
pyhf Documentation

Release 0.5.3

Lukas Heinrich, Matthew Feickert, Giordon Stark

Oct 20, 2020

CONTENTS

1	HistFactory	3
2	Declarative Formats	7
3	Additional Material	9
3.1	Footnotes	9
3.2	Bibliography	9
4	Likelihood Specification	11
4.1	Workspace	11
4.2	Channel	11
4.3	Sample	12
4.4	Modifiers	13
4.5	Data	15
4.6	Measurements	15
4.7	Observations	16
4.8	Toy Example	16
4.9	Additional Material	17
5	Fundamentals	19
5.1	Piecewise Linear Interpolation	19
5.2	Tensorizing Interpolators	24
5.3	Empirical Test Statistics	41
6	Examples	47
6.1	ShapeFactor	47
6.2	XML Import/Export	50
6.3	Visualization with Altair	56
6.4	Hello World, pyhf style	57
6.5	Multi-bin Poisson	58
6.6	Multibin Coupled HistoSys	63
6.7	Binned HEP Statistical Analysis in Python	66
7	Outreach	73
7.1	Abstract	73
7.2	Presentations	73
7.3	Tutorials	74
7.4	Posters	75
7.5	In the Media	75
8	Installation	77

8.1	Install latest stable release from PyPI...	77
8.2	Install latest development version from GitHub...	78
8.3	Updating pyhf	79
9	Developing	81
9.1	Testing	81
9.2	Publishing	82
9.3	Context Files and Archive Metadata	82
10	FAQ	83
10.1	Questions	83
10.2	Troubleshooting	84
11	Translations	85
11.1	HistFitter	85
11.2	TRExFitter	86
12	Command Line API	89
12.1	pyhf	89
13	Python API	101
13.1	Top-Level	101
13.2	Probability Distribution Functions (PDFs)	103
13.3	Making Models from PDFs	107
13.4	Backends	116
13.5	Optimizers	144
13.6	Modifiers	147
13.7	Interpolators	149
13.8	Inference	152
13.9	Exceptions	163
13.10	Utilities	166
13.11	Contrib	167
14	Use and Citations	169
14.1	Citation	169
14.2	Use in Publications	169
14.3	Published Likelihoods	170
15	Roadmap (2019-2020)	171
15.1	Overview and Goals	171
15.2	Time scale	172
15.3	Roadmap	172
15.4	Presentations During Roadmap Timeline	174
16	Contributors	175
17	pure-python fitting/limit-setting/interval estimation HistFactory-style	177
17.1	Hello World	177
17.2	What does it support	178
17.3	Todo	179
17.4	A one bin example	179
17.5	A two bin example	181
17.6	Installation	182
17.7	Questions	182
17.8	Citation	183

17.9 Authors	183
17.10 Milestones	183
17.11 Acknowledgements	183
18 Indices and tables	185
Bibliography	187
Python Module Index	189
Index	191

Measurements in High Energy Physics (HEP) rely on determining the compatibility of observed collision events with theoretical predictions. The relationship between them is often formalised in a statistical *model* $f(x|)$ describing the probability of data x given model parameters θ . Given observed data, the *likelihood* $\mathcal{L}()$ then serves as the basis to test hypotheses on the parameters θ . For measurements based on binned data (*histograms*), the Poisson family of statistical models has been widely used in both Standard Model measurements [intro-4] as well as searches for new physics [intro-5]. In this package, a declarative, plain-text format for describing Poisson -based likelihoods is presented that is targeted for reinterpretation and long-term preservation in analysis data repositories such as HEPData [intro-3].

HISTFACTORY

Statistical models described using [intro-2] center around the simultaneous measurement of disjoint binned distributions (*channels*) observed as event counts . For each channel, the overall expected event rate¹ is the sum over a number of physics processes (*samples*). The sample rates may be subject to parametrised variations, both to express the effect of *free parameters*² and to account for systematic uncertainties as a function of *constrained parameters* . The degree to which the latter can cause a deviation of the expected event rates from the nominal rates is limited by *constraint terms*. In a frequentist framework these constraint terms can be viewed as *auxiliary measurements* with additional global observable data , which paired with the channel data completes the observation $x = (,)$. In addition to the partition of the full parameter set into free and constrained parameters $= (,)$, a separate partition $= (,)$ will be useful in the context of hypothesis testing, where a subset of the parameters are declared *parameters of interest* and the remaining ones as *nuisance parameters* .

$$f(x) = f(x | \underbrace{\text{free}}_{\text{constrained}}, \underbrace{\text{parameters of interest}}_{\text{nuisance parameters}}) \quad (1.1)$$

Thus, the overall structure of a probability model is a product of the analysis-specific model term describing the measurements of the channels and the analysis-independent set of constraint terms:

$$f(, |,) = \underbrace{\prod_{c \in \text{channels}} \prod_{b \in \text{bins}_c} \text{Pois}(n_{cb} | \nu_{cb}(,))}_{\text{Simultaneous measurement of multiple channels}} \underbrace{\prod_{a \in \text{constraint terms}} c(a |)}_{\text{constraint terms for } x \text{ 201C auxiliary measurements } x \text{ 201D}}, \quad (1.2)$$

where within a certain integrated luminosity we observe n_{cb} events given the expected rate of events $\nu_{cb}(,)$ as a function of unconstrained parameters and constrained parameters . The latter has corresponding one-dimensional constraint terms $c(a |)$ with auxiliary data a constraining the parameter . The event rates ν_{cb} are defined as

$$\nu_{cb}(,) = \sum_{s \in \text{samples}} \nu_{scb}(,) = \sum_{s \in \text{samples}} \underbrace{\left(\prod_{\kappa \in \kappa} \kappa_{scb}(,) \right)}_{\text{multiplicative modifiers}} \underbrace{\left(\nu_{scb}^0(,) + \sum_{\Delta \in \Delta} \Delta_{scb}(,) \right)}_{\text{additive modifiers}}. \quad (1.3)$$

The total rates are the sum over sample rates ν_{scb} , each determined from a *nominal rate* ν_{scb}^0 and a set of multiplicative and additive denoted *rate modifiers* $\kappa()$ and $\Delta()$. These modifiers are functions of (usually a single) model parameters. Starting from constant nominal rates, one can derive the per-bin event rate modification by iterating over all sample rate modifications as shown in (1.3).

As summarised in *Modifiers and Constraints*, rate modifications are defined in for bin b , sample s , channel c . Each modifier is represented by a parameter $\phi \in \{\gamma, \alpha, \lambda, \mu\}$. By convention bin-wise parameters are denoted with γ and

¹ Here rate refers to the number of events expected to be observed within a given data-taking interval defined through its integrated luminosity. It often appears as the input parameter to the Poisson distribution, hence the name “rate”.

² These *free parameters* frequently include the of a given process, i.e. its cross-section normalised to a particular reference cross-section such as that expected from the Standard Model or a given BSM scenario.

interpolation parameters with α . The luminosity λ and scale factors μ affect all bins equally. For constrained modifiers, the implied constraint term is given as well as the necessary input data required to construct it. σ_b corresponds to the relative uncertainty of the event rate, whereas δ_b is the event rate uncertainty of the sample relative to the total event rate $\nu_b = \sum_s \nu_{sb}^0$.

Modifiers implementing uncertainties are paired with a corresponding default constraint term on the parameter limiting the rate modification. The available modifiers may affect only the total number of expected events of a sample within a given channel, i.e. only change its normalisation, while holding the distribution of events across the bins of a channel, i.e. its “shape”, invariant. Alternatively, modifiers may change the sample shapes. Here supports correlated an uncorrelated bin-by-bin shape modifications. In the former, a single nuisance parameter affects the expected sample rates within the bins of a given channel, while the latter introduces one nuisance parameter for each bin, each with their own constraint term. For the correlated shape and normalisation uncertainties, makes use of interpolating functions, f_p and g_p , constructed from a small number of evaluations of the expected rate at fixed values of the parameter α ³. For the remaining modifiers, the parameter directly affects the rate.

Table 1: Modifiers and Constraints

Description	Modification	Constraint Term c	Input
Uncorrelated Shape	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \text{Pois}(r_b = \sigma_b^{-2} \rho_b = \sigma_b^{-2} \gamma_b)$	σ_b
Correlated Shape	$\Delta_{scb}(\alpha) = f_p(\alpha \Delta_{scb, \alpha=-1}, \Delta_{scb, \alpha=1})$	$\text{Gaus}(a = 0 \alpha, \sigma = 1)$	$\Delta_{scb, \alpha=\pm 1}$
Normalisation Unc.	$\kappa_{scb}(\alpha) = g_p(\alpha \kappa_{scb, \alpha=-1}, \kappa_{scb, \alpha=1})$	$\text{Gaus}(a = 0 \alpha, \sigma = 1)$	$\kappa_{scb, \alpha=\pm 1}$
MC Stat. Uncertainty	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \text{Gaus}(a_{\gamma_b} = 1 \gamma_b, \delta_b)$	$\delta_b^2 = \sum_s \delta_{sb}^2$
Luminosity	$\kappa_{scb}(\lambda) = \lambda$	$\text{Gaus}(l = \lambda_0 \lambda, \sigma_\lambda)$	$\lambda_0, \sigma_\lambda$
Normalisation	$\kappa_{scb}(\mu_b) = \mu_b$		
Data-driven Shape	$\kappa_{scb}(\gamma_b) = \gamma_b$		

Given the likelihood $\mathcal{L}()$, constructed from observed data in all channels and the implied auxiliary data, *measurements* in the form of point and interval estimates can be defined. The majority of the parameters are *nuisance parameters* — parameters that are not the main target of the measurement but are necessary to correctly model the data. A small subset of the unconstrained parameters may be declared as *parameters of interest* for which measurements hypothesis tests are performed, e.g. profile likelihood methods [intro-1]. The *Symbol Notation* table provides a summary of all the notation introduced in this documentation.

³ This is usually constructed from the nominal rate and measurements of the event rate at $\alpha = \pm 1$, where the value of the modifier at $\alpha = \pm 1$ must be provided and the value at $\alpha = 0$ corresponds to the corresponding identity operation of the modifier, i.e. $f_p(\alpha = 0) = 0$ and $g_p(\alpha = 0) = 1$ for additive and multiplicative modifiers respectively. See Section 4.1 in [intro-2].

Table 2: Symbol Notation

Symbol	Name
$f(x)$	model
$\mathcal{L}()$	likelihood
$x = \{, \}$	full dataset (including auxiliary data)
	channel data (or event counts)
	auxiliary data
$\nu()$	calculated event rates
$= \{, \} = \{, \}$	all parameters
	free parameters
	constrained parameters
	parameters of interest
	nuisance parameters
$\kappa()$	multiplicative rate modifier
$\Delta()$	additive rate modifier
$c(a)$	constraint term for constrained parameter
σ	relative uncertainty in the constrained parameter

DECLARATIVE FORMATS

While flexible enough to describe a wide range of LHC measurements, the design of the specification is sufficiently simple to admit a *declarative format* that fully encodes the statistical model of the analysis. This format defines the channels, all associated samples, their parameterised rate modifiers and implied constraint terms as well as the measurements. Additionally, the format represents the mathematical model, leaving the implementation of the likelihood minimisation to be analysis-dependent and/or language-dependent. Originally XML was chosen as a specification language to define the structure of the model while introducing a dependence on to encode the nominal rates and required input data of the constraint terms [intro-2]. Using this specification, a model can be constructed and evaluated within the framework.

This package introduces an updated form of the specification based on the ubiquitous plain-text JSON format and its schema-language *JSON Schema*. Described in more detail in *Likelihood Specification*, this schema fully specifies both structure and necessary constrained data in a single document and thus is implementation independent.

ADDITIONAL MATERIAL

3.1 Footnotes

3.2 Bibliography

LIKELIHOOD SPECIFICATION

The structure of the JSON specification of models follows closely the original XML-based specification [likelihood-2].

4.1 Workspace

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "$id": "https://scikit-hep.org/pyhf/schemas/1.0.0/workspace.json",
  "$ref": "defs.json#/definitions/workspace"
}
```

The overall document in the above code snippet describes a *workspace*, which includes

- **channels:** The channels in the model, which include a description of the samples within each channel and their possible parametrised modifiers.
- **measurements:** A set of measurements, which define among others the parameters of interest for a given statistical analysis objective.
- **observations:** The observed data, with which a likelihood can be constructed from the model.

A workspace consists of the channels, one set of observed data, but can include multiple measurements. If provided a JSON file, one can quickly check that it conforms to the provided workspace specification as follows:

```
import json, requests, jsonschema
workspace = json.load(open('/path/to/analysis_workspace.json'))
# if no exception is raised, it found and parsed the schema
schema = requests.get('https://scikit-hep.org/pyhf/schemas/1.0.0/workspace.json').
    → json()
# If no exception is raised by validate(), the instance is valid.
jsonschema.validate(instance=workspace, schema=schema)
```

4.2 Channel

A channel is defined by a channel name and a list of samples [likelihood-1].

```
{
  "channel": {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
```

(continues on next page)

(continued from previous page)

```

        "samples": { "type": "array", "items": {"$ref": "#/definitions/sample"},
        ↪ "minItems": 1 }
    },
    "required": ["name", "samples"],
    "additionalProperties": false
},
}

```

The Channel specification consists of a list of channel descriptions. Each channel, an analysis region encompassing one or more measurement bins, consists of a name field and a samples field (see [Channel](#)), which holds a list of sample definitions (see [Sample](#)). Each sample definition in turn has a name field, a data field for the nominal event rates for all bins in the channel, and a modifiers field of the list of modifiers for the sample.

4.3 Sample

A sample is defined by a sample name, the sample event rate, and a list of modifiers [[likelihood-1](#)].

```

{
  "sample": {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
      "data": { "type": "array", "items": { "type": "number", "minItems": 1 },
      "modifiers": {
        "type": "array",
        "items": {
          "anyOf": [
            { "$ref": "#/definitions/modifier/histosys" },
            { "$ref": "#/definitions/modifier/lumi" },
            { "$ref": "#/definitions/modifier/normfactor" },
            { "$ref": "#/definitions/modifier/normsys" },
            { "$ref": "#/definitions/modifier/shapefactor" },
            { "$ref": "#/definitions/modifier/shapesys" },
            { "$ref": "#/definitions/modifier/staterror" }
          ]
        }
      }
    },
    "required": ["name", "data", "modifiers"],
    "additionalProperties": false
  },
}

```

4.4 Modifiers

The modifiers that are applicable for a given sample are encoded as a list of JSON objects with three fields. A name field, a type field denoting the class of the modifier, and a data field which provides the necessary input data as denoted in *Modifiers and Constraints*.

Based on the declared modifiers, the set of parameters and their constraint terms are derived implicitly as each type of modifier unambiguously defines the constraint terms it requires. Correlated shape modifiers and normalisation uncertainties have compatible constraint terms and thus modifiers can be declared that *share* parameters by re-using a name¹ for multiple modifiers. That is, a variation of a single parameter causes a shift within sample rates due to both shape and normalisation variations.

We review the structure of each modifier type below.

4.4.1 Uncorrelated Shape (shapesys)

To construct the constraint term, the relative uncertainties σ_b are necessary for each bin. Therefore, we record the absolute uncertainty as an array of floats, which combined with the nominal sample data yield the desired σ_b . An example is shown below:

```
{ "name": "mod_name", "type": "shapesys", "data": [1.0, 1.5, 2.0] }
```

An example of an uncorrelated shape modifier with three absolute uncertainty terms for a 3-bin channel.

Warning: Nuisance parameters will not be allocated for any bins where either

- the samples nominal expected rate is zero, or
- the absolute uncertainty is zero.

These values are, in the context of uncorrelated shape uncertainties, unphysical. If this situation occurs, one needs to go back and understand the inputs as this is undefined behavior in HistFactory.

The previous example will allocate three nuisance parameters for `mod_name`. The following example will allocate only two nuisance parameters for a 3-bin channel:

```
{ "name": "mod_name", "type": "shapesys", "data": [1.0, 0.0, 2.0] }
```

4.4.2 Correlated Shape (histosys)

This modifier represents the same source of uncertainty which has a different effect on the various sample shapes, hence a correlated shape. To implement an interpolation between sample distribution shapes, the distributions with a “downward variation” (“lo”) associated with $\alpha = -1$ and an “upward variation” (“hi”) associated with $\alpha = +1$ are provided as arrays of floats. An example is shown below:

```
{ "name": "mod_name", "type": "histosys", "data": { "hi_data": [20,15], "lo_data": [10,
↪ 10] } }
```

An example of a correlated shape modifier with absolute shape variations for a 2-bin channel.

¹ The name of a modifier specifies the parameter set it is controlled by. Modifiers with the same name share parameter sets.

4.4.3 Normalisation Uncertainty (normsys)

The normalisation uncertainty modifies the sample rate by a overall factor $\kappa(\alpha)$ constructed as the interpolation between downward (“lo”) and upward (“hi”) as well as the nominal setting, i.e. $\kappa(-1) = \kappa_{\alpha=-1}$, $\kappa(0) = 1$ and $\kappa(+1) = \kappa_{\alpha=+1}$. In the modifier definition we record $\kappa_{\alpha=+1}$ and $\kappa_{\alpha=-1}$ as floats. An example is shown below:

```
{ "name": "mod_name", "type": "normsys", "data": { "hi": 1.1, "lo": 0.9 } }
```

An example of a normalisation uncertainty modifier with scale factors recorded for the up/down variations of an n -bin channel.

4.4.4 MC Statistical Uncertainty (statererror)

As the sample counts are often derived from Monte Carlo (MC) datasets, they necessarily carry an uncertainty due to the finite sample size of the datasets. As explained in detail in [\[likelihood-2\]](#), adding uncertainties for each sample would yield a very large number of nuisance parameters with limited utility. Therefore a set of bin-wise scale factors γ_b is introduced to model the overall uncertainty in the bin due to MC statistics. The constrained term is constructed as a set of Gaussian constraints with a central value equal to unity for each bin in the channel. The scales σ_b of the constraint are computed from the individual uncertainties of samples defined within the channel relative to the total event rate of all samples: $\delta_{csb} = \sigma_{csb} / \sum_s \nu_{scb}^0$. As not all samples are within a channel are estimated from MC simulations, only the samples with a declared statistical uncertainty modifier enter the sum. An example is shown below:

```
{ "name": "mod_name", "type": "statererror", "data": [0.1] }
```

An example of a statistical uncertainty modifier.

4.4.5 Luminosity (lumi)

Sample rates derived from theory calculations, as opposed to data-driven estimates, are scaled to the integrated luminosity corresponding to the observed data. As the luminosity measurement is itself subject to an uncertainty, it must be reflected in the rate estimates of such samples. As this modifier is of global nature, no additional per-sample information is required and thus the data field is nulled. This uncertainty is relevant, in particular, when the parameter of interest is a signal cross-section. The luminosity uncertainty σ_λ is provided as part of the parameter configuration included in the measurement specification discussed in [Measurements](#). An example is shown below:

```
{ "name": "mod_name", "type": "lumi", "data": null }
```

An example of a luminosity modifier.

4.4.6 Unconstrained Normalisation (normfactor)

The unconstrained normalisation modifier scales the event rates of a sample by a free parameter μ . Common use cases are the signal rate of a possible BSM signal or simultaneous in-situ measurements of background samples. Such parameters are frequently the parameters of interest of a given measurement. No additional per-sample data is required. An example is shown below:

```
{ "name": "mod_name", "type": "normfactor", "data": null }
```

An example of a normalisation modifier.

4.4.7 Data-driven Shape (shapefactor)

In order to support data-driven estimation of sample rates (e.g. for multijet backgrounds), the data-driven shape modifier adds free, bin-wise multiplicative parameters. Similarly to the normalisation factors, no additional data is required as no constraint is defined. An example is shown below:

```
{ "name": "mod_name", "type": "shapefactor", "data": null }
```

An example of an uncorrelated shape modifier.

4.5 Data

The data provided by the analysis are the observed data for each channel (or region). This data is provided as a mapping from channel name to an array of floats, which provide the observed rates in each bin of the channel. The auxiliary data is not included as it is an input to the likelihood that does not need to be archived and can be determined automatically from the specification. An example is shown below:

```
{ "chan_name_one": [10, 20], "chan_name_two": [4, 0] }
```

An example of channel data.

4.6 Measurements

Given the data and the model definitions, a measurement can be defined. In the current schema, the measurements defines the name of the parameter of interest as well as parameter set configurations.² Here, the remaining information not covered through the channel definition is provided, e.g. for the luminosity parameter. For all modifiers, the default settings can be overridden where possible:

- **inits**: Initial value of the parameter.
- **bounds**: Interval bounds of the parameter.
- **auxdata**: Auxiliary data for the associated constraint term.
- **sigmas**: Associated uncertainty of the parameter.

An example is shown below:

```
{
  "name": "MyMeasurement",
  "config": {
    "poi": "SignalCrossSection", "parameters": [
      { "name": "lumi", "auxdata": [1.0], "sigmas": [0.017], "bounds": [[0.915, 1.085]], "inits": [1.0] },
      { "name": "mu_ttbar", "bounds": [[0, 5]] },
      { "name": "rw_1CR", "fixed": true }
    ]
  }
}
```

² In this context a parameter set corresponds to a named lower-dimensional subspace of the full parameters. In many cases these are one-dimensional subspaces, e.g. a specific interpolation parameter α or the luminosity parameter λ . For multi-bin channels, however, e.g. all bin-wise nuisance parameters of the uncorrelated shape modifiers are grouped under a single name. Therefore in general a parameter set definition provides arrays of initial values, bounds, etc.

An example of a measurement. This measurement, which scans over the parameter of interest `SignalCrossSection`, is setting configurations for the luminosity modifier, changing the default bounds for the normfactor modifier named `mu_ttbar`, and specifying that the modifier `rw_1CR` is held constant (fixed).

4.7 Observations

This is what we evaluate the hypothesis testing against, to determine the compatibility of signal+background hypothesis to the background-only hypothesis. This is specified as a list of objects, with each object structured as

- **name**: the channel for which the observations are recorded
- **data**: the bin-by-bin observations for the named channel

An example is shown below:

```
{
  "name": "channel1",
  "data": [110.0, 120.0]
}
```

An example of an observation. This observation recorded for a 2-bin channel `channel1`, has values 110.0 and 120.0.

4.8 Toy Example

```
{
  "channels": [
    { "name": "singlechannel",
      "samples": [
        { "name": "signal",
          "data": [5.0, 10.0],
          "modifiers": [ { "name": "mu", "type": "normfactor", "data": null } ]
        },
        { "name": "background",
          "data": [50.0, 60.0],
          "modifiers": [ { "name": "uncorr_bkguncrt", "type": "shapesys", "data": [
            ↪ [5.0, 12.0] } ]
        }
      ]
    }
  ],
  "observations": [
    { "name": "singlechannel", "data": [50.0, 60.0] }
  ],
  "measurements": [
    { "name": "Measurement", "config": { "poi": "mu", "parameters": [] } }
  ],
  "version": "1.0.0"
}
```

In the above example, we demonstrate a simple measurement of a single two-bin channel with two samples: a signal sample and a background sample. The signal sample has an unconstrained normalisation factor μ , while the background sample carries an uncorrelated shape systematic controlled by parameters γ_1 and γ_2 . The background uncertainty for the bins is 10% and 20% respectively.

4.9 Additional Material

4.9.1 Footnotes

4.9.2 Bibliography

FUNDAMENTALS

Notebooks:

```
[1]: %pylab inline
from ipywidgets import interact
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Populating the interactive namespace from numpy and matplotlib

5.1 Piecewise Linear Interpolation

References: <https://cds.cern.ch/record/1456844/files/CERN-OPEN-2012-016.pdf>

We wish to understand interpolation using the piecewise linear function. This is `intercode=0` in the above reference. This function is defined as (nb: vector denotes bold)

$$\eta_s(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) + \underbrace{\sum_{p \in \text{Syst}} I_{\text{lin.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-)}_{\text{deltas to calculate}}$$

with

$$I_{\text{lin.}}(\alpha; I^0, I^+, I^-) = \begin{cases} \alpha(I^+ - I^0) & \alpha \geq 0 \\ \alpha(I^0 - I^-) & \alpha < 0 \end{cases}$$

In this notebook, we'll demonstrate the technical implementation of these interpolations starting from simple dimensionality and increasing the dimensions as we go along. In all situations, we'll consider a single systematic that we wish to interpolate, such as Jet Energy Scale (JES).

Let's define the interpolate function. This function will produce the deltas we would like to calculate and sum with the nominal measurement to determine the interpolated measurements value.

```
[2]: def interpolate_deltas(down, nom, up, alpha):
    delta_up = up - nom
    delta_down = nom - down
    if alpha > 0:
        return delta_up*alpha
    else:
        return delta_down*alpha
```

Why are we calculating deltas? This is some additional foresight that you, the reader, may not have yet. Multiple interpolation schemes exist but they all rely on calculating the change with respect to the nominal measurement (the delta).

5.1.1 Case 1: The Single-binned Histogram

Let's first start with considering evaluating the total number of events after applying JES corrections. This is the single-bin case. Code that runs through event selection will vary the JES parameter and provide three histograms, each with a single bin. These three histograms represent the nominal-, up-, and down- variations of the JES nuisance parameter.

When processing, we find that there are 10 events nominally, and when we vary the JES parameter downwards, we only measure 8 events. When varying upwards, we measure 15 events.

```
[3]: down_1 = np.array([8])
     nom_1  = np.array([10])
     up_1   = np.array([15])
```

We would like to generate a function $f(\alpha_{\text{JES}})$ that linearly interpolates the number of events for us so we can scan the phase-space for calculating PDFs. The `interpolate_deltas()` function defined above does this for us.

```
[4]: alphas = np.linspace(-1.,1.)
     deltas = [interpolate_deltas(down_1, nom_1, up_1, alpha) for alpha in alphas]
     deltas[:5]

[4]: [array([-2.]),
     array([-1.91836735]),
     array([-1.83673469]),
     array([-1.75510204]),
     array([-1.67346939])]
```

So now that we've generated the deltas from the nominal measurement, we can plot this to see how the linear interpolation works in the single-bin case, where we plot the measured values in black, and the interpolation in dashed, blue.

```
[5]: plt.plot(alphas, [nom_1+delta for delta in deltas], linestyle='--')
     plt.scatter((-1,0,1), (down_1,nom_1,up_1), color='k')
     plt.xlabel(r'$\alpha_{\mathrm{JES}}$')
     plt.ylabel(r'Events')

[5]: Text(0,0.5,'Events')
```



Here, we can imagine building a 1-dimensional tensor (column-vector) of measurements as a function of α_{JES} with each row in the column vector corresponding to a given α_{JES} value.

5.1.2 Case 2: The Multi-binned Histogram

Now, let's increase the computational difficulty a little by increasing the dimensionality. Assume instead of a single-bin measurement, we have more measurements! We are good physicists after all. Imagine continuing on the previous example, where we add more bins, perhaps because we got more data. Imagine that this was binned by collection year, where we observed 10 events in the first year, 10.5 the next year, and so on...

```
[6]: down_hist = np.linspace(8,10,11)
     nom_hist = np.linspace(10,13,11)
     up_hist = np.linspace(15,20,11)
```

Now, we still need to interpolate. Just like before, we have varied JES upwards and downwards to determine the corresponding histograms of variations. In order to interpolate, we need to interpolate by bin for each bin in the three histograms we have here (or three measurements if you prefer).

Let's go ahead and plot these histograms as a function of the bin index with black as the nominal measurements, red and blue as the down and up variations respectively. The black points are the measurements we have, and for each bin, we would like to interpolate to get an interpolated histogram that represents the measurement as a function of α_{JES} .

```
[7]: def plot_measurements(down_hist, nom_hist, up_hist):
     bincenters = np.arange(len(nom_hist))
     for i, h in enumerate(zip(up_hist, nom_hist, down_hist)):
         plt.scatter([i]*len(h), h, color='k', alpha=0.5)

     for c,h in zip(['r','k','b'],[down_hist, nom_hist, up_hist]):
         plt.plot(bincenters, h, color=c, linestyle='-', alpha=0.5)

     plt.xlabel('Bin index in histogram')
     plt.ylabel('Events')

plot_measurements(down_hist, nom_hist, up_hist)
```



What does this look like if we evaluate at a single $\alpha_{\text{JES}} = 0.5$? We'll write a function that interpolates and then plots the interpolated values as a function of bin index, in green, dashed.

```
[8]: def plot_interpolated_histogram(alpha, down_hist, nom_hist, up_hist):
     bincenters = np.arange(len(nom_hist))
     interpolated_vals = [nominal + interpolate_deltas(down, nominal, up, alpha) for
     ↪down, nominal, up in zip(down_hist, nom_hist, up_hist)]
```

(continues on next page)

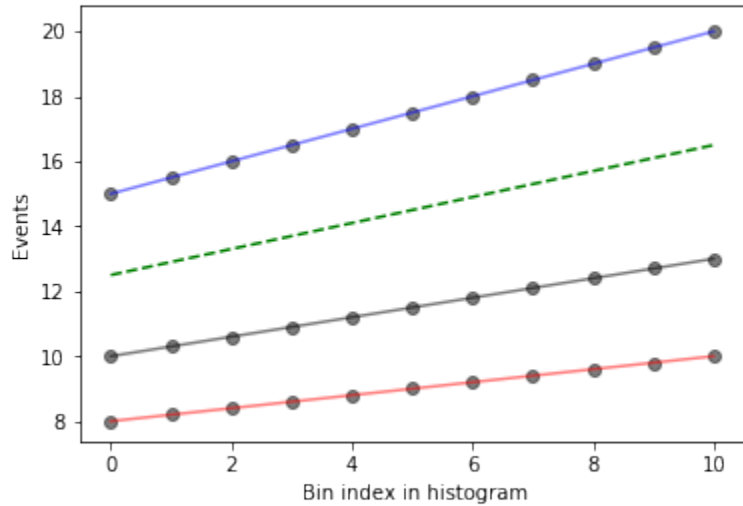
(continued from previous page)

```

plot_measurements(down_hist, nom_hist, up_hist)
plt.plot(bincenters, interpolated_vals, color = 'g', linestyle='--')

plot_interpolated_histogram(0.5, down_hist, nom_hist, up_hist)

```



We can go one step further in visualization and see what it looks like for different α_{JES} using iPyWidget's interactivity. Change the slider to get an idea of how the interpolation works.

```

[9]: x = interact(lambda alpha: plot_interpolated_histogram(alpha, down_hist, nom_hist, up_
↪ hist), alpha = (-1,1,0.1))

aW50ZXJhY3RpdmUoY2hpbGRyZW49KEZsb2F0U2xpZGVyKHZhbHVlPTAuMCwgZGVzY3JpcHRpb249dSdhbHB0YSdScsIG1heD0xLjAs

```

The magic in `plot_interpolated_histogram()` happens to be that for a given α_{JES} , we iterate over all measurements bin-by-bin to calculate the interpolated value

```

[nominal + interpolate_deltas(down, nominal, up, alpha) for down, nominal, up in zip(
↪ ..hists...)]

```

So you can imagine that we're building up a 2-dimensional tensor with each row corresponding to a different α_{JES} and each column corresponding to the bin index of the histograms (or measurements). Let's go ahead and build a 3-dimensional representation of our understanding so far!

```

[10]: def interpolate_alpha_range(alphas, down_hist, nom_hist, up_hist):
        at_alphas = []
        for alpha in alphas:
            interpolated_hist_at_alpha = [nominal + interpolate_deltas(down, nominal, up,
↪ alpha) for down, nominal, up in zip(down_hist, nom_hist, up_hist)]
            at_alphas.append(interpolated_hist_at_alpha)
        return np.array(at_alphas)

```

And then with this, we are interpolating over all histograms bin-by-bin and producing a 2-dimensional tensor with each row corresponding to a specific value of α_{JES} .

```

[11]: alphas = np.linspace(-1,1,11)

```

(continues on next page)

(continued from previous page)

```

interpolated_vals_at_alphas = interpolate_alpha_range(alphas, down_hist, nom_hist, up_
↪hist)

print(interpolated_vals_at_alphas[alphas==-1])
print(interpolated_vals_at_alphas[alphas==0])
print(interpolated_vals_at_alphas[alphas==1])

[[ 8.   8.2  8.4  8.6  8.8  9.   9.2  9.4  9.6  9.8 10. ]]
[[10.  10.3 10.6 10.9 11.2 11.5 11.8 12.1 12.4 12.7 13. ]]
[[15.  15.5 16.  16.5 17.  17.5 18.  18.5 19.  19.5 20. ]]

```

We have a way to generate the 2-dimensional tensor. Let's go ahead and add in all dimensions. Additionally, we'll add in some extra code to show the projection of the 2-d plots that we made earlier to help understand the 3-d plot a bit better. Like before, let's plot specifically colored lines for $\alpha_{\text{JES}} = 0.5$ as well as provide an interactive session.

```

[13]: def plot_wire(alpha):
    alphas = np.linspace(-1,1,51)
    at_alphas = interpolate_alpha_range(alphas, down_hist, nom_hist, up_hist)
    bincenters = np.arange(len(nom_hist))
    x,y = np.meshgrid(bincenters,alphas)
    z = np.asarray(at_alphas)
    bottom = np.zeros_like(x)
    fig = plt.figure(figsize=(10, 10))
    ax1 = fig.add_subplot(111, projection='3d')
    ax1.plot_wireframe(x, y, z, alpha = 0.3)

    x,y = np.meshgrid(bincenters,[alpha])
    z = interpolate_alpha_range([alpha], down_hist, nom_hist, up_hist)

    ax1.plot_wireframe(x, y, z, edgecolor = 'g', linestyle='--')
    ax1.set_xlim(0,10)
    ax1.set_ylim(-1.0,1.5)
    ax1.set_zlim(0,25)
    ax1.view_init(azim=-125)
    ax1.set_xlabel('Bin Index')
    ax1.set_ylabel(r'$\alpha_{\mathrm{JES}}$')
    ax1.set_zlabel('Events')

    # add in 2D plot goodness

    for c,h,zs in zip(['r','k','b'],[down_hist, nom_hist, up_hist],[-1.0,0.0,1.0]):
        ax1.plot(bincenters, h, color=c, linestyle='-', alpha=0.5, zdir='y', zs=zs)
        ax1.plot(bincenters, h, color=c, linestyle='-', alpha=0.25, zdir='y', zs=1.5)

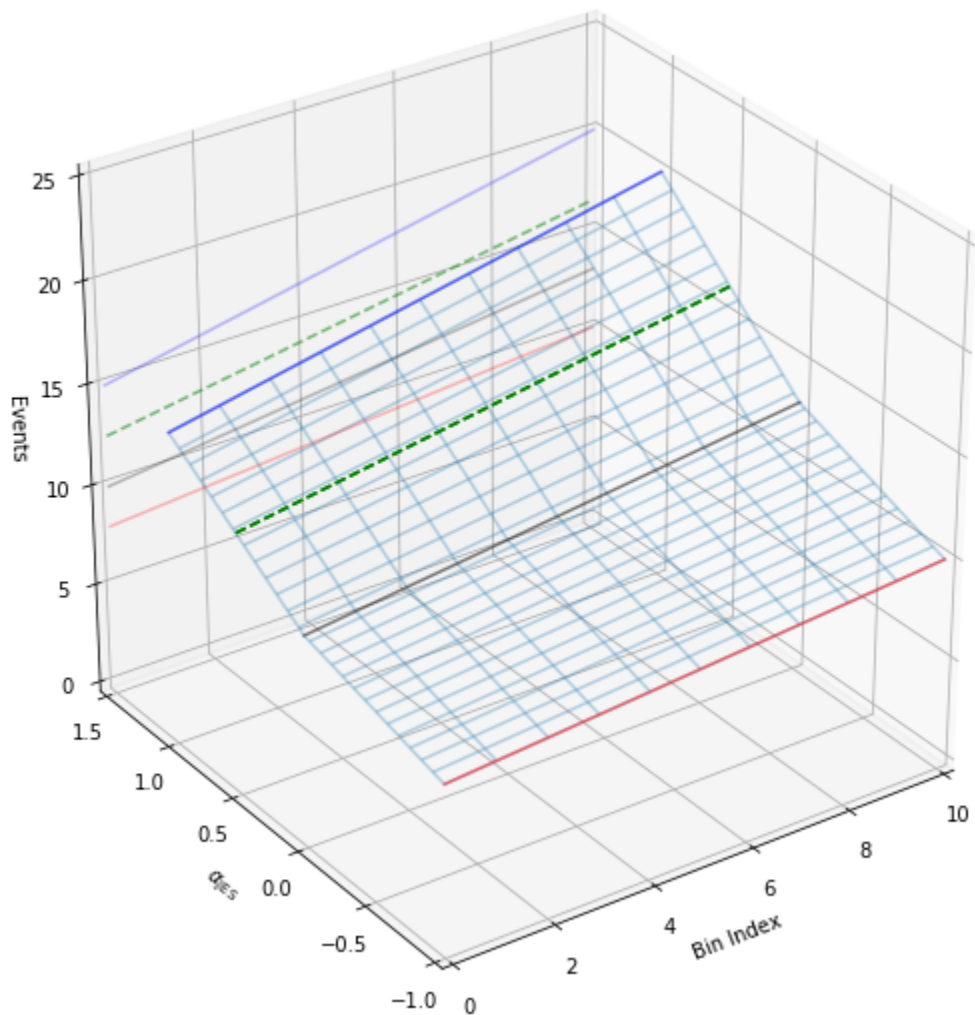
    ax1.plot(bincenters, z.T, color = 'g', linestyle='--', zdir='y', zs=alpha)
    ax1.plot(bincenters, z.T, color = 'g', linestyle='--', alpha=0.5, zdir='y', zs=1.
↪5)

    plt.show()

plot_wire(0.5)

interact(plot_wire,alpha = (-1,1,0.1))

```



aW50ZXJhY3RpdmoUoY2hpbGRyZW49KEZsb2F0U2xpZGVyKHZhbHVlPTAuMCwgZGVzY3JpcHRpb249dSdhbHB0YScsIG1heD0xLjAs

```
[13]: <function __main__.plot_wire>
```

5.2 Tensorizing Interpolators

This notebook will introduce some tensor algebra concepts about being able to convert from calculations inside for-loops into a single calculation over the entire tensor. It is assumed that you have some familiarity with what interpolation functions are used for in `pyhf`.

To get started, we'll load up some functions we wrote whose job is to generate sets of histograms and alphas that we will compute interpolations for. This allows us to generate random, structured input data that we can use to test the tensorized form of the interpolation function against the original one we wrote. For now, we will consider only the `numpy` backend for simplicity, but can replace `np` to `pyhf.tensorlib` to achieve identical functionality.

