zfit Documentation

Release 0.4.0

zfit

Jan 06, 2020

Contents

The zfit package is a model fitting library based on TensorFlow and optimised for simple and direct manipulation of probability density functions. The main focus is on the scalability, parallelisation and a user friendly experience framework (no cython, no C++ needed to extend). The basic idea is to offer a pythonic oriented alternative to the very successful RooFit library from the ROOT data analysis package. While RooFit has provided a stable platform for most of the needs of the High Energy Physics (HEP) community in the last few years, it has become increasingly difficult to integrate all the developments in the scientific Python ecosystem into RooFit due to its monolithic nature. Conversely, the core of zfit aims at becoming a solid ground for model fitting while providing enough flexibility to incorporate state-of-art tools and to allow scalability going to larger datasets. This challenging task is tackled by following two basic design pillars:

- The skeleton and extension of the code is minimalist, simple and finite: the zfit library is exclusively designed for the purpose of model fitting and sampling—opposite to the self-contained RooFit/ROOT frameworks—with no attempt to extend its functionalities to features such as statistical methods or plotting. This design philosophy is well exemplified by examining maximum likelihood fits: while zfit works as a backend for likelihood fits and can be integrated to packages such as lauztat and matplotlib, RooFit performs the fit, the statistical treatment and plotting within. This wider scope of RooFit results in a lack of flexibility with respect to new minimisers, statistic methods and, broadly speaking, any new tool that might come.
- Another paramount aspect of zfit is its design for optimal parallelisation and scalability. Even though the choice of TensorFlow as backend introduces a strong software dependency, its use provides several interesting features in the context of model fitting. The key concept is that TensorFlow is built under the dataflow programming model. Put it simply, TensorFlow creates a computational graph with the operations as the nodes of the graph and tensors to its edges. Hence, the computation only happens when the graph is executed in a session, which simplifies the parallelisation by identifying the dependencies between the edges and operations or even the partition across multiple devices (more details can be found in the TensorFlow guide). The architecture of zfit is built upon this idea and it aims to provide a high level interface to these features, *i.e.*, most of the operations of graphs and evaluations are hidden for the user, leaving a natural and friendly model fitting and sampling experience.

The zfit package is Free software, using an Open Source license. Both the software and this document are works in progress. Source code can be found in our github page.

CHAPTER 1

Getting started with zfit

The zfit library provides a simple model fitting and sampling framework for a broad list of applications. This section is designed to give an overview of the main concepts and features in the context of likelihood fits in a *crash course* manner. The simplest example is to generate, fit and plot a Gaussian distribution.

The first step is to naturally import `zfit` and verify if the installation has been done successfully:

```
>>> import tensorflow as tf
>>> import zfit
>>> print("TensorFlow version:", tf.__version__)
TensorFlow version: 1.12.0
```

Since we want to generate/fit a Gaussian within a given range, the domain of the PDF is defined by an *observable space*. This can be created using the *Space* class

```
>>> obs = zfit.Space('x', limits=(-10, 10))
```

The best interpretation of the observable at this stage is that it defines the name and range of the observable axis.

Using this domain, we can now create a simple Gaussian PDF. The most common PDFs are already pre-defined within the *pdf* module, including a simple Gaussian. First, we have to define the parameters of the PDF and their limits using the *Parameter* class:

```
>>> mu = zfit.Parameter("mu", 2.4, -1, 5)
>>> sigma = zfit.Parameter("sigma", 1.3,  0, 5)
```

With these parameters we can instantiate the Gaussian PDF from the library

```
>>> gauss = zfit.pdf.Gauss(obs=obs, mu=mu, sigma=sigma)
```

It is recommended to pass the arguments of the PDF as keyword arguments.

The next stage is to create a dataset to be fitted. There are several ways of producing this within the zfit framework (see the *Data* section). In this case, for simplicity we simply produce it using numpy and the `Data.from_numpy` method:

```
>>> data_np = np.random.normal(0, 1, size=10000)
>>> data = zfit.Data.from_numpy(obs=obs, array=data_np)
```

Now we have all the ingredients in order to perform a maximum likelihood fit. Conceptually this corresponds to three basic steps:

1. create a loss function, in our case a negative log-likelihood $\log \mathcal{L}$;
2. instantiate our choice of minimiser; and
3. and minimise the log-likelihood.

```
>>> # Stage 1: create an unbinned likelihood with the given PDF and dataset
>>> nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

>>> # Stage 2: instantiate a minimiser (in this case a basic minuit
>>> minimizer = zfit.minimize.Minuit()

>>> # Stage 3: minimise the given negative likelihood
>>> result = minimizer.minimize(nll)
```

This corresponds to the most basic example where the negative likelihood is defined within the pre-determined observable range and all the parameters in the PDF are floated in the fit. It is often the case that we want to only vary a given set of parameters. In this case it is necessary to specify which are the parameters to be floated (so all the remaining ones are fixed to their initial values).

```
>>> # Stage 3: minimise the given negative likelihood but floating only specific␣
↪parameters (e.g. mu)
>>> result = minimizer.minimize(nll, params=[mu])
```

It is important to highlight that conceptually zfit separates the minimisation of the loss function with respect to the error calculation, in order to give the freedom of calculating this error whenever needed and to allow the use of external error calculation packages. Most minimisers will implement their CPU-intensive error calculating with the `error` method. As an example, with the *`Minuit`* one can calculate the `MINOS` with:

```
>>> param_errors = result.error()
>>> for var, errors in param_errors.items():
...   print('{}: ^{{+{}}}_{{{}}}'.format(var.name, errors['upper'], errors['lower']))
mu: ^{+0.00998104141841555}_{-0.009981515893414316}
sigma: ^{+0.007099472590970696}_{-0.007016265476493973 4}
```

Once we've performed the fit and obtained the corresponding uncertainties, it is now important to examine the fit results. The object `result` (*`FitResult`*) has all the relevant information we need:

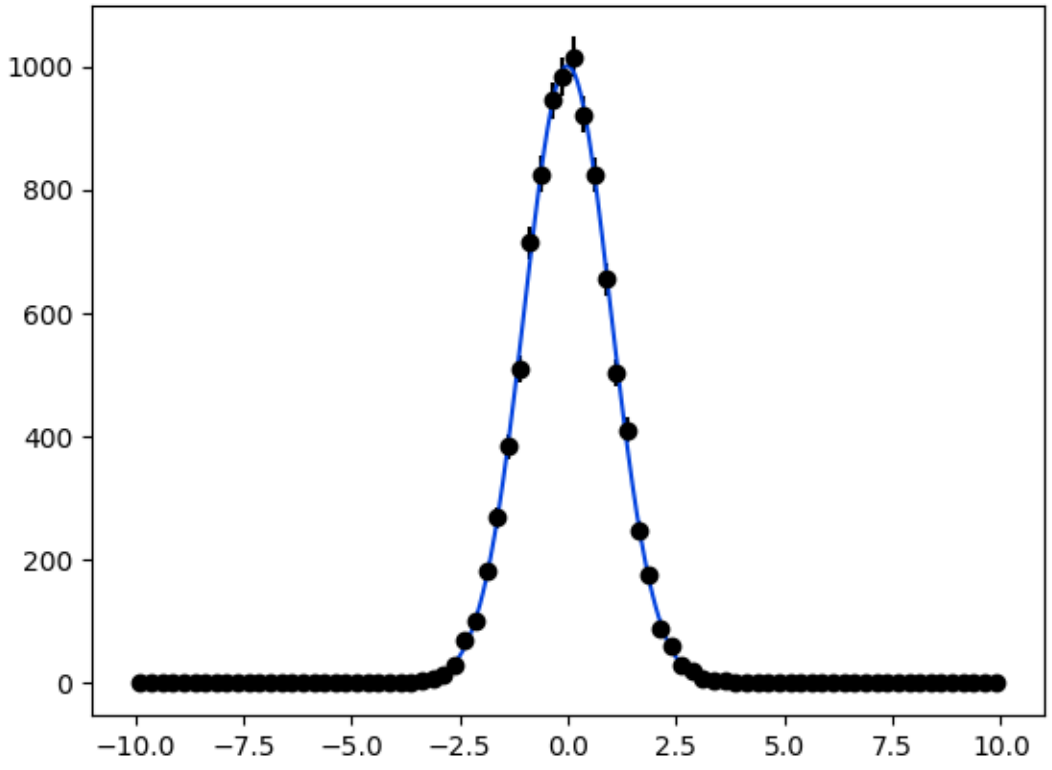
```
>>> print("Function minimum:", result.fmin)
Function minimum: 14170.396450111948
>>> print("Converged:", result.converged)
Converged: True
>>> print("Full minimizer information:", result.info)
Full minimizer information: {'n_eval': 56, 'original': {'fval': 14170.396450111948,
↪'edm': 2.8519671693442587e-10,
'nfcn': 56, 'up': 0.5, 'is_valid': True, 'has_valid_parameters': True, 'has_accurate_
↪covar': True, 'has_posdef_covar': True,
'has_made_posdef_covar': False, 'hesse_failed': False, 'has_covariance': True, 'is_
↪above_max_edm': False, 'has_reached_call_limit': False}}
```

Similarly one can obtain information on the fitted parameters with

```
>>> # Information on all the parameters in the fit
>>> params = result.params

>>> # Printing information on specific parameters, e.g. mu
>>> print("mu={}".format(params[mu]['value']))
mu=0.012464509810750313
```

As already mentioned, there is no dedicated plotting feature within zfit. However, we can easily use external libraries, such as `matplotlib`, to do the job:

```
>>> # Some simple matplotlib configurations
>>> import matplotlib.pyplot as plt
>>> lower, upper = obs.limits
>>> data_np = zfit.run(data)
>>> counts, bin_edges = np.histogram(data_np, 80, range=(lower[-1][0], upper[0][0]))
>>> bin_centres = (bin_edges[:-1] + bin_edges[1:])/2.
>>> err = np.sqrt(counts)
>>> plt.errorbar(bin_centres, counts, yerr=err, fmt='o', color='xkcd:black')

>>> x_plot = np.linspace(lower[-1][0], upper[0][0], num=1000)
>>> y_plot = zfit.run(gauss.pdf(x_plot, norm_range=obs))

>>> plt.plot(x_plot, y_plot*data_np.shape[0]/80*obs.area(), color='xkcd:blue')
>>> plt.show()
```



The plotting example above presents a distinctive feature that had not been shown in the previous exercises: the

specific call to `zfit.run`, a specialised wrapper around `tf.Session().run`. While actions like `minimize` or `sample` return Python objects (including numpy arrays or scalars), functions like `pdf` or `integrate` return TensorFlow graphs, which are lazy-evaluated. To obtain the value of these PDFs, we need to execute the graph by using `zfit.run`.

