pyhf Documentation

Release 0.2.1

Lukas Heinrich, Matthew Feickert

Dec 18, 2019

CONTENTS

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 . Given observed data, the *likelihood* $\mathcal{L}()$ then serves as the basis to test hypotheses on the parameters . For measurements based on binned data (*histograms*), the family of statistical models has been widely used in both Standard Model measurements [4] as well as searches for new physics [5]. In this package, a declarative, plain-text format for describing -based likelihoods is presented that is targeted for reinterpretation and long-term preservation in analysis data repositories such as HEPData [3].

CHAPTER

ONE

HISTFACTORY

Statistical models described using [2] center around the simultaneous measurement of disjoint binned distributions (*channels*) observed as event counts . For each channel, the overall expected event rate[1] 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* .

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

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:

$$f(,|,) = \underbrace{\prod_{c\in\text{ channels}}\prod_{b\in\text{ bins}_c}\text{Pois}\left(n_{cb}\,|\,\nu_{cb}\,(,)\right)}_{\substack{\text{Simultaneous measurement}\\ \text{of multiple channels}}}\qquad\underbrace{\prod_{\in} c(a|\,)}_{\substack{\text{constraint terms}\\ \text{for } x201C\text{auxiliary measurements}x201D}}\quad, \tag{1.2}$$

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

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

The total rates are the sum over sample rates ν_{csb}, each determined from a *nominal rate* ν^0_{scb} 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	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \mathrm{Pois}\left(r_b = \sigma_b^{-2} \middle\vert \rho_b = \sigma_b^{-2}\gamma_b\right)$	σ_b
Correlated Shape	$\Delta_{scb}(\alpha) = f_p\left(\alpha \middle\vert \Delta_{scb,\alpha=-1}, \Delta_{scb,\alpha=1}\right)$	$\mathrm{Gaus}\left(a = 0 \middle\vert \alpha, \sigma = 1\right)$	$\Delta_{scb,\alpha=\pm1}$
Normalisation Unc.	$\kappa_{scb}(\alpha) = g_p\left(\alpha \middle\vert \kappa_{scb,\alpha=-1}, \kappa_{scb,\alpha=1}\right)$	$\mathrm{Gaus}\left(a = 0 \middle\vert \alpha, \sigma = 1\right)$	$\kappa_{scb,\alpha=\pm1}$
MC Stat. Uncertainty	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \mathrm{Gaus}\left(a_{\gamma_b} = 1 \middle\vert \gamma_b, \delta_b\right)$	$\delta_b^2 = \sum_s \delta_{sb}^2$
Luminosity	$\kappa_{scb}(\lambda) = \lambda$	$\mathrm{Gaus}\left(l = \lambda_0 \middle\vert \lambda, \sigma_\lambda\right)$	$\lambda_0, \sigma_\lambda$
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 [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 [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

CHAPTER

TWO

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 [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.

CHAPTER

THREE

ADDITIONAL MATERIAL

3.1 Footnotes

3.2 Bibliography

CHAPTER

FOUR

LIKELIHOOD SPECIFICATION

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

4.1 Workspace

```
{
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "https://diana-hep.org/pyhf/schemas/1.0.0/workspace.json",
    "type": "object",
    "properties": {
        "channels": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪channel"} },
        "measurements": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪measurement"} },
        "observations": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪observation" } },
        "version": { "const": "1.0.0" }
    },
    "additionalProperties": false,
    "required": ["channels", "measurements", "observations", "version"]
}
```

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://diana-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 [1].

```
{
    "channel": {
        "type": "object",
        "properties": {
            "name": { "type": "string" },
            "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 [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 name[1] 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.

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.

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.

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

4.4.4 MC Statistical Uncertainty (staterror)

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 [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: $\delta_{csb} = \sigma_{csb} / \sum_s \nu^0_{scb}$. 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": "staterror", "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 }
        ]
    }
}
```

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`).

[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.

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

CHAPTER

FIVE

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 `interpcode=0` in the above reference. This function is defined as (nb: vector denotes bold)

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

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 interplations 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.

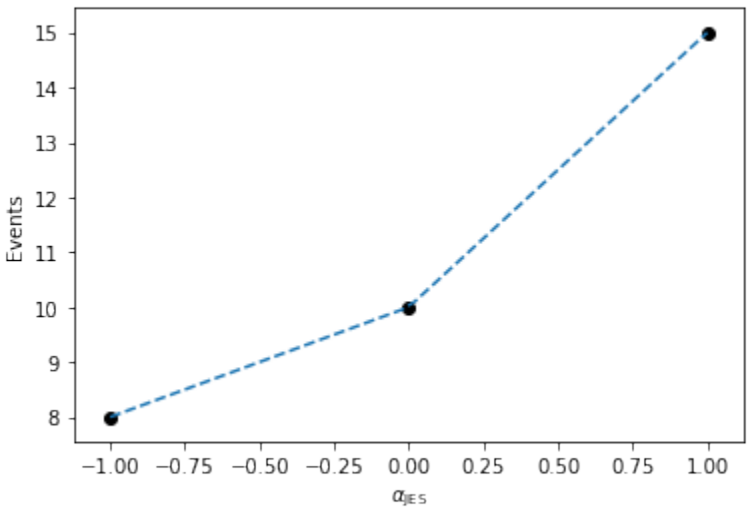
[4]:
```
alphas = np.linspace(-1.,1.)
deltas = [interpolate_deltas(down_1, nom_1, up_1, alpha) for alpha in alphas]
deltas[:5]
```