The function `random_histosets_alphasets_pair` will produce a pair `(histogramsets,`

alphasets) of histograms and alphas for those histograms that represents the type of input we wish to interpolate on.

```
[1]: import numpy as np

def random_histosets_alphasets_pair(nsysts = 150, nhistos_per_syst_upto = 300,
    ↳ nalphas = 1, nbins_upto = 1):
    def generate_shapes(histogramssets, alphasets):
        h_shape = [len(histogramssets), 0, 0, 0]
        a_shape = (len(alphasets), max(map(len, alphasets)))
        for hs in histogramssets:
            h_shape[1] = max(h_shape[1], len(hs))
            for h in hs:
                h_shape[2] = max(h_shape[2], len(h))
                for sh in h:
                    h_shape[3] = max(h_shape[3], len(sh))
        return tuple(h_shape), a_shape

    def filled_shapes(histogramssets, alphasets):
        # pad our shapes with NaNs
        histos, alphas = generate_shapes(histogramssets, alphasets)
        histos, alphas = np.ones(histos) * np.nan, np.ones(alphas) * np.nan
        for i, syst in enumerate(histogramssets):
            for j, sample in enumerate(syst):
                for k, variation in enumerate(sample):
                    histos[i, j, k, :len(variation)] = variation
        for i, alphasets in enumerate(alphasets):
            alphas[i, :len(alphasets)] = alphasets
        return histos, alphas

    nsyst_histos = np.random.randint(1, 1+nhistos_per_syst_upto, size=nsysts)
    nhistograms = [np.random.randint(1, nbins_upto+1, size=n) for n in nsyst_histos]
    random_alphas = [np.random.uniform(-1, 1, size=nalphas) for n in nsyst_histos]

    random_histogramssets = [
        [# all histos affected by systematic $nh
        [# sample $i, systematic $nh
            np.random.uniform(10*i+j, 10*i+j+1, size = nbins).tolist() for j in_
    ↳ range(3)
        ] for i, nbins in enumerate(nh)
    ] for nh in nhistograms
    ]
    h, a = filled_shapes(random_histogramssets, random_alphas)
    return h, a
```

5.2.1 The (slow) interpolations

In all cases, the way we do interpolations is as follows:

1. Loop over both the `histogramssets` and `alphasets` simultaneously (e.g. using python's `zip()`)
2. Loop over all histograms set in the set of histograms sets that correspond to the histograms affected by a given systematic
3. Loop over all of the alphas in the set of alphas
4. Loop over all the bins in the histogram sets simultaneously (e.g. using python's `zip()`)
5. Apply the interpolation across the same bin index

This is already exhausting to think about, so let's put this in code form. Depending on the kind of interpolation being done, we'll pass in `func` as an argument to the top-level interpolation loop to switch between linear (`interpcode=0`) and non-linear (`interpcode=1`).

```
[2]: def interpolation_looper(histogramssets, alphasets, func):
    all_results = []
    for histoset, alpha in zip(histogramssets, alphasets):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphasets:
                alpha_result = []
                for down,nom,up in zip(histo[0],histo[1],histo[2]):
                    v = func(down, nom, up, alpha)
                    alpha_result.append(v)
                histo_result.append(alpha_result)
    return all_results
```

And we can also define our linear and non-linear interpolations we'll consider in this notebook that we wish to tensorize.

```
[3]: def interpolation_linear(histogramssets,alphasets):
    def summand(down, nom, up, alpha):
        delta_up = up - nom
        delta_down = nom - down
        if alpha > 0:
            delta = delta_up*alpha
        else:
            delta = delta_down*alpha
        return nom + delta
    return interpolation_looper(histogramssets, alphasets, summand)

def interpolation_nonlinear(histogramssets,alphasets):
    def product(down, nom, up, alpha):
        delta_up = up/nom
        delta_down = down/nom
        if alpha > 0:
            delta = delta_up**alpha
        else:
            delta = delta_down**(-alpha)
        return nom*delta
    return interpolation_looper(histogramssets, alphasets, product)
```

We will also define a helper function that allows us to pass in two functions we wish to compare the outputs for:

```
[4]: def compare_fns(func1, func2):
    h,a = random_histosets_alphasets_pair()

    def _func_runner(func, histsets, alphasets):
        return np.asarray(func(histsets,alphasets))

    old = _func_runner(func1, h, a)
    new = _func_runner(func2, h, a)

    return (np.all(old[~np.isnan(old)] == new[~np.isnan(new)]), (h,a))
```

For the rest of the notebook, we will detail in explicit form how the linear interpolator gets tensorized, step-by-step.

The same sequence of steps will be shown for the non-linear interpolator – but it is left up to the reader to understand the steps.

5.2.2 Tensorizing the Linear Interpolator

Step 0

Step 0 requires converting the innermost conditional check on `alpha > 0` into something tensorizable. This also means the calculation itself is going to become tensorized. So we will convert from

```
if alpha > 0:
    delta = delta_up*alpha
else:
    delta = delta_down*alpha
```

to

```
delta = np.where(alpha > 0, delta_up*alpha, delta_down*alpha)
```

Let's make that change now, and let's check to make sure we still do the calculation correctly.

```
[5]: # get the internal calculation to use tensorlib backend
def new_interpolation_linear_step0(histogramssets, alphasets):
    all_results = []
    for histoset, alphaset in zip(histogramssets, alphasets):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphaset:
                alpha_result = []
                for down, nom, up in zip(histo[0], histo[1], histo[2]):
                    delta_up = up - nom
                    delta_down = nom - down
                    delta = np.where(alpha > 0, delta_up*alpha, delta_down*alpha)
                    v = nom + delta
                    alpha_result.append(v)
                histo_result.append(alpha_result)
            return all_results
```

And does the calculation still match?

```
[6]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step0)
print(result)

True
```

```
[7]: %%timeit
interpolation_linear(h,a)

189 ms ± 6.14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[8]: %%timeit
new_interpolation_linear_step0(h,a)
```

```
255 ms ± 11.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Great! We're a little bit slower right now, but that's expected. We're just getting started.

Step 1

In this step, we would like to remove the innermost `zip()` call over the histogram bins by calculating the interpolation between the histograms in one fell swoop. This means, instead of writing something like

```
for down,nom,up in zip(histo[0],histo[1],histo[2]):
    delta_up = up - nom
    ...
```

one can instead write

```
delta_up = histo[2] - histo[1]
...
```

taking advantage of the automatic broadcasting of operations on input tensors. This sort of feature of the tensor backends allows us to speed up code, such as interpolation.

```
[9]: # update the delta variations to remove the zip() call and remove most-nested loop
def new_interpolation_linear_step1(histogramssets,alphasets):
    all_results = []
    for histoset, alpha in zip(histogramssets,alphasets):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphaset:
                alpha_result = []
                deltas_up = histo[2]-histo[1]
                deltas_dn = histo[1]-histo[0]
                calc_deltas = np.where(alpha > 0, deltas_up*alpha, deltas_dn*alpha)
                v = histo[1] + calc_deltas
                alpha_result.append(v)
                histo_result.append(alpha_result)
    return all_results
```

And does the calculation still match?

```
[10]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step1)
print(result)

True
```

```
[11]: %timeit
interpolation_linear(h,a)

188 ms ± 7.14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[12]: %timeit
new_interpolation_linear_step1(h,a)

492 ms ± 42.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Great!

Step 2

In this step, we would like to move the giant array of the deltas calculated to the beginning – outside of all loops – and then only take a subset of it for the calculation itself. This allows us to figure out the entire structure of the input for the rest of the calculations as we slowly move towards including `einsum()` calls (einstein summation). This means we would like to go from

```
for histo in histoset:
    delta_up = histo[2] - histo[1]
    ...
```

to

```
all_deltas = ...
for nh, histo in enumerate(histoset):
    deltas = all_deltas[nh]
    ...
```

Again, we are taking advantage of the automatic broadcasting of operations on input tensors to calculate all the deltas in a single action.

```
[13]: # figure out the giant array of all deltas at the beginning and only take subsets of
      ↪ it for the calculation
def new_interpolation_linear_step2(histogramssets, alphasets):
    all_results = []

    allset_all_histo_deltas_up = histogramssets[:, :, 2] - histogramssets[:, :, 1]
    allset_all_histo_deltas_dn = histogramssets[:, :, 1] - histogramssets[:, :, 0]

    for nset, (histoset, alphaset) in enumerate(zip(histogramssets, alphasets)):
        set_result = []

        all_histo_deltas_up = allset_all_histo_deltas_up[nset]
        all_histo_deltas_dn = allset_all_histo_deltas_dn[nset]

        for nh, histo in enumerate(histoset):
            alpha_deltas = []
            for alpha in alphaset:
                alpha_result = []
                deltas_up = all_histo_deltas_up[nh]
                deltas_dn = all_histo_deltas_dn[nh]
                calc_deltas = np.where(alpha > 0, deltas_up*alpha, deltas_dn*alpha)
                alpha_deltas.append(calc_deltas)
            set_result.append([histo[1] + d for d in alpha_deltas])
        all_results.append(set_result)
    return all_results
```

And does the calculation still match?

```
[14]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step2)
      print(result)
```

```
True
```

```
[15]: %timeit
      interpolation_linear(h,a)
```

```
179 ms ± 12.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[16]: %%timeit
new_interpolation_linear_step2(h,a)

409 ms ± 20.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Great!

Step 3

In this step, we get to introduce einstein summation to generalize the calculations we perform across many dimensions in a more concise, straightforward way. See [this blog post](#) for some more details on einstein summation notation. In short, it allows us to write

$$c_j = \sum_i \sum_k A_{ik} B_{kj} \quad \rightarrow \quad \text{einsum}("ij, jk \rightarrow i", A, B)$$

in a much more elegant way to express many kinds of common tensor operations such as dot products, transposes, outer products, and so on. This step is generally the hardest as one needs to figure out the corresponding `einsum` that keeps the calculation preserved (and matching). To some extent it requires a lot of trial and error until you get a feel for how einstein summation notation works.

As a concrete example of a conversion, we wish to go from something like

```
for nh, histo in enumerate(histoset):
    for alpha in alphasets:
        deltas_up = all_histo_deltas_up[nh]
        deltas_dn = all_histo_deltas_dn[nh]
        calc_deltas = np.where(alpha > 0, deltas_up*alpha, deltas_dn*alpha)
        ...
```

to get rid of the loop over alpha

```
for nh, histo in enumerate(histoset):
    alphas_times_deltas_up = np.einsum('i, j->ij', alphasets, all_histo_deltas_up[nh])
    alphas_times_deltas_dn = np.einsum('i, j->ij', alphasets, all_histo_deltas_dn[nh])
    masks = np.einsum('i, j->ij', alphasets > 0, np.ones_like(all_histo_deltas_dn[nh]))

    alpha_deltas = np.where(masks, alphas_times_deltas_up, alphas_times_deltas_dn)
    ...
```

In this particular case, we need an outer product that multiplies across the `alphasets` to the corresponding `histoset` for the up/down variations. Then we just need to select from either the up variation calculation or the down variation calculation based on the sign of alpha. Try to convince yourself that the einstein summation does what the for-loop does, but a little bit more concisely, and perhaps more clearly! How does the function look now?

```
[17]: # remove the loop over alphas, starts using einsum to help generalize to more_
      ↪ dimensions
def new_interpolation_linear_step3(histogramssets, alphasets):
    all_results = []

    allset_all_histo_deltas_up = histogramssets[:, :, 2] - histogramssets[:, :, 1]
    allset_all_histo_deltas_dn = histogramssets[:, :, 1] - histogramssets[:, :, 0]

    for nset, (histoset, alphasets) in enumerate(zip(histogramssets, alphasets)):
```

(continues on next page)

(continued from previous page)

```

    set_result = []

    all_histo_deltas_up = allset_all_histo_deltas_up[nset]
    all_histo_deltas_dn = allset_all_histo_deltas_dn[nset]

    for nh, histo in enumerate(histoset):
        alphas_times_deltas_up = np.einsum('i, j->ij', alphas, all_histo_deltas_
↪up[nh])
        alphas_times_deltas_dn = np.einsum('i, j->ij', alphas, all_histo_deltas_
↪dn[nh])
        masks = np.einsum('i, j->ij', alphas > 0, np.ones_like(all_histo_deltas_
↪dn[nh]))

        alpha_deltas = np.where(masks, alphas_times_deltas_up, alphas_times_
↪deltas_dn)
        set_result.append([histo[1] + d for d in alpha_deltas])

    all_results.append(set_result)
    return all_results

```

And does the calculation still match?

```
[18]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step3)
      print(result)
```

```
True
```

```
[19]: %timeit
      interpolation_linear(h,a)
```

```
166 ms ± 11.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[20]: %timeit
      new_interpolation_linear_step3(h,a)
```

```
921 ms ± 133 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Great! Note that we've been getting a little bit slower during these steps. It will all pay off in the end when we're fully tensorized! A lot of the internal steps are overkill with the heavy einstein summation and broadcasting at the moment, especially for how many loops in we are.

Step 4

Now in this step, we will move the einstein summations to the outer loop, so that we're calculating it once! This is the big step, but a little bit easier because all we're doing is adding extra dimensions into the calculation. The underlying calculation won't have changed. At this point, we'll also rename from *i* and *j* to *a* and *b* for alpha and bin (as in the bin in the histogram). To continue the notation as well, here's a summary of the dimensions involved:

- *s* will be for the set under consideration (e.g. the modifier)
- *a* will be for the alpha variation
- *h* will be for the histogram affected by the modifier
- *b* will be for the bin of the histogram

So we wish to move the `einsum` code from

```
for nset, (histoset, alphaset) in enumerate(zip(histogramssets, alphasets)):
    ...

    for nh, histo in enumerate(histoset):
        alphas_times_deltas_up = np.einsum('i,j->ij', alphaset, all_histo_deltas_up[nh])
        ...
```

to

```
all_alphas_times_deltas_up = np.einsum('...', alphaset, all_histo_deltas_up)
for nset, (histoset, alphaset) in enumerate(zip(histogramssets, alphasets)):
    ...

    for nh, histo in enumerate(histoset):
        ...
```

So how does this new function look?

```
[21]: # move the einsums to outer loops to get ready to get rid of all loops
def new_interpolation_linear_step4(histogramssets, alphasets):
    allset_all_histo_deltas_up = histogramssets[:, :, 2] - histogramssets[:, :, 1]
    allset_all_histo_deltas_dn = histogramssets[:, :, 1] - histogramssets[:, :, 0]
    allset_all_histo_nom = histogramssets[:, :, 1]

    allsets_all_histos_alphas_times_deltas_up = np.einsum('sa,shb->shab', alphasets,
    ↪allset_all_histo_deltas_up)
    allsets_all_histos_alphas_times_deltas_dn = np.einsum('sa,shb->shab', alphasets,
    ↪allset_all_histo_deltas_dn)
    allsets_all_histos_masks = np.einsum('sa,s...u->s...au', alphasets > 0, np.ones_
    ↪like(allset_all_histo_deltas_dn))

    allsets_all_histos_deltas = np.where(allsets_all_histos_masks, allsets_all_histos_
    ↪alphas_times_deltas_up, allsets_all_histos_alphas_times_deltas_dn)

    all_results = []
    for nset, histoset in enumerate(histogramssets):
        all_histos_deltas = allsets_all_histos_deltas[nset]
        set_result = []
        for nh, histo in enumerate(histoset):
            set_result.append([d + histoset[nh, 1] for d in all_histos_deltas[nh]])
        all_results.append(set_result)
    return all_results
```

And does the calculation still match?

```
[22]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step4)
print(result)
```

```
True
```

```
[23]: %%timeit
interpolation_linear(h,a)
```

```
160 ms ± 5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[24]: %%timeit
new_interpolation_linear_step4(h,a)
```

```
119 ms ± 3.19 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Great! And look at that huge speed up in time already, just from moving the multiple, heavy einstein summation calculations up through the loops. We still have some more optimizing to do as we still have explicit loops in our code. Let's keep at it, we're almost there!

Step 5

The hard part is mostly over. We have to now think about the nominal variations. Recall that we were trying to add the nominals to the deltas in order to compute the new value. In practice, we'll return the delta variation only, but we'll show you how to get rid of this last loop. In this case, we want to figure out how to change code like

```
all_results = []
for nset, histoset in enumerate(histogramssets):
    all_histos_deltas = allsets_all_histos_deltas[nset]
    set_result = []
    for nh, histo in enumerate(histoset):
        set_result.append([d + histoset[nh,1] for d in all_histos_deltas[nh]])
    all_results.append(set_result)
```

to get rid of that most-nested loop

```
all_results = []
for nset, histoset in enumerate(histogramssets):
    # look ma, no more loops inside!
```

So how does this look?

```
[25]: # slowly getting rid of our loops to build the right output tensor -- gotta think_
↳ about nominals
def new_interpolation_linear_step5(histogramssets, alphasets):
    allset_all_histo_deltas_up = histogramssets[:, :, 2] - histogramssets[:, :, 1]
    allset_all_histo_deltas_dn = histogramssets[:, :, 1] - histogramssets[:, :, 0]
    allset_all_histo_nom = histogramssets[:, :, 1]

    allsets_all_histos_alphas_times_deltas_up = np.einsum('sa, shb->shab', alphasets,
↳ allset_all_histo_deltas_up)
    allsets_all_histos_alphas_times_deltas_dn = np.einsum('sa, shb->shab', alphasets,
↳ allset_all_histo_deltas_dn)
    allsets_all_histos_masks = np.einsum('sa, s...u->s...au', alphasets > 0, np.ones_
↳ like(allset_all_histo_deltas_dn))

    allsets_all_histos_deltas = np.where(allsets_all_histos_masks, allsets_all_histos_
↳ alphas_times_deltas_up, allsets_all_histos_alphas_times_deltas_dn)

    all_results = []

    for nset, (_, alphaset) in enumerate(zip(histogramssets, alphasets)):
        all_histos_deltas = allsets_all_histos_deltas[nset]
        noms = histogramssets[nset, :, 1]

        all_histos_noms_repeated = np.einsum('a, hn->han', np.ones_like(alphaset), noms)

        set_result = all_histos_deltas + all_histos_noms_repeated
        all_results.append(set_result)
    return all_results
```

And does the calculation still match?

```
[26]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step5)
      print(result)
```

```
True
```

```
[27]: %%timeit
      interpolation_linear(h,a)
```

```
160 ms ± 8.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[28]: %%timeit
      new_interpolation_linear_step5(h,a)
```

```
1.57 ms ± 75.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Fantastic! And look at the speed up. We're already faster than the for-loop and we're not even done yet.

Step 6

The final frontier. Also probably the best Star Wars episode. In any case, we have one more for-loop that needs to die in a slab of carbonite. This should be much easier now that you're more comfortable with tensor broadcasting and einstein summations.

What does the function look like now?

```
[29]: def new_interpolation_linear_step6(histogramssets,alphasets):
      allset_allhisto_deltas_up = histogramssets[:, :, 2] - histogramssets[:, :, 1]
      allset_allhisto_deltas_dn = histogramssets[:, :, 1] - histogramssets[:, :, 0]
      allset_allhisto_nom = histogramssets[:, :, 1]

      #x is dummy index

      allsets_allhistos_alphas_times_deltas_up = np.einsum('sa,shb->shab',alphasets,
      ↪allset_allhisto_deltas_up)
      allsets_allhistos_alphas_times_deltas_dn = np.einsum('sa,shb->shab',alphasets,
      ↪allset_allhisto_deltas_dn)
      allsets_allhistos_masks = np.einsum('sa,sxu->sxau',np.where(alphasets > 0, np.
      ↪ones(alphasets.shape), np.zeros(alphasets.shape)),np.ones(allset_allhisto_deltas_dn.
      ↪shape))

      allsets_allhistos_deltas = np.where(allsets_allhistos_masks,allsets_allhistos_
      ↪alphas_times_deltas_up, allsets_allhistos_alphas_times_deltas_dn)
      allsets_allhistos_noms_repeated = np.einsum('sa,shb->shab',np.ones(alphasets.
      ↪shape),allset_allhisto_nom)
      set_results = allsets_allhistos_deltas + allsets_allhistos_noms_repeated
      return set_results
```

And does the calculation still match?

```
[30]: result, (h,a) = compare_fns(interpolation_linear, new_interpolation_linear_step6)
      print(result)
```

```
True
```

```
[31]: %%timeit
      interpolation_linear(h,a)
```



```
156 ms ± 6.29 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[32]: %%timeit
new_interpolation_linear_step6(h,a)

468 µs ± 37.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

And we're done tensorizing it. There are some more improvements that could be made to make this interpolation calculation even more robust – but for now we're done.

5.2.3 Tensorizing the Non-Linear Interpolator

This is very, very similar to what we've done for the case of the linear interpolator. As such, we will provide the resulting functions for each step, and you can see how things perform all the way at the bottom. Enjoy and learn at your own pace!

```
[33]: def interpolation_nonlinear(histogramssets,alphasets):
    all_results = []
    for histoset, alphaset in zip(histogramssets,alphasets):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphaset:
                alpha_result = []
                for down,nom,up in zip(histo[0],histo[1],histo[2]):
                    delta_up = up/nom
                    delta_down = down/nom
                    if alpha > 0:
                        delta = delta_up**alpha
                    else:
                        delta = delta_down**(-alpha)
                    v = nom*delta
                    alpha_result.append(v)
                histo_result.append(alpha_result)
    return all_results

def new_interpolation_nonlinear_step0(histogramssets,alphasets):
    all_results = []
    for histoset, alphaset in zip(histogramssets,alphasets):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphaset:
                alpha_result = []
                for down,nom,up in zip(histo[0],histo[1],histo[2]):
                    delta_up = up/nom
                    delta_down = down/nom
                    delta = np.where(alpha > 0, np.power(delta_up, alpha), np.
↪power(delta_down, np.abs(alpha)))
                    v = nom*delta
                    alpha_result.append(v)
                histo_result.append(alpha_result)
```

(continues on next page)

(continued from previous page)

```

    return all_results

def new_interpolation_nonlinear_step1(histogramssets, alphaset):
    all_results = []
    for histoset, alphaset in zip(histogramssets, alphaset):
        all_results.append([])
        set_result = all_results[-1]
        for histo in histoset:
            set_result.append([])
            histo_result = set_result[-1]
            for alpha in alphaset:
                alpha_result = []
                deltas_up = np.divide(histo[2], histo[1])
                deltas_down = np.divide(histo[0], histo[1])
                bases = np.where(alpha > 0, deltas_up, deltas_down)
                exponents = np.abs(alpha)
                calc_deltas = np.power(bases, exponents)
                v = histo[1] * calc_deltas
                alpha_result.append(v)
            histo_result.append(alpha_result)
        return all_results

def new_interpolation_nonlinear_step2(histogramssets, alphaset):
    all_results = []

    allset_all_histo_deltas_up = np.divide(histogramssets[:, :, 2], histogramssets[:, :,
↪1])
    allset_all_histo_deltas_dn = np.divide(histogramssets[:, :, 0], histogramssets[:, :,
↪1])

    for nset, (histoset, alphaset) in enumerate(zip(histogramssets, alphaset)):
        set_result = []

        all_histo_deltas_up = allset_all_histo_deltas_up[nset]
        all_histo_deltas_dn = allset_all_histo_deltas_dn[nset]

        for nh, histo in enumerate(histoset):
            alpha_deltas = []
            for alpha in alphaset:
                alpha_result = []
                deltas_up = all_histo_deltas_up[nh]
                deltas_down = all_histo_deltas_dn[nh]
                bases = np.where(alpha > 0, deltas_up, deltas_down)
                exponents = np.abs(alpha)
                calc_deltas = np.power(bases, exponents)
                alpha_deltas.append(calc_deltas)
            set_result.append([histo[1]*d for d in alpha_deltas])
        all_results.append(set_result)
    return all_results

def new_interpolation_nonlinear_step3(histogramssets, alphaset):
    all_results = []

    allset_all_histo_deltas_up = np.divide(histogramssets[:, :, 2], histogramssets[:, :,
↪1])
    allset_all_histo_deltas_dn = np.divide(histogramssets[:, :, 0], histogramssets[:, :,
↪1])

```

(continues on next page)

(continued from previous page)

```

for nset, (histoset, alphasets) in enumerate(zip(histogramssets, alphasets)):
    set_result = []

    all_histo_deltas_up = allset_all_histo_deltas_up[nset]
    all_histo_deltas_dn = allset_all_histo_deltas_dn[nset]

    for nh, histo in enumerate(histoset):
        # bases and exponents need to have an outer product, to essentially tile_
        ↪ or repeat over rows/cols
        bases_up = np.einsum('a,b->ab', np.ones(alphasets.shape), all_histo_deltas_
        ↪ up[nh])
        bases_dn = np.einsum('a,b->ab', np.ones(alphasets.shape), all_histo_deltas_
        ↪ dn[nh])
        exponents = np.einsum('a,b->ab', np.abs(alphasets), np.ones(all_histo_
        ↪ deltas_up[nh].shape))

        masks = np.einsum('a,b->ab', alphasets > 0, np.ones(all_histo_deltas_dn[nh].
        ↪ shape))
        bases = np.where(masks, bases_up, bases_dn)
        alpha_deltas = np.power(bases, exponents)
        set_result.append([histo[1]*d for d in alpha_deltas])

    all_results.append(set_result)
return all_results

def new_interpolation_nonlinear_step4(histogramssets, alphasets):
    all_results = []

    allset_all_histo_nom = histogramssets[:, :, 1]
    allset_all_histo_deltas_up = np.divide(histogramssets[:, :, 2], allset_all_histo_
    ↪ nom)
    allset_all_histo_deltas_dn = np.divide(histogramssets[:, :, 0], allset_all_histo_
    ↪ nom)

    bases_up = np.einsum('sa,shb->shab', np.ones(alphasets.shape), allset_all_histo_
    ↪ deltas_up)
    bases_dn = np.einsum('sa,shb->shab', np.ones(alphasets.shape), allset_all_histo_
    ↪ deltas_dn)
    exponents = np.einsum('sa,shb->shab', np.abs(alphasets), np.ones(allset_all_histo_
    ↪ deltas_up.shape))

    masks = np.einsum('sa,shb->shab', alphasets > 0, np.ones(allset_all_histo_deltas_up.
    ↪ shape))
    bases = np.where(masks, bases_up, bases_dn)

    allsets_all_histos_deltas = np.power(bases, exponents)

    all_results = []
    for nset, histoset in enumerate(histogramssets):
        all_histos_deltas = allsets_all_histos_deltas[nset]
        set_result = []
        for nh, histo in enumerate(histoset):
            set_result.append([histoset[nh, 1]*d for d in all_histos_deltas[nh]])
        all_results.append(set_result)
    return all_results

```

(continues on next page)

(continued from previous page)

```

def new_interpolation_nonlinear_step5(histogramssets, alphaset):
    all_results = []

    allset_all_histo_nom = histogramssets[:, :, 1]
    allset_all_histo_deltas_up = np.divide(histogramssets[:, :, 2], allset_all_histo_
↪nom)
    allset_all_histo_deltas_dn = np.divide(histogramssets[:, :, 0], allset_all_histo_
↪nom)

    bases_up = np.einsum('sa,shb->shab', np.ones(alphaset.shape), allset_all_histo_
↪deltas_up)
    bases_dn = np.einsum('sa,shb->shab', np.ones(alphaset.shape), allset_all_histo_
↪deltas_dn)
    exponents = np.einsum('sa,shb->shab', np.abs(alphaset), np.ones(allset_all_histo_
↪deltas_up.shape))

    masks = np.einsum('sa,shb->shab', alphaset > 0, np.ones(allset_all_histo_deltas_up.
↪shape))
    bases = np.where(masks, bases_up, bases_dn)

    allsets_all_histos_deltas = np.power(bases, exponents)

    all_results = []
    for nset, (_, alphasets) in enumerate(zip(histogramssets, alphaset)):
        all_histos_deltas = allsets_all_histos_deltas[nset]
        noms = allset_all_histo_nom[nset]
        all_histos_noms_repeated = np.einsum('a,hn->han', np.ones_like(alphasets), noms)
        set_result = all_histos_deltas * all_histos_noms_repeated
        all_results.append(set_result)
    return all_results

def new_interpolation_nonlinear_step6(histogramssets, alphaset):
    all_results = []

    allset_all_histo_nom = histogramssets[:, :, 1]
    allset_all_histo_deltas_up = np.divide(histogramssets[:, :, 2], allset_all_histo_
↪nom)
    allset_all_histo_deltas_dn = np.divide(histogramssets[:, :, 0], allset_all_histo_
↪nom)

    bases_up = np.einsum('sa,shb->shab', np.ones(alphaset.shape), allset_all_histo_
↪deltas_up)
    bases_dn = np.einsum('sa,shb->shab', np.ones(alphaset.shape), allset_all_histo_
↪deltas_dn)
    exponents = np.einsum('sa,shb->shab', np.abs(alphaset), np.ones(allset_all_histo_
↪deltas_up.shape))

    masks = np.einsum('sa,shb->shab', alphaset > 0, np.ones(allset_all_histo_deltas_up.
↪shape))
    bases = np.where(masks, bases_up, bases_dn)

    allsets_all_histos_deltas = np.power(bases, exponents)
    allsets_all_histos_noms_repeated = np.einsum('sa,shb->shab', np.ones(alphaset.
↪shape), allset_all_histo_nom)
    set_results = allsets_all_histos_deltas * allsets_all_histos_noms_repeated
    return set_results

```

```
[34]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step0)
print(result)

True
```

```
[35]: %%timeit
interpolation_nonlinear(h,a)

149 ms ± 9.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[36]: %%timeit
new_interpolation_nonlinear_step0(h,a)

527 ms ± 29.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[37]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step1)
print(result)

True
```

```
[38]: %%timeit
interpolation_nonlinear(h,a)

150 ms ± 5.21 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[39]: %%timeit
new_interpolation_nonlinear_step1(h,a)

456 ms ± 17.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[40]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step2)
print(result)

True
```

```
[41]: %%timeit
interpolation_nonlinear(h,a)

154 ms ± 4.49 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[42]: %%timeit
new_interpolation_nonlinear_step2(h,a)

412 ms ± 31 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[43]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step3)
print(result)

True
```

```
[44]: %%timeit
interpolation_nonlinear(h,a)
```

```
145 ms ± 5.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[45]: %%timeit
new_interpolation_nonlinear_step3(h,a)
```

```
1.28 s ± 74.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[46]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step4)
print(result)
```

```
True
```

```
[47]: %%timeit
interpolation_nonlinear(h,a)
```

```
147 ms ± 8.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[48]: %%timeit
new_interpolation_nonlinear_step4(h,a)
```

```
120 ms ± 3.06 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[49]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step5)
print(result)
```

```
True
```

```
[50]: %%timeit
interpolation_nonlinear(h,a)
```

```
151 ms ± 5.29 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[51]: %%timeit
new_interpolation_nonlinear_step5(h,a)
```

```
2.65 ms ± 57.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[52]: result, (h,a) = compare_fns(interpolation_nonlinear, new_interpolation_nonlinear_
↪step6)
print(result)
```

```
True
```

```
[53]: %%timeit
interpolation_nonlinear(h,a)
```

```
156 ms ± 3.35 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[54]: %%timeit
new_interpolation_nonlinear_step6(h,a)
```

```
1.49 ms ± 16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

5.3 Empirical Test Statistics

In this notebook we will compute test statistics empirically from pseudo-experiment and establish that they behave as assumed in the asymptotic approximation.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import pyhf

np.random.seed(0)
plt.rcParams.update({"font.size": 14})
```

First, we define the statistical model we will study.

- the signal expected event rate is 10 events
- the background expected event rate is 100 events
- a 10% uncertainty is assigned to the background

The expected event rates are chosen to lie comfortably in the asymptotic regime.

```
[2]: model = pyhf.simplemodels.hepdata_like(
    signal_data=[10.0], bkg_data=[100.0], bkg_uncerts=[10.0]
)
```

The test statistics based on the profile likelihood described in [arXiv:1007.1727](#) cover scenarios for both

- POI allowed to float to negative values (unbounded; $\mu \in [-10, 10]$)
- POI constrained to non-negative values (bounded; $\mu \in [0, 10]$)

For consistency, test statistics (t_μ, q_μ) associated with bounded POIs are usually denoted with a tilde ($\tilde{t}_\mu, \tilde{q}_\mu$).

We set up the bounds for the fit as follows

```
[3]: unbounded_bounds = model.config.suggested_bounds()
unbounded_bounds[model.config.poi_index] = (-10, 10)

bounded_bounds = model.config.suggested_bounds()
```

Next we draw some synthetic datasets (also referred to as “toys” or pseudo-experiments). We will “throw” 300 toys:

```
[4]: true_poi = 1.0
n_toys = 300
toys = model.make_pdf(pyhf.tensorlib.astensor([true_poi, 1.0])).sample((n_toys,))
```

In the asymptotic treatment the test statistics are described as a function of the data’s best-fit POI value $\hat{\mu}$.

So let’s run some fits so we can plot the empirical test statistics against $\hat{\mu}$ to observe the emergence of the asymptotic behavior.

```
[5]: pars = np.asarray(
    [pyhf.infer.mle.fit(toy, model, par_bounds=unbounded_bounds) for toy in toys]
)
```

We can now calculate all four test statistics described in [arXiv:1007.1727](#)

```
[6]: test_poi = 1.0
    tmu = np.asarray(
        [
            pyhf.infer.test_statistics.tmu(
                test_poi,
                toy,
                model,
                init_pars=model.config.suggested_init(),
                par_bounds=unbounded_bounds,
            )
            for toy in toys
        ]
    )
```

```
[7]: tmu_tilde = np.asarray(
    [
        pyhf.infer.test_statistics.tmu_tilde(
            test_poi,
            toy,
            model,
            init_pars=model.config.suggested_init(),
            par_bounds=bounded_bounds,
        )
        for toy in toys
    ]
)
```

```
[8]: qmu = np.asarray(
    [
        pyhf.infer.test_statistics.qmu(
            test_poi,
            toy,
            model,
            init_pars=model.config.suggested_init(),
            par_bounds=unbounded_bounds,
        )
        for toy in toys
    ]
)
```

```
[9]: qmu_tilde = np.asarray(
    [
        pyhf.infer.test_statistics.qmu_tilde(
            test_poi,
            toy,
            model,
            init_pars=model.config.suggested_init(),
            par_bounds=bounded_bounds,
        )
        for toy in toys
    ]
)
```

Let's plot all the test statistics we have computed

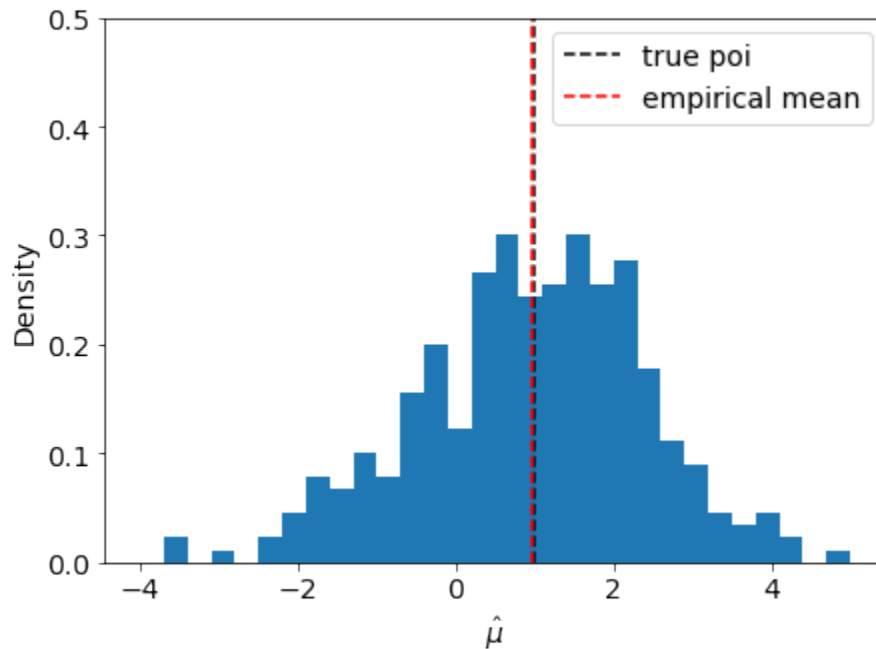
```
[10]: muhat = pars[:, model.config.poi_index]
    muhat_sigma = np.std(muhat)
```


We can check the asymptotic assumption that $\hat{\mu}$ is distributed normally around it's true value $\mu' = 1$

```
[11]: fig, ax = plt.subplots()
fig.set_size_inches(7, 5)

ax.set_xlabel("$\hat{\mu}$")
ax.set_ylabel("Density")
ax.set_ylim(top=0.5)

ax.hist(muhat, bins=np.linspace(-4, 5, 31), density=True)
ax.axvline(true_poi, label="true poi", color="black", linestyle="dashed")
ax.axvline(np.mean(muhat), label="empirical mean", color="red", linestyle="dashed")
ax.legend();
```



Here we define the asymptotic profile likelihood test statistics:

$$t_{\mu} = \frac{(\mu - \hat{\mu})^2}{\sigma^2}$$

$$\tilde{t}_{\mu} = \begin{cases} t_{\mu}, & \hat{\mu} > 0 \\ t_{\mu} - t_0, & \text{else} \end{cases}$$

$$q_{\mu} = \begin{cases} t_{\mu}, & \hat{\mu} < \mu \\ 0, & \text{else} \end{cases}$$

$$\tilde{q}_{\mu} = \begin{cases} \tilde{t}_{\mu}, & \hat{\mu} < \mu \\ 0, & \text{else} \end{cases}$$

```
[12]: def tmu_asymp(mutest, muhat, sigma):
    return (mutest - muhat) ** 2 / sigma ** 2
```

(continues on next page)

(continued from previous page)