1.1 What did just happen?

The core idea of TensorFlow is to use dataflow *graphs*, in which *sessions* run part of the graphs that are required. Since zfit has TensorFlow at its core, it also preserves this feature, but wrapper functions are used to hide the graph generation and graph running two-stage procedure in the case of high-level functions such as `minimize`. However, it is worth noting that most of the internal objects that are built by zfit are intrinsically graphs that are executed by running the session:

```
zfit.run(TensorFlow_object)
```

One example is the Gauss PDF that has been shown above. The object `gauss` contains all the functions you would expect from a PDF, such as calculating a probability, calculating its integral, etc. As an example, let's calculate the probability for given values

```
>>> from zfit import z
>>> consts = [-1, 0, 1]
>>> probs = gauss.pdf(consts, norm_range=(-np.infty, np.infty))

>>> # And now execute the tensorflow graph
>>> result = zfit.run(probs)
>>> print("x values: {}\nresult:   {}".format(consts, result))
x values: [-1, 0, 1]
result:   [0.24262615 0.39670691 0.24130008]
```

Integrating a given PDF for a given normalisation range also returns a graph, so it needs to be run using

```
>>> from zfit import z
>>> consts = [-1, 0, 1]
>>> probs = gauss.pdf(consts, norm_range=(-np.infty, np.infty))
```

```
>>> # And now execute the tensorflow graph
>>> result = zfit.run(probs)
>>> print("x values: {}\nresult:   {}".format(consts, result))
x values: [-1, 0, 1]
result:   [0.24262615 0.39670691 0.24130008]
```

Integrating a given PDF for a given normalisation range also returns a graph, so it needs to be run using

```
>>> from zfit import ztf
>>> consts = [-1, 0, 1]
>>> probs = gauss.pdf(consts, norm_range=(-np.infty, np.infty))
```

```
>>> # And now execute the tensorflow graph
>>> result = zfit.run(probs)
>>> print("x values: {}\nresult:   {}".format(consts, result))
x values: [-1, 0, 1]
result:   [0.24262615 0.39670691 0.24130008]
```

Integrating a given PDF for a given normalisation range also returns a graph, so it needs to be run using `zfit.run`:

```
>>> with gauss.set_norm_range((-1e6, 1e6)):
...     print(zfit.run(gauss.integrate((-0.6, 0.6))))
...     print(zfit.run(gauss.integrate((-3, 3))))
...     print(zfit.run(gauss.integrate((-100, 100))))
0.4492509559828224
0.9971473939649167
1.0
```

CHAPTER 2

zfit introduction

Following and introduction to the elements of zfit

2.1 Space, Observable and Range

Inside zfit, *Space* defines the domain of objects by specifying the observables/axes and *maybe* also the limits. Any model and data needs to be specified in a certain domain, which is usually done using the `obs` argument. It is crucial that the axis used by the observable of the data and the model match, and this matching is handle by the *Space* class.

```
obs = zfit.Space("x")
model = zfit.pdf.Gauss(obs=obs, ...)
data = zfit.Data.from_numpy(obs=obs, ...)
```

2.1.1 Definitions

Space: an *n*-dimensional definition of a domain (either by using one or more observables or axes), with or without limits.

Note: *compared to 'RooFit', a space is **not* the equivalent of an observable but rather corresponds to an object combining **a set** of observables (which of course can be of size 1). Furthermore, there is a **strong** distinction in zfit between a *Space* (or observables) and a *Parameter*, both conceptually and in terms of implementation and usage.*

Observable: a string defining the axes; a named axes.

(for advanced usage only, can be skipped on first read) **Axis**: integer defining the axes *internally* of a model. There is always a mapping of observables <-> axes *once inside a model*.

Limit The range on a certain axis. Typically defines an interval.

Since every object has a well defined domain, it is possible to combine them in an unambiguous way

```
obs1 = zfit.Space(['x', 'y'])
obs2 = zfit.Space(['z', 'y'])

model1 = zfit.pdf.Gauss(obs=obs1, ...)
model2 = zfit.pdf.Gauss(obs=obs2, ...)

# creating a composite pdf
product = model1 * model2
# OR, equivalently
product = zfit.pdf.ProductPDF([model1, model2])
```

The `product` is now defined in the space with observables *['x', 'y', 'z']*. Any `Data` object to be combined with `product` has to be specified in the same space.

```
# create the space
combined_obs = obs1 * obs2

data = zfit.Data.from_numpy(obs=combined_obs, ...)
```

Now we have a `Data` object that is defined in the same domain as *product* and can be used to build a loss function.

2.1.2 Limits

In many places, just defining the observables is not enough and an interval, specified by its limits, is required. Examples are a normalization range, the limits of an integration or sampling in a certain region.

Simple, 1-dimensional limits can be specified as follows. Operations like addition (creating a space with two intervals) or combination (increase the dimensionality) are also possible.

```
simple_limit1 = zfit.Space(obs='obs1', limits=(-5, 1))
simple_limit2 = zfit.Space(obs='obs1', limits=(3, 7.5))

added_limits = simple_limit1 + simple_limit2
```

In this case, *added_limits* is now a *Space* with observable *'obs1'* defined in the intervals (-5, 1) and (3, 7.5). This can be useful, *e.g.*, when fitting in two regions. An example of the product of different *Space* instances has been shown before as `combined_obs`.

Defining limits

To define simple, 1-dimensional limits, a tuple with two numbers is enough. For anything more complicated, the definition works as follows:

```
first_limit_lower = (low_1_obs1, low_1_obs2,...)
first_limit_upper = (up_1_obs1, up_1_obs2,...)

second_limit_lower = (low_2_obs1, low_2_obs2,...)
second_limit_upper = (up_2_obs1, up_2_obs2,...)

...

lower = (first_limit_lower, second_limit_lower, ...)
upper = (first_limit_upper, second_limit_upper, ...)

limits = (lower, upper)
```

(continues on next page)

(continued from previous page)

```
space1 = zfit.Space(obs=['obs1', 'obs2', ...], limits=limits)
```

This defines the area from

- *low_1_obs1* to *up_1_obs1* in the first observable *'obs1'*;
- *low_1_obs2* to *up_1_obs2* in the second observable *'obs2'*;
- ...

the area from

- *low_2_obs1* to *up_2_obs1* in the first observable *'obs1'*;
- *low_2_obs2* to *up_2_obs2* in the second observable *'obs2'*;
- ...

and so on.

A working code example of *Space* handling is provided in *spaces.py* in examples.

2.2 Parameter

Several objects in zfit, most importantly models, have one or more parameter which typically parametrise a function or distribution. There are two different kinds of parameters in zfit:

- Independent: can be changed in a fit (or explicitly be set to *fixed*).
- Dependent: **cannot** be directly changed but _may_ depend on independent parameters.

2.2.1 Independent Parameter

To create a parameter that can be changed, *e.g.*, to fit a model, a *Parameter* has to be instantiated.

The syntax is as follows:

```
param1 = zfit.Parameter("param_name_human_readable", start_value[, lower_limit, upper_
↪limit])
```

Furthermore, a `step_size` can be specified. If not, it is set to a default value around 0.001. *Parameter* can have limits (tested with *has_limits()*), which will clip the value to the limits given by *lower_limit()* and *upper_limit()*. While this closely follows the RooFit syntax, it is very important to note that the optional limits of the parameter behave differently: if not given, the parameter will be "unbounded", not fixed (as in RooFit). Parameters are therefore floating by default, but their value can be fixed by setting the attribute `floating` to `False` or already specifying it in the init.

The value of the parameter can be changed with the *set_value()* method. Using this method as a context manager, the value can also temporarily changed. However, be aware that anything _dependent_ on the parameter will have a value with the parameter evaluated with the new value at run-time:

```
>>> mu = zfit.Parameter("mu_one", 1)  # no limits, but FLOATING (!)
>>> with mu.set_value(3):
...     # in here, mu has the value 3
...     mu_val = zfit.run(mu)  # 3
...     five_mu = 5 * mu
```

(continues on next page)

(continued from previous page)

```
...     five_mu_val = zfit.run(five_mu)  # is evaluated with mu = 5. -> five_mu_val is␣
↪15

>>> # here, mu is again 1
>>> mu_val_after = zfit.run(mu)  # 1
>>> five_mu_val_after = zfit.run(five_mu)  # is evaluated with mu = 1! -> five_mu_val_
↪after is 5
```

2.2.2 Dependent Parameter

A parameter can be composed of several other parameters. We can use any `Tensor` for that and the dependency will be detected automatically. They can be used equivalently to *`Parameter`*.

```
>>> mu2 = zfit.Parameter("mu_two", 7)
>>> dependent_func = lambda: mu * 5 + mu2  # or any kind of computation
>>> dep_param = zfit.ComposedParameter("dependent_param", dependent_func,␣
↪dependents=[mu, mu2])

>>> dependents = dep_param.get_dependents()  # returns ordered-set(mu, mu2)
```

A special case of the above is *`ComplexParameter`*: it takes a complex `tf.Tensor` as input and provides a few special methods (like *`real()`*, `ComplexParameterconj()` etc.) to easier deal with them. Additionally, the *`from_cartesian()`* and *`from_polar()`* methods can be used to initialize polar parameters from floats, avoiding the need of creating complex `tf.Tensor` objects.

2.3 Building a model

In order to build a generic model the concept of function and distributed density functions (PDFs) need to be clarified. The PDF, or density of a continuous random variable, of X is a function f(x) that describes the relative likelihood for this random variable to take on a given value. In this sense, for any two numbers a and b with $a \leq b$,

$P(a \leq X \leq b) = \int_a^b f(X)dx$

That is, the probability that X takes on a value in the interval $[a, b]$ is the area above this interval and under the graph of the density function. In other words, in order to a function to be a PDF it must satisfy two criteria: 1. $f(x) \geq 0$ for all x; 2. $\int_{-\infty}^{\infty} f(x)dx =$ are under the entire graph of $f(x) = 1$. In zfit these distinctions are respected, *i.e.*, a function can be converted into a PDF by imposing the basic two criteria above... _basic-model:

2.3.1 Predefined PDFs and basic properties

A series of predefined PDFs are available to the users and can be easily accessed using autocompletion (if available). In fact, all of these can also be seen in

```
>>> print(zfit.pdf.__all__)
['BasePDF', 'BaseFunctor', 'Exponential', 'CrystalBall', 'DoubleCB', 'Gauss', 'Uniform
↪', 'TruncatedGauss', 'WrapDistribution', 'Chebyshev', 'Legendre', 'Chebyshev2',
↪'Hermite', 'Laguerre', 'RecursivePolynomial', 'ProductPDF', 'SumPDF', 'ZPDF',
↪'SimplePDF', 'SimpleFunctorPDF']
```

These include the basic function but also some operations discussed below. Let's consider the simple example of a `CrystalBall`. PDF objects must also be initialised giving their named parameters. For example:

```
>>> obs = zfit.Space('x', limits=(4800, 6000))

>>> # Creating the parameters for the crystal ball
>>> mu = zfit.Parameter("mu", 5279, 5100, 5300)
>>> sigma = zfit.Parameter("sigma", 20, 0, 50)
>>> a = zfit.Parameter("a", 1, 0, 10)
>>> n = zfit.Parameter("n", 1, 0, 10)

>>> # Single crystal Ball
>>> model_cb = zfit.pdf.CrystalBall(obs=obs, mu=mu, sigma=sigma, alpha=a, n=n)
```

In this case the CB object corresponds to a normalised PDF. The main properties of a PDF, e.g. the probability for a given normalisation range or even to set a temporary normalisation range can be given as

```
>>> # Get the probabilities of some random generated events
>>> probs = model_cb.pdf(x=np.random.random(10))
>>> # And now execute the tensorflow graph
>>> result = zfit.run(probs)
>>> print(result)
[3.34187765e-05 3.34196917e-05 3.34202989e-05 3.34181458e-05
 3.34172973e-05 3.34209238e-05 3.34164538e-05 3.34210950e-05
 3.34201199e-05 3.34209360e-05]

>>> # The norm range of the pdf can be changed any time by
>>> model_cb.set_norm_range((5000, 6000))
```

Another feature for the PDF is to calculate its integral in a certain limit. This can be easily achieved by

```
>>> # Calculate the integral between 5000 and 5250 over the PDF normalized
>>> integral_norm = model_cb.integrate(limits=(5000, 5250))
```

In this case the CB has been normalised using the range defined in the observable. Conversely, the `norm_range` in which the PDF is normalised can also be specified as input.

2.3.2 Composite PDF

A common feature in building composite models it the ability to combine in terms of sum and products different PDFs. There are two ways to create such models, either with the class API or with simple Python syntax. Let's consider a second crystal ball with the same mean position and width, but different tail parameters

```
>>> # New tail parameters for the second CB
>>> a2 = zfit.Parameter("a2", -1, 0, -10)
>>> n2 = zfit.Parameter("n2", 1, 0, 10)

>>> # New crystal Ball function defined in the same observable range
>>> model_cb2 = zfit.pdf.CrystalBall(obs=obs, mu=mu, sigma=sigma, alpha=a2, n=n2)
```

We can now combine these two PDFs to create a double Crystal Ball with a single mean and width, either using arithmetic operations

```
>>> # First needs to define a parameters that represent
>>> # the relative fraction between the two PDFs
>>> frac = zfit.Parameter("frac", 0.5, 0, 1)

>>> # Two different ways to combine
>>> double_cb = frac * model_cb + model_cb2
```

Or through the *zfit.pdf.SumPDF* class:

```
>>> # or via the class API
>>> double_cb_class = zfit.pdf.SumPDF(pdfs=[model_cb, model_cb2], fracs=frac)
```

Notice that the new PDF has the same observables as the original ones, as they coincide. Alternatively one could consider having PDFs for different axis, which would then create a totalPDF with higher dimension.