[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

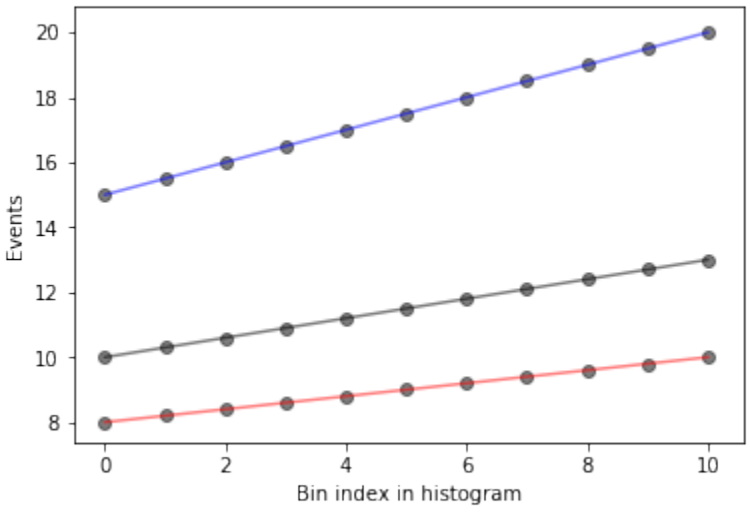
Now, let's increase the computational difficulty a little by increasing the dimensionality. Assume instead of a single-bin measurement, we have more measurements! We are good physicists after all. Imagine continuing on the previous example, where we add more bins, perhaps because we got more data. Imagine 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)
```



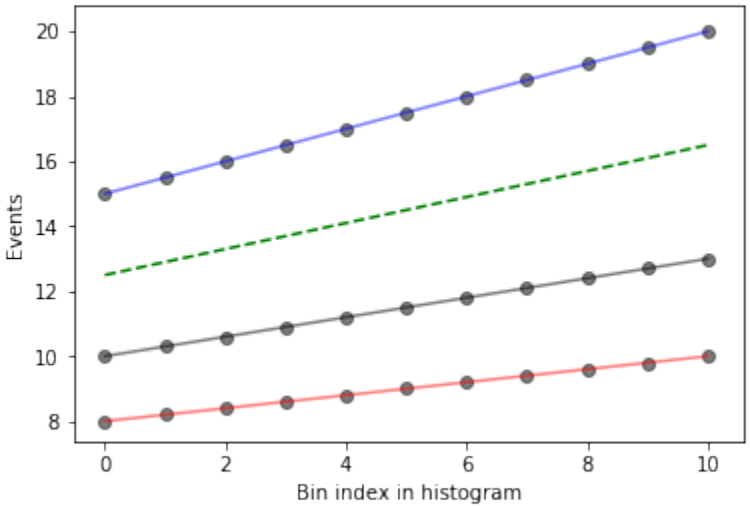
What does this look like if we evaluate at a single $\alpha_{\text{JES}} = 0.5$? We'll write a function that interpolates and then plots the interpolated values as a function of bin index, in green, dashed.

[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))
```

```
aW50ZXJhY3RpdmUoY2hpbGRyZW49KEZsb2F0U2xpZGVyKHZhbHVlPTAuMCwgZGVzY3JpcHRpb249dSdhbHBoYScsIG1heD0xLjAs
```

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)

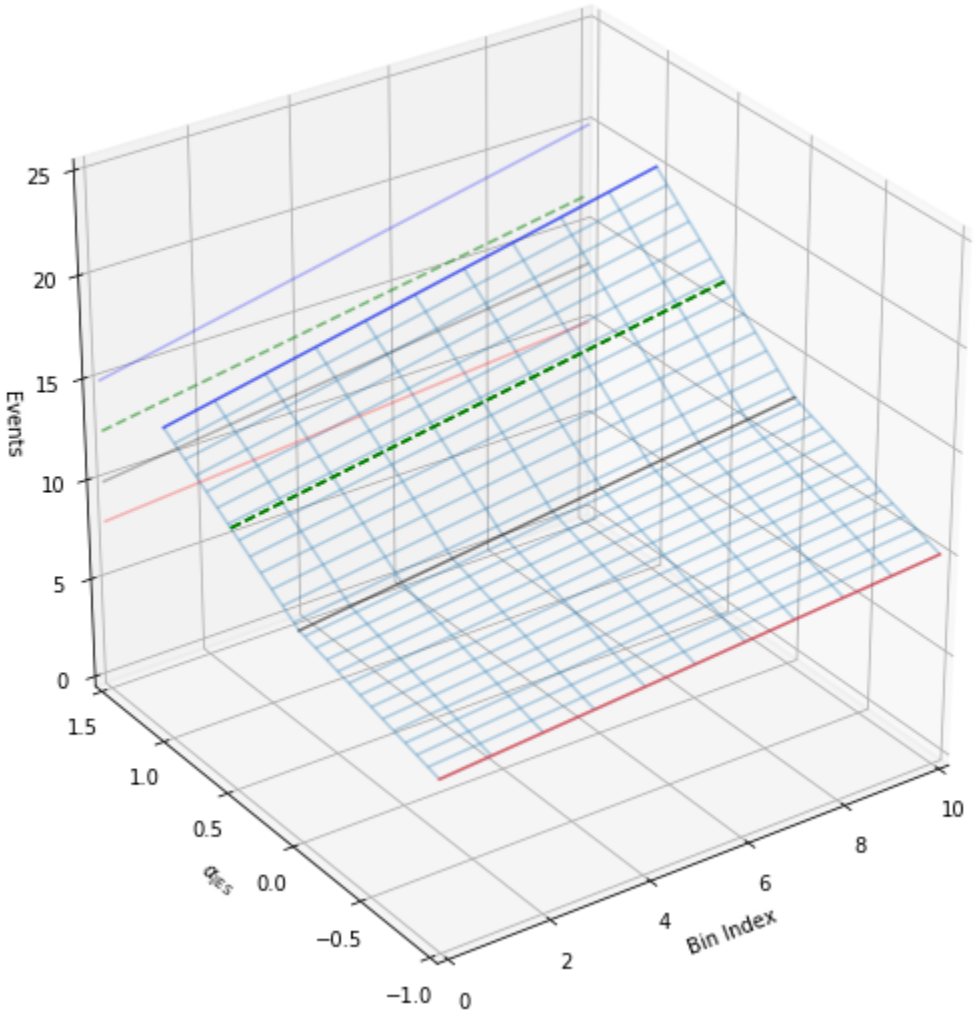
(continued from previous page)

```
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))
```



aW50ZXJhY3RpdmUoY2hpbGRyZW49KEZsb2F0U2xpZGVyKHZhbHVlPTAuMCwgZGVzY3JpcHRpb249dSdhbHBoYScsIG1heD0xLjAs

[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,alphaset in enumerate(alphasets):
            alphas[i,:len(alphaset)] = alphaset
        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 = nbin).tolist() for j in␣
↪range(3)
            ] for i,nbin 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, 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]):
                    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, histssets, alphasets):
        return np.asarray(func(histssets,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, 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 = []
                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

$$c_j = \sum_i \sum_k = A_{ik} B_{kj} \qquad \rightarrow \qquad \texttt{einsum("ij,jk->i", A, B)}$$