```

def tmu_tilde_asymp(mutest, muhat, sigma):
    a = tmu_asymp(mutest, muhat, sigma)
    b = tmu_asymp(mutest, muhat, sigma) - tmu_asymp(0.0, muhat, sigma)
    return np.where(muhat > 0, a, b)

def qmu_asymp(mutest, muhat, sigma):
    return np.where(
        muhat < mutest, tmu_asymp(mutest, muhat, sigma), np.zeros_like(muhat)
    )

def qmu_tilde_asymp(mutest, muhat, sigma):
    return np.where(
        muhat < mutest, tmu_tilde_asymp(mutest, muhat, sigma), np.zeros_like(muhat)
    )

```

And now we can compare them to the empirical values:

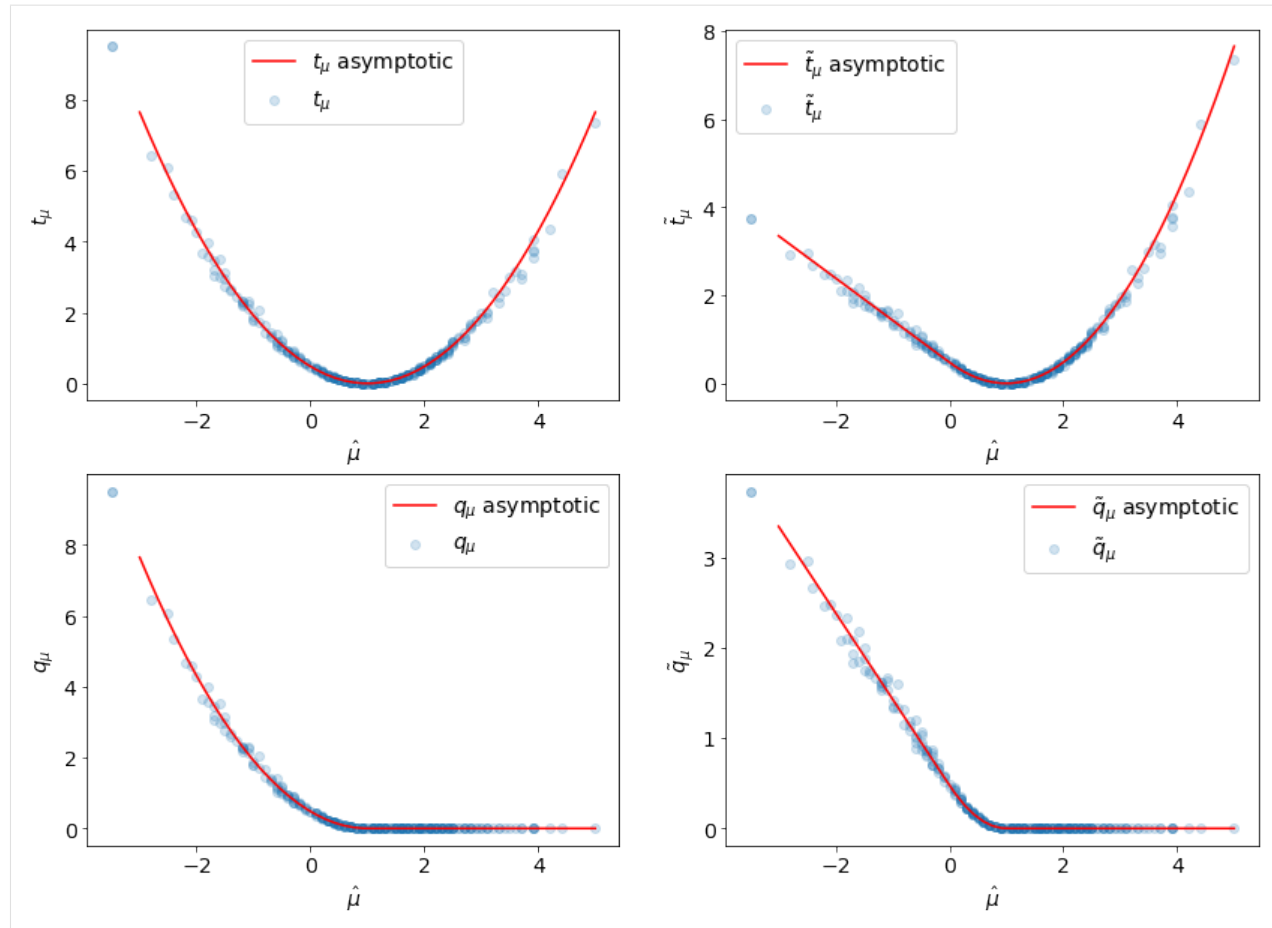
```

[13]: muhat_asymp = np.linspace(-3, 5)
fig, axarr = plt.subplots(2, 2)
fig.set_size_inches(14, 10)

labels = ["$t_{\mu}$", "$\tilde{t}_{\mu}$", "$q_{\mu}$", "$\tilde{q}_{\mu}$"]
data = [
    (tmu, tmu_asymp),
    (tmu_tilde, tmu_tilde_asymp),
    (qmu, qmu_asymp),
    (qmu_tilde, qmu_tilde_asymp),
]

for ax, (label, d) in zip(axarr.ravel(), zip(labels, data)):
    empirical, asymp_func = d
    ax.scatter(muhat, empirical, alpha=0.2, label=label)
    ax.plot(
        muhat_asymp,
        asymp_func(1.0, muhat_asymp, muhat_sigma),
        label=f"{label} asymptotic",
        c="r",
    )
    ax.set_xlabel(r"$\hat{\mu}$")
    ax.set_ylabel(f"{label}")
    ax.legend(loc="best")

```



EXAMPLES

Try out in Binder!

Notebooks:

6.1 ShapeFactor

```
[1]: %pylab inline

Populating the interactive namespace from numpy and matplotlib

[2]: import logging
import json

import pyhf
from pyhf import Model

logging.basicConfig(level = logging.INFO)

[3]: def prep_data(sourcedata):
    spec = {
        'channels': [
            {
                'name': 'signal',
                'samples': [
                    {
                        'name': 'signal',
                        'data': sourcedata['signal']['bindata']['sig'],
                        'modifiers': [
                            {
                                'name': 'mu',
                                'type': 'normfactor',
                                'data': None
                            }
                        ]
                    }
                ],
            },
            {
                'name': 'bkg1',
                'data': sourcedata['signal']['bindata']['bkg1'],
                'modifiers': [
                    {
                        'name': 'coupled_shapefactor',
```

(continues on next page)

(continued from previous page)

```

        'type': 'shapefactor',
        'data': None
    }
    ]
    },
    {
        'name': 'control',
        'samples': [
            {
                'name': 'background',
                'data': sourcedata['control']['bindata']['bkg1'],
                'modifiers': [
                    {
                        'name': 'coupled_shapefactor',
                        'type': 'shapefactor',
                        'data': None
                    }
                ]
            }
        ]
    }
]
}
pdf = Model(spec)
data = []
for c in pdf.spec['channels']:
    data += sourcedata[c['name']]['bindata']['data']
data = data + pdf.config.auxdata
return data, pdf

```

```

[4]: source = {
    "channels": {
        "signal": {
            "binning": [2,-0.5,1.5],
            "bindata": {
                "data": [220.0, 230.0],
                "bkg1": [100.0, 70.0],
                "sig": [ 20.0, 20.0]
            }
        },
        "control": {
            "binning": [2,-0.5,1.5],
            "bindata": {
                "data": [200.0, 300.0],
                "bkg1": [100.0, 100.0]
            }
        }
    }
}

data, pdf = prep_data(source['channels'])
print('data: {}'.format(data))

init_pars = pdf.config.suggested_init()

```

(continues on next page)

(continued from previous page)

```
print('expected data: {}'.format(pdf.expected_data(init_pars)))

par_bounds = pdf.config.suggested_bounds()

INFO:pyhf.pdf:Validating spec against schema: /home/jovyan/pyhf/src/pyhf/data/spec.
→json
INFO:pyhf.pdf:adding modifier mu (1 new nuisance parameters)
INFO:pyhf.pdf:adding modifier coupled_shapefactor (2 new nuisance parameters)

data: [220.0, 230.0, 200.0, 300.0]
expected data: [120.  90. 100. 100.]
```

```
[5]: print('initialization parameters: {}'.format(pdf.config.suggested_init()))

unconpars = pyhf.infer.mle.fit(data, pdf)
print('parameters post unconstrained fit: {}'.format(unconpars))

initialization parameters: [1.0, 1.0, 1.0]
parameters post unconstrained fit: [0.99981412 2.00002042 3.00006469]

/home/jovyan/pyhf/src/pyhf/tensor/numpy_backend.py:173: RuntimeWarning: divide by_
→zero encountered in log
    return n * np.log(lam) - lam - gammaln(n + 1.0)
```

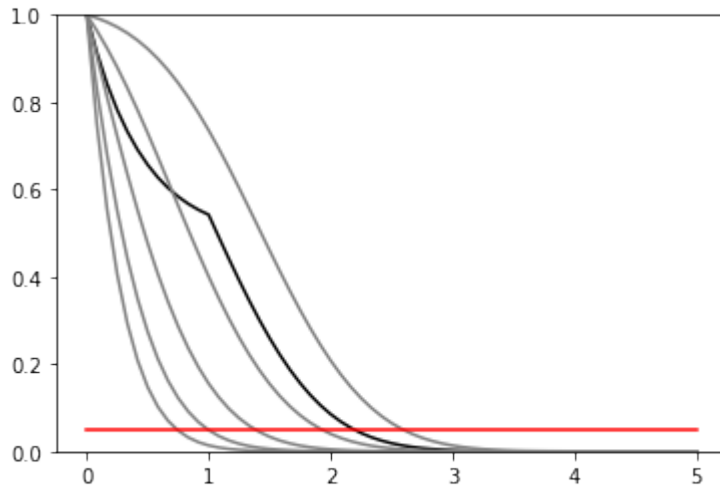
```
[6]: def plot_results(testmus, cls_obs, cls_exp, poi_tests, test_size = 0.05):
    plt.plot(poi_tests, cls_obs, c = 'k')
    for i, c in zip(range(5), ['grey', 'grey', 'grey', 'grey', 'grey']):
        plt.plot(poi_tests, cls_exp[i], c = c)
    plt.plot(testmus, [test_size]*len(testmus), c = 'r')
    plt.ylim(0,1)

def invert_interval(test_mus, cls_obs, cls_exp, test_size=0.05):
    crossing_test_stats = {'exp': [], 'obs': None}
    for cls_exp_sigma in cls_exp:
        crossing_test_stats['exp'].append(
            np.interp(
                test_size, list(reversed(cls_exp_sigma)), list(reversed(test_mus))
            )
        )
    crossing_test_stats['obs'] = np.interp(
        test_size, list(reversed(cls_obs)), list(reversed(test_mus))
    )
    return crossing_test_stats

poi_tests = np.linspace(0, 5, 61)
tests = [pyhf.infer.hypotest(poi_test, data, pdf, init_pars, par_bounds, return_
→expected_set=True)
        for poi_test in poi_tests]
cls_obs = np.array([test[0] for test in tests]).flatten()
cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]

print('\n')
plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
invert_interval(poi_tests, cls_obs, cls_exp)
```

```
[6]: {'exp': [0.741381412468345,  
            0.9949353526744877,  
            1.3845144105754894,  
            1.9289946435921614,  
            2.594077794516857],  
      'obs': 2.1945970333027187}
```



6.2 XML Import/Export

```
[1]: # NB: python -m pip install pyhf[xmlio]  
import pyhf
```

```
[2]: !ls -lavh ../../validation/xmlimport_input
```

```
total 1752  
drwxr-xr-x  7 kratsg  staff   238B Oct 16 22:20 .  
drwxr-xr-x 21 kratsg  staff   714B Apr  4 14:26 ..  
drwxr-xr-x  6 kratsg  staff   204B Feb 27 17:13 config  
drwxr-xr-x  7 kratsg  staff   238B Feb 27 23:41 data  
-rw-r--r--  1 kratsg  staff  850K Oct 16 22:20 log  
drwxr-xr-x 17 kratsg  staff   578B Nov 15 12:24 results  
-rw-r--r--  1 kratsg  staff   21K Oct 16 22:20 scan.pdf
```

6.2.1 Importing

In order to convert HistFactory XML+ROOT to the pyhf JSON spec for likelihoods, you need to point the command-line interface `pyhf xml2json` at the top-level XML file. Additionally, as the HistFactory XML specification often uses relative paths, you might need to specify the base directory `--basedir` from which all other files are located, as specified in the top-level XML. The command will be of the format

```
pyhf xml2json {top-level XML} --basedir {base directory}
```

This will print the JSON representation of the XML+ROOT specified. If you wish to store this as a JSON file, you simply need to redirect it


```
pyhf xml2json {top-level XML} --basedir {base directory} > spec.json
```

```
[3]: !pyhf xml2json --hide-progress ../../../../validation/xmlimport_input/config/example.xml_
    ↪--basedir ../../../../validation/xmlimport_input | tee xml_importexport.json
```

```
{
  "channels": [
    {
      "name": "channel1",
      "samples": [
        {
          "data": [
            20.0,
            10.0
          ],
          "modifiers": [
            {
              "data": {
                "hi": 1.05,
                "lo": 0.95
              },
              "name": "syst1",
              "type": "normsys"
            },
            {
              "data": null,
              "name": "SigXsecOverSM",
              "type": "normfactor"
            }
          ],
          "name": "signal"
        },
        {
          "data": [
            100.0,
            0.0
          ],
          "modifiers": [
            {
              "data": null,
              "name": "lumi",
              "type": "lumi"
            },
            {
              "data": [
                5.000000074505806,
                0.0
              ],
              "name": "staterror_channel1",
              "type": "staterror"
            },
            {
              "data": {
                "hi": 1.05,
                "lo": 0.95
              },
              "name": "syst2",
```

(continues on next page)

(continued from previous page)

```

        "type": "normsys"
    },
    ],
    "name": "background1"
},
{
    "data": [
        0.0,
        100.0
    ],
    "modifiers": [
        {
            "data": null,
            "name": "lumi",
            "type": "lumi"
        },
        {
            "data": [
                0.0,
                10.0
            ],
            "name": "staterror_channel1",
            "type": "staterror"
        },
        {
            "data": {
                "hi": 1.05,
                "lo": 0.95
            },
            "name": "syst3",
            "type": "normsys"
        }
    ],
    "name": "background2"
}
]
}
],
"data": {
    "channel1": [
        122.0,
        112.0
    ]
},
"toplvl": {
    "measurements": [
        {
            "config": {
                "parameters": [
                    {
                        "auxdata": [
                            1.0
                        ],
                        "bounds": [
                            [
                                0.5,
                                1.5

```

(continues on next page)

(continued from previous page)

```

        ],
        "fixed": true,
        "inits": [
            1.0
        ],
        "name": "lumi",
        "sigmas": [
            0.1
        ]
    },
    {
        "fixed": true,
        "name": "alpha_syst1"
    }
],
"poi": "SigXsecOverSM"
},
"name": "GaussExample"
},
{
    "config": {
        "parameters": [
            {
                "auxdata": [
                    1.0
                ],
                "bounds": [
                    [
                        0.5,
                        1.5
                    ]
                ],
                "fixed": true,
                "inits": [
                    1.0
                ],
                "name": "lumi",
                "sigmas": [
                    0.1
                ]
            },
            {
                "fixed": true,
                "name": "alpha_syst1"
            }
        ],
        "poi": "SigXsecOverSM"
    },
    "name": "GammaExample"
},
{
    "config": {
        "parameters": [
            {
                "auxdata": [
                    1.0

```

(continues on next page)

(continued from previous page)

```

        ],
        "bounds": [
            [
                0.5,
                1.5
            ]
        ],
        "fixed": true,
        "inits": [
            1.0
        ],
        "name": "lumi",
        "sigmas": [
            0.1
        ]
    },
    {
        "fixed": true,
        "name": "alpha_syst1"
    }
],
"poi": "SigXsecOverSM"
},
"name": "LogNormExample"
},
{
    "config": {
        "parameters": [
            {
                "auxdata": [
                    1.0
                ],
                "bounds": [
                    [
                        0.5,
                        1.5
                    ]
                ],
                "fixed": true,
                "inits": [
                    1.0
                ],
                "name": "lumi",
                "sigmas": [
                    0.1
                ]
            },
            {
                "fixed": true,
                "name": "alpha_syst1"
            }
        ],
        "poi": "SigXsecOverSM"
    },
    "name": "ConstExample"
}
],

```

(continues on next page)

(continued from previous page)

```

    "resultprefix": "./results/example"
}
}

```

6.2.2 Exporting

In order to convert the pyhf JSON to the HistFactory XML+ROOT spec for likelihoods, you need to point the command-line interface `pyhf json2xml` at the JSON file you want to convert. As everything is specified in a single file, there is no need to deal with base directories or looking up additional files. This will produce output XML+ROOT in the `--output-dir=.` directory (your current working directory), storing XML configs under `--specroot=config` and the data file under `--dataroot=data`. The XML configs are prefixed with `--resultprefix=FitConfig` by default, so that the top-level XML file will be located at `{output_dir}/{prefix}.xml`. The command will be of the format

```
pyhf json2xml {JSON spec}
```

Note that the output directory must already exist.

```
[4]: !mkdir -p output
!pyhf json2xml xml_importexport.json --output-dir output
!ls -lavh output/*

/Users/jovyan/pyhf/src/pyhf/writexml.py:120: RuntimeWarning: invalid value_
↪encountered in true_divide
  attrs['HistoName'], np.divide(modifierspec['data'], sampledata).tolist()
-rw-r--r--  1 kratsg  staff   822B Apr  9 09:36 output/FitConfig.xml

output/config:
total 8
drwxr-xr-x  3 kratsg  staff   102B Apr  9 09:36 .
drwxr-xr-x  5 kratsg  staff   170B Apr  9 09:36 ..
-rw-r--r--  1 kratsg  staff   1.0K Apr  9 09:36 FitConfig_channel1.xml

output/data:
total 96
drwxr-xr-x  3 kratsg  staff   102B Apr  9 09:36 .
drwxr-xr-x  5 kratsg  staff   170B Apr  9 09:36 ..
-rw-r--r--  1 kratsg  staff   46K Apr  9 09:36 data.root

```

```
[5]: !rm xml_importexport.json
!rm -rf output/
```

```
[1]: import pyhf
import pandas
import numpy as np
import altair as alt

```

6.3 Visualization with Altair

Altair is a python API for generating Vega visualization specifications. We demonstrate how to use this to build an interactive chart of pyhf results.

6.3.1 Preparing the data

Altair reads the data as a pandas dataframe, so we create one.

```
[2]: model = pyhf.simplemodels.hepdata_like([7], [20], [5])
    data = [25] + model.config.auxdata

[3]: muscan = np.linspace(0, 5, 31)
    results = [
        pyhf.infer.hypotest(mu, data, model, return_expected_set=True) for mu in muscan
    ]

[4]: data = np.concatenate(
    [
        muscan.reshape(-1, 1),
        np.asarray([r[0] for r in results]).reshape(-1, 1),
        np.asarray([r[1] for r in results]).reshape(-1, 5),
    ],
    axis=1,
)
df = pandas.DataFrame(data, columns=["mu", "obs"] + [f"exp_{i}" for i in range(5)])
df.head()
```

	mu	obs	exp_0	exp_1	exp_2	exp_3	exp_4
0	0.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1	0.166667	0.885208	0.670809	0.771258	0.870322	0.949235	0.989385
2	0.333333	0.795986	0.438838	0.581516	0.743696	0.890881	0.975022
3	0.500000	0.726450	0.279981	0.428500	0.623443	0.825621	0.956105
4	0.666667	0.672216	0.174235	0.308524	0.512383	0.754629	0.931866

6.3.2 Defining the Chart

We need to filled areas for the 1,2 sigma bands and two lines for the expected and observed CLs value. For interactivity we add a hovering label of the observed result

```
[5]: band1 = (
    alt.Chart(df)
    .mark_area(opacity=0.5, color="green")
    .encode(x="mu", y="exp_1", y2="exp_3")
)

band2 = (
    alt.Chart(df)
    .mark_area(opacity=0.5, color="yellow")
    .encode(x="mu", y="exp_0", y2="exp_4")
)

line1 = alt.Chart(df).mark_line(color="black").encode(x="mu", y="obs")
```

(continues on next page)

(continued from previous page)

```

line2 = (
    alt.Chart(df).mark_line(color="black", strokeDash=[5, 5]).encode(x="mu", y="exp_2
↪")
)

nearest = alt.selection_single(
    nearest=True, on="mouseover", fields=["mu"], empty="none"
)

point = (
    alt.Chart(df)
    .mark_point(color="black")
    .encode(x="mu", y="obs", opacity=alt.condition(nearest, alt.value(1), alt.
↪value(0)))
    .add_selection(nearest)
)

text = line1.mark_text(aligned="left", dx=5, dy=-5).encode(
    text=alt.condition(nearest, "obs", alt.value(" "))
)

band2 + band1 + line1 + line2 + point + text

[5]: alt.LayerChart(...)

```

6.4 Hello World, pyhf style

Two bin counting experiment with a background uncertainty

```
[1]: import pyhf
```

Returning the observed and expected CL_s

```

[2]: model = pyhf.simplemodels.hepdata_like(
    signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
)
data = [51, 48] + model.config.auxdata
test_mu = 1.0
CLs_obs, CLs_exp = pyhf.infer.hypotest(
    test_mu, data, model, qtilde=True, return_expected=True
)
print(f"Observed: {CLs_obs}, Expected: {CLs_exp}")

Observed: 0.052515541856109765, Expected: 0.06445521290832805

```

Returning the observed CL_s , CL_{s+b} , and CL_b

```

[3]: CLs_obs, p_values = pyhf.infer.hypotest(
    test_mu, data, model, qtilde=True, return_tail_probs=True
)
print(f"Observed CL_s: {CLs_obs}, CL_sb: {p_values[0]}, CL_b: {p_values[1]}")

Observed CL_s: 0.052515541856109765, CL_sb: 0.023324961200974572, CL_b: 0.
↪44415349012077077

```

A reminder that

$$CL_s = \frac{CL_{s+b}}{CL_b} = \frac{p_{s+b}}{1 - p_b}$$

```
[4]: assert CLs_obs == p_values[0] / p_values[1]
```

Returning the expected CL_s band values

```
[5]: import numpy as np
```

```
[6]: CLs_obs, CLs_exp_band = pyhf.infer.hypotest(
    test_mu, data, model, qtilde=True, return_expected_set=True
)
print(f"Observed CL_s: {CLs_obs}\n")
for p_value, n_sigma in enumerate(np.arange(-2, 3)):
    print(
        "Expected CL_s{:}: {}".format(
            " " " if n_sigma == 0 else "({}) ".format(n_sigma), CLs_exp_band[p_
↪value],
        )
    )
```

```
Observed CL_s: 0.052515541856109765
```

```
Expected CL_s(-2 ): 0.0026064088679947964
```

```
Expected CL_s(-1 ): 0.013820657528619273
```

```
Expected CL_s      : 0.06445521290832805
```

```
Expected CL_s(1 ): 0.23526103626937836
```

```
Expected CL_s(2 ): 0.5730418174887743
```

6.5 Multi-bin Poisson

```
[1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
[2]: import logging
import json

import pyhf
from pyhf import Model, optimizer
from pyhf.simplemodels import hepdata_like

from scipy.interpolate import griddata
import scrapbook as sb
```

```
[3]: def plot_results(testmus, cls_obs, cls_exp, poi_tests, test_size=0.05):
    plt.plot(poi_tests, cls_obs, c="k")
    for idx, color in zip(range(5), 5 * ["grey"]):
        plt.plot(poi_tests, cls_exp[idx], c=color)
    plt.plot(testmus, [test_size] * len(testmus), c="r")
    plt.ylim(0, 1)
```

(continues on next page)

(continued from previous page)

```

def invert_interval(test_mus, cls_obs, cls_exp, test_size=0.05):
    crossing_test_stats = {'exp': [], 'obs': None}
    for cls_exp_sigma in cls_exp:
        crossing_test_stats['exp'].append(
            np.interp(
                test_size, list(reversed(cls_exp_sigma)), list(reversed(test_mus))
            )
        )
    crossing_test_stats['obs'] = np.interp(
        test_size, list(reversed(cls_obs)), list(reversed(test_mus))
    )
    return crossing_test_stats

def plot_histo(ax, binning, data):
    bin_width = (binning[2]-binning[1])/binning[0]
    bin_leftedges = np.linspace(binning[1],binning[2],binning[0]+1)[: -1]
    bin_centers = [le + bin_width/2. for le in bin_leftedges]
    ax.bar(bin_centers,data,1, alpha=0.5)

def plot_data(ax, binning, data):
    errors = [math.sqrt(d) for d in data]
    bin_width = (binning[2]-binning[1])/binning[0]
    bin_leftedges = np.linspace(binning[1],binning[2],binning[0]+1)[: -1]
    bin_centers = [le + bin_width/2. for le in bin_leftedges]
    ax.bar(bin_centers,data,0, yerr=errors, linewidth=0, error_kw = dict(ecolor='k',
↪ elinewidth = 1))
    ax.scatter(bin_centers, data, c = 'k')

```

```
[4]: validation_datadir = '../..validation/data'
```

```

[5]: source = json.load(open(validation_datadir + '/1bin_example1.json'))
pdf = hepdata_like(source['bindata']['sig'], source['bindata']['bkg'], source['bindata']
↪ ['bkgerr'])
data = source['bindata']['data'] + pdf.config.auxdata

init_pars = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

poi_tests = np.linspace(0, 5, 41)
tests = [pyhf.infer.hypotest(poi_test, data, pdf, init_pars, par_bounds, return_
↪ expected_set=True)
         for poi_test in poi_tests]
cls_obs = np.array([test[0] for test in tests]).flatten()
cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]

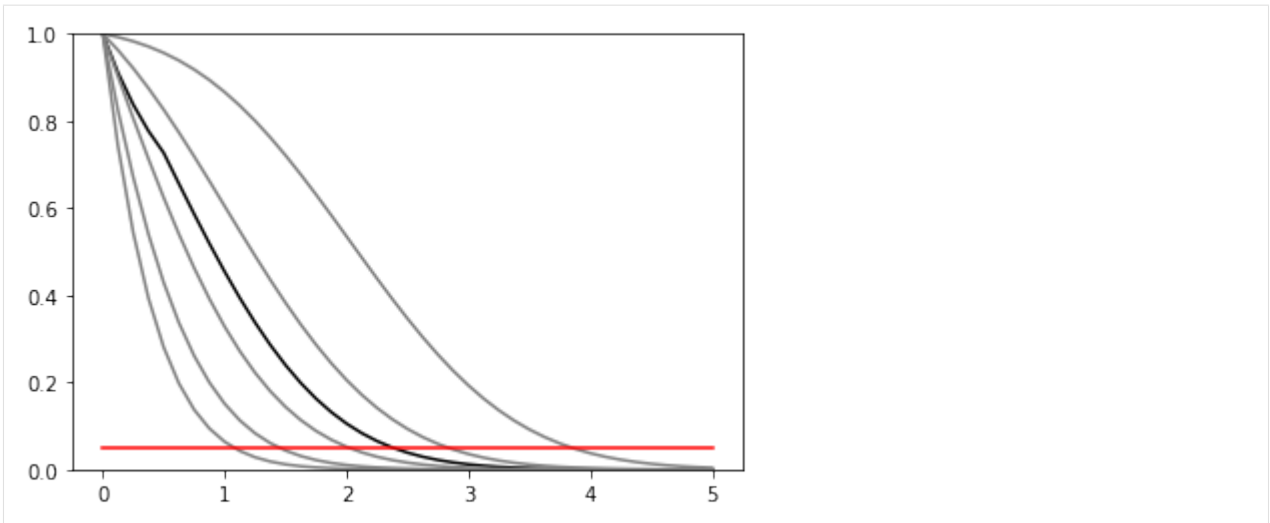
plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
invert_interval(poi_tests, cls_obs, cls_exp)

```

```

[5]: {'exp': [1.081060678053728,
1.4517179965651272,
2.0200754881420275,
2.834863384648174,
3.848756749431549],
'obs': 2.3800254370628036}

```



```
[6]: source = {
    "binning": [2, -0.5, 1.5],
    "bindata": {
        "data": [120.0, 145.0],
        "bkg": [100.0, 150.0],
        "bkgerr": [15.0, 20.0],
        "sig": [30.0, 45.0]
    }
}

my_observed_counts = source['bindata']['data']

pdf = hepdata_like(source['bindata']['sig'], source['bindata']['bkg'], source['bindata']
    ↪ ['bkgerr'])
data = my_observed_counts + pdf.config.auxdata

binning = source['binning']

nompars = pdf.config.suggested_init()

bonlypars = [x for x in nompars]
bonlypars[pdf.config.poi_index] = 0.0
nom_bonly = pdf.expected_data(bonlypars, include_auxdata = False)

nom_sb = pdf.expected_data(nompars, include_auxdata = False)

init_pars = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

print(init_pars)

bestfit_pars = pyhf.infer.mle.fit(data, pdf, init_pars, par_bounds)
bestfit_cts = pdf.expected_data(bestfit_pars, include_auxdata = False)

[1.0, 1.0, 1.0]
```

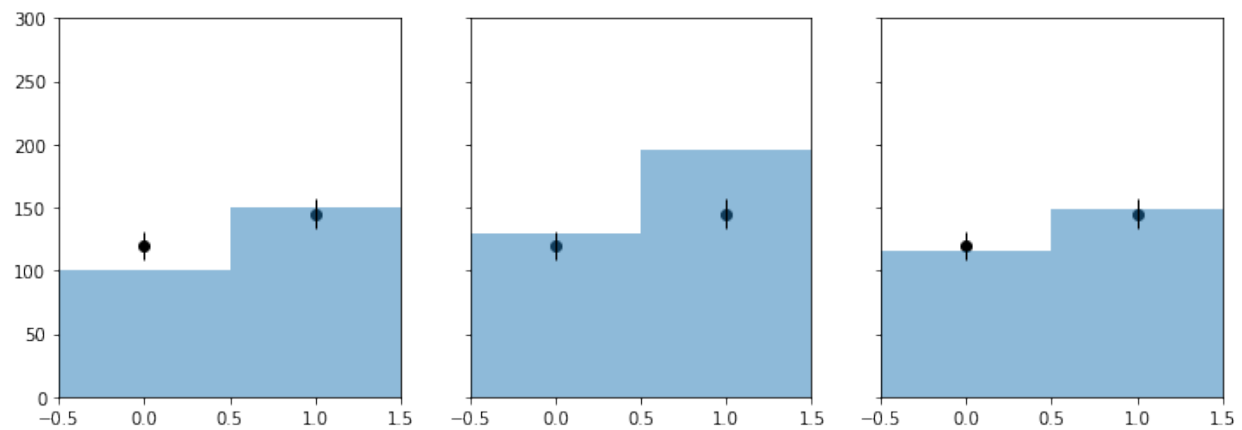
```
[7]: f, axarr = plt.subplots(1,3,sharey=True)
f.set_size_inches(12,4)

plot_histo(axarr[0], binning, nom_bonly)
plot_data(axarr[0], binning, my_observed_counts)
axarr[0].set_xlim(binning[1:])

plot_histo(axarr[1], binning, nom_sb)
plot_data(axarr[1], binning, my_observed_counts)
axarr[1].set_xlim(binning[1:])

plot_histo(axarr[2], binning, bestfit_cts)
plot_data(axarr[2], binning, my_observed_counts)
axarr[2].set_xlim(binning[1:])

plt.ylim(0,300);
```



```
[8]: ##
## DUMMY 2D thing
##

def signal(m1, m2):
    massscale = 150.
    minmass = 100.
    countscale = 2000

    effective_mass = np.sqrt(m1**2 + m2**2)
    return [countscale*np.exp(-(effective_mass-minmass)/massscale), 0]

def CLs(m1,m2):
    signal_counts = signal(m1, m2)
    pdf = hepdata_like(signal_counts, source['bindata']['bkg'], source['bindata']['
    ↪ 'bkger'])
    try:
        cls_obs, cls_exp_set = pyhf.infer.hypotest(1.0, data, pdf, init_pars, par_
    ↪ bounds, return_expected_set=True)
        return cls_obs, cls_exp_set, True
    except AssertionError:
        print('fit failed for mass points ({}, {})'.format(m1, m2))
        return None, None, False
```

```
[9]: nx, ny = 15, 15
    grid = grid_x, grid_y = np.mgrid[100:1000:complex(0, nx), 100:1000:complex(0, ny)]
    X = grid.T.reshape(nx * ny, 2)
    results = [CLs(m1, m2) for m1, m2 in X]

[10]: X = np.array([x for x, (_,_, success) in zip(X, results) if success])
    yobs = np.array([obs for obs, exp, success in results if success]).flatten()
    yexp = [np.array([exp[i] for obs, exp, success in results if success]).flatten() for
    ↪ i in range(5)]

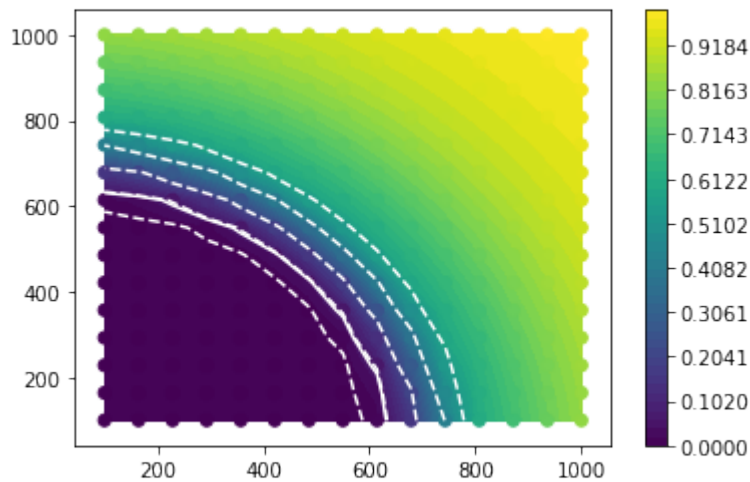
[11]: int_obs = griddata(X, yobs, (grid_x, grid_y), method='linear')

    int_exp = [griddata(X, yexp[i], (grid_x, grid_y), method='linear') for i in range(5)]

    plt.contourf(grid_x, grid_y, int_obs, levels = np.linspace(0,1))
    plt.colorbar()

    plt.contour(grid_x, grid_y, int_obs, levels = [0.05], colors = 'w')
    for level in int_exp:
        plt.contour(grid_x, grid_y, level, levels = [0.05], colors = 'w', linestyle =
        ↪ 'dashed')

    plt.scatter(X[:,0], X[:,1], c = yobs, vmin = 0, vmax = 1);
```



```
[12]: sb.glue("number_2d_successpoints", len(X))
```

Data type cannot be displayed: application/scrapbook.scrap.json+json

6.6 Multibin Coupled HistoSys

```
[1]: %pylab inline
Populating the interactive namespace from numpy and matplotlib

[2]: import logging
import json

import pyhf
from pyhf import Model

logging.basicConfig(level=logging.INFO)

[3]: def prep_data(sourcedata):
    spec = {
        "channels": [
            {
                "name": "signal",
                "samples": [
                    {
                        "name": "signal",
                        "data": sourcedata["signal"]["bindata"]["sig"],
                        "modifiers": [
                            {"name": "mu", "type": "normfactor", "data": None}
                        ],
                    },
                    {
                        "name": "bkg1",
                        "data": sourcedata["signal"]["bindata"]["bkg1"],
                        "modifiers": [
                            {
                                "name": "coupled_histosys",
                                "type": "histosys",
                                "data": {
                                    "lo_data": sourcedata["signal"]["bindata"]["bkg1_
↪dn"],
                                    "hi_data": sourcedata["signal"]["bindata"]["bkg1_
↪up"],
                                },
                            }
                        ],
                    }
                ],
            },
            {
                "name": "bkg2",
                "data": sourcedata["signal"]["bindata"]["bkg2"],
                "modifiers": [
                    {
                        "name": "coupled_histosys",
                        "type": "histosys",
                        "data": {
                            "lo_data": sourcedata["signal"]["bindata"]["bkg2_
↪dn"],
                            "hi_data": sourcedata["signal"]["bindata"]["bkg2_
↪up"],
                        },
                    }
                ],
            }
        ]
    }
```

(continues on next page)

(continued from previous page)

```

        ],
        },
    ],
    },
    {
        "name": "control",
        "samples": [
            {
                "name": "background",
                "data": sourcedata["control"]["bindata"]["bkg1"],
                "modifiers": [
                    {
                        "name": "coupled_histosys",
                        "type": "histosys",
                        "data": {
                            "lo_data": sourcedata["control"]["bindata"]["bkg1_
↪dn"],
                            "hi_data": sourcedata["control"]["bindata"]["bkg1_
↪up"],
                        },
                    },
                ],
            },
        ],
    },
]
}
pdf = Model(spec)
data = []
for c in pdf.spec["channels"]:
    data += sourcedata[c["name"]]["bindata"]["data"]
data = data + pdf.config.auxdata
return data, pdf

```

```
[4]: validation_datadir = "../validation/data"
```

```
[5]: source = json.load(open(validation_datadir + "/2bin_2channel_coupledhisto.json"))
```

```

data, pdf = prep_data(source["channels"])

print(data)

init_pars = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

unconpars = pyhf.infer.mle.fit(data, pdf, init_pars, par_bounds)
print("parameters post unconstrained fit: {}".format(unconpars))

conpars = pyhf.infer.mle.fixed_poi_fit(0.0, data, pdf, init_pars, par_bounds)
print("parameters post constrained fit: {}".format(conpars))

pdf.expected_data(conpars)

[170.0, 220.0, 110.0, 105.0, 0.0]
parameters post unconstrained fit: [1.53170588e-12 2.21657891e+00]
parameters post constrained fit: [0.          2.21655133]

```

```
[5]: array([116.08275666, 133.24826999, 183.24826999, 98.08967672,
          2.21655133])
```

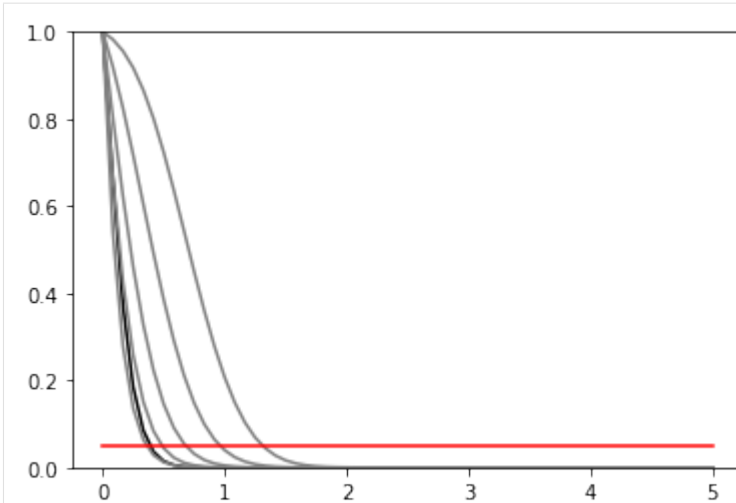
```
[6]: def plot_results(test_mus, cls_obs, cls_exp, poi_tests, test_size=0.05):
    plt.plot(poi_tests, cls_obs, c="k")
    for i, c in zip(range(5), ["grey", "grey", "grey", "grey", "grey"]):
        plt.plot(poi_tests, cls_exp[i], c=c)
    plt.plot(poi_tests, [test_size] * len(test_mus), c="r")
    plt.ylim(0, 1)
```

```
[7]: def invert_interval(test_mus, cls_obs, cls_exp, test_size=0.05):
    crossing_test_stats = {"exp": [], "obs": None}
    for cls_exp_sigma in cls_exp:
        crossing_test_stats["exp"].append(
            np.interp(
                test_size, list(reversed(cls_exp_sigma)), list(reversed(test_mus))
            )
        )
    crossing_test_stats["obs"] = np.interp(
        test_size, list(reversed(cls_obs)), list(reversed(test_mus))
    )
    return crossing_test_stats
```

```
[8]: poi_tests = np.linspace(0, 5, 61)
    tests = [
        pyhf.infer.hypotest(
            poi_test, data, pdf, init_pars, par_bounds, return_expected_set=True
        )
        for poi_test in poi_tests
    ]
    cls_obs = np.array([test[0] for test in tests]).flatten()
    cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]
```

```
[9]: print("\n")
    plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
    invert_interval(poi_tests, cls_obs, cls_exp)
```

```
[9]: {'exp': [0.3654675198094938,
            0.4882076670368835,
            0.683262284467055,
            0.9650584704888153,
            1.3142329292131938],
      'obs': 0.3932476110375604}
```



```
[1]: %pylab inline
      Populating the interactive namespace from numpy and matplotlib

[2]: from pathlib import Path
      import pyhf
      import pyhf.readxml
      from ipywidgets import interact, fixed
```

6.7 Binned HEP Statistical Analysis in Python

6.7.1 HistFactory

HistFactory is a popular framework to analyze binned event data and commonly used in High Energy Physics. At its core it is a template for building a statistical model from individual binned distribution (‘Histograms’) and variations on them (‘Systematics’) that represent auxiliary measurements (for example an energy scale of the detector which affects the shape of a distribution)

6.7.2 pyhf

pyhf is a work-in-progress standalone implementation of the HistFactory p.d.f. template and an implementation of the test statistics and asymptotic formulae described in the paper by Cowan, Cranmer, Gross, Vitells: *Asymptotic formulae for likelihood-based tests of new physics* [[arXiv:1007.1727](https://arxiv.org/abs/1007.1727)].

Models can be defined using JSON specification, but existing models based on the XML + ROOT file scheme are readable as well.

6.7.3 The Demo

The input data for the statistical analysis was built generated using the containerized workflow engine [yadage](#) (see demo from KubeCon 2018 [\[youtube\]](#)). Similarly to Binder this utilizes modern container technology for reproducible science. Below you see the execution graph leading up to the model input data at the bottom.