A simple extension of these operations is if we want to instead of a sum of PDFs, to model a two-dimensional Gaussian (e.g.):

```
>>> # Defining two Gaussians in two different axis (obs)
>>> mu1 = zfit.Parameter("mu1", 1.)
>>> sigma1 = zfit.Parameter("sigma1", 1.)
>>> gauss1 = zfit.pdf.Gauss(obs="obs1", mu=mu1, sigma=sigma1)

>>> mu2 = zfit.Parameter("mu2", 1.)
>>> sigma2 = zfit.Parameter("sigma2", 1.)
>>> gauss2 = zfit.pdf.Gauss(obs="obs2", mu=mu2, sigma=sigma2)

>>> # Producing the product of two PDFs
>>> prod_gauss = gauss1 * gauss2
>>> # Or alternatively
>>> prod_gauss_class = zfit.pdf.ProductPDF(pdfs=[gauss2, gauss1])  # notice the␣
↪different order or the pdf
```

The new PDF is now in two dimensions. The order of the observables follows the order of the PDFs given.

```
>>> print("python syntax product obs", prod_gauss.obs)
[python syntax product obs ('obs1', 'obs2')]
>>> print("class API product obs", prod_gauss_class.obs)
[class API product obs ('obs2', 'obs1')]
```

2.3.3 Extended PDF

In the event there are different *species* of distributions in a given observable, the simple sum of PDFs does not a priori provides the absolute number of events for each specie but rather the fraction as seen above. An example is a Gaussian mass distribution with an exponential background, e.g.

$$P = f_S \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} + (1 - f_S) e^{-\alpha x}$$

Since we are interested to express a measurement of the number of events, the expression $M(x) = N_S S(x) + N_B B(x)$ respect that M(x) is normalised to $N_S + N_B = N$ instead of one. This means that $M(x)$ is not a true PDF but rather an expression for two quantities, the shape and the number of events in the distributions.

An extended PDF can be easily implemented in zfit in two ways:

```
>>> # Create a parameter for the number of events
>>> yieldGauss = zfit.Parameter("yieldGauss", 100, 0, 1000)

>>> # Extended PDF using a predefined method
>>> extended_gauss_method = gauss.create_extended(yieldGauss)
>>> # Or simply with a Python syntax of multiplying a PDF with the parameter
>>> extended_gauss_python = yieldGauss * gauss
```

2.3.4 Custom PDF

A fundamental design choice of zfit is the ability to create custom PDFs and functions in an easy way. Let's consider a simplified implementation

```
>>> class MyGauss(zfit.pdf.ZPDF):
...     """Simple implementation of a Gaussian similar to :py:class`~zfit.pdf.Gauss`␣
↪class"""
...     _N_OBS = 1  # dimension, can be omitted
...     _PARAMS = ['mean', 'std']  # the name of the parameters

>>> def _unnormalized_pdf(self, x):
...     x = zfit.ztf.unstack_x(x)
...     mean = self.params['mean']
...     std  = self.params['std']
...     return zfit.ztf.exp(- ((x - mean)/std)**2)
```

This is the basic information required for this custom PDF. With this new PDF one can access the same feature of the predefined PDFs, e.g.

```
>>> obs = zfit.Space("obs1", limits=(-4, 4))

>>> mean = zfit.Parameter("mean", 1.)
>>> std  = zfit.Parameter("std", 1.)
>>> my_gauss = MyGauss(obs='obs1', mean=mean, std=std)

>>> # For instance integral probabilities
>>> integral = my_gauss.integrate(limits=(-1, 2))
>>> probs    = my_gauss.pdf(data, norm_range=(-3, 4))
```

Finally, we could also improve the description of the PDF by providing a analytical integral for the `MyGauss` PDF:

```
>>> def gauss_integral_from_any_to_any(limits, params, model):
...     (lower,), (upper,) = limits.limits
...     mean = params['mean']
...     std = params['std']
...     # Write you integral
...     return 42. # Dummy value

>>> # Register the integral
>>> limits = zfit.Space.from_axes(axes=0, limits=(zfit.Space.ANY_LOWER, zfit.Space.
↪ANY_UPPER))
>>> MyGauss.register_analytic_integral(func=gauss_integral_from_any_to_any,␣
↪limits=limits)
```

Sampling from a Model

In order to sample from model, there are two different methods, *`sample()`* for **advanced** sampling returning a Tensor, and *`create_sampler()`* for **multiple sampling** as used for toys.

Tensor sampling

The sample from *`sample()`* is a Tensor that samples when executed. This is for an advanced usecase only

Playing with toys: Multiple samplings

The method *create_sampler()* returns a sampler that can be used like a Data object (e.g. for building a *ZfitLoss*). The sampling itself is *not yet done* but only when *resample()* is invoked. The sample generated depends on the original pdf at this point, e.g. parameters have the value they have when the *resample()* is invoked. To have certain parameters fixed, they have to be specified *either* on *create_sampler()* via *fixed_params*, on *resample()* by specifying which parameter will take which value via *param_values* or by changing the attribute of *Sampler*.

How typically toys look like: .. _playing_with_toys:

A typical example of toys would therefore look like

```
>>> # create a model depending on mu, sigma

>>> sampler = model.create_sampler(n=1000, fixed_params=True)
>>> nll = zfit.loss.UnbinnedNLL(model=model, data=sampler)

>>> minimizer = zfit.minimize.Minuit()

>>> for run_number in n_runs:
...     # initialize the parameters randomly
...     sampler.resample()  # now the resampling gets executed
...
...     mu.set_value(np.random.normal())
...     sigma.set_value(abs(np.random.normal()))
...
...     result = minimizer.minimize(nll)
...
...     # safe the result, collect the values, calculate errors...
```

Here we fixed all parameters as they have been initialized and then sample. If we do not provide any arguments to *resample*, this will always sample now from the distribution with the parameters set to the

values when the sampler was created.

To give another, though not very useful example:

```
>>> # create a model depending on mu1, sigma1, mu2, sigma2

>>> sampler = model.create_sampler(n=1000, fixed_params=[mu1, mu2])
>>> nll = zfit.loss.UnbinnedNLL(model=model, data=sampler)

>>> sampler.resample()  # now it sampled

>>> # do something with nll
>>> minimizer.minimize(nll)  # minimize

>>> sampler.resample()
>>> # note that the nll, being dependent on `sampler`, also changed!
```

The sample is now resampled with the *current values* (minimized values) of *sigma1*, *sigma2* and with the initial values of *mu1*, *mu2* (because they have been fixed).

We can also specify the parameter values explicitly by using the following argument. Reusing the example above

```
>>> sigma.set_value(np.random.normal())
>>> sampler.resample(param_values={sigma1: 5})
```

The sample (and therefore also the sample the *nll* depends on) is now sampled with *sigma1* set to 5.

If some parameters are constrained from external measurements, usually Gaussian constraints, then sampling of those parameters might be needed to obtain an unbiased sample from the model. Example:

```
>>> # same model depending on mu1, sigma1, mu2, sigma2

>>> constraint = zfit.constraint.GaussianConstraint(params=[sigma1, sigma2], mu=[1.0,␣
↪0.5], sigma=[0.1, 0.05])

>>> n_samples = 1000

>>> sampler = model.create_sampler(n=n_samples, fixed_params=[mu1, mu2])
>>> nll = zfit.loss.UnbinnedNLL(model=model, data=sampler, constraints=constraint)

>>> constr_values = constraint.sample(n=n_samples)

>>> for i in range(n_samples):
>>>     sampler.resample(param_values={sigma1: constr_values[sigma1][i],
>>>                                    sigma2: constr_values[sigma2][i]})
>>>     # do something with nll
>>>     minimizer.minimize(nll)  # minimize
```

2.4 Data

An easy and fast data manipulation are among the crucial aspects in High Energy Particle physics data analysis. With the increasing data availability (e.g. with the advent of LHC), this challenge has been pursued in different manners. Common strategies vary from multidimensional arrays with attached row/column labels (e.g. `DataFrame` in *pandas*) or compressed binary formats (e.g. ROOT). While each of these data structure designs has their own advantages in terms of speed and acessibility, the data concept inplemented in `zfit` follows closely the features of `DataFrame` in *pandas*.

The `Data` class provides a simple and structured access/manipulation of *data* – similarly to concept of multidimensional arrays approach from *pandas*. The key feature of `Data` is its relation to the *`Space`* or more explicitly its axis or name. A more equally convention is to name the role of the *`Space`* in this context as the *observable* under investigation. Note that no explicit range for the *`Space`* is required at the moment of the data definition, since this is only required at the moment some calculation is needed (e.g. integrals, fits, etc).

2.4.1 Import dataset from a ROOT file

With the proliferation of the ROOT framework in the context of particle physics, it is often the case that the user will have access to a ROOT file in their analysis. A simple method has been used to handle this conversion:

```
>>> data = zfit.Data.from_root(root_file,
...                            root_tree,
...                            branches)
```

where `root_file` is the path to the ROOT file, `root_tree` is the tree name and branches are the list (or a single) of branches that the user wants to import from the ROOT file.

From the default conversion of the dataset there are two optional funcionalities for the user, i.e. the use of weights and the rename of the specified branches. The nominal structure follows:

```
>>> data = zfit.Data.from_root(root_file,
...                            root_tree,
...                            branches,
```

(continues on next page)

(continued from previous page)

```
...                                     branches_alias=None,
...                                     weights=None)
```

The `branches_alias` can be seen as a list of strings that renames the original `branches`. The `weights` has two different implementations: (1) either a 1-D column is provided with shape equals to the data (nevents) or (2) a column of the ROOT file by using a string corresponding to a column. Note that in case of multiple weights are required, the weight manipulation has to be performed by the user beforehand, e.g. using Numpy/pandas or similar.

Note: The implementation of the `from_root` method makes uses of the uproot packages, which uses Numpy to cast bocks of data from the ROOT file as Numpy arrays in time optimised manner. This also means that the *goodies* from uproot can also be used by specifying the root_dir_options, such as cuts in the dataset. However, this can be applied later when examining the produced dataset and it is the advised implementation of this.

2.4.2 Import dataset from a pandas DataFrame or Numpy ndarray

A very simple manipulation of the dataset is provided via the pandas DataFrame. Naturally this is simplified since the *Space* (observable) is not mandatory, and can be obtained directly from the columns:

```
>>> data = zfit.Data.from_pandas(pandas_DataFrame,
...                                      obs=None,
...                                      weights=None)
```

In the case of Numpy, the only difference is that as input is required a numpy ndarray and the *Space* (obs) is mandatory:

```
>>> data = zfit.Data.from_numpy(numpy_ndarray,
...                                     obs,
...                                     weights=None)
```

2.5 Loss

A *loss function* can be defined as a measurement of the discrepancy between the observed data and the predicted data by the fitted function. To some extent it can be visualised as a metric of the goodness of a given prediction as you change the settings of your algorithm. For example, in a general linear model the loss function is essentially the sum of squared deviations from the fitted line or plane. A more useful application in the context of High Energy Physics (HEP) is the Maximum Likelihood Estimator (MLE). The MLE is a specific type of probability model estimation, where the loss function is the negative log-likelihood (NLL).

In zfit, loss functions inherit from the *BaseLoss* class and they follow a common interface, in which the model, the dataset **must** be given, and where parameter constraints in form of a dictionary *{param: constraint}* **may** be given. As an example, we can create an unbinned negative log-likelihood loss (*UnbinnedNLL*) from the model described in the Basic model section and the data from the *Data section*:

```
>>> my_loss = zfit.loss.UnbinnedNLL(model_cb,
>>>                                 data)
```

2.5.1 Adding constraints

Constraints (or, in general, penalty terms) can be added to the loss function either by using the `constraints` keyword when creating the loss object or by using the *`add_constraints()`* method. These constraints are specified as a list of penalty terms, which can be any object inheriting from `BaseConstraint` that is simply added to the calculation of the loss.