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 alphaset:
        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',alphaset,all_histo_deltas_up[nh])
    alphas_times_deltas_dn = np.einsum('i,j->ij',alphaset,all_histo_deltas_dn[nh])
    masks = np.einsum('i,j->ij',alphaset > 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 `alphaset` 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, alphaset) 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',alphaset,all_histo_deltas_
↪up[nh])
            alphas_times_deltas_dn = np.einsum('i,j->ij',alphaset,all_histo_deltas_
↪dn[nh])
            masks = np.einsum('i,j->ij',alphaset > 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,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 = []
                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,alphasets):
    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,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_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,alphasets):
    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, 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):
            # bases and exponents need to have an outer product, to esentially tile␣
↪or repeat over rows/cols
            bases_up = np.einsum('a,b->ab', np.ones(alphaset.shape), all_histo_deltas_
↪up[nh])
            bases_dn = np.einsum('a,b->ab', np.ones(alphaset.shape), all_histo_deltas_
↪dn[nh])
            exponents = np.einsum('a,b->ab', np.abs(alphaset), np.ones(all_histo_
↪deltas_up[nh].shape))

            masks = np.einsum('a,b->ab',alphaset > 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,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,(_,alphaset) in enumerate(zip(histogramssets,alphasets)):
        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(alphaset),noms)
        set_result = all_histos_deltas * all_histos_noms_repeated
        all_results.append(set_result)
    return all_results

def new_interpolation_nonlinear_step6(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)
    allsets_allhistos_noms_repeated = np.einsum('sa,shb->shab', np.ones(alphasets.
↪shape), allset_all_histo_nom)
    set_results = allsets_all_histos_deltas * allsets_allhistos_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)
```

CHAPTER

SIX

EXAMPLES

Try out in Binder!

Notebooks:

6.1 Hello World, `pyhf` style

Two bin counting experiment with a background uncertainty

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

Returning the observed and expected CL_s

[2]:
```
pdf = pyhf.simplemodels.hepdata_like(signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0],␣
↪bkg_uncerts=[3.0, 7.0])
CLs_obs, CLs_exp = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected=True)
print('Observed: {}, Expected: {}'.format(CLs_obs, CLs_exp))
```

```
Observed: [0.05290116], Expected: [0.06445521]
```

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

[3]:
```
CLs_obs, p_values = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_tail_probs=True)
print('Observed CL_s: {}, CL_sb: {}, CL_b: {}'.format(CLs_obs, p_values[0], p_
↪values[1]))
```

```
Observed CL_s: [0.05290116], CL_sb: [0.0236], CL_b: [0.44611493]
```

A reminder that

$$\mathrm{CL}_s = \frac{\mathrm{CL}_{s+b}}{\mathrm{CL}_b} = \frac{p_{s+b}}{1-p_b}$$

[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.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected_set=True)
print('Observed CL_s: {}\n'.format(CLs_obs))
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: [0.05290116]

Expected CL_s(-2 ): [0.00260641]
Expected CL_s(-1 ): [0.01382066]
Expected CL_s      : [0.06445521]
Expected CL_s(1 ): [0.23526104]
Expected CL_s(2 ): [0.57304182]
```

Returning the test statistics for the observed and Asimov data

[7]:
```
CLs_obs, test_statistics = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata,␣
↪pdf, return_test_statistics=True)
print('q_mu: {}, Asimov q_mu: {}'.format(test_statistics[0], test_statistics[1]))
```

```
q_mu: [3.93824492], Asimov q_mu: [3.41886758]
```

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

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

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

6.2 Binned HEP Statistical Analysis in Python

6.2.1 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)

6.2.2 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].

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

6.2.3 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('<img src="{}">'.format(anim))
```

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

6.2.4 Read in the Model from XML and ROOT

The ROOT files are read using scikit-hep's uproot module.

[4]:
```
parsed = pyhf.readxml.parse('meas.xml',os.getcwd())
workspace = pyhf.Workspace(parsed)
obs_data = workspace.observations['channel1']
```

From the parsed data, we construct a probability density function (p.d.f). As the model includes systematics a number of implied "auxiliary measurements" must be added to the observed data distribution.

[5]:
```
pdf = pyhf.Model({'channels': parsed['channels'], 'parameters': parsed['measurements
↪'][0]['config']['parameters']}, poiname = 'SigXsecOverSM')
data = obs_data + pdf.config.auxdata
```

The p.d.f is build from one data-drived "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]:
```
par_name_dict = {k: v['slice'].start for k,v in pdf.config.par_map.items()}
print('Samples:\n {}'.format(pdf.config.samples))
print('Parameters:\n {}'.format(par_name_dict))
```

```
Samples:
 ['mc1', 'mc2', 'qcd', 'signal']
Parameters:
 {'lumi': 0, 'SigXsecOverSM': 1, 'mc1_weight_var1': 2, 'mc1_shape_conv': 3, 'mc2_
↪weight_var1': 4, 'mc2_shape_conv': 5}
```

[7]:
```
all_par_settings = {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)
```

(continues on next page)

(continued from previous page)

```
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, obs_data, c='k', alpha=1., 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]
            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, obs_data, c = 'k', alpha = 1., zorder=99)
```

6.2.5 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 oberserve the effect on the samples. For example changing the parameter of interest `SigXsecOverSM` (or μ) controls the overall normalization of the (BSM) signal sample (μ=0 for background-only and μ=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(obs_data))

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),␣
↪IntSlider(value=5, descrip...
```

[9]:
```
nominal = pdf.config.suggested_init()
background_only = pdf.config.suggested_init()
background_only[pdf.config.poi_index] = 0.0
best_fit = pyhf.optimizer.unconstrained_bestfit(
    pyhf.utils.loglambdav, data, pdf, pdf.config.suggested_init(), pdf.config.
↪suggested_bounds())
```

```
/Users/jovyan/pyhf/src/pyhf/tensor/numpy_backend.py:184: RuntimeWarning: invalid␣
↪value encountered in log
  return n * np.log(lam) - lam - gammaln(n + 1.0)
```