```
[3]: import base64
      from IPython.core.display import display, HTML
      anim = base64.b64encode(open('workflow.gif', 'rb').read()).decode('ascii')
      HTML(''.format(anim))

[3]: <IPython.core.display.HTML object>
```

6.7.4 Read in the Model from XML and ROOT

The ROOT files are read using scikit-hep’s [uproot](#) module.

```
[4]: spec = pyhf.readxml.parse('meas.xml', Path.cwd())
      workspace = pyhf.Workspace(spec)
```

From the `meas.xml` spec, we construct a probability density function (p.d.f). As the model includes systematics, it will be a simultaneous joint p.d.f. of the main model (poisson) and constraint model. The latter is defined by the implied “auxiliary measurements”.

```
[5]: pdf = workspace.model(measurement_name='meas')
      data = workspace.data(pdf)
      # what is the measurement?
      workspace.get_measurement(measurement_name='meas')

[5]: {'name': 'meas',
      'config': {'poi': 'SigXsecOverSM',
                  'parameters': [{'name': 'lumi',
                                   'auxdata': [1.0],
                                   'bounds': [[0.5, 1.5]],
                                   'inits': [1.0],
                                   'sigmas': [0.1]},
                                   {'name': 'SigXsecOverSM',
                                    'bounds': [[0.0, 3.0]],
                                    'inits': [1.0],
                                    'fixed': False}]]}}
```

The p.d.f is build from one data-driven “qcd” (or multijet) estimate and two Monte Carlo-based background samples and is parametrized by five parameters: One parameter of interest `SigXsecOverSM` and four *nuisance parameters* that affect the shape of the two Monte Carlo background estimates (both weight-only and shape systematics)

```
[6]: print('Samples:\n {}'.format(workspace.samples))
      print('Parameters:\n {}'.format(workspace.parameters))

Samples:
 ['mc1', 'mc2', 'qcd', 'signal']
Parameters:
 ['SigXsecOverSM', 'lumi', 'mc1_shape_conv', 'mc1_weight_var1', 'mc2_shape_conv',
  ↪ 'mc2_weight_var1']
```

```
[7]: par_name_dict = {k: v["slice"].start for k, v in pdf.config.par_map.items()}
      all_par_settings = {
```

(continues on next page)

(continued from previous page)

```

n[0]: tuple(m)
for n, m in zip(
    sorted(reversed(list(par_name_dict.items()))), key=lambda x: x[1]),
    pdf.config.suggested_bounds(),
)
}
default_par_settings = {n[0]: sum(tuple(m)) / 2.0 for n, m in all_par_settings.
    ↪items()}

def get_mc_counts(pars):
    deltas, factors = pdf._modifications(pars)
    allsum = pyhf.tensorlib.concatenate(
        deltas + [pyhf.tensorlib.astensor(pdf.nominal_rates)]
    )
    nom_plus_delta = pyhf.tensorlib.sum(allsum, axis=0)
    nom_plus_delta = pyhf.tensorlib.reshape(
        nom_plus_delta, (1,) + pyhf.tensorlib.shape(nom_plus_delta)
    )
    allfac = pyhf.tensorlib.concatenate(factors + [nom_plus_delta])
    return pyhf.tensorlib.product(allfac, axis=0)

animate_plot_pieces = None

def init_plot(fig, ax, par_settings):
    global animate_plot_pieces

    nbins = sum(list(pdf.config.channel_nbins.values()))
    x = np.arange(nbins)
    data = np.zeros(nbins)
    items = []
    for i in [3, 2, 1, 0]:
        items.append(ax.bar(x, data, 1, alpha=1.0))
    animate_plot_pieces = (
        items,
        ax.scatter(x, workspace.data(pdf, with_aux=False), c="k", alpha=1.0, ↪
    ↪zorder=99),
    )

def animate(ax=None, fig=None, **par_settings):
    global animate_plot_pieces
    items, obs = animate_plot_pieces
    pars = pyhf.tensorlib.astensor(pdf.config.suggested_init())
    for k, v in par_settings.items():
        pars[par_name_dict[k]] = v

    mc_counts = get_mc_counts(pars)
    rectangle_collection = zip(*map(lambda x: x.patches, items))

    for rectangles, binvalues in zip(rectangle_collection, mc_counts[:, 0].T):
        offset = 0
        for sample_index in [3, 2, 1, 0]:
            rect = rectangles[sample_index]
            binvalue = binvalues[sample_index]

```

(continues on next page)

(continued from previous page)

```

        rect.set_y(offset)
        rect.set_height(binvalue)
        offset += rect.get_height()

fig.canvas.draw()

def plot(ax=None, order=[3, 2, 1, 0], **par_settings):
    pars = pyhf.tensorlib.astensor(pdf.config.suggested_init())
    for k, v in par_settings.items():
        pars[par_name_dict[k]] = v

    mc_counts = get_mc_counts(pars)
    bottom = None
    # nb: bar_data[0] because evaluating only one parset
    for i, sample_index in enumerate(order):
        data = mc_counts[sample_index][0]
        x = np.arange(len(data))
        ax.bar(x, data, 1, bottom=bottom, alpha=1.0)
        bottom = data if i == 0 else bottom + data
    ax.scatter(x, workspace.data(pdf, with_aux=False), c="k", alpha=1.0, zorder=99)

```

6.7.5 Interactive Exploration of a HistFactory Model

One advantage of a pure-python implementation of Histfactory is the ability to explore the pdf interactively within the setting of a notebook. Try moving the sliders and observe the effect on the samples. For example changing the parameter of interest SigXsecOverSM (or μ) controls the overall normalization of the (BSM) signal sample ($\mu=0$ for background-only and $\mu=1$ for the nominal signal-plus-background hypothesis)

```

[8]: %matplotlib notebook
fig, ax = plt.subplots(1, 1)
fig.set_size_inches(10, 5)
ax.set_ylim(0, 1.5 * np.max(workspace.data(pdf, with_aux=False)))

init_plot(fig, ax, default_par_settings)
interact(animate, fig=fixed(fig), ax=fixed(ax), **all_par_settings);

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

interactive(children=(FloatSlider(value=1.0, description='lumi', max=1.5, min=0.5),
↪FloatSlider(value=1.5, des...

```

```

[9]: nominal = pdf.config.suggested_init()
background_only = pdf.config.suggested_init()
background_only[pdf.config.poi_index] = 0.0
best_fit = pyhf.infer.mle.fit(data, pdf)

/Users/jovyan/pyhf/src/pyhf/tensor/numpy_backend.py:253: RuntimeWarning: invalid_
↪value encountered in log
    return n * np.log(lam) - lam - gammaln(n + 1.0)

```

6.7.6 Fitting

We can now fit the statistical model to the observed data. The best fit of the signal strength is close to the background-only hypothesis.

```
[10]: f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, sharex=True)
f.set_size_inches(18, 4)
ax1.set_ylim(0, 1.5*np.max(workspace.data(pdf, with_aux=False)))
ax1.set_title(u'nominal signal + background  $\mu = 1$ ')
plot(ax = ax1, **{k: nominal[v] for k, v in par_name_dict.items()})

ax2.set_title(u'nominal background-only  $\mu = 0$ ')
plot(ax = ax2, **{k: background_only[v] for k, v in par_name_dict.items()})

ax3.set_title(u'best fit  $\mu = {:.3g}$ '.format(best_fit[pdf.config.poi_index]))
plot(ax = ax3, **{k: best_fit[v] for k, v in par_name_dict.items()});

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

6.7.7 Interval Estimation (Computing Upper Limits on μ)

A common task in the statistical evaluation of High Energy Physics data analyses is the estimation of confidence intervals of parameters of interest. The general strategy is to perform a series of hypothesis tests and then *invert* the tests in order to obtain an interval with the correct coverage properties.

A common figure of merit is a modified p-value, CLs. Here we compute an upper limit based on a series of CLs tests.

```
[11]: def invert_interval(test_mus, hypo_tests, test_size=0.05):
    cls_obs = np.array([test[0] for test in hypo_tests]).flatten()
    cls_exp = [
        np.array([test[1][i] for test in hypo_tests]).flatten() for i in range(5)
    ]
    crossing_test_stats = {"exp": [], "obs": None}
    for cls_exp_sigma in cls_exp:
        crossing_test_stats["exp"].append(
            np.interp(
                test_size, list(reversed(cls_exp_sigma)), list(reversed(test_mus))
            )
        )
    crossing_test_stats["obs"] = np.interp(
        test_size, list(reversed(cls_obs)), list(reversed(test_mus))
    )
    return crossing_test_stats
```

```
[12]: mu_tests = np.linspace(0, 1, 16)
hypo_tests = [
    pyhf.infer.hypotest(
        mu,
        data,
        pdf,
        pdf.config.suggested_init(),
        pdf.config.suggested_bounds(),
        return_expected_set=True,
    )
]
```

(continues on next page)

(continued from previous page)

```

    for mu in mu_tests
]

```

```
[13]: import pyhf.contrib.viz.brazil
```

```

fig, (ax1) = plt.subplots(1, 1)
fig.set_size_inches(7, 5)

ax1.set_title(u"Hypothesis Tests")
ax1.set_ylabel(u"CLs")
ax1.set_xlabel(u" $\mu$ ")

pyhf.contrib.viz.brazil.plot_results(ax1, mu_tests, hypo_tests);

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```

```
[14]: results = invert_interval(mu_tests, hypo_tests)
```

```

print('Observed Limit: {:.2f}'.format(results['obs']))
print('-----')
for i, n_sigma in enumerate(np.arange(-2,3)):
    print('Expected Limit {}: {:.3f}'.format(' if n_sigma==0 else '({} )'.format(n_
    sigma), results['exp'][i]))

Observed Limit: 0.60
-----
Expected Limit(-2 ): 0.266
Expected Limit(-1 ): 0.363
Expected Limit: 0.505
Expected Limit(1 ): 0.707
Expected Limit(2 ): 0.956

```


OUTREACH

We are always interested in talking about pyhf. See the abstract and a list of previously given presentations and feel free to invite us to your next conference/workshop/meeting!

7.1 Abstract

The HistFactory p.d.f. template [CERN-OPEN-2012-016] is per-se independent of its implementation in ROOT and it is useful to be able to run statistical analysis outside of the ROOT, RooFit, RooStats framework. pyhf is a pure-python implementation of that statistical model for multi-bin histogram-based analysis and its interval estimation is based on the asymptotic formulas of “Asymptotic formulae for likelihood-based tests of new physics” [1007.1727]. pyhf supports modern computational graph libraries such as TensorFlow and PyTorch in order to make use of features such as auto-differentiation and GPU acceleration.

```
The HistFactory p.d.f. template
\href{https://cds.cern.ch/record/1456844}{[CERN-OPEN-2012-016]} is
per-se independent of its implementation in ROOT and it is useful to be
able to run statistical analysis outside of the ROOT, RooFit, RooStats
framework. pyhf is a pure-python implementation of that statistical
model for multi-bin histogram-based analysis and its interval
estimation is based on the asymptotic formulas of "Asymptotic formulae
for likelihood-based tests of new physics"
\href{https://arxiv.org/abs/1007.1727}{[arXiv:1007.1727]}. pyhf
supports modern computational graph libraries such as TensorFlow and
PyTorch in order to make use of features such as autodifferentiation
and GPU acceleration.
```

7.2 Presentations

This list will be updated with talks given on pyhf:

- Matthew Feickert. Likelihood preservation and statistical reproduction of searches for new physics. CHEP 2019, Nov 2019. URL: <https://indico.cern.ch/event/773049/contributions/3476143/>.
- Matthew Feickert. pyhf: pure-Python implementation of HistFactory. PyHEP 2019 Workshop, Oct 2019. URL: <https://indico.cern.ch/event/833895/contributions/3577824/>.
- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: a pure Python implementation of HistFactory with tensors and autograd. DIANA Meeting - pyhf, October 2018. URL: <https://indico.cern.ch/event/759480/>.

- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: pure-Python implementation of HistFactory models with autograd. 2018 US LHC Users Association Meeting, October 2018. URL: <https://indico.fnal.gov/event/17566/session/0/contribution/99>.
- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: pure-Python implementation of HistFactory models with autograd. (Internal) 3rd ATLAS Machine Learning Workshop, October 2018. URL: <https://indico.cern.ch/event/735932/contributions/3136879/>.
- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: pure-Python implementation of HistFactory models with autograd. (Internal) Joint Machine Learning & Statistics Fora Meeting, September 2018. URL: <https://indico.cern.ch/event/757657/contributions/3141134/>.
- Lukas Heinrich. Gaussian Process Shape Estimation and Systematics. (Internal) Joint Machine Learning & Statistics Fora Meeting, Dec 2018. URL: <https://indico.cern.ch/event/777561/contributions/3234669/>.
- Lukas Heinrich. HEP in the Cloud Computing and Open Science Era. EP-IT Data science seminar, Oct 2019. URL: <https://indico.cern.ch/event/840837/>.
- Lukas Heinrich. Likelihoods associated with statistical fits used in searches for new physics on HEPData and use of RECAST. (Internal) ATLAS Weekly Meeting, Nov 2019. URL: <https://indico.cern.ch/event/864395/contributions/3642165/>.
- Lukas Heinrich. Traditional inference with machine learning tools. 1st Pan-European Advanced School on Statistics in High Energy Physics, Oct 2019. URL: <https://indico.desy.de/indico/event/22731/session/4/contribution/19>.
- Lukas Heinrich. pyhf: Full Run-2 ATLAS likelihoods. (Internal) Joint Machine Learning & Statistics Fora Meeting, May 2019. URL: <https://indico.cern.ch/event/817483/contributions/3412907/>.
- Lukas Heinrich, Matthew Feickert, Giordon Stark, and Kyle Cranmer. pyhf: A standalone HistFactory Implementation. (Re)interpreting the results of new physics searches at the LHC Workshop, May 2018. URL: <https://indico.cern.ch/event/702612/contributions/2958658/>.
- Giordon Stark. Likelihood Preservation and Reproduction. West Coast LHC Jamboree 2019, Oct 2019. URL: <https://indico.cern.ch/event/848030/contributions/3616614/>.
- Giordon Stark. New techniques for use of public likelihoods for reinterpretation of search results. 27th International Conference on Supersymmetry and Unification of Fundamental Interactions (SUSY2019), May 2019. URL: <https://indico.cern.ch/event/746178/contributions/3396797/>.

7.3 Tutorials

This list will be updated with tutorials and schools given on pyhf:

- Matthew Feickert. pyhf: Accelerating analyses and preserving likelihoods. PyHEP 2020 Workshop (pyhf v0.4.4), Jul 2020. URL: <https://indico.cern.ch/event/882824/contributions/3931292/>.
- Lukas Heinrich. Introduction to pyhf. (Internal) ATLAS Induction Day + Software Tutorial (pyhf v0.1.2), Oct 2019. URL: <https://indico.cern.ch/event/831761/contributions/3484275/>.
- Lukas Heinrich. pyhf tutorial. (Internal) ATLAS Induction Day + Software Tutorial (pyhf v0.4.4), Jul 2020. URL: <https://indico.cern.ch/event/892952/contributions/3853306/>.

7.4 Posters

This list will be updated with posters presented on pyhf:

- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: a pure Python statistical fitting library for High Energy Physics with tensors and autograd. July 2019. 18th Scientific Computing with Python Conference (SciPy 2019). URL: <http://conference.scipy.org/proceedings/scipy2019/slides.html>, doi:10.25080/Majora-7ddc1dd1-019.
- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: pure Python implementation of HistFactory. November 2019. 24th International Conference on computing in High Energy & Nuclear Physics (CHEP 2019). URL: <https://indico.cern.ch/event/773049/contributions/3476180/>.
- Lukas Heinrich, Matthew Feickert, Giordon Stark, and Kyle Cranmer. pyhf: auto-differentiable binned statistical models. 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2019), March 2019. URL: <https://indico.cern.ch/event/708041/contributions/3272095/>.

7.5 In the Media

This list will be updated with media publications featuring pyhf:

- Katarina Anthony. New open release streamlines interactions with theoretical physicists. *ATLAS News*, December 2019. URL: <https://atlas.cern/updates/atlas-news/new-open-likelihoods>.
- Katarina Anthony. New open release allows theorists to explore LHC data in a new way. *CERN News*, January 2020. URL: <https://home.cern/news/news/knowledge-sharing/new-open-release-allows-theorists-explore-lhc-data-new-way>.

INSTALLATION

To install, we suggest first setting up a [virtual environment](#)

```
# Python3  
python3 -m venv pyhf
```

and activating it

```
source pyhf/bin/activate
```

8.1 Install latest stable release from PyPI...

8.1.1 ... with NumPy backend

```
python -m pip install pyhf
```

8.1.2 ... with TensorFlow backend

```
python -m pip install pyhf[tensorflow]
```

8.1.3 ... with PyTorch backend

```
python -m pip install pyhf[torch]
```

8.1.4 ... with JAX backend

```
python -m pip install pyhf[jax]
```

8.1.5 ... with all backends

```
python -m pip install pyhf[backends]
```

8.1.6 ... with xml import/export functionality

```
python -m pip install pyhf[xmlio]
```

8.2 Install latest development version from GitHub...

8.2.1 ... with NumPy backend

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf"
```

8.2.2 ... with TensorFlow backend

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf[tensorflow]"
```

8.2.3 ... with PyTorch backend

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf[torch]"
```

8.2.4 ... with JAX backend

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf[jax]"
```

8.2.5 ... with all backends

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf[backends]"
```

8.2.6 ... with xml import/export functionality

```
python -m pip install --ignore-installed -U "git+https://github.com/scikit-hep/pyhf.  
↪git#egg=pyhf[xmlio]"
```

8.3 Updating pyhf

Rerun the installation command. As the upgrade flag, `-U`, is used then the libraries will be updated.

DEVELOPING

To develop, we suggest using [virtual environments](#) together with `pip` or using [pipenv](#). Once the environment is activated, clone the repo from GitHub

```
git clone https://github.com/scikit-hep/pyhf.git
```

and install all necessary packages for development

```
python -m pip install --ignore-installed -U -e .[complete]
```

Then setup the Git pre-commit hook for [Black](#) by running

```
pre-commit install
```

as the `rev` gets updated through time to track changes of different hooks, simply run

```
pre-commit autoupdate
```

to have pre-commit install the new version.

9.1 Testing

9.1.1 Data Files

A function-scoped fixture called `datadir` exists for a given test module which will automatically copy files from the associated test modules data directory into a temporary directory for the given test execution. That is, for example, if a test was defined in `test_schema.py`, then data files located in `test_schema/` will be copied to a temporary directory whose path is made available by the `datadir` fixture. Therefore, one can do:

```
def test_patchset(datadir):  
    data_file = open(datadir.join('test.txt'))  
    ...
```

which will load the copy of `test.txt` in the temporary directory. This also works for parameterizations as this will effectively sandbox the file modifications made.

9.1.2 TestPyPI

pyhf tests packaging and distributing by publishing each commit to master to [TestPyPI](#). In addition, installation of the latest test release from TestPyPI can be tested with

```
python -m pip install --extra-index-url https://test.pypi.org/simple/ --pre pyhf
```

Note: This adds TestPyPI as an [additional package index to search](#) when installing pyhf specifically. PyPI will still be the default package index pip will attempt to install from for all dependencies.

9.2 Publishing

Publishing to PyPI and TestPyPI is automated through the PyPA's [PyPI publish GitHub Action](#) and the pyhf [Tag Creator GitHub Actions workflow](#). A release can be created from any PR created by a core developer by adding a `bumpversion` tag to it that corresponds to the release type: [major](#), [minor](#), [patch](#). Once the PR is tagged with the label, the GitHub Actions bot will post a comment with information on the actions it will take once the PR is merged. When the PR has been reviewed, approved, and merged, the Tag Creator workflow will automatically create a new release with `bump2version` and then deploy the release to PyPI.

9.3 Context Files and Archive Metadata

The `.zenodo.json` and `codemeta.json` files have the version number automatically updated through `bump2version`, though their additional metadata should be checked periodically by the dev team (probably every release). The `codemeta.json` file can be generated automatically **from a PyPI install** of pyhf using `codemetapy`

```
codemetapy --no-extras pyhf > codemeta.json
```

though the author metadata will still need to be checked and revised by hand. The `.zenodo.json` is currently generated by hand, so it is worth using `codemeta.json` as a guide to edit it.

Frequently Asked Questions about `pyhf` and its use.

10.1 Questions

10.1.1 Where can I ask questions about `pyhf` use?

If you have a question about the use of `pyhf` not covered in the [documentation](#), please ask a question on [Stack Overflow](#) with the `[pyhf]` tag, which the `pyhf` dev team [watches](#).

If you believe you have found a bug in `pyhf`, please report it in the [GitHub Issues](#).

10.1.2 How can I get updates on `pyhf`?

If you're interested in getting updates from the `pyhf` dev team and release announcements you can join the [pyhf-announcements](#) mailing list.

10.1.3 Is it possible to set the backend from the CLI?

Yes. Use the `--backend` option for `pyhf cls` to specify a tensor backend. The default backend is NumPy. For more information see `pyhf cls --help`.

10.1.4 Does `pyhf` support Python 2?

No. Like the rest of the Python community, as of January 2020 the latest releases of `pyhf` no longer support Python 2. The last release of `pyhf` that was compatible with Python 2.7 is [v0.3.4](#), which can be installed with

```
python -m pip install pyhf~=0.3
```

10.1.5 I only have access to Python 2. How can I use pyhf?

It is recommended that `pyhf` is used as a standalone step in any analysis, and its environment need not be the same as the rest of the analysis. As Python 2 is not supported it is suggested that you setup a Python 3 runtime on whatever machine you're using. If you're using a cluster, talk with your system administrators to get their help in doing so. If you are unable to get a Python 3 runtime, versioned Docker images of `pyhf` are distributed through [Docker Hub](#).

Once you have Python 3 installed, see the [Installation](#) page to get started.

10.1.6 I validated my workspace by comparing pyhf and HistFactory, and while the expected CLs matches, the observed CLs is different. Why is this?

Make sure you're using the right test statistic (q or \tilde{q}) in both situations. In `HistFactory`, the asymptotics calculator, for example, will do something more involved for the observed CLs if you choose a different test statistic.

10.1.7 I ran validation to compare HistFitter and pyhf, but they don't match exactly. Why not?

`pyhf` is validated against `HistFactory`. `HistFitter` makes some particular implementation choices that `pyhf` doesn't reproduce. Instead of trying to compare `pyhf` and `HistFitter` you should try to validate them both against `HistFactory`.

10.1.8 How is pyhf typeset?

As you may have guessed from this page, `pyhf` is typeset in all lowercase. This is largely historical, as the core developers had just always typed it that way and it seemed a bit too short of a library name to write as `PyHF`. When typesetting in LaTeX the developers recommend introducing the command

```
\newcommand{\pyhf}{\texttt{pyhf}}
```

If the journal you are publishing in requires you to use `textsc` for software names it is okay to instead use

```
\newcommand{\pyhf}{\textsc{pyhf}}
```

10.2 Troubleshooting

- `import torch` or `import pyhf` causes a Segmentation fault (core dumped)

This is may be the result of a conflict with the NVIDIA drivers that you have installed on your machine. Try uninstalling and completely removing all of them from your machine

```
# On Ubuntu/Debian
sudo apt-get purge nvidia*
```

and then installing the latest versions.

TRANSLATIONS

One key goal of `pyhf` is to provide seamless translations between other statistical frameworks and `pyhf`. This page details the various ways to translate from a tool you might already be using as part of an existing analysis to `pyhf`. Many of these solutions involve extracting out the `HistFactory` workspace and then running `pyhf xml2json` which provides a single JSON workspace that can be loaded directly into `pyhf`.

11.1 HistFitter

In order to go from `HistFitter` to `pyhf`, the first step is to extract out the `HistFactory` workspaces. Assuming you have an existing configuration file, `config.py`, you likely run an exclusion fit like so:

```
HistFitter.py -f -D "before,after,corrMatrix" -F excl config.py
```

The name of output workspace files depends on four parameters you define in your `config.py`:

- `analysisName` is from `configMgr.analysisName`
- `prefix` is defined in `configMgr.addFitConfig({prefix})`
- `measurementName` is the first measurement you define via `fitConfig.addMeasurement(name={measurementName}, ...)`
- `channelName` are the names of channels you define via `fitConfig.addChannel("cuts", [{channelName}], ...)`
- `cachePath` is where `HistFitter` stores the cached histograms, defined by `configMgr.histCacheFile` which defaults to `data/{analysisName}.root`

To dump the `HistFactory` workspace, you will modify the above to skip the fit `-f` and plotting `-D` so you end up with

```
HistFitter.py -wx -F excl config.py
```

The `-w` flag tells `HistFitter` to (re)create the `HistFactory` workspace stored in `results/{analysisName}/{prefix}_combined_{measurementName}.root`. The `-x` flag tells `HistFitter` to dump the XML files into `config/{analysisName}/`, with the top-level file being `{prefix}.xml` and all other files being `{prefix}_{channelName}_cuts.xml`.

Typically, `prefix = 'FitConfig'` and `measurementName = 'NormalMeasurement'`. For example, if the following exists in your `config.py`

```
from configManager import configMgr
# ...
configMgr.analysisName = '3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg'
configMgr.histCacheFile = 'cache/{0:s}.root'.format(configMgr.analysisName)
```

(continues on next page)

(continued from previous page)

```
# ...
fitConfig = configMgr.addFitConfig("Excl")
# ...
channel = fitConfig.addChannel("cuts", ['SR_0L'], 1, 0.5, 1.5)
# ...
meas1=fitConfig.addMeasurement(name="DefaultMeasurement",lumi=1.0,lumiErr=0.029)
meas1.addPOI("mu_SIG1")
# ...
meas2=fitConfig.addMeasurement(name="DefaultMeasurement",lumi=1.0,lumiErr=0.029)
meas2.addPOI("mu_SIG2")
```

Then, you expect the following files to be made:

- config/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg/Excl.xml
- config/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg/Excl_SR_0L_cuts.xml
- cache/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg.root
- results/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg/
Excl_combined_DefaultMeasurement.root

These are all the files you need in order to use `pyhf xml2json`. At this point, you could run

```
pyhf xml2json config/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg/Excl.xml
```

which will read all of the XML files and load the histogram data from the histogram cache.

The `HistFactory` workspace in `results/` contains all of the information necessary to re-build the XML files again. For debugging purposes, the `pyhf` developers will often ask for your workspace file, which means `results/3b_tag21.2.27-1_RW_ExpSyst_36100_multibin_bkg/Excl_combined_DefaultMeasurement.root`. If you want to generate the XML, you can open this file in ROOT and run `DefaultMeasurement->PrintXML()` which puts all of the XML files into the current directory you are in.

11.2 TRExFitter

Note: For more details on this section, please refer to the ATLAS-internal [TRExFitter documentation](#).

In order to go from TRExFitter to `pyhf`, the good news is that the `RooWorkspace` files (XML and ROOT) are already made for you. For a given configuration which looks like

```
Job: "pyhf_example"
Label: "..."
```

You can expect some files to be made after the `n/h` and `w` steps:

- pyhf_example/RooStats/pyhf_example.xml
- pyhf_example/RooStats/pyhf_example_Signal_region.xml
- pyhf_example/Histograms/pyhf_example_Signal_region_histos.root

These are all the files you need in order to use `pyhf xml2json`. At this point, you could run

```
pyhf xml2json pyhf_example/RooStats/pyhf_example.xml
```

which will read all of the XML files and load the histogram data from the histogram cache.

Warning: There are a few caveats one needs to be aware of with this conversion:

- Uncorrelated shape systematics cannot be pruned, see Issue [#662](#).
- Custom expressions for normalization factors cannot be used, see Issue [#850](#).

COMMAND LINE API

12.1 pyhf

Top-level CLI entrypoint.

```
pyhf [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit.

12.1.1 cls

Compute CLs value(s) for a given pyhf workspace.

Example:

```
$ curl -sL https://git.io/JJYDE | pyhf cls

{
  "CLs_exp": [
    0.07807427911686156,
    0.17472571775474618,
    0.35998495263681285,
    0.6343568235898907,
    0.8809947004472013
  ],
  "CLs_obs": 0.3599845631401915
}
```

```
pyhf cls [OPTIONS] [WORKSPACE]
```

Options

--output-file <output_file>
The location of the output json file. If not specified, prints to screen.

--measurement <measurement>

-p, --patch <patch>

--testpoi <testpoi>

--teststat <teststat>

Options qlqtilde

--backend <backend>
The tensor backend used for the calculation.

Options numpy/pytorch/tensorflow/jax/np/ptorch/ltf

--optimizer <optimizer>
The optimizer used for the calculation.

Options scipy/minuit

--optconf <optconf>

Arguments

WORKSPACE
Optional argument

12.1.2 combine

Combine two workspaces into a single workspace.

See `pyhf.workspace.Workspace.combine()` for more information.

```
pyhf combine [OPTIONS] [WORKSPACE_ONE] [WORKSPACE_TWO]
```

Options

-j, --join <join>
The join operation to apply when combining the two workspaces.

Options none/outer/left outer/right outer

--output-file <output_file>
The location of the output json file. If not specified, prints to screen.

--merge-channels, --no-merge-channels
Whether or not to deeply merge channels. Can only be done with left/right outer joins.

Arguments

WORKSPACE_ONE

Optional argument

WORKSPACE_TWO

Optional argument

12.1.3 completions

Generate shell completion code.

```
pyhf completions [OPTIONS] SHELL
```

Arguments

SHELL

Required argument

12.1.4 contrib

Contrib experimental operations.

Note: Requires installation of the `contrib` extra.

```
$ python -m pip install pyhf[contrib]
```

```
pyhf contrib [OPTIONS] COMMAND [ARGS]...
```

download

Download the patchset archive from the remote URL and extract it in a directory at the path given.

Example:

```
$ pyhf contrib download --verbose https://doi.org/10.17182/hepdata.90607.v3/r3 1Lbb-
↳likelihoods

1Lbb-likelihoods/patchset.json
1Lbb-likelihoods/README.md
1Lbb-likelihoods/BkgOnly.json
```

Raises: `InvalidArchiveHost`: if the provided archive host name is not known to be valid

```
pyhf contrib download [OPTIONS] ARCHIVE_URL OUTPUT_DIRECTORY
```

Options

- v, --verbose**
Enables verbose mode
- f, --force**
Force download from non-approved host
- c, --compress**
Keep the archive in a compressed tar.gz form

Arguments

- ARCHIVE_URL**
Required argument
- OUTPUT_DIRECTORY**
Required argument

12.1.5 digest

Use hashing algorithm to calculate the workspace digest.

Returns: digests (`dict`): A mapping of the hashing algorithms used to the computed digest for the workspace.

Example:

```
$ curl -sL https://raw.githubusercontent.com/scikit-hep/pyhf/master/docs/examples/
→ json/2-bin_1-channel.json | pyhf digest
sha256:dad8822af55205d60152cbe4303929042dbd9d4839012e055e7c6b6459d68d73
```

```
pyhf digest [OPTIONS] [WORKSPACE]
```

Options

- a, --algorithm <algorithm>**
The hashing algorithm used to compute the workspace digest.
- j, --json, -p, --plaintext**
Output the hash values as a JSON dictionary or plaintext strings

Arguments

- WORKSPACE**
Optional argument

12.1.6 fit

Perform a maximum likelihood fit for a given pyhf workspace.

Example:

```
$ curl -sL https://git.io/JJYDE | pyhf fit --value

{
  "mle_parameters": {
    "mu": [
      0.00017298628839781602
    ],
    "uncorr_bkguncrt": [
      1.0000015671710816,
      0.9999665895859197
    ]
  },
  "twice_nll": 23.19636590468879
}
```

```
pyhf fit [OPTIONS] [WORKSPACE]
```

Options

--output-file <output_file>

The location of the output json file. If not specified, prints to screen.

--measurement <measurement>

-p, --patch <patch>

--value

Flag for returning the fitted value of the objective function.

--backend <backend>

The tensor backend used for the calculation.

Options numpy|pytorch|tensorflow|jax|npl|torch|tf

--optimizer <optimizer>

The optimizer used for the calculation.

Options scipy|minuit

--optconf <optconf>

Arguments

WORKSPACE

Optional argument

12.1.7 inspect

Inspect a pyhf JSON document.

Example:

```
$ curl -sL https://raw.githubusercontent.com/scikit-hep/pyhf/master/docs/examples/
↪ json/2-bin_1-channel.json | pyhf inspect
      Summary
      -----
      channels  1
      samples   2
      parameters 2
      modifiers 2

      channels  nbins
      -----
singlechannel  2

      samples
      -----
      background
      signal

      parameters  constraint  modifiers
      -----
              mu  unconstrained  normfactor
uncorr_bkguncrt  constrained_by_poisson  shapsys

      measurement      poi      parameters
      -----
(*) Measurement      mu      (none)
```

```
pyhf inspect [OPTIONS] [WORKSPACE]
```

Options

--output-file <output_file>

The location of the output json file. If not specified, prints to screen.

--measurement <measurement>

Arguments

WORKSPACE

Optional argument

12.1.8 json2xml

Convert pyhf JSON back to XML + ROOT files.

```
pyhf json2xml [OPTIONS] [WORKSPACE]
```

Options

```
--output-dir <output_dir>
--specroot <specroot>
--dataroot <dataroot>
--resultprefix <resultprefix>
-p, --patch <patch>
```

Arguments

WORKSPACE
Optional argument

12.1.9 patchset

Operations involving patchsets.

```
pyhf patchset [OPTIONS] COMMAND [ARGS]...
```

apply

Apply a patch from patchset to the background-only workspace specification.

Raises: *InvalidPatchLookup*: if the provided patch name is not in the patchset
PatchSetVerificationError: if the patchset cannot be verified against the workspace specification

Returns: workspace (*Workspace*): The patched background-only workspace.

```
pyhf patchset apply [OPTIONS] [BACKGROUND_ONLY] [PATCHSET]
```

Options

```
--name <name>
    The name of the patch to extract.
--output-file <output_file>
    The location of the output json file. If not specified, prints to screen.
```

Arguments

BACKGROUND_ONLY

Optional argument

PATCHSET

Optional argument

extract

Extract a patch from a patchset.

Raises: *InvalidPatchLookup*: if the provided patch name is not in the patchset

Returns: jsonpatch (*list*): A list of jsonpatch operations to apply to a workspace.

```
pyhf patchset extract [OPTIONS] [PATCHSET]
```

Options

--name <name>

The name of the patch to extract.

--output-file <output_file>

The location of the output json file. If not specified, prints to screen.

--with-metadata, --without-metadata

Include patchset metadata in output.

Arguments

PATCHSET

Optional argument

verify

Verify the patchset digests against a background-only workspace specification. Verified if no exception was raised.

Raises: *PatchSetVerificationError*: if the patchset cannot be verified against the workspace specification

Returns: None

```
pyhf patchset verify [OPTIONS] [BACKGROUND_ONLY] [PATCHSET]
```

Arguments

BACKGROUND_ONLY

Optional argument

PATCHSET

Optional argument

12.1.10 prune

Prune components from the workspace.

See `pyhf.workspace.Workspace.prune()` for more information.

```
pyhf prune [OPTIONS] [WORKSPACE]
```

Options

```
--output-file <output_file>
    The location of the output json file. If not specified, prints to screen.
-c, --channel <CHANNEL>...>
-s, --sample <SAMPLE>...>
-m, --modifier <MODIFIER>...>
-t, --modifier-type <modifier_type>
    Options histosysllumilnormfactorlnormsyslshapefactorlshapessystaterror
--measurement <MEASUREMENT>...>
```

Arguments

WORKSPACE
Optional argument

12.1.11 rename

Rename components of the workspace.

See `pyhf.workspace.Workspace.rename()` for more information.

```
pyhf rename [OPTIONS] [WORKSPACE]
```

Options

```
--output-file <output_file>
    The location of the output json file. If not specified, prints to screen.
-c, --channel <PATTERN> <REPLACE>...>
-s, --sample <PATTERN> <REPLACE>...>
-m, --modifier <PATTERN> <REPLACE>...>
--measurement <PATTERN> <REPLACE>...>
```

Arguments

WORKSPACE

Optional argument

12.1.12 sort

Sort the workspace.

See `pyhf.workspace.Workspace.sorted()` for more information.

Example:

```
$ curl -sL https://raw.githubusercontent.com/scikit-hep/pyhf/master/docs/examples/
↳ json/2-bin_1-channel.json | pyhf sort | jq '.' | md5
8be5186ec249d2704e14dd29ef05ffb0
```

```
$ curl -sL https://raw.githubusercontent.com/scikit-hep/pyhf/master/docs/examples/
↳ json/2-bin_1-channel.json | jq -S '.channels|=sort_by(.name)|.channels|.
↳ samples|=sort_by(.name)|.channels|.samples|.modifiers|=sort_by(.name,.type)|.
↳ observations|=sort_by(.name)' | md5
8be5186ec249d2704e14dd29ef05ffb0
```

```
pyhf sort [OPTIONS] [WORKSPACE]
```

Options

--output-file <output_file>

The location of the output json file. If not specified, prints to screen.

Arguments

WORKSPACE

Optional argument

12.1.13 xml2json

Entrypoint XML: The top-level XML file for the PDF definition.