Useful implementations of penalties can be found in the *`zfit.constraint`* module. For example, if we wanted to add a gaussian constraint on the `mu` parameter of the previous model, we would write:

```
>>> constraint = zfit.constraint.GaussianConstraint(params=mu, mu=5279., sigma=10.))

>>> my_loss = zfit.loss.UnbinnedNLL(model_cb,
>>>                                 data,
>>>                                 constraints=constraint)
```

Custom penalties can also be added to the loss function, for instance if you want to set limits on a parameter:

```
>>> def custom_constraint(param, max_value):
        return tf.cond(tf.greater_equal(param, max_value), lambda: 10000., lambda: 0.)
```

The custom penalty needs to be callable to be added to the loss function

```
>>> my_loss.add_constraints(lambda: custom_constraint(mu, 5400))
```

or equivalently

```
>>> simple_constraint = zfit.constraint.SimpleConstraint(lambda: custom_constraint(mu,
↪ 5400))
>>> my_loss.add_constraints(simple_constraint)
```

In this example if the value of `param` is larger than `max_value` a large value is added the loss function driving it away from the minimum.

2.5.2 Simultaneous fits

There are currently two loss functions implementations in the `zfit` library, the *`UnbinnedNLL`* and *`ExtendedUnbinnedNLL`* classes, which cover non-extended and extended negative log-likelihoods.

A very common use case of likelihood fits in HEP is the possibility to examine simultaneously different datasets (that can be independent or somehow correlated). To build loss functions for simultaneous fits, the addition operator can be used (the particular combination that is performed depends on the type of loss function):

```
>>> models = [model1, model2]
>>> datasets = [data1, data2]
>>> my_loss1 = zfit.loss.UnbinnedNLL(models[0], datasets[0], fit_range=(-10, 10))
>>> my_loss2 = zfit.loss.UnbinnedNLL(models[1], datasets[1], fit_range=(-10, 10))
>>> my_loss_sim_operator = my_loss1 + my_loss2
```

The same result can be achieved by passing a list of PDFs on instantiation, along with the same number of datasets:

```
>>> # Adding a list of models and datasets
>>> my_loss_sim = zfit.loss.UnbinnedNLL(model=[model1, model2, ...], data=[data1,␣
↪data2, ...])
```

2.6 Minimization

Minimizer objects are the last key element in the API framework of zfit. In particular, these are connected to the loss function and have an internal state that can be queried at any moment.

The zfit library is designed such that it is trivial to introduce new sets of minimizers. The only requirement in its initialisation is that a loss function **must** be given. Additionally, the parameters to be minimize, the tolerance, its name, as well as any other argument needed to configure the particular algorithm **may** be given.

2.6.1 Baseline minimizers

There are three minimizers currently included in the package: `Minuit`, `Scipy` and `Adam` TensorFlow optimiser. Let's show how these can be initialised:

```
>>> # Minuit minimizer
>>> minimizer_minuit = zfit.minimize.Minuit()
>>> # Scipy minimizer
>>> minimizer_scipy = zfit.minimize.Scipy()
>>> # Adam's Tensorflow minimizer
>>> minimizer_adam = zfit.minimize.Adam()
```

A wrapper for TensorFlow optimisers is also available to allow to easily integrate new ideas in the framework. For instance, the Adam minimizer could have been initialised by

```
>>> # Adam's TensorFlor optimiser using a wrapper
>>> minimizer_wrapper = zfit.minimize.WrapOptimizer(tf.keras.optimizer.Adam())
```

Any of these minimizers can then be used to minimize the loss function we created in *previous section*, e.g.

```
>>> result = minimizer_minuit.minimize(loss=my_loss)
```

The choice of which parameters of your model should be floating in the fit can also be made at this stage

```
>>> # In the case of a Gaussian (e.g.)
>>> result = minimizer_minuit.minimize(loss=my_loss, params=[mu, sigma])
```

Only the parameters given in `params` are floated in the optimisation process. If this argument is not provided or `params=None`, all the floating parameters in the loss function are floated in the minimization process.

The result of the fit is return as a *`FitResult`* object, which provides access the minimiser state. zfit separates the minimisation of the loss function with respect to the error calculation in order to give the freedom of calculating this error whenever needed. The *`error()`* method can be used to perform the CPU-intensive error calculation.

```
>>> param_errors = result.error()
>>> for var, errors in param_errors.items():
...   print('{}: ^{{+{}}}_{{-{}}}'.format(var.name, errors['upper'], errors['lower']))
mu: ^{+0.00998104141841555}_{--0.009981515893414316}
sigma: ^{+0.007099472590970696}_{--0.0070162654764939734}
```

The `result` object also provides access the minimiser state:

```
>>> print("Function minimum:", result.fmin)
Function minimum: 14170.396450111948
>>> print("Converged:", result.converged)
Converged: True
>>> print("Full minimizer information:", result.info)
```

(continues on next page)

(continued from previous page)

```
Full minimizer information: {'n_eval': 56, 'original': {'fval': 14170.396450111948,
↪'edm': 2.8519671693442587e-10,
'nfcn': 56, 'up': 0.5, 'is_valid': True, 'has_valid_parameters': True, 'has_accurate_
↪covar': True, 'has_posdef_covar': True,
'has_made_posdef_covar': False, 'hesse_failed': False, 'has_covariance': True, 'is_
↪above_max_edm': False, 'has_reached_call_limit': False}}
```

and the fitted parameters

```
>>> # Information on all the parameters in the fit
>>> params = result.params

>>> # Printing information on specific parameters, e.g. mu
>>> print("mu={}".format(params[mu]['value']))
mu=0.012464509810750313
```

CHAPTER 3

zfit Project

Following and introduction to the elements of zfit

3.1 Installation

3.1.1 Stable release

To install zfit, run this command in your terminal:

```
$ pip install zfit
```

This is the preferred method to install zfit, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

3.1.2 From sources

The sources for zfit can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/zfit/zfit
```

Or download the tarball:

```
$ curl  -OL https://github.com/zfit/zfit/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

3.2 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

- You can report bugs at https://github.com/zfit/zfit/issues.
- You can send feedback by filing an issue at https://github.com/zfit/zfit/issues or,

for more informal discussions, you can also join our Gitter channel.

3.2.1 Get Started!

Ready to contribute? Here's how to set up *zfit* for local development.

1. Fork the *zfit* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/zfit.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv zfit
$ cd zfit/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests:

```
$ py.test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website. The test suite is going to run again, testing all the necessary Python versions.

3.2.2 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the necessary explanations in the corresponding rst file in the docs. If any math is involved, please document the exact formulae implemented in the docstring/docs.
3. The pull request should work for Python 3.6 and 3.7. Check https://travis-ci.org/zfit/zfit/pull_requests and make sure that the tests pass for all supported Python versions.

3.3 Upgrade guide

3.3.1 Upgrade from zfit 0.3.x to 0.4.0

zfit moved from TensorFlow 1.x to 2.x. The main difference is that in 1.x, you would mostly built a graph all the time and execute it when needed. In TF 2.x, this has gone and happens implicitly if a function is decorated with the right decorator. But it is also possible to build no graph at all and execute the code _eagerly_, just as Numpy would. So writing just TF 2.x code is "no different", if not wrapped by a `tf.function()`, than executing Numpy code.

In short: write TF 2.x as if you would write Numpy. If something is supposed to _change_, it has to be newly generated each time, e.g. be a function that can be called.

zfit offers objects that still keep track of everything.

Consequences for zfit:

Dependents

this implies that zfit does not rely on the graph structure anymore. Therefore, dependencies have to be given manually (although in the future, certain automatitions can surely be added).

Affected from this is the *ComposedParameter*. Instead of giving a Tensor, a function returning a value has to be given _and_ the dependents have to be specified explicitly.

```
mu = zfit.Parameter(...)
shift = zfit.Parameter(...)
def shifted_mu_func():
    return mu + shift