6.2.6 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(obs_data))
ax1.set_title(u'nominal signal + background μ = 1')
plot(ax = ax1, **{k: nominal[v] for k,v in par_name_dict.items()})

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

ax3.set_title(u'best fit μ = {:.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>
```

6.2.7 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 plot_results(ax, test_mus, cls_obs, cls_exp, test_size=0.05):
    ax.plot(mu_tests, cls_obs, c = 'k')
    for i,c in zip(range(5),['k','k','k','k','k']):
        ax.plot(mu_tests, cls_exp[i], c = c, linestyle = 'dotted' if i!=2 else 'dashed
↪')
    ax.fill_between(test_mus,cls_exp[0],cls_exp[-1], facecolor = 'y')
    ax.fill_between(test_mus,cls_exp[1],cls_exp[-2], facecolor = 'g')
    ax.plot(test_mus,[test_size]*len(test_mus), c = 'r')
    ax.set_ylim(0,1)
```

[12]:
```
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
```

[13]:
```
mu_tests = np.linspace(0, 1, 16)
hypo_tests = [pyhf.utils.hypotest(mu, data, pdf, pdf.config.suggested_init(), pdf.
↪config.suggested_bounds(),
                                  return_expected_set=True, return_test_
↪statistics=True)
              for mu in mu_tests]

test_stats = np.array([test[-1][0] for test in hypo_tests]).flatten()
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)]

fig, (ax1,ax2) = plt.subplots(1, 2)
fig.set_size_inches(15, 5)

ax1.set_title(u'Hypothesis Tests')
ax1.set_ylabel(u'CLs')
ax1.set_xlabel(u'μ')
plot_results(ax1, mu_tests, cls_obs, cls_exp)


ax2.set_title(u'Test Statistic')
ax2.set_xlabel(u'μ')
ax2.plot(mu_tests,test_stats);
```

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

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

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

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
```

6.3 XML Import/Export

[1]:
```
# NB: 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
```

6.3.1 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",
```

(continues on next page)

(continued from previous page)


```
            "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
                    ],

                        {
                            "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",
                            "type": "normsys"
                        }
                    ],
                    "name": "background1"
                },
                {
                    "data": [
                        0.0,
                        100.0
```


(continues on next page)

(continued from previous page)


```
                ],
                "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
                            ]
                        ],
                        "fixed": true,
                        "inits": [
                            1.0
                        ],
                        "name": "lumi",
                        "sigmas": [
                            0.1
```


(continues on next page)

(continued from previous page)


```
                        ]
                    },
                    {
                        "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
                        ],
                        "bounds": [
                            [
                                0.5,
                                1.5
                            ]
                        ],
                        "fixed": true,
                        "inits": [
```


(continues on next page)

(continued from previous page)


```
                                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"
            }
        ],
        "resultprefix": "./results/example"
    }
}
```

6.3.2 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/
```

6.4 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 =  {
```

(continues on next page)

(continued from previous page)

```
        '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',
                                '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": {
```

(continues on next page)

(continued from previous page)

```
      "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()
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.optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, data, pdf,
                                                 pdf.config.suggested_init(), pdf.
↪config.suggested_bounds())
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)
```

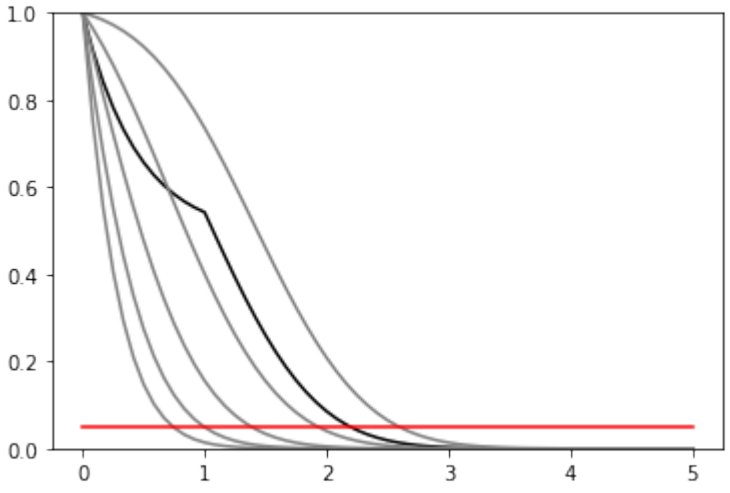
[6]:
```
def plot_results(testmus, cls_obs, cls_exp, poi_tests, test_size = 0.05):
    plt.plot(poi_tests,cls_obs, c = 'k')
    for i,c in zip(range(5),['grey','grey','grey','grey','grey']):
        plt.plot(poi_tests, cls_exp[i], c = c)
    plt.plot(testmus,[test_size]*len(testmus), c = 'r')
    plt.ylim(0,1)

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(
```

(continues on next page)

(continued from previous page)

```
            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.utils.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}
```



6.5 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 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

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'
```

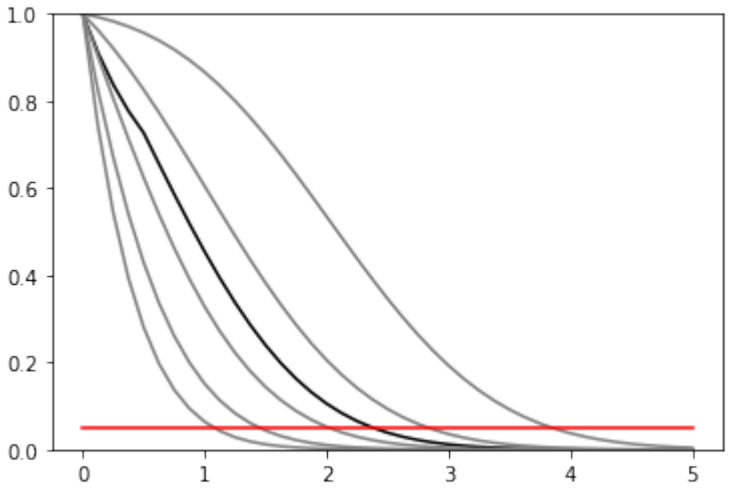
[5]:
```
source = json.load(open(validation_datadir + '/1bin_example1.json'))
pdf = hepdata_like(source['bindata']['sig'], source['bindata']['bkg'], source['bindata
↪']['bkgerr'])
```

(continues on next page)

(continued from previous page)