```
pyhf xml2json [OPTIONS] ENTRYPOINT_XML
```

Options

--basedir <basedir>

The base directory for the XML files to point relative to.

--output-file <output_file>

The location of the output json file. If not specified, prints to screen.

--track-progress, --hide-progress

Arguments

ENTRYPOINT_XML

Required argument

13.1 Top-Level

<i>default_backend</i>	NumPy backend for pyhf
<i>default_optimizer</i>	Optimizer that uses <code>scipy.optimize.minimize()</code> .
<i>tensorlib</i>	NumPy backend for pyhf
<i>optimizer</i>	Optimizer that uses <code>scipy.optimize.minimize()</code> .
<i>get_backend()</i>	Get the current backend and the associated optimizer
<i>set_backend</i> (backend[, custom_optimizer, ...])	Set the backend and the associated optimizer

13.1.1 pyhf.default_backend

`pyhf.default_backend = <pyhf.tensor.numpy_backend.numpy_backend object>`
NumPy backend for pyhf

13.1.2 pyhf.default_optimizer

`pyhf.default_optimizer = <pyhf.optimize.scipy_optimizer object>`
Optimizer that uses `scipy.optimize.minimize()`.

13.1.3 pyhf.tensorlib

`pyhf.tensorlib` = <pyhf.tensor.numpy_backend.numpy_backend object>
NumPy backend for pyhf

13.1.4 pyhf.optimizer

`pyhf.optimizer` = <pyhf.optimize.scipy_optimizer object>
Optimizer that uses `scipy.optimize.minimize()`.

13.1.5 pyhf.get_backend

`pyhf.get_backend()`
Get the current backend and the associated optimizer

Example

```
>>> import pyhf
>>> backend, optimizer = pyhf.get_backend()
>>> backend
<pyhf.tensor.numpy_backend.numpy_backend object at 0x...>
>>> optimizer
<pyhf.optimize.scipy_optimizer object at 0x...>
```

Returns backend, optimizer

13.1.6 pyhf.set_backend

`pyhf.set_backend(backend, custom_optimizer=None, precision=None)`
Set the backend and the associated optimizer

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> pyhf.tensorlib.name
'tensorflow'
>>> pyhf.tensorlib.precision
'32b'
>>> pyhf.set_backend(b"pytorch", precision="64b")
>>> pyhf.tensorlib.name
'pytorch'
>>> pyhf.tensorlib.precision
'64b'
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> pyhf.tensorlib.name
'numpy'
>>> pyhf.tensorlib.precision
'64b'
```

Parameters

- **backend** (`str` or `pyhf.tensor` backend) – One of the supported pyhf backends: NumPy, TensorFlow, PyTorch, and JAX
- **custom_optimizer** (`pyhf.optimize` optimizer) – Optional custom optimizer defined by the user
- **precision** (`str`) – Floating point precision to use in the backend: 64b or 32b. Default is backend dependent.

Returns None

13.2 Probability Distribution Functions (PDFs)

<i>Normal</i>	The Normal distribution with mean <code>loc</code> and standard deviation <code>scale</code> .
<i>Poisson</i>	The Poisson distribution with rate parameter <code>rate</code> .
<i>Independent</i>	A probability density corresponding to the joint distribution of a batch of identically distributed random variables.
<i>Simultaneous</i>	A probability density corresponding to the joint distribution of multiple non-identical component distributions

13.2.1 Normal

class `pyhf.probability.Normal` (`loc`, `scale`)Bases: `pyhf.probability._SimpleDistributionMixin`The Normal distribution with mean `loc` and standard deviation `scale`.**Example**

```
>>> import pyhf
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> pyhf.probability.Normal(means, stds)
<pyhf.probability.Normal object at 0x...>
```

Methods`__init__` (`loc`, `scale`)**Parameters**

- **loc** (tensor or `float`) – The mean of the Normal distribution
- **scale** (tensor or `float`) – The standard deviation of the Normal distribution

expected_data ()

The expectation value of the Normal distribution.

Example

```
>>> import pyhf
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> normals = pyhf.probability.Normal(means, stds)
>>> normals.expected_data()
array([5., 8.]
```

Returns The mean of the Normal distribution (which is the `loc`)

Return type Tensor

13.2.2 Poisson

class `pyhf.probability.Poisson(rate)`
Bases: `pyhf.probability._SimpleDistributionMixin`
The Poisson distribution with rate parameter `rate`.

Example

```
>>> import pyhf
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> pyhf.probability.Poisson(rates)
<pyhf.probability.Poisson object at 0x...>
```

Methods

`__init__(rate)`

Parameters `rate` (tensor or `float`) – The mean of the Poisson distribution (the expected number of events)

`expected_data()`

The expectation value of the Poisson distribution.

Example

```
>>> import pyhf
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> poissons = pyhf.probability.Poisson(rates)
>>> poissons.expected_data()
array([5., 8.]
```

Returns The mean of the Poisson distribution (which is the `rate`)

Return type Tensor

13.2.3 Independent

class pyhf.probability.**Independent** (*batched_pdf*, *batch_size=None*)

Bases: pyhf.probability._SimpleDistributionMixin

A probability density corresponding to the joint distribution of a batch of identically distributed random variables.

Example

```
>>> import pyhf
>>> import numpy.random as random
>>> random.seed(0)
>>> rates = pyhf.tensorlib.astensor([10.0, 10.0])
>>> poissons = pyhf.probability.Poisson(rates)
>>> independent = pyhf.probability.Independent(poissons)
>>> independent.sample()
array([10, 11])
```

Methods

__init__ (*batched_pdf*, *batch_size=None*)

Parameters

- **batched_pdf** (pyhf.probability distribution) – The batch of pdfs of the same type (e.g. Poisson)
- **batch_size** (int) – The size of the batch

log_prob (*value*)

The log of the probability density function at the given value. As the distribution is a joint distribution of the same type, this is the sum of the log probabilities of each of the distributions the compose the joint.

Example

```
>>> import pyhf
>>> import numpy.random as random
>>> random.seed(0)
>>> rates = pyhf.tensorlib.astensor([10.0, 10.0])
>>> poissons = pyhf.probability.Poisson(rates)
>>> independent = pyhf.probability.Independent(poissons)
>>> values = pyhf.tensorlib.astensor([8.0, 9.0])
>>> independent.log_prob(values)
-4.262483801927939
>>> broadcast_value = pyhf.tensorlib.astensor([11.0])
>>> independent.log_prob(broadcast_value)
-4.347743645878765
```

Parameters **value** (tensor or float) – The value at which to evaluate the distribution

Returns The value of $\log(f(x|\theta))$ for $x = \text{value}$

Return type Tensor

13.2.4 Simultaneous

class pyhf.probability.Simultaneous(*pdfobjs, tensorview, batch_size=None*)

Bases: `object`

A probability density corresponding to the joint distribution of multiple non-identical component distributions

Example

```
>>> import pyhf
>>> import numpy.random as random
>>> from pyhf.tensor.common import _TensorViewer
>>> random.seed(0)
>>> poissons = pyhf.probability.Poisson(pyhf.tensorlib.astensor([1.,100.]))
>>> normals = pyhf.probability.Normal(pyhf.tensorlib.astensor([1.,100.]), pyhf.
↳ tensorlib.astensor([1.,2.]))
>>> tv = _TensorViewer([[0,2],[1,3]])
>>> sim = pyhf.probability.Simultaneous([poissons,normals], tv)
>>> sim.sample((4,))
array([[ 2.          ,  1.3130677, 101.          ,  98.29180852],
       [ 1.          , -1.55298982,  97.          , 101.30723719],
       [ 1.          ,  1.8644362 , 118.          ,  98.51566996],
       [ 0.          ,  3.26975462,  99.          ,  97.09126865]])
```

Methods

__init__(*pdfobjs, tensorview, batch_size=None*)

Construct a simultaneous pdf.

Parameters

- **pdfobjs** (`Distribution`) – The constituent pdf objects
- **tensorview** (`_TensorViewer`) – The `_TensorViewer` defining the data composition
- **batch_size** (`int`) – The size of the batch

expected_data()

The expectation value of the probability density function.

Returns The expectation value of the distribution $E[f(\theta)]$

Return type `Tensor`

log_prob(*value*)

The log of the probability density function at the given value.

Parameters **value** (`tensor`) – The observed value

Returns The value of $\log(f(x|\theta))$ for $x = \text{value}$

Return type `Tensor`

sample(*sample_shape=()*)

The collection of values sampled from the probability density function.

Parameters **sample_shape** (`tuple`) – The shape of the sample to be returned

Returns The values $x \sim f(\theta)$ where x has shape `sample_shape`

Return type Tensor

13.3 Making Models from PDFs

<i>Model</i>	The main pyhf model class.
<i>_ModelConfig</i>	
<i>Workspace</i>	A JSON-serializable object that is built from an object that follows the <code>workspace.json schema</code> .
<i>PatchSet</i>	A way to store a collection of patches (<i>Patch</i>).
<i>Patch</i>	A way to store a patch definition as part of a patchset (<i>PatchSet</i>).
<i>simplemodels.hepdata_like</i>	Construct a simple single channel <i>Model</i> with a <i>shapesys</i> modifier representing an uncorrelated background uncertainty.

13.3.1 Model

class `pyhf.pdf.Model` (*spec*, *batch_size=None*, ***config_kwargs*)
 Bases: `object`

The main pyhf model class.

Attributes

nominal_rates
 Nominal value of bin rates of the main model.

Methods

__init__ (*spec*, *batch_size=None*, ***config_kwargs*)
 Construct a HistFactory Model.

Parameters

- **spec** (`jsonable`) – The HistFactory JSON specification
- **batch_size** (`None` or `int`) – Number of simultaneous (batched) Models to compute.
- **config_kwargs** – Possible keyword arguments for the model configuration

Returns The Model instance.

Return type model (*Model*)

constraint_logpdf (*auxdata*, *pars*)
 Compute the log value of the constraint pdf.

Parameters

- **auxdata** (`tensor`) – The auxiliary measurement data
- **pars** (`tensor`) – The parameter values

Returns The log density value

Return type Tensor

expected_actualdata (*pars*)

Compute the expected value of the main model.

Parameters **pars** (tensor) – The parameter values

Returns The expected data of the main model (no auxiliary data)

Return type Tensor

expected_auxdata (*pars*)

Compute the expected value of the auxiliary measurements.

Parameters **pars** (tensor) – The parameter values

Returns The expected data of the auxiliary pdf

Return type Tensor

expected_data (*pars*, *include_auxdata=True*)

Compute the expected value of the main model

Parameters **pars** (tensor) – The parameter values

Returns The expected data of the main and auxiliary model

Return type Tensor

logpdf (*pars*, *data*)

Compute the log value of the full density.

Parameters

- **pars** (tensor) – The parameter values
- **data** (tensor) – The measurement data

Returns The log density value

Return type Tensor

mainlogpdf (*maindata*, *pars*)

Compute the log value of the main term.

Parameters

- **maindata** (tensor) – The main measurement data
- **pars** (tensor) – The parameter values

Returns The log density value

Return type Tensor

make_pdf (*pars*)

Construct a pdf object for a given set of parameter values.

Parameters **pars** (tensor) – The model parameters

Returns A distribution object implementing the main measurement pdf of HistFactory

Return type pdf

pdf (*pars*, *data*)

Compute the density at a given observed point in data space of the full model.

Parameters

- **pars** (tensor) – The parameter values
- **data** (tensor) – The measurement data

Returns The density value

Return type Tensor

13.3.2 `_ModelConfig`

class `pyhf.pdf._ModelConfig` (*spec*, ***config_kwargs*)
 Bases: `pyhf.mixins._ChannelSummaryMixin`

Methods

`__init__` (*spec*, ***config_kwargs*)
 Initialize self. See `help(type(self))` for accurate signature.

`par_slice` (*name*)

`param_set` (*name*)

`set_poi` (*name*)

`suggested_bounds` ()

`suggested_fixed` ()
 Identify the fixed parameters in the model.

Returns A list of booleans, `True` for fixed and `False` for not fixed.

Return type List

Something like the following to build `fixed_vals` appropriately:

```
fixed_pars = pdf.config.suggested_fixed()
inits = pdf.config.suggested_init()
fixed_vals = [
    (index, init)
    for index, (init, is_fixed) in enumerate(zip(inits, fixed_pars))
    if is_fixed
]
```

`suggested_init` ()

13.3.3 `Workspace`

class `pyhf.workspace.Workspace` (*spec*, ***config_kwargs*)
 Bases: `pyhf.mixins._ChannelSummaryMixin`, `dict`

A JSON-serializable object that is built from an object that follows the `workspace.json` [schema](#).

Attributes

`valid_joins = ['none', 'outer', 'left outer', 'right outer']`

Methods

`__init__(spec, **config_kwargs)`

Workspaces hold the model, data and measurements.

classmethod `build(model, data, name='measurement')`

Build a workspace from model and data.

Parameters

- **model** (`Model`) – A model to store into a workspace
- **data** (`tensor`) – A array holding observations to store into a workspace
- **name** (`str`) – The name of the workspace measurement

Returns A new workspace object

Return type `Workspace`

classmethod `combine(left, right, join='none', merge_channels=False)`

Return a new workspace specification that is the combination of the two workspaces.

The new workspace must also be a valid workspace. A combination of workspaces is done by combining the set of:

- channels,
- observations, and
- measurements

between the two workspaces. If the two workspaces have modifiers that follow the same naming convention, then correlations across the two workspaces may be possible. In particular, the *lumi* modifier will be fully-correlated.

If the two workspaces have the same measurement (with the same POI), those measurements will get merged.

Raises `InvalidWorkspaceOperation` – The workspaces have common channel names, incompatible measurements, or incompatible schema versions.

Parameters

- **left** (`Workspace`) – A workspace
- **right** (`Workspace`) – Another workspace
- **join** (`str`) – How to join the two workspaces. Pick from “none”, “outer”, “left outer”, or “right outer”.
- **merge_channels** (`bool`) – Whether or not to merge channels when performing the combine. This is only done with “outer”, “left outer”, and “right outer” options.

Returns A new combined workspace object

Return type `Workspace`

data (*model*, *with_aux=True*)

Return the data for the supplied model with or without auxiliary data from the model.

The model is needed as the order of the data depends on the order of the channels in the model.

Raises `KeyError` – Invalid or missing channel

Parameters

- **model** (`Model`) – A model object adhering to the schema model.json
- **with_aux** (`bool`) – Whether to include auxiliary data from the model or not

Returns data

Return type `list`

get_measurement (*poi_name=None*, *measurement_name=None*, *measurement_index=None*)

Get (or create) a measurement object.

The following logic is used:

1. if the poi name is given, create a measurement object for that poi
2. if the measurement name is given, find the measurement for the given name
3. if the measurement index is given, return the measurement at that index
4. if there are measurements but none of the above have been specified, return the 0th measurement

Raises `InvalidMeasurement` – If the measurement was not found

Parameters

- **poi_name** (`str`) – The name of the parameter of interest to create a new measurement from
- **measurement_name** (`str`) – The name of the measurement to use
- **measurement_index** (`int`) – The index of the measurement to use

Returns A measurement object adhering to the schema defs.json#/definitions/measurement

Return type `dict`

model (***config_kwargs*)

Create a model object with/without patches applied.

See `pyhf.workspace.Workspace.get_measurement()` and `pyhf.pdf.Model` for possible keyword arguments.

Parameters

- **patches** – A list of JSON patches to apply to the model specification
- **config_kwargs** – Possible keyword arguments for the measurement and model configuration

Returns A model object adhering to the schema model.json

Return type `Model`

prune (*modifiers=None*, *modifier_types=None*, *samples=None*, *channels=None*, *measurements=None*)

Return a new, pruned workspace specification. This will not modify the original workspace.

The pruned workspace must also be a valid workspace.

Parameters

- **modifiers** – A `str` or a `list` of modifiers to prune.
- **modifier_types** – A `str` or a `list` of modifier types to prune.
- **samples** – A `str` or a `list` of samples to prune.
- **channels** – A `str` or a `list` of channels to prune.
- **measurements** – A `str` or a `list` of measurements to prune.

Returns A new workspace object with the specified components removed

Return type *Workspace*

Raises *InvalidWorkspaceOperation* – An item name to prune does not exist in the workspace.

rename (*modifiers=None, samples=None, channels=None, measurements=None*)

Return a new workspace specification with certain elements renamed.

This will not modify the original workspace. The renamed workspace must also be a valid workspace.

Parameters

- **modifiers** – A `dict` mapping old modifier name to new modifier name.
- **samples** – A `dict` mapping old sample name to new sample name.
- **channels** – A `dict` mapping old channel name to new channel name.
- **measurements** – A `dict` mapping old measurement name to new measurement name.

Returns A new workspace object with the specified components renamed

Return type *Workspace*

Raises *InvalidWorkspaceOperation* – An item name to rename does not exist in the workspace.

classmethod sorted (*workspace*)

Return a new workspace specification that is sorted.

Parameters **workspace** (*Workspace*) – A workspace to sort

Returns A new sorted workspace object

Return type *Workspace*

13.3.4 PatchSet

class `pyhf.patchset.PatchSet` (*spec, **config_kwargs*)

Bases: `object`

A way to store a collection of patches (*Patch*).

It contains *metadata* about the PatchSet itself:

- a high-level *description* of what the patches represent or the analysis it is for
- a list of *references* where the patchset is sourced from (e.g. hepdata)
- a list of *digests* corresponding to the background-only workspace the patchset was made for
- the *labels* of the dimensions of the phase-space for what the patches cover

In addition to the above metadata, the PatchSet object behaves like a:

- smart list allowing you to iterate over all the patches defined

- smart dictionary allowing you to access a patch by the patch name or the patch values

The below example shows various ways one can interact with a *PatchSet* object.

Example

```
>>> import pyhf
>>> patchset = pyhf.PatchSet({
...     "metadata": {
...         "references": { "hepdata": "ins1234567" },
...         "description": "example patchset",
...         "digests": { "md5": "098f6bcd4621d373cade4e832627b4f6" },
...         "labels": ["x", "y"]
...     },
...     "patches": [
...         {
...             "metadata": {
...                 "name": "patch_name_for_2100x_800y",
...                 "values": [2100, 800]
...             },
...             "patch": [
...                 {
...                     "op": "add",
...                     "path": "/foo/0/bar",
...                     "value": {
...                         "foo": [1.0]
...                     }
...                 }
...             ]
...         }
...     ]
... },
...     "version": "1.0.0"
... })
>>> patchset.version
'1.0.0'
>>> patchset.references
{'hepdata': 'ins1234567'}
>>> patchset.description
'example patchset'
>>> patchset.digests
{'md5': '098f6bcd4621d373cade4e832627b4f6'}
>>> patchset.labels
['x', 'y']
>>> patchset.patches
[<pyhf.patchset.Patch object 'patch_name_for_2100x_800y(2100, 800)' at 0x...>]
>>> patchset['patch_name_for_2100x_800y']
<pyhf.patchset.Patch object 'patch_name_for_2100x_800y(2100, 800)' at 0x...>
>>> patchset[(2100,800)]
<pyhf.patchset.Patch object 'patch_name_for_2100x_800y(2100, 800)' at 0x...>
>>> patchset[[2100,800]]
<pyhf.patchset.Patch object 'patch_name_for_2100x_800y(2100, 800)' at 0x...>
>>> patchset[2100,800]
<pyhf.patchset.Patch object 'patch_name_for_2100x_800y(2100, 800)' at 0x...>
>>> for patch in patchset:
...     print(patch.name)
...
```

(continues on next page)

(continued from previous page)

```
patch_name_for_2100x_800y
>>> len(patchset)
1
```

Attributes

description

The description in the PatchSet metadata

digests

The digests in the PatchSet metadata

labels

The labels in the PatchSet metadata

metadata

The metadata of the PatchSet

patches

The patches in the PatchSet

references

The references in the PatchSet metadata

version

The version of the PatchSet

Methods

__init__ (*spec*, ****config_kwargs**)

Construct a PatchSet.

Parameters

- **spec** (*jsonable*) – The patchset JSON specification
- **config_kwargs** – Possible keyword arguments for the patchset validation

Returns The PatchSet instance.

Return type patchset (*PatchSet*)

apply (*spec*, *key*)

Apply the patch associated with the key to the background-only workspace specification.

Parameters

- **spec** (*Workspace*) – The workspace specification to verify the patchset against.
- **key** (*str* or *tuple* of *int/float*) – The key to look up the associated patch - either a name or a set of values.

Raises

- *InvalidPatchLookup* – if the provided patch name is not in the patchset
- *PatchSetVerificationError* – if the patchset cannot be verified against the workspace specification

Returns The background-only workspace with the patch applied.

Return type workspace (*Workspace*)

verify (*spec*)

Verify the patchset digests against a background-only workspace specification. Verified if no exception was raised.

Parameters *spec* (*Workspace*) – The workspace specification to verify the patchset against.

Raises *PatchSetVerificationError* – if the patchset cannot be verified against the workspace specification

Returns None

13.3.5 Patch

class pyhf.patchset.**Patch** (*spec*)

Bases: jsonpatch.JsonPatch

A way to store a patch definition as part of a patchset (*PatchSet*).

It contains *metadata* about the Patch itself:

- a descriptive *name*
- a list of the *values* for each dimension in the phase-space the associated *PatchSet* is defined for, see *labels*

In addition to the above metadata, the Patch object behaves like the underlying `jsonpatch.JsonPatch`.

Attributes

metadata

The metadata of the patch

name

The name of the patch

values

The values of the associated labels for the patch

Methods

__init__ (*spec*)

Construct a Patch.

Parameters *spec* (*jsonable*) – The patch JSON specification

Returns The Patch instance.

Return type patch (*Patch*)

13.3.6 hepdata_like

class pyhf.simplemodels.hepdata_like(*signal_data*, *bkg_data*, *bkg_uncerts*,
batch_size=None)

Construct a simple single channel *Model* with a *shapesys* modifier representing an uncorrelated background uncertainty.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> model.schema
'model.json'
>>> model.config.channels
['singlechannel']
>>> model.config.samples
['background', 'signal']
>>> model.config.parameters
['mu', 'uncorr_bkguncrt']
>>> model.expected_data(model.config.suggested_init())
array([ 62.          ,  63.          , 277.77777778,  55.18367347])
```

Parameters

- **signal_data** (*list*) – The data in the signal sample
- **bkg_data** (*list*) – The data in the background sample
- **bkg_uncerts** (*list*) – The statistical uncertainty on the background sample counts
- **batch_size** (*None* or *int*) – Number of simultaneous (batched) Models to compute

Returns The statistical model adhering to the `model.json` schema

Return type *Model*

13.4 Backends

The computational backends that pyhf provides interfacing for the vector-based calculations.

<code>numpy_backend.numpy_backend</code>	NumPy backend for pyhf
<code>pytorch_backend.pytorch_backend</code>	PyTorch backend for pyhf
<code>tensorflow_backend.tensorflow_backend</code>	TensorFlow backend for pyhf
<code>jax_backend.jax_backend</code>	JAX backend for pyhf

13.4.1 numpy_backend

class pyhf.tensor.numpy_backend.**numpy_backend**(**kwargs)

Bases: `object`

NumPy backend for pyhf

Attributes

default_do_grad

dtypesmap

name

precision

Methods

__init__(**kwargs)

Initialize self. See help(type(self)) for accurate signature.

abs(*tensor*)

astensor(*tensor_in*, *dtype*='float')

Convert to a NumPy array.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> tensor = pyhf.tensorlib.astensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> tensor
array([[1., 2., 3.],
       [4., 5., 6.]])
>>> type(tensor)
<class 'numpy.ndarray'>
```

Parameters **tensor_in** (*Number or Tensor*) – Tensor object

Returns A multi-dimensional, fixed-size homogenous array.

Return type *numpy.ndarray*

boolean_mask(*tensor*, *mask*)

clip(*tensor_in*, *min_value*, *max_value*)

Clips (limits) the tensor values to be within a specified min and max.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> a = pyhf.tensorlib.astensor([-2, -1, 0, 1, 2])
>>> pyhf.tensorlib.clip(a, -1, 1)
array([-1., -1.,  0.,  1.,  1.])
```

Parameters

- **tensor_in** (tensor) – The input tensor object
- **min_value** (scalar or tensor or `None`) – The minimum value to be clipped to
- **max_value** (scalar or tensor or `None`) – The maximum value to be clipped to

Returns A clipped *tensor*

Return type NumPy ndarray

concatenate (*sequence*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

- **sequence** – sequence of tensors
- **axis** – dimension along which to concatenate

Returns the concatenated tensor

Return type output

conditional (*predicate*, *true_callable*, *false_callable*)

Runs a callable conditional on the boolean value of the evaluation of a predicate

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> tensorlib = pyhf.tensorlib
>>> a = tensorlib.astensor([4])
>>> b = tensorlib.astensor([5])
>>> tensorlib.conditional((a < b)[0], lambda: a + b, lambda: a - b)
array([9.])
```

Parameters

- **predicate** (scalar) – The logical condition that determines which callable to evaluate
- **true_callable** (callable) – The callable that is evaluated when the predicate evaluates to true
- **false_callable** (callable) – The callable that is evaluated when the predicate evaluates to false

Returns The output of the callable that was evaluated

Return type NumPy ndarray

divide (*tensor_in_1*, *tensor_in_2*)

einsum (*subscripts, *operands*)

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations. The best way to understand this function is to try the examples below, which show how many common NumPy functions can be implemented as calls to einsum.

Parameters

- **subscripts** – str, specifies the subscripts for summation
- **operands** – list of array_like, these are the tensors for the operation

Returns the calculation based on the Einstein summation convention

Return type tensor

erf (*tensor_in*)

The error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erf(a)
array([-0.99532227, -0.84270079,  0.          ,  0.84270079,  0.99532227])
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the error function at the given points.

Return type NumPy ndarray

erfinv (*tensor_in*)

The inverse of the error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erfinv(pyhf.tensorlib.erf(a))
array([-2., -1.,  0.,  1.,  2.])
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the inverse of the error function at the given points.

Return type NumPy ndarray

exp (*tensor_in*)

gather (*tensor, indices*)

isfinite (*tensor*)

log (*tensor_in*)

normal (*x, mu, sigma*)

The probability density function of the Normal distribution evaluated at *x* given parameters of mean of *mu* and standard deviation of *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> pyhf.tensorlib.normal(0.5, 0., 1.)
0.3520653267642995
>>> values = pyhf.tensorlib.astensor([0.5, 2.0])
>>> means = pyhf.tensorlib.astensor([0., 2.3])
>>> sigmas = pyhf.tensorlib.astensor([1., 0.8])
>>> pyhf.tensorlib.normal(values, means, sigmas)
array([0.35206533, 0.46481887])
```

Parameters

- **x** (tensor or float) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns Value of Normal(*x*|*mu*, *sigma*)

Return type NumPy float

normal_cdf (*x, mu=0, sigma=1*)

The cumulative distribution function for the Normal distribution

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> pyhf.tensorlib.normal_cdf(0.8)
0.7881446014166034
>>> values = pyhf.tensorlib.astensor([0.8, 2.0])
>>> pyhf.tensorlib.normal_cdf(values)
array([0.7881446 , 0.97724987])
```

Parameters

- **x** (tensor or float) – The observed value of the random variable to evaluate the CDF for
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The CDF

Return type NumPy float

normal_dist (*mu, sigma*)

The Normal distribution with mean *mu* and standard deviation *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> normals = pyhf.tensorlib.normal_dist(means, stds)
>>> normals.log_prob(values)
array([-1.41893853, -2.22579135])
```

Parameters

- **mu** (tensor or `float`) – The mean of the Normal distribution
- **sigma** (tensor or `float`) – The standard deviation of the Normal distribution

Returns The Normal distribution class

Return type Normal distribution

normal_logpdf (*x*, *mu*, *sigma*)

ones (*shape*)

outer (*tensor_in_1*, *tensor_in_2*)

poisson (*n*, *lam*)

The continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at *n* given the parameter *lam*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> pyhf.tensorlib.poisson(5., 6.)
0.16062314104797995
>>> values = pyhf.tensorlib.astensor([5., 9.])
>>> rates = pyhf.tensorlib.astensor([6., 8.])
>>> pyhf.tensorlib.poisson(values, rates)
array([0.16062314, 0.12407692])
```

Parameters

- **n** (tensor or `float`) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (tensor or `float`) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continous approximation to Poisson(nllam)

Return type NumPy float

poisson_dist (*rate*)

The Poisson distribution with rate parameter *rate*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> poissons = pyhf.tensorlib.poisson_dist(rates)
>>> poissons.log_prob(values)
array([-1.74030218, -2.0868536 ])
```

Parameters *rate* (tensor or float) – The mean of the Poisson distribution (the expected number of events)

Returns The Poisson distribution class

Return type Poisson distribution

poisson_logpdf (*n*, *lam*)

power (*tensor_in_1*, *tensor_in_2*)

product (*tensor_in*, *axis=None*)

reshape (*tensor*, *newshape*)

shape (*tensor*)

simple_broadcast (**args*)

Broadcast a sequence of 1 dimensional arrays.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> pyhf.tensorlib.simple_broadcast(
...     pyhf.tensorlib.astensor([1]),
...     pyhf.tensorlib.astensor([2, 3, 4]),
...     pyhf.tensorlib.astensor([5, 6, 7]))
[array([1., 1., 1.]), array([2., 3., 4.]), array([5., 6., 7.])]
```

Parameters *args* (*Array of Tensors*) – Sequence of arrays

Returns The sequence broadcast together.

Return type list of Tensors

sqrt (*tensor_in*)

stack (*sequence*, *axis=0*)

sum (*tensor_in*, *axis=None*)

tile (*tensor_in*, *repeats*)

Repeat tensor data along a specific dimension

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> a = pyhf.tensorlib.astensor([[1.0], [2.0]])
>>> pyhf.tensorlib.tile(a, (1, 2))
array([[1., 1.],
       [2., 2.]])
```

Parameters

- **tensor_in** (tensor) – The tensor to be repeated
- **repeats** (tensor) – The tuple of multipliers for each dimension

Returns The tensor with repeated axes

Return type NumPy ndarray

tolist (*tensor_in*)

where (*mask, tensor_in_1, tensor_in_2*)

zeros (*shape*)

13.4.2 pytorch_backend

class pyhf.tensor.pytorch_backend.**pytorch_backend** (***kwargs*)

Bases: `object`

PyTorch backend for pyhf

Attributes

default_do_grad

dtypesmap

name

precision

Methods

__init__ (***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

abs (*tensor*)

astensor (*tensor_in, dtype='float'*)

Convert to a PyTorch Tensor.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> tensor = pyhf.tensorlib.astensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> tensor
tensor([[1., 2., 3.],
        [4., 5., 6.]])
>>> type(tensor)
<class 'torch.Tensor'>
```

Parameters `tensor_in` (*Number or Tensor*) – Tensor object

Returns A multi-dimensional matrix containing elements of a single data type.

Return type torch.Tensor

boolean_mask (*tensor, mask*)

clip (*tensor_in, min_value, max_value*)

Clips (limits) the tensor values to be within a specified min and max.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> a = pyhf.tensorlib.astensor([-2, -1, 0, 1, 2])
>>> pyhf.tensorlib.clip(a, -1, 1)
tensor([-1., -1., 0., 1., 1.])
```

Parameters

- **tensor_in** (*tensor*) – The input tensor object
- **min_value** (*scalar or tensor or None*) – The minimum value to be clipped to
- **max_value** (*scalar or tensor or None*) – The maximum value to be clipped to

Returns A clipped *tensor*

Return type PyTorch tensor

concatenate (*sequence, axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

- **sequence** – sequence of tensors
- **axis** – dimension along which to concatenate

Returns the concatenated tensor

Return type output

conditional (*predicate, true_callable, false_callable*)

Runs a callable conditional on the boolean value of the evaluation of a predicate

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> tensorlib = pyhf.tensorlib
>>> a = tensorlib.astensor([4])
>>> b = tensorlib.astensor([5])
>>> tensorlib.conditional((a < b)[0], lambda: a + b, lambda: a - b)
tensor([9.])
```

Parameters

- **predicate** (scalar) – The logical condition that determines which callable to evaluate
- **true_callable** (callable) – The callable that is evaluated when the predicate evaluates to true
- **false_callable** (callable) – The callable that is evaluated when the predicate evaluates to false

Returns The output of the callable that was evaluated

Return type PyTorch Tensor

divide (*tensor_in_1*, *tensor_in_2*)

einsum (*subscripts*, **operands*)

This function provides a way of computing multilinear expressions (i.e. sums of products) using the Einstein summation convention.

Parameters

- **subscripts** – str, specifies the subscripts for summation
- **operands** – list of array_like, these are the tensors for the operation

Returns the calculation based on the Einstein summation convention

Return type tensor

erf (*tensor_in*)

The error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erf(a)
tensor([-0.9953, -0.8427,  0.0000,  0.8427,  0.9953])
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the error function at the given points.

Return type PyTorch Tensor

erfinv (*tensor_in*)

The inverse of the error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erfinv(pyhf.tensorlib.erf(a))
tensor([-2.0000, -1.0000,  0.0000,  1.0000,  2.0000])
```

Parameters `tensor_in` (tensor) – The input tensor object

Returns The values of the inverse of the error function at the given points.

Return type PyTorch Tensor

exp (*tensor_in*)

gather (*tensor*, *indices*)

isfinite (*tensor*)

log (*tensor_in*)

normal (*x*, *mu*, *sigma*)

The probability density function of the Normal distribution evaluated at *x* given parameters of mean of *mu* and standard deviation of *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> pyhf.tensorlib.normal(0.5, 0., 1.)
tensor(0.3521)
>>> values = pyhf.tensorlib.astensor([0.5, 2.0])
>>> means = pyhf.tensorlib.astensor([0., 2.3])
>>> sigmas = pyhf.tensorlib.astensor([1., 0.8])
>>> pyhf.tensorlib.normal(values, means, sigmas)
tensor([0.3521, 0.4648])
```

Parameters

- **x** (tensor or float) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns Value of Normal(*x*|*mu*, *sigma*)

Return type PyTorch FloatTensor

normal_cdf (*x*, *mu*=0.0, *sigma*=1.0)

The cumulative distribution function for the Normal distribution

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> pyhf.tensorlib.normal_cdf(0.8)
tensor(0.7881)
>>> values = pyhf.tensorlib.astensor([0.8, 2.0])
>>> pyhf.tensorlib.normal_cdf(values)
tensor([0.7881, 0.9772])
```

Parameters

- **x** (tensor or float) – The observed value of the random variable to evaluate the CDF for
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The CDF

Return type PyTorch FloatTensor

normal_dist (*mu*, *sigma*)

The Normal distribution with mean *mu* and standard deviation *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> normals = pyhf.tensorlib.normal_dist(means, stds)
>>> normals.log_prob(values)
tensor([-1.4189, -2.2258])
```

Parameters

- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The Normal distribution class

Return type PyTorch Normal distribution

normal_logpdf (*x*, *mu*, *sigma*)

ones (*shape*)

outer (*tensor_in_1*, *tensor_in_2*)

poisson (*n*, *lam*)

The continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at *n* given the parameter *lam*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> pyhf.tensorlib.poisson(5., 6.)
tensor(0.1606)
>>> values = pyhf.tensorlib.astensor([5., 9.])
>>> rates = pyhf.tensorlib.astensor([6., 8.])
>>> pyhf.tensorlib.poisson(values, rates)
tensor([0.1606, 0.1241])
```

Parameters

- **n** (tensor or float) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (tensor or float) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continuous approximation to Poisson(nlam)

Return type PyTorch FloatTensor

poisson_dist (*rate*)

The Poisson distribution with rate parameter *rate*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> poissons = pyhf.tensorlib.poisson_dist(rates)
>>> poissons.log_prob(values)
tensor([-1.7403, -2.0869])
```

Parameters **rate** (tensor or float) – The mean of the Poisson distribution (the expected number of events)

Returns The Poisson distribution class

Return type PyTorch Poisson distribution

poisson_logpdf (*n*, *lam*)

power (*tensor_in_1*, *tensor_in_2*)

product (*tensor_in*, *axis=None*)

reshape (*tensor*, *newshape*)

shape (*tensor*)

simple_broadcast (**args*)

Broadcast a sequence of 1 dimensional arrays.

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> pyhf.tensorlib.simple_broadcast(
...     pyhf.tensorlib.astensor([1]),
...     pyhf.tensorlib.astensor([2, 3, 4]),
...     pyhf.tensorlib.astensor([5, 6, 7]))
[tensor([1., 1., 1.]), tensor([2., 3., 4.]), tensor([5., 6., 7.])]
```

Parameters **args** (*Array of Tensors*) – Sequence of arrays

Returns The sequence broadcast together.

Return type list of Tensors

sqrt (*tensor_in*)

stack (*sequence, axis=0*)

sum (*tensor_in, axis=None*)

tile (*tensor_in, repeats*)

Repeat tensor data along a specific dimension

Example

```
>>> import pyhf
>>> pyhf.set_backend("pytorch")
>>> a = pyhf.tensorlib.astensor([[1.0], [2.0]])
>>> pyhf.tensorlib.tile(a, (1, 2))
tensor([[1., 1.],
        [2., 2.]])
```

Parameters

- **tensor_in** (*tensor*) – The tensor to be repeated
- **repeats** (*tensor*) – The tuple of multipliers for each dimension

Returns The tensor with repeated axes

Return type PyTorch tensor

tolist (*tensor_in*)

where (*mask, tensor_in_1, tensor_in_2*)

zeros (*shape*)

13.4.3 tensorflow_backend

class pyhf.tensor.tensorflow_backend.tensorflow_backend(**kwargs)
Bases: `object`

TensorFlow backend for pyhf

Attributes

default_do_grad

dtypesmap

name

precision

Methods

__init__ (**kwargs)
Initialize self. See help(type(self)) for accurate signature.

abs (tensor)

astensor (tensor_in, dtype='float')
Convert to a TensorFlow Tensor.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> tensor = pyhf.tensorlib.astensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> tensor
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> type(tensor)
<class 'tensorflow.python.framework.ops.EagerTensor'>
```

Parameters **tensor_in** (*Number or Tensor*) – Tensor object

Returns A symbolic handle to one of the outputs of a *tf.Operation*.