shifted_mu = zfit.params.ComposedParameter(shifted_mu_func, dependents=[mu, shift])
```

The same is true for the *SimpleLoss*

3.4 Changelog

3.4.1 Develop

Major Features and Improvements

Behavioral changes

Bug fixes and small changes

Requirement changes

Thanks

3.4.2 0.4.0 (7.1.2020)

This release switched to TensorFlow 2.0 eager mode. In case this breaks things for you and you need **urgently** a running version, install a version < 0.4.1. It is highly recommended to upgrade and make the small changes required.

Please read the *upgrade guide <docs/project/upgrade_guide.rst>* on a more detailed explanation how to upgrade.

TensorFlow 2.0 is eager executing and uses functions to abstract the performance critical parts away.

Major Features and Improvements

- Dependents (currently, and probably also in the future) need more manual tracking. This has mostly an effect on CompositeParameters and SimpleLoss, which now require to specify the dependents by giving the objects it depends (indirectly) on. For example, it is sufficient to give a *ComplexParameter* (which itself is not independent but has dependents) to a *SimpleLoss* as dependents (assuming the loss function depends on it).
- *ComposedParameter* does no longer allow to give a Tensor but requires a function that, when evaluated, returns the value. It depends on the *dependents* that are now required.
- Added numerical differentiation, which allows now to wrap any function with *z.py_function* (*zfit.z*). This can be switched on with *zfit.settings.options['numerical_grad'] = True*
- Added gradient and hessian calculation options to the loss. Support numerical calculation as well.
- Add caching system for graph to prevent recursive graph building
- changed backend name to *z* and can be used as *zfit.z* or imported from it. Added:
  - *function* decorator that can be used to trace a function. Respects dependencies of inputs and automatically caches/invalidates the graph and recreates.
  - *py_function*, same as *tf.py_function*, but checks and may extends in the future
  - *math* module that contains autodiff and numerical differentiation methods, both working with tensors.

Behavioral changes

- EDM goal of the minuit minimizer has been reduced by a factor of 10 to 10E-3 in agreement with the goal in RooFits Minuit minimizer. This can be varied by specifying the tolerance.
- known issue: the *projection_pdf* has troubles with the newest TF version and may not work properly (runs out of memory)

Bug fixes and small changes

Requirement changes

- added numdifftools (for numerical differentiation)

Thanks

3.4.3 0.3.7 (6.12.19)

This is a legacy release to add some fixes, next release is TF 2 eager mode only release.

Major Features and Improvements

- mostly TF 2.0 compatibility in graph mode, tests against 1.x and 2.x

Behavioral changes

Bug fixes and small changes

- *get_dependents* returns now an OrderedSet
- errordef is now a (hidden) attribute and can be changed
- fix bug in polynomials

Requirement changes

- added ordered-set

3.4.4 0.3.6 (12.10.19)

Special release for conda deployment and version fix (TF 2.0 is out)

This is the last release before breaking changes occur

Major Features and Improvements

- added ConstantParameter and *zfit.param* namespace
- Available on conda-forge

Behavioral changes

- an implicitly created parameter with a Python numerical (e.g. when instantiating a model) will be converted to a ConstantParameter instead of a fixed Parameter and therefore cannot be set to floating later on.

Bug fixes and small changes

- added native support TFP distributions for analytic sampling
- fix Gaussian (TFP Distribution) Constraint with mixed up order of parameters
- *from_numpy* automatically converts to default float regardless the original numpy dtype, *dtype* has to be used as an explicit argument

Requirement changes

- TensorFlow >= 1.14 is required

Thanks

- Chris Burr for the conda-forge deployment

3.4.5 0.3.4 (30-07-19)

This is the last release before breaking changes occur

Major Features and Improvements

- create *Constraint* class which allows for more fine grained control and information on the applied constraints.
- Added Polynomial models
- Improved and fixed sampling (can still be slightly biased)

Behavioral changes

None

Bug fixes and small changes

- fixed various small bugs

Thanks

for the contribution of the Constraints to Matthieu Marinangeli <matthieu.marinangeli@cern.ch>

3.4.6 0.3.3 (15-05-19)

Fixed Partial numeric integration

Bugfixes mostly, a few major fixes. Partial numeric integration works now.

Bugfixes

- data_range cuts are now applied correctly, also in several dimensions when a subset is selected (which happens internally of some Functors, e.g. ProductPDF). Before, only the selected obs was respected for cuts.
- parital integration had a wrong take on checking limits (now uses supports).

3.4.7 0.3.2 (01-05-19)

With 0.3.2, bugfixes and three changes in the API/behavior

Breaking changes

- tfp distributions wrapping is now different with dist_kwargs allowing for non-Parameter arguments (like other dists)
- sampling allows now for importance sampling (sampler in Model specified differently)
- *model.sample* now also returns a tensor, being consistent with *pdf* and *integrate*

Bugfixes

- shape handling of tfp dists was "wrong" (though not producing wrong results!), fixed. TFP distributions now get a tensor with shape (nevents, nobs) instead of a list of tensors with (nevents,)

Improvements

- refactor the sampling for more flexibility and performance (less graph constructed)
- allow to use more sophisticated importance sampling (e.g. phasespace)
- on-the-fly normalization (experimentally) implemented with correct gradient

3.4.8 0.3.1 (30-04-19)

Minor improvements and bugfixes including:

- improved importance sampling allowing to preinstantiate objects before it's called inside the while loop
- fixing a problem with *ztf.sqrt*

3.4.9 0.3.0 (2019-03-20)

Beta stage and first pip release

3.4.10 0.0.1 (2018-03-22)

- First creation of the package.

3.5 Development Lead

- zfit <zfit@physik.uzh.ch>

3.6 Authors

Jonas Eschle <Jonas.Eschle@cern.ch>
Albert Puig <apuignav@gmail.com>
Rafael Silva Coutinho <rsilvaco@cern.ch>
Matthieu Marinangeli <matthieu.marinangeli@cern.ch>

3.7 Contributors

Chris Burr <christopher.burr@cern.ch>
Abhijit Mathad <amathad@cern.ch>
Oliver Lantwin <oliver.lantwin@cern.ch>

CHAPTER 4

zfit API documentation

The API documentation of zfit can be found below. Most classes and functions are documented with docstrings, but don't hesitate to contact us if this documentation is insufficient!

4.1 zfit package

Top-level package for zfit.

class zfit.**Parameter**(*name*, *value*, *lower_limit=None*, *upper_limit=None*, *step_size=None*, *floating=True*, *dtype=tf.float64*, ***kwargs*)

Bases: *zfit.core.parameter.ZfitParameterMixin*, *zfit.core.parameter.TFBaseVariable*, *zfit.core.parameter.BaseParameter*

Class for fit parameters, derived from TF Variable class.

name : name of the parameter, value : starting value lower_limit : lower limit upper_limit : upper limit step_size : step size (set to 0 for fixed parameters)

class SaveSliceInfo(*full_name=None*, *full_shape=None*, *var_offset=None*, *var_shape=None*, *save_slice_info_def=None*, *import_scope=None*)

Bases: object

Information on how to save this Variable as a slice.

Provides internal support for saving variables as slices of a larger variable. This API is not public and is subject to change.

Available properties:

- full_name
- full_shape
- var_offset
- var_shape

Create a *SaveSliceInfo*.

Parameters

- **full_name** – Name of the full variable of which this *Variable* is a slice.
- **full_shape** – Shape of the full variable, as a list of int.
- **var_offset** – Offset of this *Variable* into the full variable, as a list of int.
- **var_shape** – Shape of this *Variable*, as a list of int.
- **save_slice_info_def** – *SaveSliceInfoDef* protocol buffer. If not *None*, recreates the SaveSliceInfo object its contents. *save_slice_info_def* and other arguments are mutually exclusive.
- **import_scope** – Optional *string*. Name scope to add. Only used when initializing from protocol buffer.

spec
Computes the spec string used for saving.

to_proto(*export_scope=None*)
Returns a SaveSliceInfoDef() proto.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Returns A *SaveSliceInfoDef* protocol buffer, or None if the *Variable* is not in the specified name scope.

__iter__()
Dummy method to prevent iteration.

Do not call.

NOTE(mrry): If we register __getitem__ as an overloaded operator, Python will valiantly attempt to iterate over the variable's Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises TypeError – when invoked.

__ne__(*other*)
Compares two variables element-wise for equality.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

aggregation

assign(*value*, *use_locking=None*, *name=None*, *read_value=True*)
Assigns a new value to this variable.

Parameters

- **value** – A *Tensor*. The new value for this variable.
- **use_locking** – If *True*, use locking during the assignment.
- **name** – The name to use for the assignment.

- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_add(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Adds a value to this variable.

Parameters

- **delta** – A *Tensor*. The value to add to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_sub(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Subtracts a value from this variable.

Parameters

- **delta** – A *Tensor*. The value to subtract from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

batch_scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable batch-wise.

Analogous to *batch_gather*. This assumes that this variable and the sparse_delta IndexedSlices have a series of leading dimensions that are the same for all of them, and the updates are performed on the last dimension of indices. In other words, the dimensions should be the following:

num_prefix_dims = sparse_delta.indices.ndims - 1 batch_dim = num_prefix_dims + 1 ‘sparse_delta.updates.shape = sparse_delta.indices.shape + var.shape[

batch_dim:]‘

where

sparse_delta.updates.shape[:num_prefix_dims] == sparse_delta.indices.shape[:num_prefix_dims] == var.shape[:num_prefix_dims]

And the operation performed can be expressed as:

‘var[i_1, . . . , i_n,

sparse_delta.indices[i_1, . . . , i_n, j]] = sparse_delta.updates[i_1, . . . , i_n, j]‘

When sparse_delta.indices is a 1D tensor, this operation is equivalent to *scatter_update*.

To avoid this operation one can looping over the first *ndims* of the variable and using *scatter_update* on the subtensors that result of slicing the first dimension. This is a valid option for *ndims = 1*, but less efficient than this implementation.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

constraint
Returns the constraint function associated with this variable.

Returns The constraint function that was passed to the variable constructor. Can be *None* if no constraint was passed.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

count_up_to(*limit*)
Increments this variable until it reaches *limit*. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Dataset.range instead.

When that Op is run it tries to increment the variable by *1*. If incrementing the variable would bring it above *limit* then the Op raises the exception *OutOfRangeError*.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for *count_up_to(self, limit)*.

Parameters **limit** – value at which incrementing the variable raises an error.

Returns A *Tensor* that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

create
The op responsible for initializing this variable.

device
The device this variable is on.

dtype
The dtype of the object

eval(*session=None*)
Evaluates and returns the value of this variable.

experimental_ref()
Returns a hashable reference object to this Variable.

Warning: Experimental API that could be changed or removed.

The primary usecase for this API is to put variables in a set/dictionary. We can't put variables in a set/dictionary as *variable.__hash__()* is no longer available starting Tensorflow 2.0.

‘‘‘python import tensorflow as tf

x = tf.Variable(5) y = tf.Variable(10) z = tf.Variable(10)

The followings will raise an exception starting 2.0 # TypeError: Variable is unhashable if Variable equality is enabled. variable_set = {x, y, z} variable_dict = {x: 'five', y: 'ten'} ‘‘‘

Instead, we can use *variable.experimental_ref()*.

‘‘‘python variable_set = {x.experimental_ref(),

y.experimental_ref(), z.experimental_ref()}

print(x.experimental_ref() in variable_set) ==> True

variable_dict = {x.experimental_ref(): 'five', y.experimental_ref(): 'ten', z.experimental_ref(): 'ten'}

print(variable_dict[y.experimental_ref()]) ==> ten ‘‘‘

Also, the reference object provides *.deref()* function that returns the original Variable.

```
`python x = tf.Variable(5) print(x.experimental_ref().deref()) ==>
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=5> `
```

floating

static from_proto(*variable_def*, *import_scope=None*)
Returns a *Variable* object created from *variable_def*.

gather_nd(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather_nd*.

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_shape()
Alias of *Variable.shape*.

graph
The *Graph* of this variable.

graph_caching_methods = []

handle
The handle by which this variable can be accessed.

has_limits

independent

initial_value
Returns the Tensor used as the initial value for the variable.

initialized_value()
Returns the value of the initialized variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun) contexts.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
`python # Initialize 'v' with a random tensor. v = tf.Variable(tf.
random.truncated_normal([10, 40])) # Use `initialized_value` to
guarantee that `v` has been # initialized before its value is used
to initialize `w`. # The random values are picked only once. w = tf.
Variable(v.initialized_value() * 2.0) `
```

Returns A *Tensor* holding the value of this variable after its initializer has run.

initializer
The op responsible for initializing this variable.

is_initialized(*name=None*)
Checks whether a resource variable has been initialized.

Outputs boolean scalar indicating whether the tensor has been initialized.

Parameters **name** – A name for the operation (optional).

Returns A *Tensor* of type *bool*.

load(*value*, *session=None*)
Load new value into this variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Variable.assign which has equivalent behavior in 2.X.

Writes new value to variable's memory. Doesn't add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See *tf.compat.v1.Session* for more information on launching a graph and on sessions.

‘‘‘python v = tf.Variable([1, 2]) init = tf.compat.v1.global_variables_initializer()

with tf.compat.v1.Session() as sess: sess.run(init) # Usage passing the session explicitly. v.load([2, 3], sess) print(v.eval(sess)) # prints [2 3] # Usage with the default session. The 'with' block # above makes 'sess' the default session. v.load([3, 4], sess) print(v.eval()) # prints [3 4]

‘‘‘

Parameters

- **value** – New variable value
- **session** – The session to use to evaluate this variable. If none, the default session is used.

Raises `ValueError` – Session is not passed and no default session

lower_limit

name
The name of the object.

numpy()

old_graph_caching_methods = []

op
The op for this variable.

params

randomize(*minval=None*, *maxval=None*, *sampler=<built-in method uniform of numpy.random.mtrand.RandomState object>*)
Update the value with a randomised value between minval and maxval.

Parameters

- **minval** (*Numerical*) –
- **maxval** (*Numerical*) –
- **()** (*sampler*) –

read_value()
Constructs an op which reads the value of this variable.

Should be used when there are multiple reads, or when it is desirable to read the value only after some condition is true.

Returns the read operation.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

scatter_add(*sparse_delta*, *use_locking=False*, *name=None*)
Adds *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be added to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered addition has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_div(*sparse_delta*, *use_locking=False*, *name=None*)
Divide this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to divide this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered division has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_max(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the max of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of max with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered maximization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_min(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the min of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of min with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered minimization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_mul(*sparse_delta*, *use_locking=False*, *name=None*)
Multiply this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to multiply this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered multiplication has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_nd_add(*indices*, *updates*, *name=None*)
Applies sparse addition to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) add = ref.scatter_nd_add(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(add)

‘‘‘

The resulting update to ref would look like this:

[1, 13, 3, 14, 14, 6, 7, 20]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_sub(*indices*, *updates*, *name=None*)

Applies sparse subtraction to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K‘th dimension of ‘ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_sub(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, -9, 3, -6, -6, 6, 7, -4]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_update(*indices*, *updates*, *name=None*)
Applies sparse assignment to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_update(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, 11, 3, 10, 9, 6, 7, 12]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_sub(*sparse_delta*, *use_locking=False*, *name=None*)
Subtracts *tf.IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be subtracted from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.

- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

set_shape(*shape*)
Unsupported.

set_value(*value: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)
Set the `Parameter` to *value* (temporarily if used in a context manager).

Parameters **value** (*float*) – The value the parameter will take on.

shape
The shape of this variable.

sparse_read(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather*.

step_size

synchronization

to_proto(*export_scope=None*)
Converts a *ResourceVariable* to a *VariableDef* protocol buffer.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Raises `RuntimeError` – If run in EAGER mode.

Returns A *VariableDef* protocol buffer, or *None* if the *Variable* is not in the specified name scope.

trainable

upper_limit

value()
A cached operation which reads the value of this variable.

class zfit.**ComposedParameter**(*name*, *value_fn*, *dependents*, *dtype=tf.float64*, ***kwargs*)
Bases: *zfit.core.parameter.BaseComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.**ComplexParameter**(*name*, *value_fn*, *dependents*, *dtype=tf.complex128*, ***kwargs*)
Bases: *zfit.core.parameter.ComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

arg

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

conj

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

static from_cartesian(*name*, *real*, *imag*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

static from_polar(*name*, *mod*, *arg*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

imag

independent

mod

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

real

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

zfit.**convert_to_parameter**(*value*, *name=None*, *prefer_floating=False*, *dependents=None*, *graph_mode=False*) → zfit.core.interfaces.ZfitParameter

Convert a *numerical* to a fixed/floating parameter or return if already a parameter.