```
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.utils.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': [1.0810606780537388,
  1.4517179965651292,
  2.0200754881420266,
  2.834863384648174,
  3.8487567494315487],
 '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
```

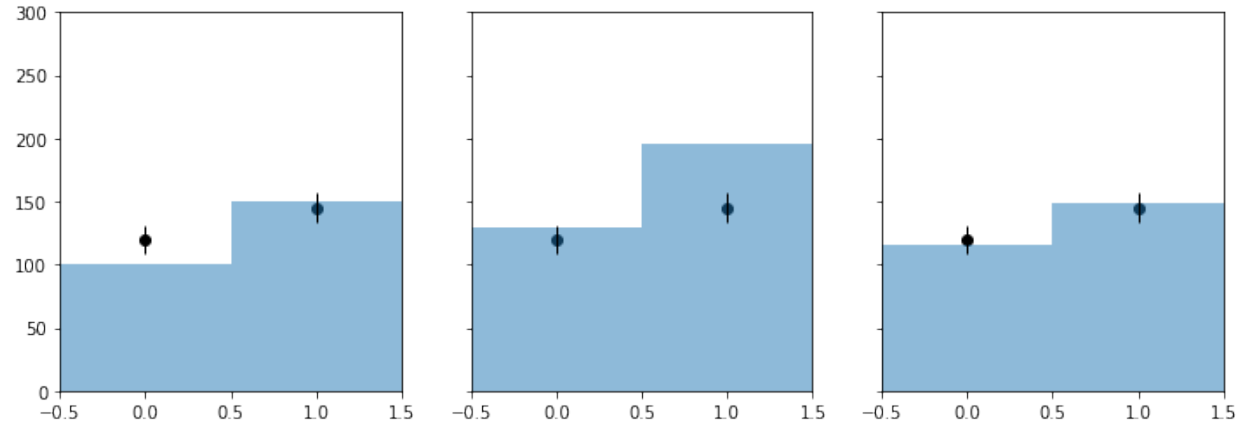
(continues on next page)

(continued from previous page)

```
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 = optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, 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);
```

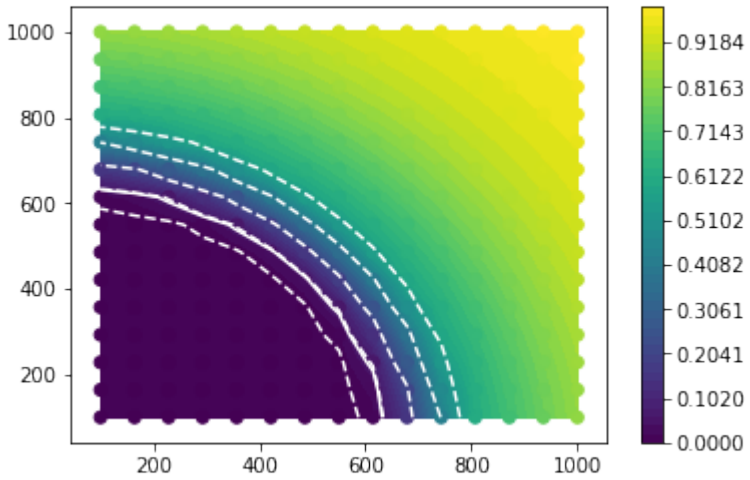


[8]:
```
##
##  DUMMY 2D thing
##

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'][
↪'bkgerr'])
    try:
        cls_obs, cls_exp_set = pyhf.utils.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', linestyles =
↪'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/papermill.record+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
                            }
                        ]
```

(continues on next page)

(continued from previous page)

```
                    },
                    {
                        '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']}
                            }
                        ]
                    }
                ]
            },
            {
                '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.optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, data, pdf,␣
↪init_pars, par_bounds)
print('parameters post unconstrained fit: {}'.format(unconpars))

conpars = pyhf.optimizer.constrained_bestfit(pyhf.utils.loglambdav, 0.0, data, pdf,␣
↪init_pars, par_bounds)
print('parameters post constrained fit: {}'.format(conpars))

pdf.expected_data(conpars)
```

```
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_histosys (1 new nuisance parameters)
```

```
[170.0, 220.0, 110.0, 105.0, 0.0]
parameters post unconstrained fit: [1.05563069e-12 4.00000334e+00]
parameters post constrained fit: [0.         4.00000146]
```

[5]:
```
array([ 1.25000007e+02,  1.60000022e+02,  2.10000022e+02, -8.00631284e-05,
        4.00000146e+00])
```

[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)

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.utils.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)]
```

(continues on next page)

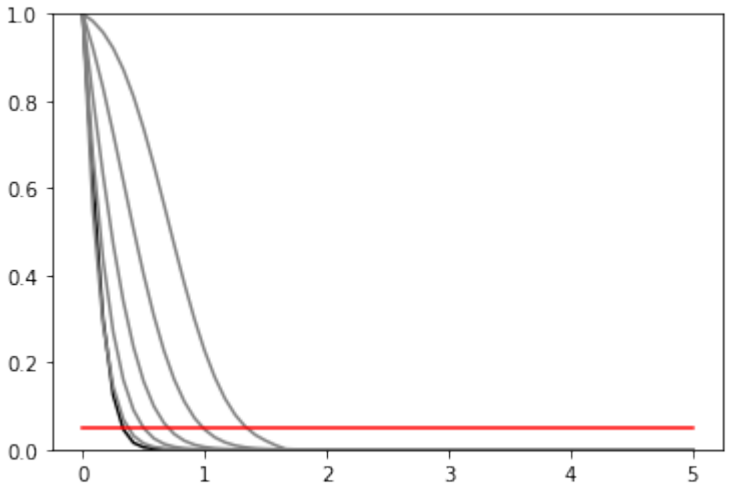
(continued from previous page)

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

```
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_histosys (1 new nuisance parameters)
```

```
[170.0, 220.0, 110.0, 105.0, 0.0]
parameters post unconstrained fit: [1.05563069e-12 4.00000334e+00]
parameters post constrained fit: [0.         4.00000146]
```

[6]:
```
{'exp': [0.37566312243018246,
  0.49824494455027707,
  0.7023047842202288,
  0.9869744452422563,
  1.3443167343146392],
 'obs': 0.329179494864517}
```



CHAPTER

SEVEN

TALKS

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" [arxiv: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. 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. (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. 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`:

- Lukas Heinrich. Introduction to pyhf. (Internal) ATLAS Induction Day + Software Tutorial, Oct 2019. URL: https://indico.cern.ch/event/831761/contributions/3484275/.