Return type *tf.Tensor*

boolean_mask (tensor, mask)

clip (tensor_in, min_value, max_value)
Clips (limits) the tensor values to be within a specified min and max.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> a = pyhf.tensorlib.astensor([-2, -1, 0, 1, 2])
>>> t = pyhf.tensorlib.clip(a, -1, 1)
>>> print(t)
tf.Tensor([-1. -1.  0.  1.  1.], shape=(5,), dtype=float32)
```

Parameters

- **tensor_in** (tensor) – The input tensor object
- **min_value** (scalar or tensor or `None`) – The minimum value to be clipped to
- **max_value** (scalar or tensor or `None`) – The maximum value to be clipped to

Returns A clipped *tensor*

Return type TensorFlow Tensor

concatenate (*sequence*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

- **sequence** – sequence of tensors
- **axis** – dimension along which to concatenate

Returns the concatenated tensor

Return type output

conditional (*predicate*, *true_callable*, *false_callable*)

Runs a callable conditional on the boolean value of the evaluation of a predicate

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> tensorlib = pyhf.tensorlib
>>> a = tensorlib.astensor([4])
>>> b = tensorlib.astensor([5])
>>> t = tensorlib.conditional((a < b)[0], lambda: a + b, lambda: a - b)
>>> print(t)
tf.Tensor([9.], shape=(1,), dtype=float32)
```

Parameters

- **predicate** (scalar) – The logical condition that determines which callable to evaluate
- **true_callable** (callable) – The callable that is evaluated when the predicate evaluates to true
- **false_callable** (callable) – The callable that is evaluated when the predicate evaluates to false

Returns The output of the callable that was evaluated

Return type TensorFlow Tensor

divide (*tensor_in_1*, *tensor_in_2*)

einsum (*subscripts*, **operands*)

A generalized contraction between tensors of arbitrary dimension.

This function returns a tensor whose elements are defined by equation, which is written in a shorthand form inspired by the Einstein summation convention.

Parameters

- **subscripts** – str, specifies the subscripts for summation
- **operands** – list of array_like, these are the tensors for the operation

Returns the calculation based on the Einstein summation convention

Return type TensorFlow Tensor

erf (*tensor_in*)

The error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> t = pyhf.tensorlib.erf(a)
>>> print(t)
tf.Tensor([-0.9953223 -0.8427007  0.          0.8427007  0.9953223], shape=(5,
↪), dtype=float32)
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the error function at the given points.

Return type TensorFlow Tensor

erfinv (*tensor_in*)

The inverse of the error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> t = pyhf.tensorlib.erfinv(pyhf.tensorlib.erf(a))
>>> print(t)
tf.Tensor([-2.000001  -0.99999964  0.          0.99999964  1.9999981 ],
↪ shape=(5,), dtype=float32)
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the inverse of the error function at the given points.

Return type TensorFlow Tensor

exp (*tensor_in*)

gather (*tensor*, *indices*)

isfinite (*tensor*)

log (*tensor_in*)

normal (*x, mu, sigma*)

The probability density function of the Normal distribution evaluated at *x* given parameters of mean of *mu* and standard deviation of *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.normal(0.5, 0., 1.)
>>> print(t)
tf.Tensor(0.35206532, shape=(), dtype=float32)
>>> values = pyhf.tensorlib.astensor([0.5, 2.0])
>>> means = pyhf.tensorlib.astensor([0., 2.3])
>>> sigmas = pyhf.tensorlib.astensor([1., 0.8])
>>> t = pyhf.tensorlib.normal(values, means, sigmas)
>>> print(t)
tf.Tensor([0.35206532 0.46481887], shape=(2,), dtype=float32)
```

Parameters

- **x** (tensor or float) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns Value of Normal(*x*|*mu*, *sigma*)

Return type TensorFlow Tensor

normal_cdf (*x, mu=0.0, sigma=1*)

Compute the value of cumulative distribution function for the Normal distribution at *x*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.normal_cdf(0.8)
>>> print(t)
tf.Tensor(0.7881446, shape=(), dtype=float32)
>>> values = pyhf.tensorlib.astensor([0.8, 2.0])
>>> t = pyhf.tensorlib.normal_cdf(values)
>>> print(t)
tf.Tensor([0.7881446 0.97724986], shape=(2,), dtype=float32)
```

Parameters

- **x** (tensor or float) – The observed value of the random variable to evaluate the CDF for
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The CDF

Return type TensorFlow Tensor

normal_dist (*mu*, *sigma*)

Construct a Normal distribution with mean *mu* and standard deviation *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> normals = pyhf.tensorlib.normal_dist(means, stds)
>>> t = normals.log_prob(values)
>>> print(t)
tf.Tensor([-1.4189385 -2.2257915], shape=(2,), dtype=float32)
```

Parameters

- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The Normal distribution class

Return type TensorFlow Probability Normal distribution

normal_logpdf (*x*, *mu*, *sigma*)

The log of the probability density function of the Normal distribution evaluated at *x* given parameters of mean of *mu* and standard deviation of *sigma*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.normal_logpdf(0.5, 0., 1.)
>>> print(t)
tf.Tensor(-1.0439385, shape=(), dtype=float32)
>>> values = pyhf.tensorlib.astensor([0.5, 2.0])
>>> means = pyhf.tensorlib.astensor([0., 2.3])
>>> sigmas = pyhf.tensorlib.astensor([1., 0.8])
>>> t = pyhf.tensorlib.normal_logpdf(values, means, sigmas)
>>> print(t)
tf.Tensor([-1.0439385 -0.7661075], shape=(2,), dtype=float32)
```

Parameters

- **x** (tensor or float) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns Value of log(Normal(*x*|*mu*, *sigma*))

Return type TensorFlow Tensor

ones (*shape*)

outer (*tensor_in_1*, *tensor_in_2*)

poisson (*n*, *lam*)

The continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at *n* given the parameter *lam*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.poisson(5., 6.)
>>> print(t)
tf.Tensor(0.16062315, shape=(), dtype=float32)
>>> values = pyhf.tensorlib.astensor([5., 9.])
>>> rates = pyhf.tensorlib.astensor([6., 8.])
>>> t = pyhf.tensorlib.poisson(values, rates)
>>> print(t)
tf.Tensor([0.16062315 0.12407687], shape=(2,), dtype=float32)
```

Parameters

- **n** (tensor or float) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (tensor or float) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continous approximation to Poisson(n|lam)

Return type TensorFlow Tensor

poisson_dist (*rate*)

Construct a Poisson distribution with rate parameter *rate*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> poissons = pyhf.tensorlib.poisson_dist(rates)
>>> t = poissons.log_prob(values)
>>> print(t)
tf.Tensor([-1.7403021 -2.086854 ], shape=(2,), dtype=float32)
```

Parameters **rate** (tensor or float) – The mean of the Poisson distribution (the expected number of events)

Returns The Poisson distribution class

Return type TensorFlow Probability Poisson distribution

poisson_logpdf (*n*, *lam*)

The log of the continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at *n* given the parameter *lam*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.poisson_logpdf(5., 6.)
>>> print(t)
tf.Tensor(-1.8286943, shape=(), dtype=float32)
>>> values = pyhf.tensorlib.astensor([5., 9.])
>>> rates = pyhf.tensorlib.astensor([6., 8.])
>>> t = pyhf.tensorlib.poisson_logpdf(values, rates)
>>> print(t)
tf.Tensor([-1.8286943 -2.086854 ], shape=(2,), dtype=float32)
```

Parameters

- **n** (tensor or float) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (tensor or float) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continuous approximation to $\log(\text{Poisson}(n|\text{lam}))$

Return type TensorFlow Tensor

power (*tensor_in_1*, *tensor_in_2*)

product (*tensor_in*, *axis=None*)

reshape (*tensor*, *newshape*)

shape (*tensor*)

simple_broadcast (*args)

Broadcast a sequence of 1 dimensional arrays.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> b = pyhf.tensorlib.simple_broadcast(
...     pyhf.tensorlib.astensor([1]),
...     pyhf.tensorlib.astensor([2, 3, 4]),
...     pyhf.tensorlib.astensor([5, 6, 7]))
>>> print([str(t) for t in b])
['tf.Tensor([1. 1. 1.], shape=(3,), dtype=float32)',
 'tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)',
 'tf.Tensor([5. 6. 7.], shape=(3,), dtype=float32)']
```

Parameters **args** (*Array of Tensors*) – Sequence of arrays

Returns The sequence broadcast together.

Return type list of Tensors

sqrt (*tensor_in*)

stack (*sequence*, *axis=0*)

sum (*tensor_in*, *axis=None*)

tile (*tensor_in*, *repeats*)

Repeat tensor data along a specific dimension

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> a = pyhf.tensorlib.astensor([[1.0], [2.0]])
>>> t = pyhf.tensorlib.tile(a, (1, 2))
>>> print(t)
tf.Tensor(
[[1. 1.]
 [2. 2.]], shape=(2, 2), dtype=float32)
```

Parameters

- **tensor_in** (*tensor*) – The tensor to be repeated
- **repeats** (*tensor*) – The tuple of multipliers for each dimension

Returns The tensor with repeated axes

Return type TensorFlow Tensor

tolist (*tensor_in*)

where (*mask*, *tensor_in_1*, *tensor_in_2*)

Apply a boolean selection mask to the elements of the input tensors.

Example

```
>>> import pyhf
>>> pyhf.set_backend("tensorflow")
>>> t = pyhf.tensorlib.where(
...     pyhf.tensorlib.astensor([1, 0, 1], dtype='bool'),
...     pyhf.tensorlib.astensor([1, 1, 1]),
...     pyhf.tensorlib.astensor([2, 2, 2]),
... )
>>> print(t)
tf.Tensor([1. 2. 1.], shape=(3,), dtype=float32)
```

Parameters

- **mask** (*bool*) – Boolean mask (boolean or tensor object of booleans)
- **tensor_in_1** (*Tensor*) – Tensor object
- **tensor_in_2** (*Tensor*) – Tensor object

Returns The result of the mask being applied to the tensors.

Return type TensorFlow Tensor

zeros (*shape*)

13.4.4 jax_backend

class pyhf.tensor.jax_backend.**jax_backend**(**kwargs)

Bases: `object`

JAX backend for pyhf

Attributes

default_do_grad

dtypesmap

name

precision

Methods

__init__(**kwargs)

Initialize self. See help(type(self)) for accurate signature.

abs(*tensor*)

astensor(*tensor_in*, *dtype*='float')

Convert to a JAX ndarray.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> tensor = pyhf.tensorlib.astensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
>>> tensor
DeviceArray([[1., 2., 3.],
             [4., 5., 6.]], dtype=float64)
>>> type(tensor)
<class 'jax.interpreters.xla.DeviceArray'>
```

Parameters **tensor_in** (*Number or Tensor*) – Tensor object

Returns A multi-dimensional, fixed-size homogenous array.

Return type *jax.interpreters.xla.DeviceArray*

boolean_mask(*tensor*, *mask*)

clip(*tensor_in*, *min_value*, *max_value*)

Clips (limits) the tensor values to be within a specified min and max.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> a = pyhf.tensorlib.astensor([-2, -1, 0, 1, 2])
>>> pyhf.tensorlib.clip(a, -1, 1)
DeviceArray([-1., -1.,  0.,  1.,  1.], dtype=float64)
```

Parameters

- **tensor_in** (tensor) – The input tensor object
- **min_value** (scalar or tensor or `None`) – The minimum value to be clipped to
- **max_value** (scalar or tensor or `None`) – The maximum value to be clipped to

Returns A clipped *tensor*

Return type JAX ndarray

concatenate (*sequence*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

- **sequence** – sequence of tensors
- **axis** – dimension along which to concatenate

Returns the concatenated tensor

Return type output

conditional (*predicate*, *true_callable*, *false_callable*)

Runs a callable conditional on the boolean value of the evaluation of a predicate

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> tensorlib = pyhf.tensorlib
>>> a = tensorlib.astensor([4])
>>> b = tensorlib.astensor([5])
>>> tensorlib.conditional((a < b)[0], lambda: a + b, lambda: a - b)
DeviceArray([9.], dtype=float64)
```

Parameters

- **predicate** (scalar) – The logical condition that determines which callable to evaluate
- **true_callable** (callable) – The callable that is evaluated when the predicate evaluates to true
- **false_callable** (callable) – The callable that is evaluated when the predicate evaluates to false

Returns The output of the callable that was evaluated

Return type JAX ndarray

divide (*tensor_in_1*, *tensor_in_2*)

einsum (*subscripts*, **operands*)

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations. The best way to understand this function is to try the examples below, which show how many common NumPy functions can be implemented as calls to einsum.

Parameters

- **subscripts** – str, specifies the subscripts for summation
- **operands** – list of array_like, these are the tensors for the operation

Returns the calculation based on the Einstein summation convention

Return type tensor

erf (*tensor_in*)

The error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erf(a)
DeviceArray([-0.99532227, -0.84270079,  0.          ,  0.84270079,
              0.99532227], dtype=float64)
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the error function at the given points.

Return type JAX ndarray

erfinv (*tensor_in*)

The inverse of the error function of complex argument.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> a = pyhf.tensorlib.astensor([-2., -1., 0., 1., 2.])
>>> pyhf.tensorlib.erfinv(pyhf.tensorlib.erf(a))
DeviceArray([-2., -1.,  0.,  1.,  2.], dtype=float64)
```

Parameters **tensor_in** (tensor) – The input tensor object

Returns The values of the inverse of the error function at the given points.

Return type JAX ndarray

exp (*tensor_in*)

gather (*tensor*, *indices*)

isfinite (*tensor*)

`log(tensor_in)`

`normal(x, mu, sigma)`

The probability density function of the Normal distribution evaluated at x given parameters of mean of μ and standard deviation of σ .

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> pyhf.tensorlib.normal(0.5, 0., 1.)
DeviceArray(0.35206533, dtype=float64)
>>> values = pyhf.tensorlib.astensor([0.5, 2.0])
>>> means = pyhf.tensorlib.astensor([0., 2.3])
>>> sigmas = pyhf.tensorlib.astensor([1., 0.8])
>>> pyhf.tensorlib.normal(values, means, sigmas)
DeviceArray([0.35206533, 0.46481887], dtype=float64)
```

Parameters

- **x** (tensor or float) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns Value of Normal($x|\mu, \sigma$)

Return type JAX ndarray

`normal_cdf(x, mu=0, sigma=1)`

The cumulative distribution function for the Normal distribution

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> pyhf.tensorlib.normal_cdf(0.8)
DeviceArray(0.7881446, dtype=float64)
>>> values = pyhf.tensorlib.astensor([0.8, 2.0])
>>> pyhf.tensorlib.normal_cdf(values)
DeviceArray([0.7881446 , 0.97724987], dtype=float64)
```

Parameters

- **x** (tensor or float) – The observed value of the random variable to evaluate the CDF for
- **mu** (tensor or float) – The mean of the Normal distribution
- **sigma** (tensor or float) – The standard deviation of the Normal distribution

Returns The CDF

Return type JAX ndarray

`normal_dist(mu, sigma)`

The Normal distribution with mean μ and standard deviation σ .

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> means = pyhf.tensorlib.astensor([5, 8])
>>> stds = pyhf.tensorlib.astensor([1, 0.5])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> normals = pyhf.tensorlib.normal_dist(means, stds)
>>> normals.log_prob(values)
DeviceArray([-1.41893853, -2.22579135], dtype=float64)
```

Parameters

- **mu** (tensor or `float`) – The mean of the Normal distribution
- **sigma** (tensor or `float`) – The standard deviation of the Normal distribution

Returns The Normal distribution class

Return type Normal distribution

normal_logpdf (*x*, *mu*, *sigma*)

ones (*shape*)

outer (*tensor_in_1*, *tensor_in_2*)

poisson (*n*, *lam*)

The continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at *n* given the parameter *lam*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> pyhf.tensorlib.poisson(5., 6.)
DeviceArray(0.16062314, dtype=float64)
>>> values = pyhf.tensorlib.astensor([5., 9.])
>>> rates = pyhf.tensorlib.astensor([6., 8.])
>>> pyhf.tensorlib.poisson(values, rates)
DeviceArray([0.16062314, 0.12407692], dtype=float64)
```

Parameters

- **n** (tensor or `float`) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (tensor or `float`) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continous approximation to Poisson(*n*/*lam*)

Return type JAX ndarray

poisson_dist (*rate*)

The Poisson distribution with rate parameter *rate*.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> rates = pyhf.tensorlib.astensor([5, 8])
>>> values = pyhf.tensorlib.astensor([4, 9])
>>> poissons = pyhf.tensorlib.poisson_dist(rates)
>>> poissons.log_prob(values)
DeviceArray([-1.74030218, -2.0868536 ], dtype=float64)
```

Parameters *rate* (tensor or float) – The mean of the Poisson distribution (the expected number of events)

Returns The Poisson distribution class

Return type Poisson distribution

poisson_logpdf (*n*, *lam*)

power (*tensor_in_1*, *tensor_in_2*)

product (*tensor_in*, *axis=None*)

reshape (*tensor*, *newshape*)

shape (*tensor*)

simple_broadcast (**args*)

Broadcast a sequence of 1 dimensional arrays.

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> pyhf.tensorlib.simple_broadcast(
...     pyhf.tensorlib.astensor([1]),
...     pyhf.tensorlib.astensor([2, 3, 4]),
...     pyhf.tensorlib.astensor([5, 6, 7]))
[DeviceArray([1., 1., 1.], dtype=float64), DeviceArray([2., 3., 4.],
dtype=float64), DeviceArray([5., 6., 7.], dtype=float64)]
```

Parameters *args* (*Array of Tensors*) – Sequence of arrays

Returns The sequence broadcast together.

Return type list of Tensors

sqr (*tensor_in*)

stack (*sequence*, *axis=0*)

sum (*tensor_in*, *axis=None*)

tile (*tensor_in*, *repeats*)

Repeat tensor data along a specific dimension

Example

```
>>> import pyhf
>>> pyhf.set_backend("jax")
>>> a = pyhf.tensorlib.astensor([[1.0], [2.0]])
>>> pyhf.tensorlib.tile(a, (1, 2))
DeviceArray([[1., 1.],
             [2., 2.]], dtype=float64)
```

Parameters

- **tensor_in** (tensor) – The tensor to be repeated
- **repeats** (tensor) – The tuple of multipliers for each dimension

Returns The tensor with repeated axes

Return type JAX ndarray

tolist (*tensor_in*)

where (*mask, tensor_in_1, tensor_in_2*)

zeros (*shape*)

13.5 Optimizers

<i>mixins.OptimizerMixin</i>	Mixin Class to build optimizers.
<i>opt_scipy.scipy_optimizer</i>	Optimizer that uses <code>scipy.optimize.minimize()</code> .
<i>opt_minuit.minuit_optimizer</i>	Optimizer that uses <code>iminuit.Minuit.migrad</code> .

13.5.1 OptimizerMixin

class `pyhf.optimize.mixins.OptimizerMixin` (***kwargs*)

Bases: `object`

Mixin Class to build optimizers.

Attributes

maxiter

verbose

Methods

__init__ (***kwargs*)
Create an optimizer.

Parameters

- **maxiter** (*int*) – maximum number of iterations. Default is 100000.
- **verbose** (*int*) – verbose output level during minimization. Default is off (0).

minimize (*objective, data, pdf, init_pars, par_bounds, fixed_vals=None, return_fitted_val=False, return_result_obj=False, do_grad=None, do_stitch=False, **kwargs*)
Find parameters that minimize the objective.

Parameters

- **objective** (*func*) – objective function
- **data** (*list*) – observed data
- **pdf** (*Model*) – The statistical model adhering to the schema model.json
- **init_pars** (*list*) – initial parameters
- **par_bounds** (*list*) – parameter boundaries
- **fixed_vals** (*list*) – fixed parameter values
- **return_fitted_val** (*bool*) – return bestfit value of the objective
- **return_result_obj** (*bool*) – return `scipy.optimize.OptimizeResult`
- **do_grad** (*bool*) – enable autodifferentiation mode. Default depends on backend (`pyhf.tensorlib.default_do_grad`).
- **do_stitch** (*bool*) – enable splicing/stitching fixed parameter.
- **kwargs** – other options to pass through to underlying minimizer

Returns

- **parameters** (*tensor*): fitted parameters
- **minimum** (*float*): if `return_fitted_val` flagged, return minimized objective value
- **result** (*scipy.optimize.OptimizeResult*): if `return_result_obj` flagged

Return type Fitted parameters or tuple of results

13.5.2 scipy_optimizer

class pyhf.optimize.opt_scipy.**scipy_optimizer**(*args, **kwargs)

Bases: *pyhf.optimize.mixins.OptimizerMixin*

Optimizer that uses `scipy.optimize.minimize()`.

Attributes

maxiter

name

verbose

Methods

__init__(*args, **kwargs)

Initialize the scipy_optimizer.

See *pyhf.optimize.mixins.OptimizerMixin* for configuration options.

13.5.3 minuit_optimizer

class pyhf.optimize.opt_minuit.**minuit_optimizer**(*args, **kwargs)

Bases: *pyhf.optimize.mixins.OptimizerMixin*

Optimizer that uses `iminuit.Minuit.migrad`.

Attributes

errordef

maxiter

name

steps

verbose

Methods

__init__(*args, **kwargs)

Create MINUIT Optimizer.

Note: `errordef` should be 1.0 for a least-squares cost function and 0.5 for negative log-likelihood function. See page 37 of <http://hep.fi.infn.it/minuit.pdf>. This parameter is sometimes called UP in the MINUIT docs.

Parameters

- **errordef** (*float*) – See minuit docs. Default is 1.0.

- **steps** (`int`) – Number of steps for the bounds. Default is 1000.

13.6 Modifiers

histosys

normfactor

normsys

shapefactor

shapesys

staterror

13.6.1 histosys

```
class pyhf.modifiers.histosys
```

```
    Bases: object
```

Attributes

```
    is_constrained = True
```

```
    op_code = 'addition'
```

```
    pdf_type = 'normal'
```

Methods

```
    classmethod required_parset (sample_data, modifier_data)
```

13.6.2 normfactor

```
class pyhf.modifiers.normfactor
```

```
    Bases: object
```

Attributes

```
    is_constrained = False
```

```
    op_code = 'multiplication'
```

```
    pdf_type = None
```

Methods

`classmethod required_parset` (*sample_data*, *modifier_data*)

13.6.3 normsys

`class pyhf.modifiers.normsys`
Bases: `object`

Attributes

`is_constrained` = `True`
`op_code` = `'multiplication'`
`pdf_type` = `'normal'`

Methods

`classmethod required_parset` (*sample_data*, *modifier_data*)

13.6.4 shapefactor

`class pyhf.modifiers.shapefactor`
Bases: `object`

Attributes

`is_constrained` = `False`
`op_code` = `'multiplication'`
`pdf_type` = `None`

Methods

`classmethod required_parset` (*sample_data*, *modifier_data*)

13.6.5 shapesys

`class pyhf.modifiers.shapesys`
Bases: `object`

Attributes

```
is_constrained = True
op_code = 'multiplication'
pdf_type = 'poisson'
```

Methods

```
classmethod required_parset (sample_data, modifier_data)
```

13.6.6 staterror

```
class pyhf.modifiers.staterror
    Bases: object
```

Attributes

```
is_constrained = True
op_code = 'multiplication'
pdf_type = 'normal'
```

Methods

```
classmethod required_parset (sample_data, modifier_data)
```

13.7 Interpolators

<code>code0</code>	The piecewise-linear interpolation strategy.
<code>code1</code>	The piecewise-exponential interpolation strategy.
<code>code2</code>	The quadratic interpolation and linear extrapolation strategy.
<code>code4</code>	The polynomial interpolation and exponential extrapolation strategy.
<code>code4p</code>	The piecewise-linear interpolation strategy, with polynomial at $ a < 1$.

13.7.1 code0

class pyhf.interpolators.**code0** (*histogramssets, subscribe=True*)
 Bases: `object`

The piecewise-linear interpolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) + \underbrace{\sum_{p \in \text{Syst}} I_{\text{lin.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-)}_{\text{deltas to calculate}}$$

with

$$I_{\text{lin.}}(\alpha; I^0, I^+, I^-) = \begin{cases} \alpha(I^+ - I^0) & \alpha \geq 0 \\ \alpha(I^0 - I^-) & \alpha < 0 \end{cases}$$

Methods

__init__ (*histogramssets, subscribe=True*)
 Piecewise-linear Interpolation.

13.7.2 code1

class pyhf.interpolators.**code1** (*histogramssets, subscribe=True*)
 Bases: `object`

The piecewise-exponential interpolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) \underbrace{\prod_{p \in \text{Syst}} I_{\text{exp.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-)}_{\text{factors to calculate}}$$

with

$$I_{\text{exp.}}(\alpha; I^0, I^+, I^-) = \begin{cases} \left(\frac{I^+}{I^0}\right)^\alpha & \alpha \geq 0 \\ \left(\frac{I^-}{I^0}\right)^{-\alpha} & \alpha < 0 \end{cases}$$

Methods

__init__ (*histogramssets, subscribe=True*)
 Piecewise-Exponential Interpolation.

13.7.3 code2

class pyhf.interpolators.**code2** (*histogramssets, subscribe=True*)
 Bases: `object`

The quadratic interpolation and linear extrapolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) + \underbrace{\sum_{p \in \text{Syst}} I_{\text{quad.lin.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-)}_{\text{deltas to calculate}}$$

with

$$I_{\text{quad.llin.}}(\alpha; I^0, I^+, I^-) = \begin{cases} (b + 2a)(\alpha - 1) & \alpha \geq 1 \\ a\alpha^2 + b\alpha & |\alpha| < 1 \\ (b - 2a)(\alpha + 1) & \alpha < -1 \end{cases}$$

and

$$a = \frac{1}{2}(I^+ + I^-) - I^0 \quad \text{and} \quad b = \frac{1}{2}(I^+ - I^-)$$

Methods

`__init__` (*histogramssets*, *subscribe=True*)
Quadratic Interpolation.

13.7.4 code4

class `pyhf.interpolators.code4` (*histogramssets*, *subscribe=True*, *alpha0=1*)
Bases: `object`

The polynomial interpolation and exponential extrapolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) \underbrace{\prod_{p \in \text{Syst}} I_{\text{polylexp.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-, \alpha_0)}_{\text{factors to calculate}}$$

with

$$I_{\text{polylexp.}}(\alpha; I^0, I^+, I^-, \alpha_0) = \begin{cases} \left(\frac{I^+}{I^0}\right)^\alpha & \alpha \geq \alpha_0 \\ 1 + \sum_{i=1}^6 a_i \alpha^i & |\alpha| < \alpha_0 \\ \left(\frac{I^-}{I^0}\right)^{-\alpha} & \alpha < -\alpha_0 \end{cases}$$

and the a_i are fixed by the boundary conditions

$$\sigma_{sb}(\alpha = \pm\alpha_0), \left. \frac{d\sigma_{sb}}{d\alpha} \right|_{\alpha=\pm\alpha_0}, \text{ and } \left. \frac{d^2\sigma_{sb}}{d\alpha^2} \right|_{\alpha=\pm\alpha_0}.$$

Namely that $\sigma_{sb}(\vec{\alpha})$ is continuous, and its first- and second-order derivatives are continuous as well.

Methods

`__init__` (*histogramssets*, *subscribe=True*, *alpha0=1*)
Polynomial Interpolation.

13.7.5 code4p

class `pyhf.interpolators.code4p` (*histogramssets*, *subscribe=True*)
Bases: `object`

The piecewise-linear interpolation strategy, with polynomial at $|a| < 1$.

$$\sigma_{sb}(\vec{\alpha}) = \sigma_{sb}^0(\vec{\alpha}) + \underbrace{\sum_{p \in \text{Syst}} I_{\text{lin.}}(\alpha_p; \sigma_{sb}^0, \sigma_{psb}^+, \sigma_{psb}^-)}_{\text{deltas to calculate}}$$

Methods

`__init__` (*histogramssets*, *subscribe=True*)
 Piecewise-Linear + Polynomial Interpolation.

13.8 Inference

<code>test_statistics.qmu(mu, data, pdf, ...)</code>	The test statistic, q_μ , for establishing an upper limit on the strength parameter, μ , as defined in Equation (14) in [1007.1727]
<code>test_statistics.qmu_tilde(mu, data, pdf, ...)</code>	The test statistic, \tilde{q}_μ , for establishing an upper limit on the strength parameter, μ , for models with bounded POI, as defined in Equation (16) in [1007.1727]
<code>test_statistics.tmu(mu, data, pdf, ...)</code>	The test statistic, t_μ , for establishing a two-sided interval on the strength parameter, μ , as defined in Equation (8) in [1007.1727]
<code>test_statistics.tmu_tilde(mu, data, pdf, ...)</code>	The test statistic, \tilde{t}_μ , for establishing a two-sided interval on the strength parameter, μ , for models with bounded POI, as defined in Equation (11) in [1007.1727]
<code>mle.twice_nll(pars, data, pdf)</code>	Two times the negative log-likelihood of the model parameters, (μ, θ) , given the observed data.
<code>mle.fit(data, pdf[, init_pars, par_bounds, ...])</code>	Run a maximum likelihood fit.
<code>mle.fixed_poi_fit(poi_val, data, pdf[, ...])</code>	Run a maximum likelihood fit with the POI value fixed.
<code>hypotest(poi_test, data, pdf[, init_pars, ...])</code>	Compute p -values and test statistics for a single value of the parameter of interest.
<code>calculators.generate_asimov_data(asimov_n, data, pdf, ...)</code>	Compute Asimov Dataset (expected yields at best-fit values) for a given POI value.
<code>calculators.AsymptoticTestStatDistribution(k, data, pdf)</code>	The distribution of the test statistic in the asymptotic case.
<code>calculators.AsymptoticCalculator(data, pdf)</code>	The Asymptotic Calculator.
<code>intervals.upperlimit(data, model, scan[, ...])</code>	Calculate an upper limit interval $(0, \text{poi_up})$ for a single Parameter of Interest (POI) using a fixed scan through POI-space.

13.8.1 pyhf.infer.test_statistics.qmu

`pyhf.infer.test_statistics.qmu(mu, data, pdf, init_pars, par_bounds, fixed_params)`

The test statistic, q_μ , for establishing an upper limit on the strength parameter, μ , as defined in Equation (14) in [1007.1727]

$$q_\mu = \begin{cases} -2 \ln \lambda(\mu), & \hat{\mu} < \mu, \\ 0, & \hat{\mu} > \mu \end{cases} \quad (13.1)$$

where $\lambda(\mu)$ is the profile likelihood ratio as defined in Equation (7)

$$\lambda(\mu) = \frac{L(\mu, \hat{\theta})}{L(\hat{\mu}, \hat{\theta})}.$$

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_mu = 1.0
>>> init_pars = model.config.suggested_init()
>>> par_bounds = model.config.suggested_bounds()
>>> par_bounds[model.config.poi_index] = [-10.0, 10.0]
>>> fixed_params = model.config.suggested_fixed()
>>> pyhf.infer.test_statistics.qmu(test_mu, data, model, init_pars, par_bounds,
↳ fixed_params)
array(3.9549891)
```

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*Tensor*) – The data to be considered
- **pdf** (*Model*) – The HistFactory statistical model used in the likelihood ratio calculation
- **init_pars** (*list*) – Values to initialize the model parameters at for the fit
- **par_bounds** (*list of lists or tuples*) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (*list*) – Parameters held constant in the fit

Returns The calculated test statistic, q_μ

Return type Float

13.8.2 pyhf.infer.test_statistics.qmu_tilde

`pyhf.infer.test_statistics.qmu_tilde(mu, data, pdf, init_pars, par_bounds, fixed_params)`

The test statistic, \tilde{q}_μ , for establishing an upper limit on the strength parameter, μ , for models with bounded POI, as defined in Equation (16) in [1007.1727]

$$\tilde{q}_\mu = \begin{cases} -2 \ln \tilde{\lambda}(\mu), & \hat{\mu} < \mu, \\ 0, & \hat{\mu} \geq \mu \end{cases} \quad (13.2)$$

where $\tilde{\lambda}(\mu)$ is the constrained profile likelihood ratio as defined in Equation (10)

$$\tilde{\lambda}(\mu) = \begin{cases} \frac{L(\mu, \hat{\boldsymbol{\theta}}(\mu))}{L(\hat{\mu}, \hat{\boldsymbol{\theta}}(0))}, & \hat{\mu} < \mu, \\ \frac{L(\mu, \hat{\boldsymbol{\theta}}(\mu))}{L(\hat{\mu}, \hat{\boldsymbol{\theta}})}, & \hat{\mu} \geq \mu. \end{cases} \quad (13.3)$$

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_mu = 1.0
>>> init_pars = model.config.suggested_init()
>>> par_bounds = model.config.suggested_bounds()
>>> fixed_params = model.config.suggested_fixed()
>>> pyhf.infer.test_statistics.qmu_tilde(test_mu, data, model, init_pars, par_
↳ bounds, fixed_params)
array(3.93824492)
```

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*tensor*) – The data to be considered
- **pdf** (*Model*) – The statistical model adhering to the schema model.json
- **init_pars** (*list*) – Values to initialize the model parameters at for the fit
- **par_bounds** (*list of lists or tuples*) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (*list*) – Parameters held constant in the fit

Returns The calculated test statistic, \tilde{q}_μ

Return type Float

13.8.3 pyhf.infer.test_statistics.tmu

`pyhf.infer.test_statistics.tmu(mu, data, pdf, init_pars, par_bounds, fixed_params)`

The test statistic, t_μ , for establishing a two-sided interval on the strength parameter, μ , as defined in Equation (8) in [1007.1727]

$$t_\mu = -2 \ln \lambda(\mu)$$

where $\lambda(\mu)$ is the profile likelihood ratio as defined in Equation (7)

$$\lambda(\mu) = \frac{L(\mu, \hat{\hat{\theta}})}{L(\hat{\mu}, \hat{\theta})}.$$

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_mu = 1.0
>>> init_pars = model.config.suggested_init()
>>> par_bounds = model.config.suggested_bounds()
>>> par_bounds[model.config.poi_index] = [-10.0, 10.0]
>>> fixed_params = model.config.suggested_fixed()
>>> pyhf.infer.test_statistics.tmu(test_mu, data, model, init_pars, par_bounds,
↳ fixed_params)
array(3.9549891)
```

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*Tensor*) – The data to be considered
- **pdf** (*Model*) – The statistical model adhering to the schema model.json
- **init_pars** (*list*) – Values to initialize the model parameters at for the fit
- **par_bounds** (*list of lists or tuples*) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (*list*) – Parameters held constant in the fit

Returns The calculated test statistic, t_μ

Return type Float

13.8.4 pyhf.infer.test_statistics.tmu_tilde

`pyhf.infer.test_statistics.tmu_tilde(mu, data, pdf, init_pars, par_bounds, fixed_params)`

The test statistic, \tilde{t}_μ , for establishing a two-sided interval on the strength parameter, μ , for models with bounded POI, as defined in Equation (11) in [1007.1727]

$$\tilde{t}_\mu = -2 \ln \tilde{\lambda}(\mu)$$

where $\tilde{\lambda}(\mu)$ is the constrained profile likelihood ratio as defined in Equation (10)

$$\tilde{\lambda}(\mu) = \begin{cases} \frac{L(\mu, \hat{\boldsymbol{\theta}}(\mu))}{L(\hat{\mu}, \hat{\boldsymbol{\theta}}(0))}, & \hat{\mu} < 0, \\ \frac{L(\mu, \hat{\boldsymbol{\theta}}(\mu))}{L(\hat{\mu}, \hat{\boldsymbol{\theta}})}, & \hat{\mu} \geq 0. \end{cases} \quad (13.4)$$

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_mu = 1.0
>>> init_pars = model.config.suggested_init()
>>> par_bounds = model.config.suggested_bounds()
>>> fixed_params = model.config.suggested_fixed()
>>> pyhf.infer.test_statistics.tmu_tilde(test_mu, data, model, init_pars, par_
↳ bounds, fixed_params)
array(3.93824492)
```

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*tensor*) – The data to be considered
- **pdf** (*Model*) – The statistical model adhering to the schema model.json
- **init_pars** (*list*) – Values to initialize the model parameters at for the fit
- **par_bounds** (*list of lists or tuples*) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (*list*) – Parameters held constant in the fit

Returns The calculated test statistic, \tilde{t}_μ

Return type Float

13.8.5 pyhf.infer.mle.twice_nll

`pyhf.infer.mle.twice_nll(pars, data, pdf)`

Two times the negative log-likelihood of the model parameters, (μ, θ) , given the observed data. It is used in the calculation of the test statistic, t_μ , as defined in Equation (8) in [1007.1727]

$$t_\mu = -2 \ln \lambda(\mu)$$

where $\lambda(\mu)$ is the profile likelihood ratio as defined in Equation (7)

$$\lambda(\mu) = \frac{L(\mu, \hat{\hat{\theta}})}{L(\hat{\mu}, \hat{\theta})}.$$

It serves as the objective function to minimize in `fit()` and `fixed_poi_fit()`.

Example

```

>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> parameters = model.config.suggested_init() # nominal parameters
>>> twice_nll = pyhf.infer.mle.twice_nll(parameters, data, model)
>>> twice_nll
array([30.77525435])
>>> -2 * model.logpdf(parameters, data) == twice_nll
array([ True])

```

Parameters

- **pars** (tensor) – The parameters of the HistFactory model
- **data** (tensor) – The data to be considered
- **pdf** (`Model`) – The statistical model adhering to the schema model.json

Returns Two times the negative log-likelihood, $-2 \ln L(\mu, \theta)$

Return type Tensor

13.8.6 pyhf.infer.mle.fit

`pyhf.infer.mle.fit` (*data*, *pdf*, *init_pars=None*, *par_bounds=None*, *fixed_params=None*, ***kwargs*)

Run a maximum likelihood fit. This is done by minimizing the objective function `twice_nll()` of the model parameters given the observed data. This is used to produce the maximal likelihood $L(\hat{\mu}, \hat{\theta})$ in the profile likelihood ratio in Equation (7) in [1007.1727]

$$\lambda(\mu) = \frac{L(\mu, \hat{\theta})}{L(\hat{\mu}, \hat{\theta})}$$

Note: `twice_nll()` is the objective function given to the optimizer and is returned evaluated at the best fit model parameters when the optional kwarg `return_fitted_val` is `True`.

Example

```

>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> bestfit_pars, twice_nll = pyhf.infer.mle.fit(data, model, return_fitted_
↪ val=True)

```

(continues on next page)

(continued from previous page)