Parameters

- **()** (*name*) –
- **()** –
- **prefer_floating** – If True, create a Parameter instead of a FixedParameter _if possible_.

class zfit.**Space**(*obs: Union[str, Iterable[str], zfit.Space], limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, name: Optional[str] = 'Space'*)

Bases: *zfit.core.interfaces.ZfitSpace*, *zfit.core.baseobject.BaseObject*

Define a space with the name (*obs*) of the axes (and it's number) and possibly it's limits.

Parameters

- **obs** (*str, List[str,..]*) –
- **()** (*limits*) –
- **name** (*str*) –

ANY = <Any>

ANY_LOWER = <Any Lower Limit>

ANY_UPPER = <Any Upper Limit>

AUTO_FILL = <object object>

add(*other: Union[zfit.Space, Iterable[zfit.Space]]*)

Add the limits of the spaces. Only works for the same obs.

In case the observables are different, the order of the first space is taken.

Parameters **other** (*Space*) –

Returns

Return type *Space*

area() → float

Return the total area of all the limits and axes. Useful, for example, for MC integration.

axes

The axes ("obs with int") the space is defined in.

Returns:

combine(*other: Union[zfit.Space, Iterable[zfit.Space]]*) → zfit.core.interfaces.ZfitSpace

Combine spaces with different obs (but consistent limits).

Parameters **other** (*Space*) –

Returns

Return type *Space*

copy(*name: Optional[str] = None*, ***overwrite_kwargs*) → zfit.Space

Create a new *Space* using the current attributes and overwriting with *overwrite_overwrite_kwargs*.

Parameters

- **name** (*str*) – The new name. If not given, the new instance will be named the same as the current one.
- **()** (***overwrite_kwargs*) –

Returns *Space*

classmethod from_axes(*axes: Union[int, Iterable[int]], limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, name: str = None*) → zfit.Space

Create a space from *axes* instead of from *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **name** (*str*) –

Returns *Space*

get_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, as_dict: bool = False, autofill: bool = False*) → Union[Tuple[int], None, Dict[str, int]]

Return the axes corresponding to the *obs* (or all if None).

Parameters

- **()** (*obs*) –
- **as_dict** (*bool*) – If True, returns a ordered dictionary with {obs: axis}
- **autofill** (*bool*) – If True and the axes are not specified, automatically fill them with the default numbering and return (not setting them).

Returns Tuple, OrderedDict

Raises

- `ValueError` – if the requested *obs* do not match with the one defined in the range
- `AxesNotSpecifiedError` – If the axes in this *Space* have not been specified.

get_obs_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*)

get_reorder_indices(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*) → Tuple[int]

Indices that would order *self.obs* as *obs* respectively *self.axes* as *axes*.

Parameters

- **()** (*axes*) –
- **()** –

Returns:

get_subspace(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, name: Optional[str] = None*) → zfit.Space

Create a *Space* consisting of only a subset of the *obs*/*axes* (only one allowed).

Parameters

- **obs** (*str, Tuple[str]*) –
- **axes** (*int, Tuple[int]*) –
- **()** (*name*) –

Returns:

iter_areas(*rel: bool = False*) → Tuple[float, ...]
Return the areas of each interval

Parameters **rel** (*bool*) – If True, return the relative fraction of each interval

Returns

Return type Tuple[float]

iter_limits(*as_tuple: bool = True*) → Union[Tuple[zfit.Space], Tuple[Tuple[Tuple[float]]], Tuple[Tuple[Tuple[float]]]]
Return the limits, either as *Space* objects or as pure limits-tuple.

This makes iterating over limits easier: *for limit in space.iter_limits()* allows to, for example, pass *limit* to a function that can deal with simple limits only or if *as_tuple* is True the *limit* can be directly used to calculate something.

Example

```
for lower, upper in space.iter_limits(as_tuple=True):
    integrals = integrate(lower, upper)  # calculate integral
integral = sum(integrals)
```

Returns

Return type List[*Space*] or List[limit,...]

limit1d
return the tuple(lower, upper).

Returns so `lower, upper = space.limit1d` for a simple, 1 obs limit.

Return type tuple(float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified limits getter for 1 obs, 1 limit only

limit2d
return the tuple(low_obs1, low_obs2, up_obs1, up_obs2).

Returns

so *low_x, low_y, up_x, up_y = space.limit2d* for a single, 2 obs limit. low_x is the lower limit in x, up_x is the upper limit in x etc.

Return type tuple(float, float, float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *limits* for exactly 2 obs, 1 limit

limits
Return the limits.

Returns:

limits1d
return the tuple(low_1, ..., low_n, up_1, ..., up_n).

Returns

so *low_1, low_2, up_1, up_2 = space.limits1d* for several, 1 obs limits. low_1 to up_1 is the first interval, low_2 to up_2 is the second interval etc.

Return type tuple(float, float, ..)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *.limits* for exactly 1 obs, n limits

lower
Return the lower limits.

Returns:

n_limits
The number of different limits.

Returns int >= 1

n_obs
Return the number of observables/axes.

Returns int >= 1

name
The name of the object.

obs
The observables ("axes with str")the space is defined in.

Returns:

obs_axes

reorder_by_indices (*indices: Tuple[int]*)
Return a *Space* reordered by the indices.

Parameters **()** (*indices*) –

upper
Return the upper limits.

Returns:

with_autofill_axes (*overwrite: bool = False*) → zfit.Space
Return a *Space* with filled axes corresponding to range(len(n_obs)).

Parameters **overwrite** (*bool*) – If *self.axes* is not None, replace the axes with the autofilled ones. If axes is already set, don't do anything if *overwrite* is False.

Returns *Space*

with_axes (*axes: Union[int, Iterable[int]]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*axes*) –

Returns *Space*

with_limits (*limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool], name: Optional[str] = None*) → zfit.Space
Return a copy of the space with the new *limits* (and the new *name*).

Parameters

- **()** (*limits*) –
- **name** (*str*) –

Returns *Space*

with_obs(*obs: Union[str, Iterable[str], zfit.Space]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*obs*) –

Returns *Space*

with_obs_axes(*obs_axes: Union[OrderedDict[str, int], Dict[str, int]], ordered: bool = False, allow_subset=False*) → zfit.Space
Return a new *Space* with reordered observables and set the *axes*.

Parameters

- **obs_axes** (*OrderedDict[str, int]*) – An ordered dict with {obs: axes}.
- **ordered** (*bool*) – If True (and the *obs_axes* is an *OrderedDict*), the
- **()** (*allow_subset*) –

Returns

Return type *Space*

zfit.**convert_to_space**(*obs: Union[str, Iterable[str], zfit.Space, None] = None, axes: Union[int, Iterable[int], None] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, *, overwrite_limits: bool = False, one_dim_limits_only: bool = True, simple_limits_only: bool = True*) → Union[None, zfit.core.limits.Space, bool]
Convert *limits* to a *Space* object if not already None or False.

Parameters

- **obs** (Union[Tuple[float, float], *Space*]) –
- **()** (*axes*) –
- **()** –
- **overwrite_limits** (*bool*) – If *obs* or *axes* is a *Space* _and_ *limits* are given, return an instance of *Space* with the new limits. If the flag is *False*, the *limits* argument will be ignored if
- **one_dim_limits_only** (*bool*) –
- **simple_limits_only** (*bool*) –

Returns

Return type Union[*Space*, False, None]

Raises `OverdefinedError` – if *obs* or *axes* is a *Space* and *axes* respectively *obs* is not *None*.

zfit.**supports**(*, *norm_range: bool = False*, *multiple_limits: bool = False*) → Callable
Decorator: Add (mandatory for some methods) on a method to control what it can handle.

If any of the flags is set to False, it will check the arguments and, in case they match a flag (say if a *norm_range* is passed while the *norm_range* flag is set to *False*), it will raise a corresponding exception (in this example a *NormRangeNotImplementedError*) that will be catched by an earlier function that knows how to handle things.

Parameters

- **norm_range** (*bool*) – If False, no norm_range argument will be passed through resp. will be *None*

- **multiple_limits** (*bool*) – If False, only simple limits are to be expected and no iteration is therefore required.

4.1.1 Subpackages

core

Submodules

basefunc

Baseclass for *Function*. Inherits from Model.

TODO(Mayou36): subclassing?

class zfit.core.basefunc.**BaseFunc**(*obs=None*, *dtype: Type[CT_co] = tf.float64*, *name: str = 'BaseFunc'*, *params: Any = None*)

Bases: *zfit.core.basemodel.BaseModel*, *zfit.core.interfaces.ZfitFunc*

TODO(docs): explain subclassing

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]*, *norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None*, *name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF
Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]
The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.

- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space

Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

basemodel

Baseclass for a Model. Handle integration and sampling

class zfit.core.basemodel.**BaseModel**(*obs: Union[str, Iterable[str], zfit.Space], params: Optional[Dict[str, zfit.core.interfaces.ZfitParameter]] = None, name: str = 'BaseModel', dtype=tf.float64, **kwargs*)
Bases: *zfit.core.baseobject.BaseNumeric*, *zfit.util.cache.Cachable*, *zfit.core.dimension.BaseDimensional*, *zfit.core.interfaces.ZfitModel*

Base class for any generic model.

TODO instructions on how to use

The base model to inherit from and overwrite *_unnormalized_pdf*.

Parameters

- **dtype** (*DType*) – the dtype of the model
- **name** (*str*) – the name of the model
- **params** (Dict(str, *Parameter*)) – A dictionary with the internal name of the parameter and the parameters itself the model depends on

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.

- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, `Space`) – the limits to integrate over
- **norm_range** (tuple, `Space`, False) – the limits to normalize over
- **name** (`str`) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.

- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.core.basemodel.**SimpleModelSubclassMixin**(**args*, ***kwargs*)
Bases: object

Subclass a model: implement the corresponding function and specify _PARAMS.

In order to create a custom model, two things have to be implemented: the class attribute _PARAMS has to be a list containing the names of the parameters and the corresponding function (_unnormalized_pdf/_func) has to be overridden.

Example:

```
class MyPDF(zfit.pdf.ZPDF):
    _PARAMS = ['mu', 'sigma']

    def _unnormalized_pdf(self, x):
        mu = self.params['mu']
        sigma = self.params['sigma']
        x = z.unstack_x(x)
        return z.exp(-z.square((x - mu) / sigma))
```

baseobject

Baseclass for most objects appearing in zfit.

class zfit.core.baseobject.**BaseNumeric**(*name*, *params*, ***kwargs*)
Bases: *zfit.util.cache.Cachable*, *zfit.core.dependents.BaseDependentsMixin*, *zfit.core.interfaces.ZfitNumeric*, *zfit.core.baseobject.BaseObject*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False*, *names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.

- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name
The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

class zfit.core.baseobject.**BaseObject**(*name*, ***kwargs*)
Bases: *zfit.core.interfaces.ZfitObject*

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

name
The name of the object.

basepdf

This module defines the *BasePdf* that can be used to inherit from in order to build a custom PDF.

The *BasePDF* implements already a lot of ready-to-use functionality like integral, automatic normalization and sampling.

Defining your own pdf

A simple example: >>> class MyGauss(BasePDF): >>> def __init__(self, mean, stddev, name="MyGauss"): >>> super().__init__(mean=mean, stddev=stddev, name=name) >>> >>> def _unnormalized_pdf(self, x): >>> return tf.exp((x - mean) ** 2 / (2 * stddev**2))

Notice that *here* we only specify the *function* and no normalization. This **No** attempt to **explicitly** normalize the function should be done inside *_unnormalized_pdf*. The normalization is handled with another method depending on the normalization range specified. (It *is* possible, though discouraged, to directly provide the *normalized probability* by overriding _pdf(), but there are other, more convenient ways to add improvements like providing an analytical integrals.)