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.
- 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/.

CHAPTER

EIGHT

INSTALLATION

To install, we suggest first setting up a virtual environment

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

```
# Python2
virtualenv --python=$(which python) pyhf
```

and activating it

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

8.1 Install latest stable release from PyPI...

8.1.1 ... with NumPy backend

```
pip install pyhf
```

8.1.2 ... with TensorFlow backend

```
pip install pyhf[tensorflow]
```

8.1.3 ... with PyTorch backend

```
pip install pyhf[torch]
```

8.1.4 ... with all backends

```
pip install pyhf[tensorflow,torch]
```

8.1.5 ... with xml import/export functionality

```
pip install pyhf[xmlio]
```

8.2 Install latest development version from GitHub...

8.2.1 ... with NumPy backend

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

8.2.2 ... with TensorFlow backend

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

8.2.3 ... with PyTorch backend

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

8.2.4 ... with all backends

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

8.2.5 ... with xml import/export functionality

```
pip install --ignore-installed -U "git+https://github.com/diana-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.

CHAPTER

NINE

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/diana-hep/pyhf.git
```

and install all necessary packages for development

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

Then setup the Git pre-commit hook for Black by running

```
pre-commit install
```

9.1 Publishing

Publishing to PyPI and TestPyPI is automated through the PyPA's PyPI publish GitHub Action. To publish a release to PyPI one simply needs to run

```
bumpversion [major|minor|patch]
```

to update the release version and get a tagged commit and then push the commit and tag to `master` with

```
git push origin master --tags
```

CHAPTER

TEN

FAQ

Frequently Asked Questions about `pyhf` and its use.

10.1 Questions

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

Not at the moment. Pull Requests are welcome.

See also:

- #266

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.

CHAPTER

ELEVEN

API

11.1 Top-Level

default_backend	NumPy backend for pyhf
default_optimizer	
tensorlib	NumPy backend for pyhf
optimizer	
get_backend()	Get the current backend and the associated optimizer
set_backend(*args, **kwargs)	

11.1.1 pyhf.default_backend

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

11.1.2 pyhf.default_optimizer

pyhf.**default_optimizer = <pyhf.optimize.opt_scipy.scipy_optimizer object>**

11.1.3 pyhf.tensorlib

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

11.1.4 pyhf.optimizer

pyhf.**optimizer = <pyhf.optimize.opt_scipy.scipy_optimizer object>**

11.1.5 pyhf.get_backend

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

Example

```
>>> import pyhf
>>> pyhf.get_backend()
(<pyhf.tensor.numpy_backend.numpy_backend object at 0x...>, <pyhf.optimize.opt_
↪scipy.scipy_optimizer object at 0x...>)
```

Returns backend, optimizer

11.1.6 pyhf.set_backend

pyhf.**set_backend**(**args*, ***kwargs*)

11.2 Probability Distribution Functions (PDFs)

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

11.2.1 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

log_prob(*value*)
The log of the probability density function at the given value.

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

Returns The value of $\log(f\,(x|\theta))$ for $x =$`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

11.2.2 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

log_prob(*value*)

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

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

Returns The value of $\log(f\,(x|\theta))$ for $x =$`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

11.2.3 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

expected_data()

The expectation value of the probability density function.

Returns The expectation value of the distribution $\mathrm{E}\left[f(\theta)\right]$

Return type Tensor

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 $x =$ 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

11.2.4 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 $\mathrm{E}[f(\theta)]$

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 $x =$ `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

11.3 Making Models from PDFs

`Workspace`
`Model`
`_ModelConfig`

11.3.1 Model

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

Attributes

nominal_rates

Methods

__init__ (*spec*, *batch_size=None*, ***config_kwargs*)
Initialize self. See help(type(self)) for accurate signature.

constraint_logpdf (*auxdata*, *pars*)

expected_actualdata (*pars*)

expected_auxdata (*pars*)

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

logpdf (*pars*, *data*)

mainlogpdf (*maindata*, *pars*)

make_pdf (*pars*)

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

Returns A distribution object implementing the main measurement pdf of HistFactory

Return type pdf

pdf(*pars*, *data*)

11.3.2 _ModelConfig

class pyhf.pdf.**_ModelConfig**(*spec*, ***config_kwargs*)
Bases: object

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_init()

11.4 Backends

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

numpy_backend.numpy_backend	NumPy backend for pyhf
pytorch_backend.pytorch_backend	
tensorflow_backend.tensorflow_backend	

11.4.1 numpy_backend

class pyhf.tensor.numpy_backend.**numpy_backend**(***kwargs*)
Bases: object

NumPy backend for pyhf

Methods

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

abs(*tensor*)

astensor(*tensor_in*, *dtype='float'*)
Convert to a NumPy array.

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(pyhf.tensor.numpy_backend())
>>> 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 cliped to
- **max_value** (*scalar* or *tensor* or *None*) – The maximum value to be cliped 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 evaulation of a predicate

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> 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` evalutes to `true`
- **false_callable** (*callable*) – The callable that is evaluated when the `predicate` evalutes 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

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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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(pyhf.tensor.numpy_backend())
>>> 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*)

11.5 Optimizers

opt_pytorch.pytorch_optimizer
opt_scipy.scipy_optimizer
opt_tflow.tflow_optimizer
opt_minuit.minuit_optimizer

11.5.1 scipy_optimizer

class pyhf.optimize.opt_scipy.**scipy_optimizer**(***kwargs*)
Bases: object

Methods

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

constrained_bestfit(*objective*, *constrained_mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

unconstrained_bestfit(*objective*, *data*, *pdf*, *init_pars*, *par_bounds*)

11.6 Modifiers

histosys
normfactor
normsys
shapefactor
shapesys
staterror

11.6.1 histosys

class pyhf.modifiers.**histosys**
Bases: object

Attributes

is_constrained = True

op_code = 'addition'

pdf_type = 'normal'

Methods

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

classmethod required_parset(*n_parameters*)

