'_i = E_i\left(1 + \sum_{j\in Syst} f_i(x_j)\right),\quad f_i(x_j) = x_j\cdot f_{ij}$$

  - Or more compl… weighting is done event-by-event

$$f_{ij} = f_i(x_j) = 1 - \frac{\sum_{k\,\in\, i^{th}\ \mathrm{bin}} W_k\cdot w_k^j(x_j)}{E_i}$$

$$E'_i = E_i\left(1 + \sum_{j\in Syst}\right.$$

$$E_i = \sum_{k\,\in\, i^{th}\ \mathrm{bin}} W_k$$

Events in bin i^{th}

Nominal weights from MC simulation and physics parameters

$$f_{ij} = f_i(x_j) = 1 - \frac{\sum_{k\,\in\, i^{th}\ \mathrm{bin}} W_k\cdot w_k^j(x_j)}{E_i}$$

$$E_i = \sum_{k\,\in\, i^{th}\ \mathrm{bin}} W_k$$

χ^2 minimization

- **To be minimized over systematic parameters**
  - Usually this is done numerically over the physically allowed systematic parameters (very CPU consuming)

χ^2 minimization

- **To be minimized over systematic parameters**
- **Then, the Jacobian over all systematics *j* must be zero**

$$\nabla_j \chi^2 = 2 \sum_{Expmnt.} \sum_{i \in Bins} \left(1 - \frac{O_i}{E'_i}\right) \frac{\partial E'_i}{\partial x_j} + 2\,\frac{\mu_j - x_j}{\sigma_j^2}$$

$\nabla \chi^2$ implementation

χ^2 minimization

- **To be minimized over systematic parameters**
- **Then, the Jacobian over the systematics must be zero**

$$\nabla_j \chi^2 = 2 \sum_{Expmnt.} \sum_{i \in Bins} \left(1 - \frac{O_i}{E'_i}\right) \frac{\partial E'_i}{\partial x_j} + 2\,\frac{\mu_j - x_j}{\sigma_j^2}$$

← $\nabla \chi^2$ *implementation*

where,

Binned syst.

$$\frac{\partial E'_i}{\partial x_j} = E_i \cdot f_{ij}$$

or

Event-by-event syst.

$$\frac{\partial E'_i}{\partial x_j} = E_i \cdot \frac{d\ f_i(x_j)}{dx_j}, \quad \frac{d\ f_i(x_j)}{dx_j} = \frac{\sum_{k\ \in\ i^{th}\ \mathrm{bin}} W_k \cdot \frac{d\ w_k^j(x_j)}{dx_j}}{\sum_{k\ \in\ i^{th}\ \mathrm{bin}} W_k}$$

χ^2 minimization

- **Further, in first approximation:**

$$\nabla_j \chi^2 = 0 \xrightarrow{E_i'^0 = E_i} \nabla_j^0 \chi^2 = 2 \sum_{Expmnt.} \sum_{i \in Bins} (E_i - O_i) \frac{d\ f_i(x_j)}{dx_j} + 2\,\frac{\mu_j - x_j}{\sigma_j^2} \approx 0$$

This removes the dependence with the rest of systematics

χ^2 minimization

- **Further, in first approximation:**

$$\nabla_j \chi^2 = 0 \xrightarrow{E_i'^0 = E_i} \nabla_j^0 \chi^2 = 2 \sum_{Expmnt.} \sum_{i \in Bins} (E_i - O_i) \frac{d\ f_i(x_j)}{dx_j} + 2 \frac{\mu_j - x_j}{\sigma_j^2} \approx 0$$

This removes the dependence with the rest of systematics

from which we can estimate the x_j parameter at minimum χ2

$$\tilde{x}_j = \mu_j + \sigma_j \sum_{Expmnt.} \sum_{i \in Bins} (E_i - O_i) \frac{d\ f_i(x_j)}{dx_j}$$

"Analytical estimate of fit priors and their bounds"

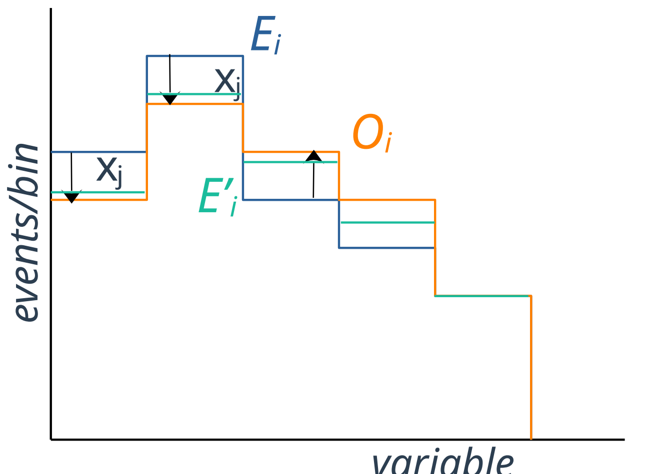
χ^2 minimization

Even more, we can estimate the allowed range for each x_j, since

$$E'_i \in [\min(E_i, O_i), \max(E_i, O_i)]$$

This condition is relaxed in case of data fit to account for larger unpredictable fluctuations

Minimization of $\chi 2$ is such that x_j makes E'_i as close as possible to O_i from E_i



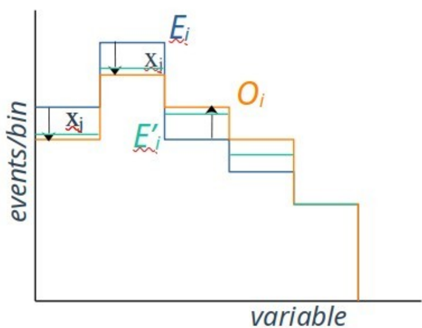
χ^2 minimization

Even more, we can estimate the allowed range for each x_j, since

$$E'_i \in [\min(E_i, O_i), \max(E_i, O_i)]$$

This condition is relaxed in case of data fit to account for larger unpredictable fluctuations

Minimization of $\chi 2$ is such that x_j makes E'_i as close as possible to O_i from E_i



then

$$x_j \in [\min(\mu_j, \tilde{x}_j), \max(\mu_j, \tilde{x}_j)]$$

"Analytical estimate of fit priors and their bounds"

OK... why would you care?

- **Implementation of the previous reduces dramatically the minimization time and the required computing time with no differences in the value of χ2**

physically allowed values

	Numerical only	w/ analytic priors and bounds	w/ $\nabla\chi^2$	w/ ana. priors and bounds + $\nabla\chi^2$
CPU time (s)	7282	3240 ($^1/_{2.25}$)	193 ($^1/_{37.7}$)	**91 ($^1/_{80}$)**

for illustration, used toy MC of 1M events of atm. ν at a HK-like experiment with 37 systematic parameters

Other Features

- **Sampling over physics parameter space:**
  - Grid of points
    - Sequential
    - Single point or range of points (for cluster)
    - Parallelization on local machine
  - Markov Chain Monte Carlo sampling
    - Parallelization on local machine
    - Parallelization on cluster (ongoing)

Other Features

- **Most of the work done at runtime, just before the fit:**
  - Binned or event-by-event re-weighting (including oscillations)
  - Binning done after binning

→ Reduces steps of the analysis, which helps the binning and parametrization optimization studies

HyperK Atmospheric ν analysis example

- ***nuflux*** **package for atm. ν flux (originally for IceCube, included low energy atm. ν)**
- **…**