Before we create an instance, we need to create the variables to initialize it >>> mean = zfit.Parameter("mean1", 2., 0.1, 4.2) # signature as in RooFit: *name, initial, lower, upper* >>> stddev = zfit.Parameter("stddev1", 5., 0.3, 10.) Let's create an instance and some example data >>> gauss = MyGauss(mean=mean, stddev=stddev) >>> example_data = np.random.random(10) Now we can get the probability >>> probs = gauss.pdf(x=example_data, norm_range=(-30., 30)) # *norm_range* specifies over which range to normalize Or the integral >>> integral = gauss.integrate(limits=(-5, 3.1), norm_range=False) # norm_range is False -> return unnormalized integral Or directly sample from it >>> sample = gauss.sample(n_draws=1000, limits=(-10, 10)) # draw 1000 samples within (-10, 10)

We can create an extended PDF, which will result in anything using a *norm_range* to not return the probability but the number probability (the function will be normalized to *yield* instead of 1 inside the *norm_range*) >>> yield1 = Parameter("yield1", 100, 0, 1000) >>> gauss_extended = gauss.create_extended(yield1) >>> gauss.is_extended True

```
>>> integral_extended = gauss.integrate(limits=(-10, 10), norm_range=(-10, 10))  #␣
↪yields approx 100
```

For more advanced methods and ways to register analytic integrals or overwrite certain methods, see also the advanced tutorials in zfit tutorials

class zfit.core.basepdf.**BasePDF**(*obs: Union[str, Iterable[str], zfit.Space], params: Dict[str, zfit.core.interfaces.ZfitParameter] = None, dtype: Type[CT_co] = tf.float64, name: str = 'BasePDF', **kwargs*)

Bases: *zfit.core.interfaces.ZfitPDF*, *zfit.core.basemodel.BaseModel*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –

- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func (*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space (*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]

Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

constraint

class zfit.core.constraint.**BaseConstraint** (*params: Dict[str, zfit.core.interfaces.ZfitParameter] = None, name: str = 'BaseConstraint', dtype=tf.float64, **kwargs*)
Bases: *zfit.core.interfaces.ZfitConstraint*, *zfit.core.baseobject.BaseNumeric*

Base class for constraints.

Parameters

- **dtype** (*DType*) – the dtype of the constraint
- **name** (*str*) – the name of the constraint
- **params** (Dict(str, *Parameter*)) – A dictionary with the internal name of the parameter and the parameters itself the constrains depends on

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name

The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n*)

Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

class zfit.core.constraint.**DistributionConstraint**(*params: Dict[str, zfit.core.interfaces.ZfitParameter], distribution: tensorflow_probability.python.distributions.distribution.Distributio dist_params, dist_kwargs=None, name: str = 'DistributionConstraint', dtype=tf.float64, **kwargs*)

Bases: *zfit.core.constraint.BaseConstraint*

Base class for constraints using a probability density function.

Parameters **distribution** (*tensorflow_probability.distributions.Distribution*) – The probability density function used to constraint the parameters

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name

The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n*)

Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

class zfit.core.constraint.**GaussianConstraint**(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], mu: Union[int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)

Bases: *zfit.core.constraint.DistributionConstraint*

Gaussian constraints on a list of parameters.

Parameters

- **params** (*list(zfit.Parameter)*) – The parameters to constraint
- **mu** (*numerical, list(numerical)*) – The central value of the constraint
- **sigma** (*numerical, list(numerical) or array/tensor*) – The standard deviations or covariance matrix of the constraint. Can either be a single value, a list of values, an array or a tensor

Raises ShapeIncompatibleError – if params, mu and sigma don't have incompatible shapes

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

distribution

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name
The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n*)
Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

class zfit.core.constraint.**SimpleConstraint**(*func: Callable, params: Optional[Dict[str, zfit.core.interfaces.ZfitParameter]], sampler: Callable = None*)
Bases: *zfit.core.constraint.BaseConstraint*

Constraint from a (function returning a) Tensor.

The parameters are named "param_{i}" with i starting from 0 and corresponding to the index of params.

Parameters

- **func** – Callable that constructs the constraint and returns a tensor.
- **dependents** – The dependents (independent *zfit.Parameter*) of the loss. If not given, the dependents are figured out automatically.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name

The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n*)
Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

data

class zfit.core.data.**Data**(*dataset: Union[tensorflow.python.data.ops.dataset_ops.DatasetV2, LightDataset], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = None, weights=None, iterator_feed_dict: Dict[KT, VT] = None, dtype: tensorflow.python.framework.dtypes.DType = None*)
Bases: *zfit.util.cache.Cachable*, *zfit.core.interfaces.ZfitData*, *zfit.core.dimension.BaseDimensional*, *zfit.core.baseobject.BaseObject*

Create a data holder from a *dataset* used to feed into *models*.

Parameters

- **()** (*dtype*) – A dataset storing the actual values
- **()** – Observables where the data is defined in
- **()** – Name of the *Data*
- **()** –
- **()** –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data_range

dtype

classmethod from_numpy(*obs: Union[str, Iterable[str], zfit.Space], array: numpy.ndarray, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create *Data* from a *np.array*.

Parameters

- **()** (*obs*) –
- **array** (*numpy.ndarray*) –
- **name** (*str*) –

Returns

Return type zfit.Data

classmethod from_pandas(*df: pandas.core.frame.DataFrame, obs: Union[str, Iterable[str], zfit.Space] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create a *Data* from a pandas DataFrame. If *obs* is *None*, columns are used as obs.

Parameters

- **df** (*pandas.DataFrame*) –
- **weights** (*tf.Tensor, None, np.ndarray, str]*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents).
- **obs** (*zfit.Space*) –
- **name** (*str*) –

classmethod from_root(*path: str*, *treepath: str*, *branches: List[str] = None*, *branches_alias: Dict[KT, VT] = None*, *weights: Union[tensorflow.python.framework.ops.Tensor*, *None*, *numpy.ndarray*, *str] = None*, *name: str = None*, *dtype: tensorflow.python.framework.dtypes.DType = None*, *root_dir_options=None*) → zfit.core.data.Data

Create a *Data* from a ROOT file. Arguments are passed to *uproot*.

Parameters

- **path** (*str*) –
- **treepath** (*str*) –
- **branches** (*List[str]]*) –
- **branches_alias** (*dict*) – A mapping from the *branches* (as keys) to the actual *observables* (as values). This allows to have different *observable* names, independent of the branch name in the file.

- **weights** (*tf.Tensor, None, np.ndarray, str*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents). Can be a column of the ROOT file by using a string corresponding to a column.
- **name** (*str*) –
- **()** (*root_dir_options*) –

Returns

Return type *zfit.Data*

classmethod from_root_iter(*path*, *treepath*, *branches=None*, *entrysteps=None*, *name=None*, ***kwargs*)

classmethod from_tensor(*obs: Union[str, Iterable[str], zfit.Space], tensor: tensorflow.python.framework.ops.Tensor, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*) → zfit.core.data.Data

Create a *Data* from a *tf.Tensor*. *Value* simply returns the tensor (in the right order).

Parameters

- **obs** (*Union[str, List[str]*) –
- **tensor** (*tf.Tensor*) –
- **name** (*str*) –

Returns

Return type zfit.core.Data

get_iteration()

graph_caching_methods = []

initialize()

iterator

n_obs

Return the number of observables.

name

The name of the object.

nevents

numpy()

obs

Return the observables.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

set_data_range(*data_range*)

set_weights(*weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray]*)
Set (temporarily) the weights of the dataset.

Parameters **weights** (*tf.Tensor*, np.ndarray, None) –

sort_by_axes(*axes: Union[int, Iterable[int]], allow_superset: bool = False*)

sort_by_obs(*obs: Union[str, Iterable[str], zfit.Space], allow_superset: bool = False*)

space
Return the *Space* object that defines the dimensionality of the object.

to_pandas(*obs: Union[str, Iterable[str], zfit.Space] = None*)
Create a *pd.DataFrame* from *obs* as columns and return it.

Parameters **()** (*obs*) – The observables to use as columns. If *None*, all observables are used.

Returns:

unstack_x(*obs: Union[str, Iterable[str], zfit.Space] = None, always_list: bool = False*)
Return the unstacked data: a list of tensors or a single Tensor.

Parameters

- **()** (*obs*) – which observables to return
- **always_list** (*bool*) – If True, always return a list (also if length 1)

Returns List(tf.Tensor)

value(*obs: Union[str, Iterable[str], zfit.Space] = None*)

weights

class zfit.core.data.**LightDataset**(*tensor*)
Bases: object

classmethod from_tensor(*tensor*)

value()

class zfit.core.data.**SampleData**(*dataset: Union[tensorflow.python.data.ops.dataset_ops.DatasetV2, LightDataset], sample_holder: tensorflow.python.framework.ops.Tensor, obs: Union[str, Iterable[str], zfit.Space] = None, weights=None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = tf.float64*)
Bases: *zfit.core.data.Data*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data_range

dtype

classmethod from_numpy(*obs: Union[str, Iterable[str], zfit.Space], array: numpy.ndarray, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)
Create *Data* from a *np.array*.

Parameters

- **()** (*obs*) –
- **array** (*numpy.ndarray*) –
- **name** (*str*) –

Returns

Return type zfit.Data

classmethod from_pandas(*df: pandas.core.frame.DataFrame, obs: Union[str, Iterable[str], zfit.Space] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)
Create a *Data* from a pandas DataFrame. If *obs* is *None*, columns are used as obs.

Parameters

- **df** (*pandas.DataFrame*) –
- **weights** (*tf.Tensor, None, np.ndarray, str*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents).
- **obs** (*zfit.Space*) –
- **name** (*str*) –

classmethod from_root(*path: str, treepath: str, branches: List[str] = None, branches_alias: Dict[KT, VT] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray, str] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None, root_dir_options=None*) → zfit.core.data.Data

Create a *Data* from a ROOT file. Arguments are passed to *uproot*.

Parameters

- **path** (*str*) –
- **treepath** (*str*) –
- **branches** (*List[str]*) –
- **branches_alias** (*dict*) – A mapping from the *branches* (as keys) to the actual *observables* (as values). This allows to have different *observable* names, independent of the branch name in the file.
- **weights** (*tf.Tensor, None, np.ndarray, str]*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents). Can be a column of the ROOT file by using a string corresponding to a column.
- **name** (*str*) –
- **()** (*root_dir_options*) –

Returns

Return type *zfit.Data*

classmethod from_root_iter(*path*, *treepath*, *branches=None*, *entrysteps=None*, *name=None*, ***kwargs*)

classmethod from_sample(*sample: tensorflow.python.framework.ops.Tensor, obs: Union[str, Iterable[str], zfit.Space], name: str = None, weights=None*)

classmethod from_tensor(*obs: Union[str, Iterable[str], zfit.Space], tensor: tensorflow.python.framework.ops.Tensor, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*) → zfit.core.data.Data

Create a *Data* from a *tf.Tensor*. *Value* simply returns the tensor (in the right order).

Parameters

- **obs** (*Union[str, List[str]*) –
- **tensor** (*tf.Tensor*) –
- **name** (*str*) –

Returns

Return type zfit.core.Data

classmethod get_cache_counting()

get_iteration()

graph_caching_methods = []

initialize()

iterator

n_obs

Return the number of observables.

name

The name of the object.

nevents

numpy()

obs
Return the observables.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

set_data_range(*data_range*)

set_weights(*weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray]*)
Set (temporarily) the weights of the dataset.

Parameters **weights** (*tf.Tensor*, np.ndarray, None) –

sort_by_axes(*axes: Union[int, Iterable[int]], allow_superset: bool = False*)

sort_by_obs(*obs: Union[str, Iterable[str], zfit.Space], allow_superset: bool = False*)

space
Return the *Space* object that defines the dimensionality of the object.

to_pandas(*obs: Union[str, Iterable[str], zfit.Space] = None*)
Create a *pd.DataFrame* from *obs* as columns and return it.

Parameters **()** (*obs*) – The observables to use as columns. If *None*, all observables are used.

Returns:

unstack_x(*obs: Union[str, Iterable[str], zfit.Space] = None, always_list: bool = False*)
Return the unstacked data: a list of tensors or a single Tensor.

Parameters

- **()** (*obs*) – which observables to return
- **always_list** (*bool*) – If True, always return a list (also if length 1)

Returns List(tf.Tensor)

value(*obs: Union[str, Iterable[str], zfit.Space] = None*)

weights

class zfit.core.data.**Sampler**(*dataset: zfit.core.data.LightDataset, sample_func: Callable, sample_holder: tensorflow.python.ops.variables.Variable, n: Union[int, float, complex, tensorflow.python.framework.ops.Tensor, Callable], weights=None, fixed_params: Dict[zfit.Parameter, Union[int, float, complex, tensorflow.python.framework.ops.Tensor]] = None, obs: Union[str, Iterable[str], zfit.Space] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = tf.float64*)
Bases: *zfit.core.data.Data*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

data_range

dtype

classmethod from_numpy(*obs: Union[str, Iterable[str], zfit.Space], array: numpy.ndarray, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create *Data* from a *np.array*.