11.6.2 normfactor

class pyhf.modifiers.**normfactor**
Bases: object

Attributes

is_constrained = False

op_code = 'multiplication'

pdf_type = None

Methods

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

classmethod required_parset(*n_parameters*)

11.6.3 normsys

class pyhf.modifiers.**normsys**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'normal'

Methods

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

classmethod required_parset(*n_parameters*)

11.6.4 shapefactor

class pyhf.modifiers.**shapefactor**
Bases: object

Attributes

is_constrained = False

op_code = 'multiplication'

pdf_type = None

Methods

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

classmethod required_parset(*n_parameters*)

11.6.5 shapesys

class pyhf.modifiers.**shapesys**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'poisson'

Methods

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

classmethod required_parset(*n_parameters*)

11.6.6 staterror

class pyhf.modifiers.**staterror**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'normal'

Methods

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

classmethod required_parset(*n_parameters*)

11.7 Interpolators

code0	The piecewise-linear interpolation strategy.
code1	The piecewise-exponential interpolation strategy.
code2	The quadratic interpolation and linear extrapolation strategy.
code4	The polynomial interpolation and exponential extrapolation strategy.
code4p	The piecewise-linear interpolation strategy, with polynomial at $\lvert a\rvert < 1$

11.7.1 code0

class pyhf.interpolators.**code0**(*histogramssets*, *subscribe=True*)
Bases: object

The piecewise-linear interpolation strategy.

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

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

11.7.2 code1

class pyhf.interpolators.**code1**(*histogramssets*, *subscribe=True*)
Bases: object

The piecewise-exponential interpolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) \underbrace{\prod_{p\in\text{Syst}} I_{\text{exp.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb})}_{\text{factors to calculate}}$$

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

11.7.3 code2

class pyhf.interpolators.**code2**(*histogramssets*, *subscribe=True*)
Bases: object

The quadratic interpolation and linear extrapolation strategy.

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

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

$$a = \frac{1}{2}(I^+ + I^-) - I^0 \qquad \text{and} \qquad b = \frac{1}{2}(I^+ - I^-)$$

Methods

__init__(*histogramssets*, *subscribe=True*)
Initialize self. See help(type(self)) for accurate signature.

11.7.4 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

$$\sigma_{sb}(\alpha = \pm\alpha_0),\ \frac{\mathrm{d}\sigma_{sb}}{\mathrm{d}\alpha}\bigg|_{\alpha=\pm\alpha_0}, \text{and } \frac{\mathrm{d}^2\sigma_{sb}}{\mathrm{d}\alpha^2}\bigg|_{\alpha=\pm\alpha_0}.$$

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

11.7.5 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*)
Initialize self. See help(type(self)) for accurate signature.

11.8 Exceptions

Various exceptions, apart from standard python exceptions, that are raised from using the `pyhf` API.

InvalidInterpCode	InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.
InvalidModifier	InvalidModifier is raised when an invalid modifier is requested.

11.8.1 InvalidInterpCode

class pyhf.exceptions.**InvalidInterpCode**
Bases: Exception

InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.

11.8.2 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

11.9 Utilities

generate_asimov_data(asimov_mu, data, pdf, ...)	
loglambdav(pars, data, pdf)	
pvals_from_teststat(sqrtqmu_v, sqrtqmuA_v[, ...])	The p-values for signal strength μ and Asimov strength μ' as defined in Equations (59) and (57) of **`arXiv:1007.1727`_**
pvals_from_teststat_expected(sqrtqmuA_v[, ...])	Computes the expected p-values CLsb, CLb and CLs for data corresponding to a given percentile of the alternate hypothesis.
qmu(mu, data, pdf, init_pars, par_bounds)	The test statistic, q_μ, for establishing an upper limit on the strength parameter, μ, as defiend in Equation (14) in **`arXiv:1007.1727`_** .
hypotest(poi_test, data, pdf[, init_pars, ...])	Computes p-values and test statistics for a single value of the parameter of interest

11.9.1 pyhf.utils.generate_asimov_data

pyhf.utils.**generate_asimov_data**(*asimov_mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

11.9.2 pyhf.utils.loglambdav

pyhf.utils.**loglambdav**(*pars*, *data*, *pdf*)

11.9.3 pyhf.utils.pvals_from_teststat

pyhf.utils.**pvals_from_teststat**(*sqrtqmu_v*, *sqrtqmuA_v*, *qtilde=False*)

The p-values for signal strength μ and Asimov strength μ' as defined in Equations (59) and (57) of arXiv:1007.1727

$$p_\mu = 1 - F\left(q_\mu|\mu'\right) = 1 - \Phi\left(q_\mu - \frac{(\mu - \mu')}{\sigma}\right)$$

with Equation (29)

$$\frac{(\mu - \mu')}{\sigma} = \sqrt{\Lambda} = \sqrt{q_{\mu,A}}$$

given the observed test statistics q_μ and $q_{\mu,A}$.

Parameters

- **sqrtqmu_v** (*Number or Tensor*) – The root of the calculated test statistic, $\sqrt{q_\mu}$
- **sqrtqmuA_v** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **qtilde** (*Bool*) – When `True` perform the calculation using the alternative test statistic, $\tilde{q}$, as defined in Equation (62) of arXiv:1007.1727

Returns The p-values for the signal + background, background only, and signal only hypotheses respectivley

Return type Tuple of Floats

11.9.4 pyhf.utils.pvals_from_teststat_expected

pyhf.utils.**pvals_from_teststat_expected**(*sqrtqmuA_v*, *nsigma=0*)

Computes the expected p-values CLsb, CLb and CLs for data corresponding to a given percentile of the alternate hypothesis.