```
>>> bestfit_pars
array([0.          , 1.0030512 , 0.96266961])
>>> twice_nll
array(24.98393521)
>>> -2 * model.logpdf(bestfit_pars, data) == twice_nll
array([ True])
```

Parameters

- **data** (tensor) – The data
- **pdf** (Model) – The statistical model adhering to the schema model.json
- **init_pars** (list) – Values to initialize the model parameters at for the fit
- **par_bounds** (list of lists or tuples) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (list) – Parameters to be held constant in the fit.
- **kwargs** – Keyword arguments passed through to the optimizer API

Returns See optimizer API

13.8.7 pyhf.infer.mle.fixed_poi_fit

`pyhf.infer.mle.fixed_poi_fit(poi_val, data, pdf, init_pars=None, par_bounds=None, fixed_params=None, **kwargs)`

Run a maximum likelihood fit with the POI value fixed. This is done by minimizing the objective function of `twice_nll()` of the model parameters given the observed data, for a given fixed value of μ . This is used to produce the constrained maximal likelihood for the given μ $L(\mu, \hat{\theta})$ in the profile likelihood ratio in Equation (7) in [1007.1727]

$$\lambda(\mu) = \frac{L(\mu, \hat{\theta})}{L(\hat{\mu}, \hat{\theta})}$$

Note: `twice_nll()` is the objective function given to the optimizer and is returned evaluated at the best fit model parameters when the optional kwarg `return_fitted_val` is `True`.

Example

```
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_poi = 1.0
>>> bestfit_pars, twice_nll = pyhf.infer.mle.fixed_poi_fit(
...     test_poi, data, model, return_fitted_val=True
```

(continues on next page)

(continued from previous page)

```

... )
>>> bestfit_pars
array([1.          , 0.97224597, 0.87553894])
>>> twice_nll
array(28.92218013)
>>> -2 * model.logpdf(bestfit_pars, data) == twice_nll
array([ True])

```

Parameters

- **data** – The data
- **pdf** (*Model*) – The statistical model adhering to the schema `model.json`
- **init_pars** (*list*) – Values to initialize the model parameters at for the fit
- **par_bounds** (*list of lists or tuples*) – The extrema of values the model parameters are allowed to reach in the fit
- **fixed_params** (*list*) – Parameters to be held constant in the fit.
- **kwargs** – Keyword arguments passed through to the optimizer API

Returns See optimizer API**13.8.8 pyhf.infer.hypotest**

`pyhf.infer.hypotest` (*poi_test*, *data*, *pdf*, *init_pars=None*, *par_bounds=None*, *fixed_params=None*, *qtilde=False*, ***kwargs*)

Compute *p*-values and test statistics for a single value of the parameter of interest.

Example

```

>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> test_poi = 1.0
>>> CLs_obs, CLs_exp_band = pyhf.infer.hypotest(
...     test_poi, data, model, qtilde=True, return_expected_set=True
... )
>>> CLs_obs
array(0.05251497)
>>> CLs_exp_band
[array(0.00260626), array(0.01382005), array(0.06445321), array(0.23525644),
↪ array(0.57303621)]

```

Parameters

- **poi_test** (*Number or Tensor*) – The value of the parameter of interest (POI)
- **data** (*Number or Tensor*) – The data considered
- **pdf** (*Model*) – The statistical model adhering to the schema `model.json`

- **init_pars** (tensor) – The initial parameter values to be used for minimization
- **par_bounds** (tensor) – The parameter value bounds to be used for minimization
- **fixed_params** (tensor) – Whether to fix the parameter to the init_pars value during minimization
- **qtilde** (bool) – When True perform the calculation using the alternative test statistic, \tilde{q}_μ , as defined under the Wald approximation in Equation (62) of [1007.1727].

Keyword Arguments

- **return_tail_probs** (bool) – Bool for returning CL_{s+b} and CL_b
- **return_expected** (bool) – Bool for returning CL_{exp}
- **return_expected_set** (bool) – Bool for returning the $(-2, -1, 0, 1, 2)\sigma$ CL_{exp} — the “Brazil band”

Returns

- CL_s : The modified p -value compared to the given threshold α , typically taken to be 0.05, defined in [1007.1727] as

$$CL_s = \frac{CL_{s+b}}{CL_b} = \frac{p_{s+b}}{1 - p_b}$$

to protect against excluding signal models in which there is little sensitivity. In the case that $CL_s \leq \alpha$ the given signal model is excluded.

- $[CL_{s+b}, CL_b]$: The signal + background model hypothesis p -value

$$CL_{s+b} = p_{s+b} = p(q \geq q_{\text{obs}} | s + b) = \int_{q_{\text{obs}}}^{\infty} f(q | s + b) dq = 1 - F(q_{\text{obs}}(\mu) | \mu')$$

and 1 minus the background only model hypothesis p -value

$$CL_b = 1 - p_b = p(q \geq q_{\text{obs}} | b) = 1 - \int_{-\infty}^{q_{\text{obs}}} f(q | b) dq = 1 - F(q_{\text{obs}}(\mu) | 0)$$

for signal strength μ and model hypothesis signal strength μ' , where the cumulative density functions $F(q(\mu) | \mu')$ are given by Equations (57) and (65) of [1007.1727] for upper-limit-like test statistic $q \in \{q_\mu, \tilde{q}_\mu\}$. Only returned when `return_tail_probs` is True.

Note: The definitions of the CL_{s+b} and CL_b used are based on profile likelihood ratio test statistics. This procedure is common in the LHC-era, but differs from procedures used in the LEP and Tevatron eras, as briefly discussed in § 3.8 of [1007.1727].

- $CL_{s,\text{exp}}$: The expected CL_s value corresponding to the test statistic under the background only hypothesis ($\mu = 0$). Only returned when `return_expected` is True.
- $CL_{s,\text{exp}}$ band: The set of expected CL_s values corresponding to the median significance of variations of the signal strength from the background only hypothesis ($\mu = 0$) at $(-2, -1, 0, 1, 2)\sigma$. That is, the p -values that satisfy Equation (89) of [1007.1727]

$$\text{band}_{N\sigma} = \mu' + \sigma \Phi^{-1}(1 - \alpha) \pm N\sigma$$

for $\mu' = 0$ and $N \in \{-2, -1, 0, 1, 2\}$. These values define the boundaries of an uncertainty band sometimes referred to as the “Brazil band”. Only returned when `return_expected_set` is True.

Return type Tuple of Floats and lists of Floats

13.8.9 pyhf.infer.calculators.generate_asimov_data

`pyhf.infer.calculators.generate_asimov_data(asimov_mu, data, pdf, init_pars, par_bounds, fixed_params)`

Compute Asimov Dataset (expected yields at best-fit values) for a given POI value.

Parameters

- **asimov_mu** (`float`) – The value for the parameter of interest to be used.
- **data** (`tensor`) – The observed data.
- **pdf** (`Model`) – The statistical model adhering to the schema `model.json`.
- **init_pars** (`tensor`) – The initial parameter values to be used for fitting.
- **par_bounds** (`tensor`) – The parameter value bounds to be used for fitting.
- **fixed_params** (`tensor`) – Parameters to be held constant in the fit.

Returns The Asimov dataset.

Return type Tensor

13.8.10 pyhf.infer.calculators.AsymptoticTestStatDistribution

class `pyhf.infer.calculators.AsymptoticTestStatDistribution(shift)`

The distribution the test statistic in the asymptotic case.

Note: These distributions are in $-\hat{\mu}/\sigma$ space. In the ROOT implementation the same sigma is assumed for both hypotheses and p -values etc are computed in that space. This assumption is necessarily valid, but we keep this for compatibility reasons.

In the $-\hat{\mu}/\sigma$ space, the test statistic (i.e. $\hat{\mu}/\sigma$) is normally distributed with unit variance and its mean at the $-\mu'$, where μ' is the true poi value of the hypothesis.

__init__ (*shift*)

Asymptotic test statistic distribution.

Parameters **shift** (`float`) – The displacement of the test statistic distribution.

Returns The asymptotic distribution of test statistic.

Return type *AsymptoticTestStatDistribution*

Methods

<code>__init__(shift)</code>	Asymptotic test statistic distribution.
<code>cdf(value)</code>	Compute the value of the cumulative distribution function for a given value of the test statistic.
<code>expected_value(nsigma)</code>	Return the expected value of the test statistic.
<code>pvalue(value)</code>	The p -value for a given value of the test statistic corresponding to signal strength μ and Asimov strength μ' as defined in Equations (59) and (57) of [1007.1727]

13.8.11 pyhf.infer.calculators.AsymptoticCalculator

class pyhf.infer.calculators.**AsymptoticCalculator** (*data*, *pdf*, *init_pars=None*,
par_bounds=None,
fixed_params=None, *qtilde=False*)

The Asymptotic Calculator.

__init__ (*data*, *pdf*, *init_pars=None*, *par_bounds=None*, *fixed_params=None*, *qtilde=False*)
 Asymptotic Calculator.

Parameters

- **data** (tensor) – The observed data.
- **pdf** (Model) – The statistical model adhering to the schema `model.json`.
- **init_pars** (tensor) – The initial parameter values to be used for fitting.
- **par_bounds** (tensor) – The parameter value bounds to be used for fitting.
- **fixed_params** (tensor) – Whether to fix the parameter to the `init_pars` value during minimization
- **qtilde** (bool) – Whether to use `qtilde` as the test statistic.

Returns The calculator for asymptotic quantities.

Return type *AsymptoticCalculator*

Methods

__init__ (<i>data</i> , <i>pdf</i> [, <i>init_pars</i> , <i>par_bounds</i> , ...])	Asymptotic Calculator.
distributions (<i>poi_test</i>)	Probability Distributions of the test statistic value under the signal + background and background-only hypothesis.
teststatistic (<i>poi_test</i>)	Compute the test statistic for the observed data under the studied model.

13.8.12 pyhf.infer.intervals.upperlimit

pyhf.infer.intervals.**upperlimit** (*data*, *model*, *scan*, *level=0.05*, *return_results=False*)

Calculate an upper limit interval (0, `poi_up`) for a single Parameter of Interest (POI) using a fixed scan through POI-space.

Example

```
>>> import numpy as np
>>> import pyhf
>>> pyhf.set_backend("numpy")
>>> model = pyhf.simplemodels.hepdata_like(
...     signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0]
... )
>>> observations = [51, 48]
>>> data = pyhf.tensorlib.astensor(observations + model.config.auxdata)
>>> scan = np.linspace(0, 5, 21)
```

(continues on next page)

(continued from previous page)

```

>>> obs_limit, exp_limits, (scan, results) = pyhf.infer.intervals.upperlimit(
...     data, model, scan, return_results=True
... )
>>> obs_limit
array(1.01764089)
>>> exp_limits
[array(0.59576921), array(0.76169166), array(1.08504773), array(1.50170482),
↪ array(2.06654952)]

```

Parameters

- **data** (tensor) – The observed data.
- **model** (`Model`) – The statistical model adhering to the schema `model.json`.
- **scan** (iterable) – Iterable of POI values.
- **level** (float) – The threshold value to evaluate the interpolated results at.
- **return_results** (bool) – Whether to return the per-point results.

Returns

- Tensor: The observed upper limit on the POI.
- Tensor: The expected upper limits on the POI.
- Tuple of Tensors: The given `scan` along with the *hypotest* results at each test POI. Only returned when `return_results` is `True`.

Return type Tuple of Tensors

13.9 Exceptions

Various exceptions, apart from standard python exceptions, that are raised from using the pyhf API.

<i>InvalidMeasurement</i>	InvalidMeasurement is raised when a specified measurement is invalid given the specification.
<i>InvalidNameReuse</i>	
<i>InvalidSpecification</i>	InvalidSpecification is raised when a specification does not validate against the given schema.
<i>InvalidPatchSet</i>	InvalidPatchSet is raised when a given patchset object does not have the right configuration, even though it validates correctly against the schema.
<i>InvalidPatchLookup</i>	InvalidPatchLookup is raised when the patch lookup from a patchset object has failed
<i>PatchSetVerificationError</i>	PatchSetVerificationError is raised when the workspace digest does not match the patchset digests as part of the verification procedure
<i>InvalidWorkspaceOperation</i>	InvalidWorkspaceOperation is raised when an operation on a workspace fails.
<i>InvalidModel</i>	InvalidModel is raised when a given model does not have the right configuration, even though it validates correctly against the schema.

continues on next page

Table 11 – continued from previous page

<i>InvalidModifier</i>	InvalidModifier is raised when an invalid modifier is requested.
<i>InvalidInterpCode</i>	InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.
<i>ImportBackendError</i>	MissingLibraries is raised when something is imported by sustained an import error due to missing additional, non-default libraries.
<i>InvalidBackend</i>	InvalidBackend is raised when trying to set a backend that does not exist.
<i>InvalidOptimizer</i>	InvalidOptimizer is raised when trying to set an optimizer that does not exist.
<i>InvalidPdfParameters</i>	InvalidPdfParameters is raised when trying to evaluate a pdf with invalid parameters.
<i>InvalidPdfData</i>	InvalidPdfData is raised when trying to evaluate a pdf with invalid data.

13.9.1 InvalidMeasurement

```
class pyhf.exceptions.InvalidMeasurement
```

```
    Bases: Exception
```

InvalidMeasurement is raised when a specified measurement is invalid given the specification.

13.9.2 InvalidNameReuse

```
class pyhf.exceptions.InvalidNameReuse
```

```
    Bases: Exception
```

13.9.3 InvalidSpecification

```
class pyhf.exceptions.InvalidSpecification (ValidationError, schema=None)
```

```
    Bases: Exception
```

InvalidSpecification is raised when a specification does not validate against the given schema.

13.9.4 InvalidPatchSet

```
class pyhf.exceptions.InvalidPatchSet
```

```
    Bases: Exception
```

InvalidPatchSet is raised when a given patchset object does not have the right configuration, even though it validates correctly against the schema.

13.9.5 InvalidPatchLookup

```
class pyhf.exceptions.InvalidPatchLookup
    Bases: Exception
```

InvalidPatchLookup is raised when the patch lookup from a patchset object has failed

13.9.6 PatchSetVerificationError

```
class pyhf.exceptions.PatchSetVerificationError
    Bases: Exception
```

PatchSetVerificationError is raised when the workspace digest does not match the patchset digests as part of the verification procedure

13.9.7 InvalidWorkspaceOperation

```
class pyhf.exceptions.InvalidWorkspaceOperation
    Bases: Exception
```

InvalidWorkspaceOperation is raised when an operation on a workspace fails.

13.9.8 InvalidModel

```
class pyhf.exceptions.InvalidModel
    Bases: Exception
```

InvalidModel is raised when a given model does not have the right configuration, even though it validates correctly against the schema.

This can occur, for example, when the provided parameter of interest to fit against does not get declared in the specification provided.

13.9.9 InvalidModifier

```
class pyhf.exceptions.InvalidModifier
    Bases: Exception
```

InvalidModifier is raised when an invalid modifier is requested. This includes:

- creating a custom modifier with the wrong structure
- initializing a modifier that does not exist, or has not been loaded

13.9.10 InvalidInterpCode

```
class pyhf.exceptions.InvalidInterpCode
    Bases: Exception
```

InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.

13.9.11 ImportError

class pyhf.exceptions.**ImportBackendError**
Bases: `Exception`

`MissingLibraries` is raised when something is imported by sustained an import error due to missing additional, non-default libraries.

13.9.12 InvalidBackend

class pyhf.exceptions.**InvalidBackend**
Bases: `Exception`

`InvalidBackend` is raised when trying to set a backend that does not exist.

13.9.13 InvalidOptimizer

class pyhf.exceptions.**InvalidOptimizer**
Bases: `Exception`

`InvalidOptimizer` is raised when trying to set an optimizer that does not exist.

13.9.14 InvalidPdfParameters

class pyhf.exceptions.**InvalidPdfParameters**
Bases: `Exception`

`InvalidPdfParameters` is raised when trying to evaluate a pdf with invalid parameters.

13.9.15 InvalidPdfData

class pyhf.exceptions.**InvalidPdfData**
Bases: `Exception`

`InvalidPdfData` is raised when trying to evaluate a pdf with invalid data.

13.10 Utilities

`load_schema(schema_id[, version])`

`validate(spec, schema_name[, version])`

`options_from_eqdelimstring(opts)`

`digest(obj[, algorithm])`

Get the digest for the provided object.

13.10.1 pyhf.utils.load_schema

`pyhf.utils.load_schema(schema_id, version=None)`

13.10.2 pyhf.utils.validate

`pyhf.utils.validate(spec, schema_name, version=None)`

13.10.3 pyhf.utils.options_from_eqdelimstring

`pyhf.utils.options_from_eqdelimstring(opts)`

13.10.4 pyhf.utils.digest

`pyhf.utils.digest(obj, algorithm='sha256')`

Get the digest for the provided object. Note: object must be JSON-serializable.

The hashing algorithms supported are in `hashlib`, part of Python's Standard Libraries.

Example

```
>>> import pyhf
>>> obj = {'a': 2.0, 'b': 3.0, 'c': 1.0}
>>> pyhf.utils.digest(obj)
'a38f6093800189b79bc22ef677baf90c75705af2cfc7ff594159eca54eaa7928'
>>> pyhf.utils.digest(obj, algorithm='md5')
'2c0633f242928eb55c3672fed5ba8612'
>>> pyhf.utils.digest(obj, algorithm='sha1')
'49a27f499e763766c9545b294880df277be6f545'
```

Raises `ValueError` – If the object is not JSON-serializable or if the algorithm is not supported.

Parameters

- **obj** (`jsonable`) – A JSON-serializable object to compute the digest of. Usually a `Workspace` object.
- **algorithm** (`str`) – The hashing algorithm to use.

Returns The digest for the JSON-serialized object provided and hash algorithm specified.

Return type `digest` (`str`)

13.11 Contrib

<code>viz.brazil</code>	Brazil Band Plots.
<code>utils.download(archive_url, output_directory)</code>	Download the patchset archive from the remote URL and extract it in a directory at the path given.

13.11.1 pyhf.contrib.viz.brazil

Brazil Band Plots.

Functions

<code>plot_results(ax, mutests, tests[, test_size])</code>	Plot a series of hypothesis tests for various POI values.
--	---

13.11.2 pyhf.contrib.utils.download

`pyhf.contrib.utils.download(archive_url, output_directory, force=False, compress=False)`
Download the patchset archive from the remote URL and extract it in a directory at the path given.

Example

```
>>> from pyhf.contrib.utils import download
>>> download("https://doi.org/10.17182/hepdata.90607.v3/r3", "1Lbb-likelihoods")
>>> import os
>>> sorted(os.listdir("1Lbb-likelihoods"))
['BkgOnly.json', 'README.md', 'patchset.json']
>>> download("https://doi.org/10.17182/hepdata.90607.v3/r3", "1Lbb-likelihoods.
↳tar.gz", compress=True)
>>> import glob
>>> glob.glob("1Lbb-likelihoods.tar.gz")
['1Lbb-likelihoods.tar.gz']
```

Parameters

- **archive_url** (`str`) – The URL of the *PatchSet* archive to download.
- **output_directory** (`str`) – Name of the directory to unpack the archive into.
- **force** (`bool`) – Force download from non-approved host. Default is `False`.
- **compress** (`bool`) – Keep the archive in a compressed `tar.gz` form. Default is `False`.

Raises `InvalidArchiveHost` – if the provided archive host name is not known to be valid

USE AND CITATIONS

14.1 Citation

The preferred BibTeX entry for citation of pyhf is

```
@software{pyhf,  
  author = "{Heinrich, Lukas and Feickert, Matthew and Stark, Giordon}",  
  title = "{pyhf: v0.5.3}",  
  version = {0.5.3},  
  doi = {10.5281/zenodo.1169739},  
  url = {https://github.com/scikit-hep/pyhf},  
}
```

14.2 Use in Publications

Updating list of citations and use cases of pyhf:

- Waleed Abdallah and others. Reinterpretation of LHC Results for New Physics: Status and Recommendations after Run 2. 2020. [arXiv:2003.07868](#).
- Gaël Alguero, Sabine Kraml, and Wolfgang Waltenberger. A SModelS interface for pyhf likelihoods. Sep 2020. [arXiv:2009.01809](#).
- J. Alison and others. Higgs Boson Pair Production at Colliders: Status and Perspectives. In B. Di Micco, M. Gouzevitch, J. Mazzitelli, and C. Vernieri, editors, *Double Higgs Production at Colliders*. 9 2019. [arXiv:1910.00012](#).
- B.C. Allanach, Tyler Corbett, and Maeve Madigan. Sensitivity of Future Hadron Colliders to Leptoquark Pair Production in the Di-Muon Di-Jets Channel. *Eur. Phys. J. C*, 80(2):170, 2020. [arXiv:1911.04455](#), [doi:10.1140/epjc/s10052-020-7722-3](#).
- Andrei Angelescu, Darius A. Faroughy, and Olcyr Sumensari. Lepton Flavor Violation and Dilepton Tails at the LHC. *Eur. Phys. J. C*, 80(7):641, 2020. [arXiv:2002.05684](#), [doi:10.1140/epjc/s10052-020-8210-5](#).
- Johann Brehmer, Felix Kling, Irina Espejo, and Kyle Cranmer. MadMiner: Machine learning-based inference for particle physics. *Comput. Softw. Big Sci.*, 4(1):3, 2020. [arXiv:1907.10621](#), [doi:10.1007/s41781-020-0035-2](#).
- G. Brooijmans and others. Les Houches 2019 Physics at TeV Colliders: New Physics Working Group Report. In 2020. [arXiv:2002.12220](#).
- Lukas Heinrich, Holger Schulz, Jessica Turner, and Ye-Ling Zhou. Constraining A_4 Leptonic Flavour Model Parameters at Colliders and Beyond. 2018. [arXiv:1810.05648](#).

- Charanjit K. Khosa, Sabine Kraml, Andre Lessa, Philipp Neuhuber, and Wolfgang Waltenberger. SModelS database update v1.2.3. *LHEP*, 158:2020, 5 2020. [arXiv:2005.00555](https://arxiv.org/abs/2005.00555), doi:10.31526/lhep.2020.158.
- Jeffrey Krupa and others. GPU coprocessors as a service for deep learning inference in high energy physics. 7 2020. [arXiv:2007.10359](https://arxiv.org/abs/2007.10359).
- ATLAS Collaboration. Reproducing searches for new physics with the ATLAS experiment through publication of full statistical likelihoods. Geneva, Aug 2019. URL: <https://cds.cern.ch/record/2684863>.

14.3 Published Likelihoods

Updating list of HEPData entries for publications using HistFactory JSON likelihoods:

- Search for bottom-squark pair production with the ATLAS detector in final states containing Higgs bosons, $b\bar{b}$ -jets and missing transverse momentum. 2019. URL: <https://doi.org/10.17182/hepdata.89408>, doi:10.17182/hepdata.89408.
- Search for chargino-neutralino production with mass splittings near the electroweak scale in three-lepton final states in $\sqrt{s} = 13$ TeV pp collisions with the ATLAS detector. 2019. URL: <https://doi.org/10.17182/hepdata.91127>, doi:10.17182/hepdata.91127.
- Search for direct stau production in events with two hadronic -leptons in $\sqrt{s} = 13$ TeV pp collisions with the ATLAS detector. 2019. URL: <https://doi.org/10.17182/hepdata.92006>, doi:10.17182/hepdata.92006.
- Search for direct production of electroweakinos in final states with one lepton, missing transverse momentum and a Higgs boson decaying into two $b\bar{b}$ -jets in (pp) collisions at $\sqrt{s}=13$ TeV with the ATLAS detector. 2020. URL: <https://doi.org/10.17182/hepdata.90607.v2>, doi:10.17182/hepdata.90607.v2.
- Search for squarks and gluinos in final states with same-sign leptons and jets using 139 fb⁻¹ of data collected with the ATLAS detector. 2020. URL: <https://doi.org/10.17182/hepdata.91214.v3>, doi:10.17182/hepdata.91214.v3.

ROADMAP (2019-2020)

This is the pyhf 2019 into 2020 Roadmap (Issue #561).

15.1 Overview and Goals

We will follow loosely Seibert's Hierarchy of Needs



(Stan Seibert, SciPy 2019)

As a general overview that will include:

- Improvements to docs

- Add lots of examples
- Add at least 5 well documented case studies
- Issue cleanup
- Adding core feature support
- “pyhf evolution”: integration with columnar data analysis systems
- GPU support and testing
- Publications
 - Submit pyhf to JOSS
 - Submit pyhf to pyOpenSci
 - Start pyhf paper in 2020
- Align with IRIS-HEP Analysis Systems NSF milestones

15.2 Time scale

The roadmap will be executed over mostly Quarter 3 of 2019 through Quarter 1 of 2020, with some projects continuing into Quarter 2 of 2020

- 2019-Q3
- 2019-Q4
- 2020-Q1
- (2020-Q2)

15.3 Roadmap

1. Documentation and Deployment

- Add docstrings to all functions and classes (Issues #38, #349) [2019-Q3]
- [Greatly revise and expand examples](#) (Issues #168, #202, #212, #325, #342, #349, #367) [2019-Q3 → 2019-Q4]
 - Add small case studies with published sbottom likelihood from HEPData
- Move to [scikit-hep](#) GitHub organization [2019-Q3]
- Develop a release schedule/criteria [2019-Q4]
- Automate deployment with [STRIKEOUT:Azure pipeline (talk with Henry Schreiner) (Issue #517)] GitHub Actions (Issue #508) [2019-Q3]
- Finalize logo and add it to website (Issue #453) [2019-Q3 → 2019-Q4]
- Write submission to [JOSS](#) (Issue #502) and write submission to [pyOpenSci](#) [2019-Q4 → 2020-Q2]
- Contribute to [IRIS-HEP Analysis Systems Milestones](#) “Initial roadmap for ecosystem coherency” and “Initial roadmap for high-level cyberinfrastructure components of analysis system” [2019-Q4 → 2020-Q2]

2. Revision and Maintenance

- Add tests using HEPData published sbottom likelihoods (Issue #518) [2019-Q3]
- Add CI with GitHub Actions and Azure Pipelines (PR #527, Issue #517) [2019-Q3]
- Investigate rewrite of pytest fixtures to use modern pytest (Issue #370) [2019-Q3 → 2019-Q4]
- Factorize out the statistical fitting portion into `pyhf.infer` (PR #531) [2019-Q3 → 2019-Q4]
- Bug squashing at large [2019-Q3 → 2020-Q2]
 - Unexpected use cases (Issues #324, #325, #529)
 - Computational edge cases (Issues #332, #445)
- Make sure that all backends reproduce sbottom results [2019-Q4 → 2020-Q2]

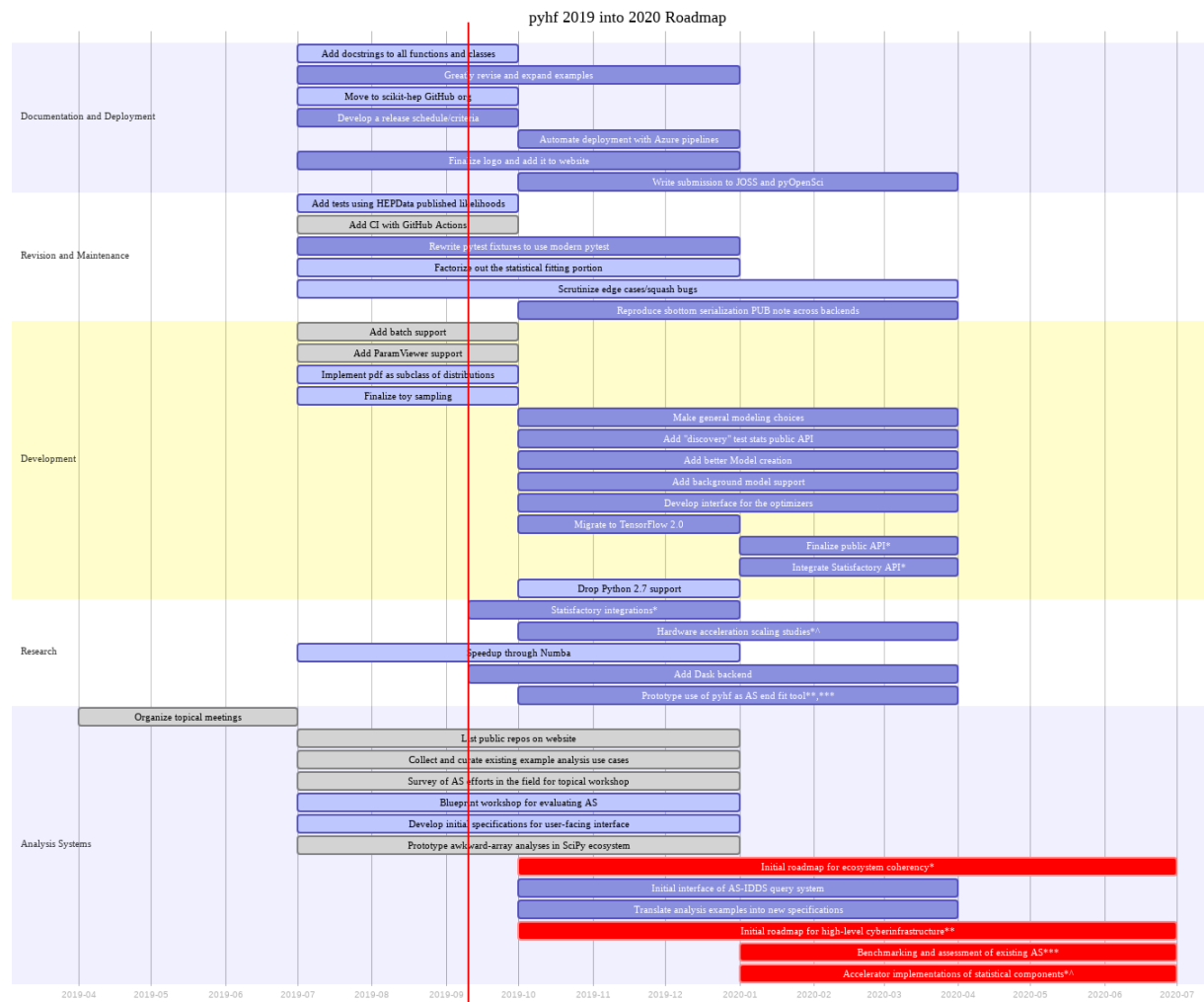
3. Development

- Batch support (PR #503) [2019-Q3]
- Add ParamViewer support (PR #519) [2019-Q3]
- Add setting of NPs constant/fixed (PR #653) [2019-Q3]
- Implement pdf as subclass of distributions (PR #551) [2019-Q3]
- Add sampling with toys (PR #558) [2019-Q3]
- Make general modeling choices (e.g., Issue #293) [2019-Q4 → 2020-Q1]
- Add “discovery” test stats (p_0) (PR #520) [2019-Q4 → 2020-Q1]
- Add better Model creation [2019-Q4 → 2020-Q1]
- Add background model support (Issues #514, #946) [2019-Q4 → 2020-Q1]
- Develop interface for the optimizers similar to tensor/backend (Issue #754, PR #951) [2019-Q4 → 2020-Q1]
- Migrate to TensorFlow v2.0 (PR #541) [2019-Q4]
- Drop Python 2.7 support at end of 2019 (Issue #469) [2019-Q4 (last week of December 2019)]
- Finalize public API [2020-Q1]
- Integrate pyfitcore/Statsfactory API [2020-Q1]

4. Research

- Add pyfitcore/Statsfactory integrations (Issue #344, [zfit Issue 120](#)) [2019-Q4]
- Hardware acceleration scaling studies (Issues #93, #301) [2019-Q4 → 2020-Q1]
- Speedup through Numba (Issue #364) [2019-Q3 → 2019-Q4]
- Dask backend (Issue #259) [2019-Q3 → 2020-Q1]
- Attempt to use pyhf as fitting tool for full Analysis Systems pipeline test in early 2020 [2019-Q4 → 2020-Q1]
- pyhf should satisfy [IRIS-HEP Analysis Systems Milestone “GPU/accelerator-based implementation of statistical and other appropriate components”](#) [2020-Q1 → 2020-Q2] and contributes to [“Benchmarking and assessment of prototype analysis system components”](#) [2020-Q3 → 2020-Q4]

15.3.1 Roadmap as Gantt Chart



15.4 Presentations During Roadmap Timeline

- Talk at IRIS-HEP Institute Retreat (September 12-13th, 2019)
- Talk at PyHEP 2019 (October 16-18th, 2019)
- Talk at CHEP 2019 (November 4-8th, 2019)
- Poster at CHEP 2019 (November 4-8th, 2019)

CONTRIBUTORS

`pyhf` is openly developed and benefits from the contributions and feedback from its users. The `pyhf` dev team would like to thank all contributors to the project for their support and help. Thank you!

Contributors include:

- Jessica Forde
- Ruggero Turra
- Tadej Novak
- Frank Sauerburger
- Lars Nielsen
- Kanishk Kalra
- Nikolai Hartmann
- Alexander Held
- Karthikeyan Singaravelan



PURE-PYTHON FITTING/LIMIT-SETTING/INTERVAL ESTIMATION HISTFACTORY-STYLE

The HistFactory p.d.f. template [CERN-OPEN-2012-016] is per-se independent of its implementation in ROOT and sometimes, it's useful to be able to run statistical analysis outside of ROOT, RooFit, RooStats framework.

This repo is a pure-python implementation of that statistical model for multi-bin histogram-based analysis and its interval estimation is based on the asymptotic formulas of “Asymptotic formulae for likelihood-based tests of new physics” [arXiv:1007.1727]. The aim is also to support modern computational graph libraries such as PyTorch and TensorFlow in order to make use of features such as autodifferentiation and GPU acceleration.

17.1 Hello World

This is how you use the pyhf Python API to build a statistical model and run basic inference:

```
>>> import pyhf
>>> model = pyhf.simplemodels.hepdata_like(signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0], bkg_uncerts=[3.0, 7.0])
>>> data = [51, 48] + model.config.auxdata
>>> test_mu = 1.0
>>> CLs_obs, CLs_exp = pyhf.infer.hypotest(test_mu, data, model, qtilde=True, return_expected=True)
>>> print(f"Observed: {CLs_obs}, Expected: {CLs_exp}")
Observed: 0.05251497423736956, Expected: 0.06445320535890459
```

Alternatively the statistical model and observational data can be read from its serialized JSON representation (see next section).