Parameters

- **sqrtqmuA_v** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **nsigma** (*Number or Tensor*) – The number of standard deviations of variations of the signal strength from the background only hypothesis ($\mu = 0$)

Returns The p-values for the signal + background, background only, and signal only hypotheses respectivley

Return type Tuple of Floats

11.9.5 pyhf.utils.qmu

pyhf.utils.**qmu**(*mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

The test statistic, q_μ, for establishing an upper limit on the strength parameter, μ, as defiend in Equation (14) in arXiv:1007.1727 .

$$q_\mu = \begin{cases} -2\ln\lambda\left(\mu\right), & \hat{\mu} < \mu, \\ 0, & \hat{\mu} > \mu \end{cases} \tag{11.1}$$

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*Tensor*) – The data to be considered
- **pdf** (pyhf.pdf.Model) – The HistFactory statistical model used in the likelihood ratio calculation
- **init_pars** (*Tensor*) – The initial parameters
- **par_bounds** (*Tensor*) – The bounds on the paramter values

Returns The calculated test statistic, q_μ

Return type Float

11.9.6 pyhf.utils.hypotest

pyhf.utils.**hypotest**(*poi_test*, *data*, *pdf*, *init_pars=None*, *par_bounds=None*, *qtilde=False*, ***kwargs*)

Computes p-values and test statistics for a single value of the parameter of interest

Parameters

- **poi_test** (*Number or Tensor*) – The value of the parameter of interest (POI)
- **data** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **pdf** (pyhf.pdf.Model) – The HistFactory statistical model
- **init_pars** (*Array or Tensor*) – The initial parameter values to be used for minimization
- **par_bounds** (*Array or Tensor*) – The parameter value bounds to be used for minimization
- **qtilde** (*Bool*) – When `True` perform the calculation using the alternative test statistic, $\tilde{q}$, as defined in Equation (62) of arXiv:1007.1727

Keyword Arguments

- **return_tail_probs** (*bool*) – Bool for returning CL_{s+b} and CL_b
- **return_expected** (*bool*) – Bool for returning $\mathrm{CL}_{\mathrm{exp}}$
- **return_expected_set** (*bool*) – Bool for returning the $(-2,-1,0,1,2)\sigma$ $\mathrm{CL}_{\mathrm{exp}}$ — the "Brazil band"
- **return_test_statistics** (*bool*) – Bool for returning q_μ and $q_{\mu,A}$

Returns

- CL_s: The p-value compared to the given threshold α, typically taken to be 0.05, defined in arXiv:1007.1727 as

$$\mathrm{CL}_s = \frac{\mathrm{CL}_{s+b}}{\mathrm{CL}_b} = \frac{p_{s+b}}{1-p_b}$$

to protect against excluding signal models in which there is little sensitivity. In the case that $\mathrm{CL}_s \leq \alpha$ the given signal model is excluded.

- $[\mathrm{CL}_{s+b}, \mathrm{CL}_b]$: The signal + background p-value and 1 minus the background only p-value as defined in Equations (75) and (76) of arXiv:1007.1727

$$\mathrm{CL}_{s+b} = p_{s+b} = \int_{q_\mathrm{obs}}^{\infty} f\left(q\,|s+b\right)\,dq = 1 - \Phi\left(\frac{q_\mathrm{obs} + 1/\sigma_{s+b}^2}{2/\sigma_{s+b}}\right)$$

$$\mathrm{CL}_b = 1 - p_b = 1 - \int_{-\infty}^{q_\mathrm{obs}} f\left(q\,|b\right)\,dq = 1 - \Phi\left(\frac{q_\mathrm{obs} - 1/\sigma_{b}^2}{2/\sigma_{b}}\right)$$

with Equations (73) and (74) for the mean

$$E\left[q\right] = \frac{1-2\mu}{\sigma^2}$$

and variance

$$V\left[q\right] = \frac{4}{\sigma^2}$$

of the test statistic q under the background only and and signal + background hypotheses. Only returned when `return_tail_probs` is `True`.

- $\mathrm{CL}_{s,\mathrm{exp}}$: The expected CL_s value corresponding to the test statistic under the background only hypothesis ($\mu = 0$). Only returned when `return_expected` is `True`.
- $\mathrm{CL}_{s,\mathrm{exp}}$ 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 arXiv:1007.1727

$$\mathrm{band}_{N\sigma} = \mu' + \sigma\,\Phi^{-1}\left(1-\alpha\right) \pm N\sigma$$

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`.

- $[q_\mu, q_{\mu,A}]$: The test statistics for the observed and Asimov datasets respectively. Only returned when `return_test_statistics` is `True`.

Return type Tuple of Floats and lists of Floats

CHAPTER

TWELVE

USE AND CITATIONS

Updating list of citations and use cases of `pyhf`:

- 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.

CHAPTER

THIRTEEN

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.

13.1 Hello World

```
>>> import pyhf
>>> pdf = pyhf.simplemodels.hepdata_like(signal_data=[12.0, 11.0], bkg_data=[50.0, 52.
↪0], bkg_uncerts=[3.0, 7.0])
>>> CLs_obs, CLs_exp = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected=True)
>>> print('Observed: {}, Expected: {}'.format(CLs_obs, CLs_exp))
Observed: [0.05290116], Expected: [0.06445521]
```

13.2 What does it support

Implemented variations:

- [x] HistoSys
- [x] OverallSys
- [x] ShapeSys
- [x] NormFactor
- [x] Multiple Channels
- [x] Import from XML + ROOT via uproot
- [x] ShapeFactor
- [x] StatError
- [x] Lumi Uncertainty

Computational Backends:

- [x] NumPy
- [x] PyTorch
- [x] TensorFlow

Available Optimizers

NumPy	Tensorflow	PyTorch	MxNet
SLSQP (`scipy.optimize`)	Newton's Method (autodiff)	Newton's Method (autodiff)	N/A
MINUIT (`iminuit`)	.	.	.

13.3 Todo

- [ ] StatConfig
- [ ] Non-asymptotic calculators

results obtained from this package are validated against output computed from HistFactory workspaces

13.4 A one bin example

```
nobs = 55, b = 50, db = 7, nom_sig = 10.
```

13.5 A two bin example

```
bin 1: nobs = 100, b = 100, db = 15., nom_sig = 30.
bin 2: nobs = 145, b = 150, db = 20., nom_sig = 45.
```

13.6 Installation

To install `pyhf` from PyPI with the NumPy backend run

```
pip install pyhf
```

and to install `pyhf` with additional backends run

```
pip install pyhf[tensorflow,torch]
```

or a subset of the options.

To uninstall run

```
pip uninstall pyhf
```

13.7 Authors

Please check the contribution statistics for a list of contributors

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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.

[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.

[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.

[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.

[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.

[1] Histfactory definitions schema. Accessed: 2019-06-20. URL: https://diana-hep.org/pyhf/schemas/1.0.0/defs.json.

[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.

INDEX

Symbols

A

B

C

D

E

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Z