```
>>> import pyhf
>>> import requests
>>> wspace = pyhf.Workspace(requests.get('https://git.io/JJYDE').json())
>>> model = wspace.model()
>>> data = wspace.data(model)
>>> test_mu = 1.0
>>> CLs_obs, CLs_exp = pyhf.infer.hypotest(test_mu, data, model, qtilde=True, return_expected=True)
```

(continues on next page)

(continued from previous page)

```
>>> print(f"Observed: {CLs_obs}, Expected: {CLs_exp}")
Observed: 0.3599840922126626, Expected: 0.3599840922126626
```

Finally, you can also use the command line interface that pyhf provides which should produce the following JSON output:

```
$ cat << EOF | tee likelihood.json | pyhf cls
{
  "channels": [
    { "name": "singlechannel",
      "samples": [
        { "name": "signal",
          "data": [12.0, 11.0],
          "modifiers": [ { "name": "mu", "type": "normfactor", "data": null} ]
        },
        { "name": "background",
          "data": [50.0, 52.0],
          "modifiers": [ { "name": "uncorr_bkguncrt", "type": "shapesys", "data": ↵
↵[3.0, 7.0]} ]
        }
      ]
    }
  ],
  "observations": [
    { "name": "singlechannel", "data": [51.0, 48.0] }
  ],
  "measurements": [
    { "name": "Measurement", "config": { "poi": "mu", "parameters": []} }
  ],
  "version": "1.0.0"
}
EOF
{
  "CLs_exp": [
    0.0026062609501074576,
    0.01382005356161206,
    0.06445320535890459,
    0.23525643861460702,
    0.573036205919389
  ],
  "CLs_obs": 0.05251497423736956
}
```

17.2 What does it support

Implemented variations:

- HistoSys
- OverallSys
- ShapeSys
- NormFactor
- Multiple Channels

- Import from XML + ROOT via `uproot`
- ShapeFactor
- StatError
- Lumi Uncertainty

Computational Backends:

- NumPy
- PyTorch
- TensorFlow
- JAX

Optimizers:

- SciPy (`scipy.optimize`)
- MINUIT (`iminuit`)

All backends can be used in combination with all optimizers. Custom user backends and optimizers can be used as well.

17.3 Todo

- StatConfig
- Non-asymptotic calculators

results obtained from this package are validated against output computed from HistFactory workspaces

17.4 A one bin example

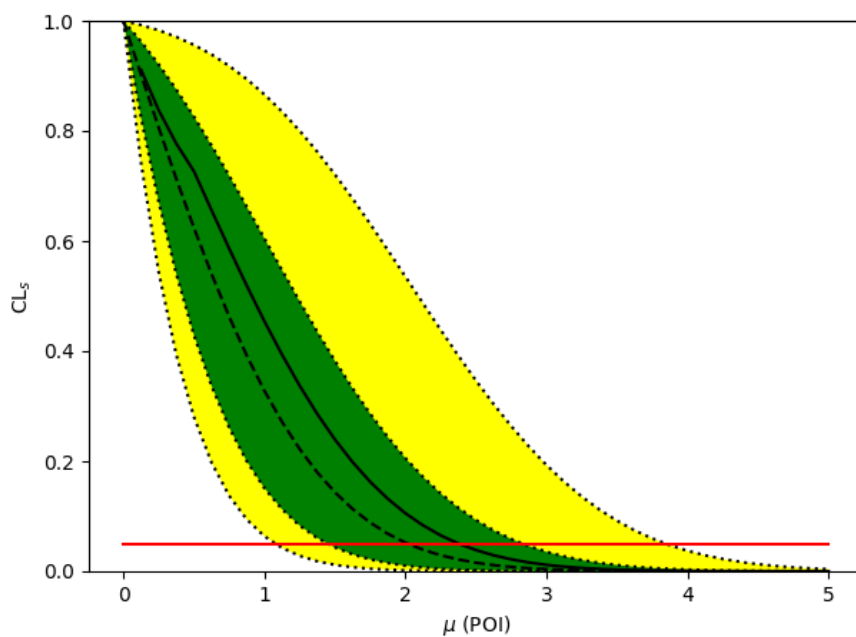
```
import pyhf
import numpy as np
import matplotlib.pyplot as plt
import pyhf.contrib.viz.brazil

pyhf.set_backend("numpy")
model = pyhf.simplemodels.hepdata_like(
    signal_data=[10.0], bkg_data=[50.0], bkg_uncerts=[7.0]
)
data = [55.0] + model.config.auxdata

poi_vals = np.linspace(0, 5, 41)
results = [
    pyhf.infer.hypotest(test_poi, data, model, qtilde=True, return_expected_set=True)
    for test_poi in poi_vals
]

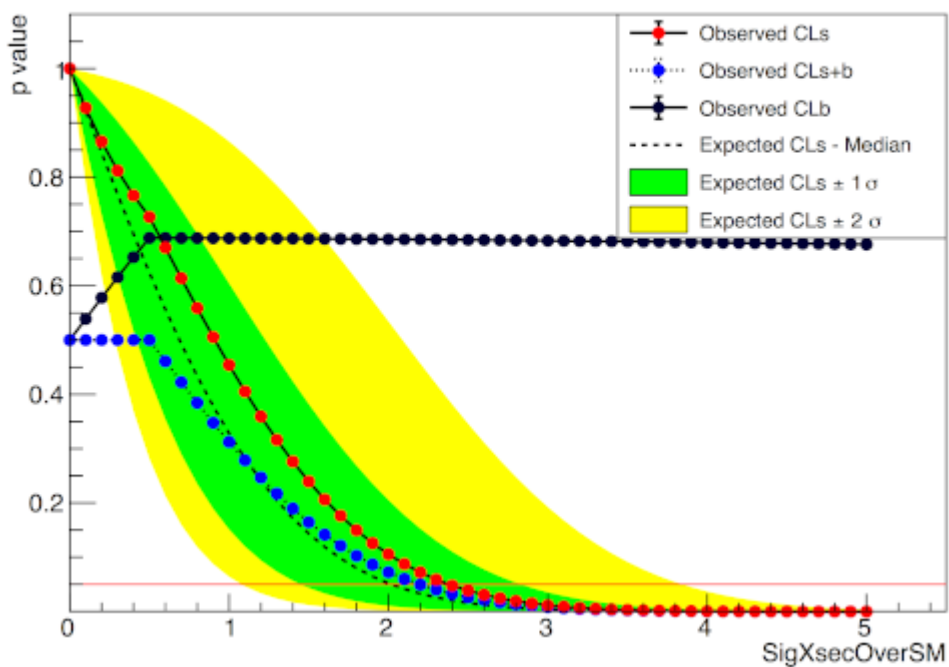
fig, ax = plt.subplots()
fig.set_size_inches(7, 5)
ax.set_xlabel(r"$\mu$ (POI)")
ax.set_ylabel(r"$\mathrm{CL}_{s}$")
pyhf.contrib.viz.brazil.plot_results(ax, poi_vals, results)
```

pyhf



ROOT

plot



17.5 A two bin example

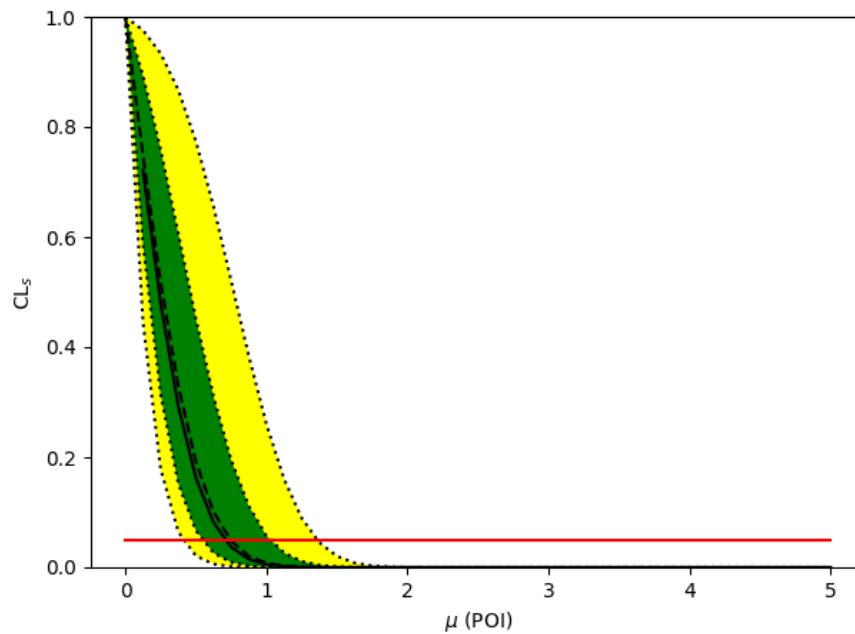
```
import pyhf
import numpy as np
import matplotlib.pyplot as plt
import pyhf.contrib.viz.brazil

pyhf.set_backend("numpy")
model = pyhf.simplemodels.hepdata_like(
    signal_data=[30.0, 45.0], bkg_data=[100.0, 150.0], bkg_uncerts=[15.0, 20.0]
)
data = [100.0, 145.0] + model.config.auxdata

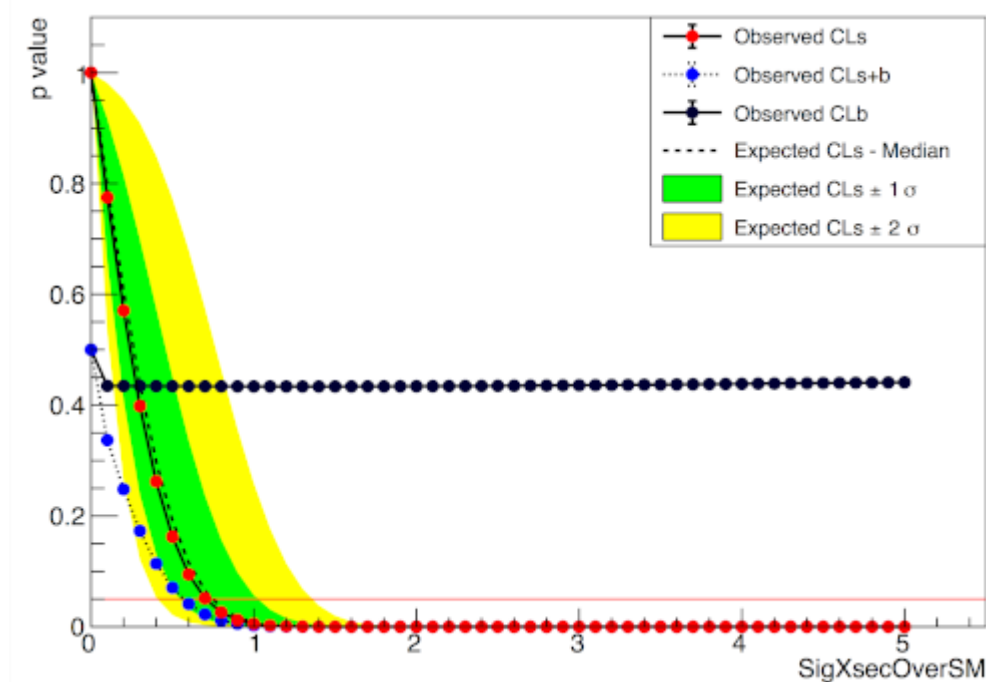
poi_vals = np.linspace(0, 5, 41)
results = [
    pyhf.infer.hypotest(test_poi, data, model, qtilde=True, return_expected_set=True)
    for test_poi in poi_vals
]

fig, ax = plt.subplots()
fig.set_size_inches(7, 5)
ax.set_xlabel(r"$\mu$ (POI)")
ax.set_ylabel(r"$\mathrm{CL}_{\mathrm{s}}$")
pyhf.contrib.viz.brazil.plot_results(ax, poi_vals, results)
```

pyhf



ROOT



17.6 Installation

To install `pyhf` from PyPI with the NumPy backend run

```
python -m pip install pyhf
```

and to install `pyhf` with all additional backends run

```
python -m pip install pyhf[backends]
```

or a subset of the options.

To uninstall run

```
python -m pip uninstall pyhf
```

17.7 Questions

If you have a question about the use of `pyhf` not covered in [the documentation](#), please ask a question on [Stack Overflow](#) with the `[pyhf]` tag, which the `pyhf` dev team watches.



If you believe you have found a bug in `pyhf`, please report it in the [GitHub Issues](#). If you're interested in getting updates from the `pyhf` dev team and release announcements you can join the [pyhf-announcements mailing list](#).

17.8 Citation

As noted in [Use and Citations](#), the preferred BibTeX entry for citation of pyhf is

```
@software{pyhf,  
  author = "{Heinrich, Lukas and Feickert, Matthew and Stark, Giordon}",  
  title = "{pyhf: v0.5.3}",  
  version = {0.5.3},  
  doi = {10.5281/zenodo.1169739},  
  url = {https://github.com/scikit-hep/pyhf},  
}
```

17.9 Authors

pyhf is openly developed by Lukas Heinrich, Matthew Feickert, and Giordon Stark.

Please check the [contribution statistics](#) for a list of contributors.

17.10 Milestones

- 2020-07-28: 1000 GitHub issues and pull requests. (See [PR #1000](#))

17.11 Acknowledgements

Matthew Feickert has received support to work on pyhf provided by NSF cooperative agreement [OAC-1836650](#) (IRIS-HEP) and grant [OAC-1450377](#) (DIANA/HEP).

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

BIBLIOGRAPHY

- [intro-1] Glen Cowan, Kyle Cranmer, Eilam Gross, and Ofer Vitells. Asymptotic formulae for likelihood-based tests of new physics. *Eur. Phys. J. C*, 71:1554, 2011. [arXiv:1007.1727](#), doi:10.1140/epjc/s10052-011-1554-0.
- [intro-2] Kyle Cranmer, George Lewis, Lorenzo Moneta, Akira Shibata, and Wouter Verkerke. HistFactory: A tool for creating statistical models for use with RooFit and RooStats. Technical Report CERN-OPEN-2012-016, New York U., New York, Jan 2012. URL: <https://cds.cern.ch/record/1456844>.
- [intro-3] Eamonn Maguire, Lukas Heinrich, and Graeme Watt. HEPData: a repository for high energy physics data. *J. Phys. Conf. Ser.*, 898(10):102006, 2017. [arXiv:1704.05473](#), doi:10.1088/1742-6596/898/10/102006.
- [intro-4] ATLAS Collaboration. Measurements of Higgs boson production and couplings in diboson final states with the ATLAS detector at the LHC. *Phys. Lett. B*, 726:88, 2013. [arXiv:1307.1427](#), doi:10.1016/j.physletb.2014.05.011.
- [intro-5] ATLAS Collaboration. Search for supersymmetry in final states with missing transverse momentum and multiple (b) -jets in proton–proton collisions at $\sqrt{s} = 13$ TeV with the ATLAS detector. ATLAS-CONF-2018-041, 2018. URL: <https://cds.cern.ch/record/2632347>.
- [likelihood-1] Histfactory definitions schema. Accessed: 2019-06-20. URL: <https://scikit-hep.org/pyhf/schemas/1.0.0/defs.json>.
- [likelihood-2] Kyle Cranmer, George Lewis, Lorenzo Moneta, Akira Shibata, and Wouter Verkerke. HistFactory: A tool for creating statistical models for use with RooFit and RooStats. Technical Report CERN-OPEN-2012-016, New York U., New York, Jan 2012. URL: <https://cds.cern.ch/record/1456844>.

PYTHON MODULE INDEX

p

`pyhf.contrib.viz.brazil`, [168](#)

Symbols

__ModelConfig (class in *pyhf.pdf*), 109
 __init__() (pyhf.infer.calculators.AsymptoticCalculator method), 162
 __init__() (pyhf.infer.calculators.AsymptoticTestStatDistribution method), 161
 __init__() (pyhf.interpolators.code0 method), 150
 __init__() (pyhf.interpolators.code1 method), 150
 __init__() (pyhf.interpolators.code2 method), 151
 __init__() (pyhf.interpolators.code4 method), 151
 __init__() (pyhf.interpolators.code4p method), 152
 __init__() (pyhf.optimize.mixins.OptimizerMixin method), 145
 __init__() (pyhf.optimize.opt_minuit.minuit_optimizer method), 146
 __init__() (pyhf.optimize.opt_scipy.scipy_optimizer method), 146
 __init__() (pyhf.patchset.Patch method), 115
 __init__() (pyhf.patchset.PatchSet method), 114
 __init__() (pyhf.pdf.Model method), 107
 __init__() (pyhf.pdf._ModelConfig method), 109
 __init__() (pyhf.probability.Independent method), 105
 __init__() (pyhf.probability.Normal method), 103
 __init__() (pyhf.probability.Poisson method), 104
 __init__() (pyhf.probability.Simultaneous method), 106
 __init__() (pyhf.tensor.jax_backend.jax_backend method), 138
 __init__() (pyhf.tensor.numpy_backend.numpy_backend method), 117
 __init__() (pyhf.tensor.pytorch_backend.pytorch_backend method), 123
 __init__() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 130
 __init__() (pyhf.workspace.Workspace method), 110
 --algorithm <algorithm>
 pyhf-digest command line option, 92
 --backend <backend>
 pyhf-cls command line option, 90
 pyhf-fit command line option, 93
 --basedir <basedir>
 pyhf-xml2json command line option, 98
 --channel <CHANNEL>...>
 pyhf-prune command line option, 97
 pyhf-channel <PATTERN> <REPLACE>...>
 pyhf-rename command line option, 97
 --compress
 pyhf-contrib-download command line option, 92
 --dataroot <dataroot>
 pyhf-json2xml command line option, 95
 --force
 pyhf-contrib-download command line option, 92
 --hide-progress
 pyhf-xml2json command line option, 98
 --join <join>
 pyhf-combine command line option, 90
 --json
 pyhf-digest command line option, 92
 --measurement <MEASUREMENT>...>
 pyhf-prune command line option, 97
 --measurement <PATTERN> <REPLACE>...>
 pyhf-rename command line option, 97
 --measurement <measurement>
 pyhf-cls command line option, 90
 pyhf-fit command line option, 93
 pyhf-inspect command line option, 94
 --merge-channels
 pyhf-combine command line option, 90
 --modifier <MODIFIER>...>
 pyhf-prune command line option, 97
 --modifier <PATTERN> <REPLACE>...>
 pyhf-rename command line option, 97
 --modifier-type <modifier_type>
 pyhf-prune command line option, 97
 --name <name>
 pyhf-patchset-apply command line option, 95
 pyhf-patchset-extract command line

option, [96](#)
--no-merge-channels
 pyhf-combine command line option, [90](#)
--optconf <optconf>
 pyhf-cls command line option, [90](#)
 pyhf-fit command line option, [93](#)
--optimizer <optimizer>
 pyhf-cls command line option, [90](#)
 pyhf-fit command line option, [93](#)
--output-dir <output_dir>
 pyhf-json2xml command line option, [95](#)
--output-file <output_file>
 pyhf-cls command line option, [90](#)
 pyhf-combine command line option, [90](#)
 pyhf-fit command line option, [93](#)
 pyhf-inspect command line option, [94](#)
 pyhf-patchset-apply command line option, [95](#)
 pyhf-patchset-extract command line option, [96](#)
 pyhf-prune command line option, [97](#)
 pyhf-rename command line option, [97](#)
 pyhf-sort command line option, [98](#)
 pyhf-xml2json command line option, [98](#)
--patch <patch>
 pyhf-cls command line option, [90](#)
 pyhf-fit command line option, [93](#)
 pyhf-json2xml command line option, [95](#)
--plaintext
 pyhf-digest command line option, [92](#)
--resultprefix <resultprefix>
 pyhf-json2xml command line option, [95](#)
--sample <PATTERN> <REPLACE>...>
 pyhf-rename command line option, [97](#)
--sample <SAMPLE>...>
 pyhf-prune command line option, [97](#)
--specroot <specroot>
 pyhf-json2xml command line option, [95](#)
--testpoi <testpoi>
 pyhf-cls command line option, [90](#)
--teststat <teststat>
 pyhf-cls command line option, [90](#)
--track-progress
 pyhf-xml2json command line option, [98](#)
--value
 pyhf-fit command line option, [93](#)
--verbose

 pyhf-contrib-download command line option, [92](#)
--version
 pyhf command line option, [89](#)
--with-metadata
 pyhf-patchset-extract command line option, [96](#)
--without-metadata
 pyhf-patchset-extract command line option, [96](#)
-a
 pyhf-digest command line option, [92](#)
-c
 pyhf-contrib-download command line option, [92](#)
 pyhf-prune command line option, [97](#)
 pyhf-rename command line option, [97](#)
-f
 pyhf-contrib-download command line option, [92](#)
-j
 pyhf-combine command line option, [90](#)
 pyhf-digest command line option, [92](#)
-m
 pyhf-prune command line option, [97](#)
 pyhf-rename command line option, [97](#)
-p
 pyhf-cls command line option, [90](#)
 pyhf-digest command line option, [92](#)
 pyhf-fit command line option, [93](#)
 pyhf-json2xml command line option, [95](#)
-s
 pyhf-prune command line option, [97](#)
 pyhf-rename command line option, [97](#)
-t
 pyhf-prune command line option, [97](#)
-v
 pyhf-contrib-download command line option, [92](#)

A

`abs()` (*pyhf.tensor.jax_backend.jax_backend method*), [138](#)
`abs()` (*pyhf.tensor.numpy_backend.numpy_backend method*), [117](#)
`abs()` (*pyhf.tensor.pytorch_backend.pytorch_backend method*), [123](#)
`abs()` (*pyhf.tensor.tensorflow_backend.tensorflow_backend method*), [130](#)
`apply()` (*pyhf.patchset.PatchSet method*), [114](#)
ARCHIVE_URL
 pyhf-contrib-download command line option, [92](#)

astensor() (pyhf.tensor.jax_backend.jax_backend method), 138

astensor() (pyhf.tensor.numpy_backend.numpy_backend method), 117

astensor() (pyhf.tensor.pytorch_backend.pytorch_backend method), 123

astensor() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 130

AsymptoticCalculator (class in pyhf.infer.calculators), 162

AsymptoticTestStatDistribution (class in pyhf.infer.calculators), 161

B

BACKGROUND_ONLY

pyhf-patchset-apply command line option, 96

pyhf-patchset-verify command line option, 96

boolean_mask() (pyhf.tensor.jax_backend.jax_backend method), 138

boolean_mask() (pyhf.tensor.numpy_backend.numpy_backend method), 117

boolean_mask() (pyhf.tensor.pytorch_backend.pytorch_backend method), 124

boolean_mask() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 130

build() (pyhf.workspace.Workspace class method), 110

C

clip() (pyhf.tensor.jax_backend.jax_backend method), 138

clip() (pyhf.tensor.numpy_backend.numpy_backend method), 117

clip() (pyhf.tensor.pytorch_backend.pytorch_backend method), 124

clip() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 130

code0 (class in pyhf.interpolators), 150

code1 (class in pyhf.interpolators), 150

code2 (class in pyhf.interpolators), 150

code4 (class in pyhf.interpolators), 151

code4p (class in pyhf.interpolators), 151

combine() (pyhf.workspace.Workspace class method), 110

concatenate() (pyhf.tensor.jax_backend.jax_backend method), 139

concatenate() (pyhf.tensor.numpy_backend.numpy_backend method), 118

concatenate() (pyhf.tensor.pytorch_backend.pytorch_backend method), 125

concatenate() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 131

conditional() (pyhf.tensor.jax_backend.jax_backend method), 139

conditional() (pyhf.tensor.numpy_backend.numpy_backend method), 118

conditional() (pyhf.tensor.pytorch_backend.pytorch_backend method), 124

conditional() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 131

constraint_logpdf() (pyhf.pdf.Model method), 107

D

data() (pyhf.workspace.Workspace method), 110

default_backend (in module pyhf), 101

default_do_grad (pyhf.tensor.jax_backend.jax_backend attribute), 138

default_do_grad (pyhf.tensor.numpy_backend.numpy_backend attribute), 117

default_do_grad (pyhf.tensor.pytorch_backend.pytorch_backend attribute), 123

default_do_grad (pyhf.tensor.tensorflow_backend.tensorflow_backend attribute), 130

default_optimizer (in module pyhf), 101

description (pyhf.patchset.PatchSet attribute), 114

digest() (in module pyhf.utils), 167

digest (pyhf.patchset.PatchSet attribute), 114

divide() (pyhf.tensor.jax_backend.jax_backend method), 139

divide() (pyhf.tensor.numpy_backend.numpy_backend method), 118

divide() (pyhf.tensor.pytorch_backend.pytorch_backend method), 125

divide() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 132

download() (in module pyhf.contrib.utils), 168

dtypesmap (pyhf.tensor.jax_backend.jax_backend attribute), 138

dtypesmap (pyhf.tensor.numpy_backend.numpy_backend attribute), 117

dtypesmap (pyhf.tensor.pytorch_backend.pytorch_backend attribute), 123

dtypesmap (pyhf.tensor.tensorflow_backend.tensorflow_backend attribute), 130

E

einsum() (pyhf.tensor.jax_backend.jax_backend method), 140

einsum() (pyhf.tensor.numpy_backend.numpy_backend method), 119

einsum() (pyhf.tensor.pytorch_backend.pytorch_backend method), 125

einsum() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 132

ENTRYPOINT_XML

pyhf-xml2json command line option,
99

erf() (pyhf.tensor.jax_backend.jax_backend method),
140

erf() (pyhf.tensor.numpy_backend.numpy_backend
method), 119

erf() (pyhf.tensor.pytorch_backend.pytorch_backend
method), 125

erf() (pyhf.tensor.tensorflow_backend.tensorflow_backend
method), 132

erfinv() (pyhf.tensor.jax_backend.jax_backend
method), 140

erfinv() (pyhf.tensor.numpy_backend.numpy_backend
method), 119

erfinv() (pyhf.tensor.pytorch_backend.pytorch_backend
method), 125

erfinv() (pyhf.tensor.tensorflow_backend.tensorflow_backend
method), 132

errordef (pyhf.optimize.opt_minuit.minuit_optimizer
attribute), 146

exp() (pyhf.tensor.jax_backend.jax_backend method),
140

exp() (pyhf.tensor.numpy_backend.numpy_backend
method), 119

exp() (pyhf.tensor.pytorch_backend.pytorch_backend
method), 126

exp() (pyhf.tensor.tensorflow_backend.tensorflow_backend
method), 132

expected_actualdata() (pyhf.pdf.Model method),
108

expected_auxdata() (pyhf.pdf.Model method), 108

expected_data() (pyhf.pdf.Model method), 108

expected_data() (pyhf.probability.Normal method),
103

expected_data() (pyhf.probability.Poisson method),
104

expected_data() (pyhf.probability.Simultaneous
method), 106

F

fit() (in module pyhf.infer.mle), 157

fixed_poi_fit() (in module pyhf.infer.mle), 158

G

gather() (pyhf.tensor.jax_backend.jax_backend
method), 140

gather() (pyhf.tensor.numpy_backend.numpy_backend
method), 119

gather() (pyhf.tensor.pytorch_backend.pytorch_backend
method), 126

gather() (pyhf.tensor.tensorflow_backend.tensorflow_backend
method), 132

generate_asimov_data() (in module
pyhf.infer.calculators), 161

get_backend() (in module pyhf), 102

get_measurement() (pyhf.workspace.Workspace
method), 111

H

hepdata_like (class in pyhf.simplemodels), 116

histosys (class in pyhf.modifiers), 147

hypotest() (in module pyhf.infer), 159

I

ImportBackendError (class in pyhf.exceptions), 166

Independent (class in pyhf.probability), 105

InvalidBackend (class in pyhf.exceptions), 166

InvalidInterpCode (class in pyhf.exceptions), 165

InvalidMeasurement (class in pyhf.exceptions), 164

InvalidModel (class in pyhf.exceptions), 165

InvalidModifier (class in pyhf.exceptions), 165

InvalidNameReuse (class in pyhf.exceptions), 164

InvalidOptimizer (class in pyhf.exceptions), 166

InvalidPatchLookup (class in pyhf.exceptions), 165

InvalidPatchSet (class in pyhf.exceptions), 164

InvalidPdfData (class in pyhf.exceptions), 166

InvalidPdfParameters (class in pyhf.exceptions),
166

InvalidSpecification (class in pyhf.exceptions),
164

InvalidWorkspaceOperation (class in
pyhf.exceptions), 165

is_constrained (pyhf.modifiers.histosys attribute),
147

is_constrained (pyhf.modifiers.normfactor at-
tribute), 147

is_constrained (pyhf.modifiers.normsys attribute),
148

is_constrained (pyhf.modifiers.shapefactor at-
tribute), 148

is_constrained (pyhf.modifiers.shapesys attribute),
149

is_constrained (pyhf.modifiers.staterror attribute),
149

isfinite() (pyhf.tensor.jax_backend.jax_backend
method), 140

isfinite() (pyhf.tensor.numpy_backend.numpy_backend
method), 119

isfinite() (pyhf.tensor.pytorch_backend.pytorch_backend
method), 126

isfinite() (pyhf.tensor.tensorflow_backend.tensorflow_backend
method), 133

J

jax_backend (class in pyhf.tensor.jax_backend), 138

L

labels (pyhf.patchset.PatchSet attribute), 114

load_schema() (in module pyhf.utils), 167
 log() (pyhf.tensor.jax_backend.jax_backend method), 140
 log() (pyhf.tensor.numpy_backend.numpy_backend method), 119
 log() (pyhf.tensor.pytorch_backend.pytorch_backend method), 126
 log() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 133
 log_prob() (pyhf.probability.Independent method), 105
 log_prob() (pyhf.probability.Simultaneous method), 106
 logpdf() (pyhf.pdf.Model method), 108

M

mainlogpdf() (pyhf.pdf.Model method), 108
 make_pdf() (pyhf.pdf.Model method), 108
 maxiter (pyhf.optimize.mixins.OptimizerMixin attribute), 145
 maxiter (pyhf.optimize.opt_minuit.minuit_optimizer attribute), 146
 maxiter (pyhf.optimize.opt_scipy.scipy_optimizer attribute), 146
 metadata (pyhf.patchset.Patch attribute), 115
 metadata (pyhf.patchset.PatchSet attribute), 114
 minimize() (pyhf.optimize.mixins.OptimizerMixin method), 145
 minuit_optimizer (class in pyhf.optimize.opt_minuit), 146
 Model (class in pyhf.pdf), 107
 model() (pyhf.workspace.Workspace method), 111
 module
 pyhf.contrib.viz.brazil, 168

N

name (pyhf.optimize.opt_minuit.minuit_optimizer attribute), 146
 name (pyhf.optimize.opt_scipy.scipy_optimizer attribute), 146
 name (pyhf.patchset.Patch attribute), 115
 name (pyhf.tensor.jax_backend.jax_backend attribute), 138
 name (pyhf.tensor.numpy_backend.numpy_backend attribute), 117
 name (pyhf.tensor.pytorch_backend.pytorch_backend attribute), 123
 name (pyhf.tensor.tensorflow_backend.tensorflow_backend attribute), 130
 nominal_rates (pyhf.pdf.Model attribute), 107
 Normal (class in pyhf.probability), 103
 normal() (pyhf.tensor.jax_backend.jax_backend method), 141

normal() (pyhf.tensor.numpy_backend.numpy_backend method), 119
 normal() (pyhf.tensor.pytorch_backend.pytorch_backend method), 126
 normal() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 133
 normal_cdf() (pyhf.tensor.jax_backend.jax_backend method), 141
 normal_cdf() (pyhf.tensor.numpy_backend.numpy_backend method), 120
 normal_cdf() (pyhf.tensor.pytorch_backend.pytorch_backend method), 126
 normal_cdf() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 133
 normal_dist() (pyhf.tensor.jax_backend.jax_backend method), 141
 normal_dist() (pyhf.tensor.numpy_backend.numpy_backend method), 120
 normal_dist() (pyhf.tensor.pytorch_backend.pytorch_backend method), 127
 normal_dist() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 134
 normal_logpdf() (pyhf.tensor.jax_backend.jax_backend method), 142
 normal_logpdf() (pyhf.tensor.numpy_backend.numpy_backend method), 121
 normal_logpdf() (pyhf.tensor.pytorch_backend.pytorch_backend method), 127
 normal_logpdf() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 134
 normfactor (class in pyhf.modifiers), 147
 normsys (class in pyhf.modifiers), 148
 numpy_backend (class in pyhf.tensor.numpy_backend), 117

O

ones() (pyhf.tensor.jax_backend.jax_backend method), 142
 ones() (pyhf.tensor.numpy_backend.numpy_backend method), 121
 ones() (pyhf.tensor.pytorch_backend.pytorch_backend method), 127
 ones() (pyhf.tensor.tensorflow_backend.tensorflow_backend method), 135
 op_code (pyhf.modifiers.histosys attribute), 147
 op_code (pyhf.modifiers.normfactor attribute), 147
 op_code (pyhf.modifiers.normsys attribute), 148
 op_code (pyhf.modifiers.shapefactor attribute), 148
 op_code (pyhf.modifiers.shapesys attribute), 149
 op_code (pyhf.modifiers.staterror attribute), 149
 optimizer (in module pyhf), 102
 OptimizerMixin (class in pyhf.optimize.mixins), 144
 options_from_eqdelimstring() (in module pyhf.utils), 167

`outer()` (`pyhf.tensor.jax_backend.jax_backend` method), 142
`outer()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 121
`outer()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 127
`outer()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 135
 OUTPUT_DIRECTORY
 pyhf-contrib-download command line option, 92

P

`par_slice()` (`pyhf.pdf._ModelConfig` method), 109
`param_set()` (`pyhf.pdf._ModelConfig` method), 109
`Patch` (class in `pyhf.patchset`), 115
`patches` (`pyhf.patchset.PatchSet` attribute), 114
 PATCHSET
 pyhf-patchset-apply command line option, 96
 pyhf-patchset-extract command line option, 96
 pyhf-patchset-verify command line option, 96
`PatchSet` (class in `pyhf.patchset`), 112
`PatchSetVerificationError` (class in `pyhf.exceptions`), 165
`pdf()` (`pyhf.pdf.Model` method), 108
`pdf_type` (`pyhf.modifiers.histosys` attribute), 147
`pdf_type` (`pyhf.modifiers.normfactor` attribute), 147
`pdf_type` (`pyhf.modifiers.normsys` attribute), 148
`pdf_type` (`pyhf.modifiers.shapefactor` attribute), 148
`pdf_type` (`pyhf.modifiers.shapesys` attribute), 149
`pdf_type` (`pyhf.modifiers.staterror` attribute), 149
`Poisson` (class in `pyhf.probability`), 104
`poisson()` (`pyhf.tensor.jax_backend.jax_backend` method), 142
`poisson()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 121
`poisson()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 127
`poisson()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 135
`poisson_dist()` (`pyhf.tensor.jax_backend.jax_backend` method), 142
`poisson_dist()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 121
`poisson_dist()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 128
`poisson_dist()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 135
`poisson_logpdf()` (`pyhf.tensor.jax_backend.jax_backend` method), 143
`poisson_logpdf()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 122
`poisson_logpdf()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 128
`poisson_logpdf()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 135
`power()` (`pyhf.tensor.jax_backend.jax_backend` method), 143
`power()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 122
`power()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 128
`power()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 136
`precision` (`pyhf.tensor.jax_backend.jax_backend` attribute), 138
`precision` (`pyhf.tensor.numpy_backend.numpy_backend` attribute), 117
`precision` (`pyhf.tensor.pytorch_backend.pytorch_backend` attribute), 123
`precision` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` attribute), 130
`product()` (`pyhf.tensor.jax_backend.jax_backend` method), 143
`product()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 122
`product()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 128
`product()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 136
`prune()` (`pyhf.workspace.Workspace` method), 111
 pyhf command line option
 --version, 89
 pyhf.contrib.viz.brazil
 module, 168
 pyhf-cls command line option
 --backend <backend>, 90
 --measurement <measurement>, 90
 --optconf <optconf>, 90
 --optimizer <optimizer>, 90
 --output-file <output_file>, 90
 --patch <patch>, 90
 --testpoi <testpoi>, 90
 --teststat <teststat>, 90
 -p, 90
 WORKSPACE, 90
 pyhf-combine command line option
 --join <join>, 90
 --merge-channels, 90
 --merge-channels, 90
 --output-file <output_file>, 90
 -j, 90
 WORKSPACE_ONE, 91
 WORKSPACE_TWO, 91

pyhf-completions command line option
SHELL, 91

pyhf-contrib-download command line option
--compress, 92
--force, 92
--verbose, 92
-c, 92
-f, 92
-v, 92
ARCHIVE_URL, 92
OUTPUT_DIRECTORY, 92

pyhf-digest command line option
--algorithm <algorithm>, 92
--json, 92
--plaintext, 92
-a, 92
-j, 92
-p, 92
WORKSPACE, 92

pyhf-fit command line option
--backend <backend>, 93
--measurement <measurement>, 93
--optconf <optconf>, 93
--optimizer <optimizer>, 93
--output-file <output_file>, 93
--patch <patch>, 93
--value, 93
-p, 93
WORKSPACE, 93

pyhf-inspect command line option
--measurement <measurement>, 94
--output-file <output_file>, 94
WORKSPACE, 94

pyhf-json2xml command line option
--dataroot <dataroot>, 95
--output-dir <output_dir>, 95
--patch <patch>, 95
--resultprefix <resultprefix>, 95
--specroot <specroot>, 95
-p, 95
WORKSPACE, 95

pyhf-patchset-apply command line option
--name <name>, 95
--output-file <output_file>, 95
BACKGROUND_ONLY, 96
PATCHSET, 96

pyhf-patchset-extract command line option
--name <name>, 96
--output-file <output_file>, 96
--with-metadata, 96
--without-metadata, 96
PATCHSET, 96

pyhf-patchset-verify command line option
BACKGROUND_ONLY, 96
PATCHSET, 96

pyhf-prune command line option
--channel <CHANNEL>..., 97
--measurement <MEASUREMENT>..., 97
--modifier <MODIFIER>..., 97
--modifier-type <modifier_type>, 97
--output-file <output_file>, 97
--sample <SAMPLE>..., 97
-c, 97
-m, 97
-s, 97
-t, 97
WORKSPACE, 97

pyhf-rename command line option
--channel <PATTERN> <REPLACE>..., 97
--measurement <PATTERN> <REPLACE>..., 97
--modifier <PATTERN> <REPLACE>..., 97
--output-file <output_file>, 97
--sample <PATTERN> <REPLACE>..., 97
-c, 97
-m, 97
-s, 97
WORKSPACE, 98

pyhf-sort command line option
--output-file <output_file>, 98
WORKSPACE, 98

pyhf-xml2json command line option
--basedir <basedir>, 98
--hide-progress, 98
--output-file <output_file>, 98
--track-progress, 98
ENTRYPOINT_XML, 99

pytorch_backend (class in *pyhf.tensor.pytorch_backend*), 123

Q

qmu () (in module *pyhf.infer.test_statistics*), 152
qmu_tilde () (in module *pyhf.infer.test_statistics*), 153

R

references (*pyhf.patchset.PatchSet* attribute), 114
rename () (*pyhf.workspace.Workspace* method), 112
required_parset () (*pyhf.modifiers.histosys* class method), 147
required_parset () (*pyhf.modifiers.normfactor* class method), 148

[required_parset\(\)](#) ([pyhf.modifiers.normsys](#) class method), 148
[required_parset\(\)](#) ([pyhf.modifiers.shapefactor](#) class method), 148
[required_parset\(\)](#) ([pyhf.modifiers.shapesys](#) class method), 149
[required_parset\(\)](#) ([pyhf.modifiers.staterror](#) class method), 149
[reshape\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[reshape\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[reshape\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 128
[reshape\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136

S

[sample\(\)](#) ([pyhf.probability.Simultaneous](#) method), 106
[scipy_optimizer](#) (class in [pyhf.optimize.opt_scipy](#)), 146
[set_backend\(\)](#) (in module [pyhf](#)), 102
[set_poi\(\)](#) ([pyhf.pdf._ModelConfig](#) method), 109
[shape\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[shape\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[shape\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 128
[shape\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136
[shapefactor](#) (class in [pyhf.modifiers](#)), 148
[shapesys](#) (class in [pyhf.modifiers](#)), 148
[SHELL](#)
 [pyhf-completions](#) command line option, 91
[simple_broadcast\(\)](#)
 ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[simple_broadcast\(\)](#)
 ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[simple_broadcast\(\)](#)
 ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 128
[simple_broadcast\(\)](#)
 ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136
[Simultaneous](#) (class in [pyhf.probability](#)), 106
[sorted\(\)](#) ([pyhf.workspace.Workspace](#) class method), 112
[sqrt\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[sqrt\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[sqrt\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 129
[sqrt\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136
[stack\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[stack\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[stack\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 129
[stack\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136
[staterror](#) (class in [pyhf.modifiers](#)), 149
[steps](#) ([pyhf.optimize.opt_minuit.minuit_optimizer](#) attribute), 146
[suggested_bounds\(\)](#) ([pyhf.pdf._ModelConfig](#) method), 109
[suggested_fixed\(\)](#) ([pyhf.pdf._ModelConfig](#) method), 109
[suggested_init\(\)](#) ([pyhf.pdf._ModelConfig](#) method), 109
[sum\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[sum\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[sum\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 129
[sum\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 136

T

[tensorflow_backend](#) (class in [pyhf.tensor.tensorflow_backend](#)), 130
[tensorlib](#) (in module [pyhf](#)), 102
[tile\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 143
[tile\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 122
[tile\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 129
[tile\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 137
[tmu\(\)](#) (in module [pyhf.infer.test_statistics](#)), 154
[tmu_tilde\(\)](#) (in module [pyhf.infer.test_statistics](#)), 155
[tolist\(\)](#) ([pyhf.tensor.jax_backend.jax_backend](#) method), 144
[tolist\(\)](#) ([pyhf.tensor.numpy_backend.numpy_backend](#) method), 123
[tolist\(\)](#) ([pyhf.tensor.pytorch_backend.pytorch_backend](#) method), 129
[tolist\(\)](#) ([pyhf.tensor.tensorflow_backend.tensorflow_backend](#) method), 137

`twice_nll()` (in module `pyhf.infer.mle`), 156

U

`upperlimit()` (in module `pyhf.infer.intervals`), 162

V

`valid_joins` (`pyhf.workspace.Workspace` attribute), 110

`validate()` (in module `pyhf.utils`), 167

`values` (`pyhf.patchset.Patch` attribute), 115

`verbose` (`pyhf.optimize.mixins.OptimizerMixin` attribute), 145

`verbose` (`pyhf.optimize.opt_minuit.minuit_optimizer` attribute), 146

`verbose` (`pyhf.optimize.opt_scipy.scipy_optimizer` attribute), 146

`verify()` (`pyhf.patchset.PatchSet` method), 115

`version` (`pyhf.patchset.PatchSet` attribute), 114

W

`where()` (`pyhf.tensor.jax_backend.jax_backend` method), 144

`where()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 123

`where()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 129

`where()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 137

WORKSPACE

`pyhf-cls` command line option, 90

`pyhf-digest` command line option, 92

`pyhf-fit` command line option, 93

`pyhf-inspect` command line option, 94

`pyhf-json2xml` command line option, 95

`pyhf-prune` command line option, 97

`pyhf-rename` command line option, 98

`pyhf-sort` command line option, 98

`Workspace` (class in `pyhf.workspace`), 109

WORKSPACE_ONE

`pyhf-combine` command line option, 91

WORKSPACE_TWO

`pyhf-combine` command line option, 91

Z

`zeros()` (`pyhf.tensor.jax_backend.jax_backend` method), 144

`zeros()` (`pyhf.tensor.numpy_backend.numpy_backend` method), 123

`zeros()` (`pyhf.tensor.pytorch_backend.pytorch_backend` method), 129

`zeros()` (`pyhf.tensor.tensorflow_backend.tensorflow_backend` method), 137