Parameters

- **()** (*obs*) –
- **array** (*numpy.ndarray*) –
- **name** (*str*) –

Returns

Return type zfit.Data

classmethod from_pandas(*df: pandas.core.frame.DataFrame, obs: Union[str, Iterable[str], zfit.Space] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create a *Data* from a pandas DataFrame. If *obs* is *None*, columns are used as obs.

Parameters

- **df** (*pandas.DataFrame*) –
- **weights** (*tf.Tensor, None, np.ndarray, str]*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents).

- **obs** (*zfit.Space*) –
- **name** (*str*) –

classmethod from_root(*path: str, treepath: str, branches: List[str] = None, branches_alias: Dict[KT, VT] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray, str] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None, root_dir_options=None*) → zfit.core.data.Data

Create a *Data* from a ROOT file. Arguments are passed to *uproot*.

Parameters

- **path** (*str*) –
- **treepath** (*str*) –
- **branches** (*List[str]*) –
- **branches_alias** (*dict*) – A mapping from the *branches* (as keys) to the actual *observables* (as values). This allows to have different *observable* names, independent of the branch name in the file.
- **weights** (*tf.Tensor, None, np.ndarray, str]*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents). Can be a column of the ROOT file by using a string corresponding to a column.
- **name** (*str*) –
- **()** (*root_dir_options*) –

Returns

Return type *zfit.Data*

classmethod from_root_iter(*path*, *treepath*, *branches=None*, *entrysteps=None*, *name=None*, ***kwargs*)

classmethod from_sample(*sample_func: Callable, n: Union[int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], fixed_params=None, name: str = None, weights=None, dtype=None*)

classmethod from_tensor(*obs: Union[str, Iterable[str], zfit.Space], tensor: tensorflow.python.framework.ops.Tensor, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*) → zfit.core.data.Data

Create a *Data* from a *tf.Tensor*. *Value* simply returns the tensor (in the right order).

Parameters

- **obs** (*Union[str, List[str]*) –
- **tensor** (*tf.Tensor*) –
- **name** (*str*) –

Returns

Return type zfit.core.Data

classmethod get_cache_counting()

get_iteration()

graph_caching_methods = []

initialize()

iterator

n_obs
Return the number of observables.

n_samples

name
The name of the object.

nevents

numpy()

obs
Return the observables.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

resample(*param_values: Mapping[KT, VT_co] = None, n: Union[int, tensorflow.python.framework.ops.Tensor] = None*)
Update the sample by newly sampling. This affects any object that used this data already.

All params that are not in the attribute *fixed_params* will use their current value for the creation of the new sample. The value can also be overwritten for one sampling by providing a mapping with *param_values* from *Parameter* to the temporary *value*.

Parameters

- **param_values** (*Dict*) – a mapping from *Parameter* to a *value*. For the current sampling, *Parameter* will use the *value*.
- **n** (*int, tf.Tensor*) – the number of samples to produce. If the *Sampler* was created with anything else then a numerical or tf.Tensor, this can't be used.

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

set_data_range(*data_range*)

set_weights(*weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray]*)
Set (temporarily) the weights of the dataset.

Parameters **weights** (*tf.Tensor*, np.ndarray, None) –

sort_by_axes(*axes: Union[int, Iterable[int]], allow_superset: bool = False*)

sort_by_obs(*obs: Union[str, Iterable[str], zfit.Space], allow_superset: bool = False*)

space
Return the *Space* object that defines the dimensionality of the object.

to_pandas(*obs: Union[str, Iterable[str], zfit.Space] = None*)
Create a *pd.DataFrame* from *obs* as columns and return it.

Parameters **()** (*obs*) – The observables to use as columns. If *None*, all observables are used.

Returns:

unstack_x(*obs: Union[str, Iterable[str], zfit.Space] = None, always_list: bool = False*)
Return the unstacked data: a list of tensors or a single Tensor.

Parameters

- **()** (*obs*) – which observables to return
- **always_list** (*bool*) – If True, always return a list (also if length 1)

Returns List(tf.Tensor)

value(*obs: Union[str, Iterable[str], zfit.Space] = None*)

weights

zfit.core.data.**feed_function**(*data*, *feed_val*)

zfit.core.data.**feed_function_for_partial_run**(*data*)

zfit.core.data.**fetch_function**(*data*)

dependents

class zfit.core.dependents.**BaseDependentsMixin**
Bases: *zfit.core.interfaces.ZfitDependentsMixin*

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

dimension

class zfit.core.dimension.**BaseDimensional**
Bases: *zfit.core.interfaces.ZfitDimensional*

axes
Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

space
Return the *Space* object that defines the dimensionality of the object.

zfit.core.dimension.**add_spaces**(*spaces: Iterable[zfit.Space]*)
Add two spaces and merge their limits if possible or return False.

Parameters **spaces** (Iterable[*Space*]) –

Returns

Return type Union[None, *Space*, bool]

Raises `LimitsIncompatibleError` – if limits of the *spaces* cannot be merged because they overlap

`zfit.core.dimension.`**`combine_spaces`**(*spaces: Iterable[zfit.Space]*)

Combine spaces with different *obs* and *limits* to one *space*.

Checks if the limits in each obs coincide *exactly*. If this is not the case, the combination is not unambiguous and *False* is returned

Parameters **`spaces`** (List[*Space*]) –

Returns

Returns False if the limits don't coincide in one or more obs. Otherwise return the *Space* with all obs from *spaces* sorted by the order of *spaces* and with the combined limits.

Return type *zfit.Space* or False

Raises

- `ValueError` – if only one space is given
- `LimitsIncompatibleError` – If the limits of one or more spaces (or within a space) overlap
- `LimitsNotSpecifiedError` – If the limits for one or more obs but not all are None.

`zfit.core.dimension.`**`common_obs`**(*spaces: Union[zfit.Space, Iterable[zfit.Space]]*) → List[str]

Extract the union of *obs* from *spaces* in the order of *spaces*.

For example:

space1.obs: ['obs1', 'obs3']
space2.obs: ['obs2', 'obs3', 'obs1']
space3.obs: ['obs2']

returns ['obs1', 'obs3', 'obs2']

Parameters **`()`** (*spaces*) – :py:class:'~zfit.Space's to extract the obs from

Returns The observables as *str*

Return type List[str]

`zfit.core.dimension.`**`get_same_obs`**

`zfit.core.dimension.`**`is_combinable`**(*spaces*)

`zfit.core.dimension.`**`limits_consistent`**(*spaces: Iterable[zfit.Space]*)

Check if space limits are the *exact* same in each obs they are defined and therefore are compatible.

In this case, if a space has several limits, e.g. from -1 to 1 and from 2 to 3 (all in the same observable), to be consistent with this limits, other limits have to have (in this obs) also the limits from -1 to 1 and from 2 to 3. Only having the limit -1 to 1 _or_ 2 to 3 is considered _not_ consistent.

This function is useful to check if several spaces with *different* observables can be _combined_.

Parameters **`spaces`** (*List[zfit.Space]*) –

Returns

Return type bool

zfit.core.dimension.**limits_overlap**(*spaces: Union[zfit.Space, Iterable[zfit.Space]], allow_exact_match: bool = False*) → bool

Check if _any_ of the limits of *spaces* overlaps with _any_ other of *spaces*.

This also checks multiple limits within one space. If *allow_exact_match* is set to true, then an *exact* overlap of limits is allowed.

Parameters

- **spaces** (*Iterable[zfit.Space]*) –
- **allow_exact_match** (*bool*) – An exact overlap of two limits is counted as "not overlapping". Example: limits from -1 to 3 and 4 to 5 to *NOT* overlap with the limits 4 to 5 *iff allow_exact_match* is True.

Returns if there are overlapping limits.

Return type bool

integration

This module contains functions for the numeric as well as the analytic (partial) integration.

class zfit.core.integration.**AnalyticIntegral**(**args*, ***kwargs*)

Bases: object

Hold analytic integrals and manage their dimensions, limits etc.

get_max_axes(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], axes: Union[int, Iterable[int]] = None*) → Tuple[int]

Return the maximal available axes to integrate over analytically for given limits

Parameters

- **limits** (*Space*) – The integral function will be able to integrate over this limits
- **axes** (*tuple*) – The axes over which (or over a subset) it will integrate

Returns

Return type Tuple[int]

get_max_integral(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], axes: Union[int, Iterable[int]] = None*) → Union[None, zfit.core.integration.Integral]

Return the integral over the *limits* with *axes* (or a subset of them).

Parameters

- **limits** (*Space*) –
- **axes** (*Tuple[int]*) –

Returns Return a callable that integrated over the given limits.

Return type Union[None, *Integral*]

integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor, None], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], axes: Union[int, Iterable[int]] = None, norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, model: zfit.core.interfaces.ZfitModel = None, params: dict = None*) → Union[float, tensorflow.python.framework.ops.Tensor]

Integrate analytically over the axes if available.

Parameters

- **x** (*numeric*) – If a partial integration is made, x are the value to be evaluated for the partial integrated function. If a full integration is performed, this should be *None*.
- **limits** (*Space*) – The limits to integrate
- **axes** (*Tuple[int]*) – The dimensions to integrate over
- **norm_range** (*bool*) – **|norm_range_arg_descr|**
- **params** (*dict*) – The parameters of the function

Returns

Return type Union[tf.Tensor, float]

Raises NotImplementedError – If the requested integral is not available.

register(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], priority: int = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral.

Parameters

- **func** (*callable*) – The integral function. Takes 1 argument.
- **axes** (*tuple*) – **|dims_arg_descr|**
- **limits** (*possible*) – **|limits_arg_descr|** *Limits* can be None if *func* works for any
- **limits** –
- **priority** (*int*) – If two or more integrals can integrate over certain limits, the one with the higher priority is taken (usually around 0-100).
- **supports_norm_range** (*bool*) – If True, norm_range will (if needed) be given to *func* as an argument.
- **supports_multiple_limits** (*bool*) – If True, multiple limits may be given as an argument to *func*.

class zfit.core.integration.**Integral**(*func: Callable, limits: zfit.core.limits.Space, priority: Union[int, float]*)

Bases: object

A lightweight holder for the integral function.

class zfit.core.integration.**Integration**(*mc_sampler*, *draws_per_dim*)

Bases: object

class zfit.core.integration.**PartialIntegralSampleData**(*sample: List[tensorflow.python.framework.ops.Tensor], space: zfit.core.interfaces.ZfitSpace*)

Bases: *zfit.core.dimension.BaseDimensional*, *zfit.core.interfaces.ZfitData*

Takes a list of tensors and "fakes" a dataset. Useful for tensors with non-matching shapes.

Parameters

- **sample** (*List[tf.Tensor]*) –
- **()** (*space*) –

axes

Return the axes.

copy (*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

sort_by_axes (*axes*, *allow_superset: bool = False*)

sort_by_obs (*obs*, *allow_superset: bool = False*)

space
Return the `Space` object that defines the dimensionality of the object.

unstack_x (*always_list=False*)

value (*obs: List[str] = None*)

weights

zfit.core.integration.**auto_integrate** (*, *norm_range: bool = False*, *multiple_limits: bool = False*) → Callable

zfit.core.integration.**chunked_average** (*func*, *x*, *num_batches*, *batch_size*, *space*, *mc_sampler*)

zfit.core.integration.**mc_integrate** (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], axes: Union[int, Iterable[int], None] = None, x: Union[float, tensorflow.python.framework.ops.Tensor, None] = None, n_axes: Optional[int] = None, draws_per_dim: int = 20000, method: str = None, dtype: Type[CT_co] = tf.float64, mc_sampler: Callable = <function sample_halton_sequence>, importance_sampling: Optional[Callable] = None*) → tensorflow.python.framework.ops.Tensor

Monte Carlo integration of *func* over *limits*.

Parameters

- **func** (*callable*) – The function to be integrated over
- **limits** (*Space*) – The limits of the integral
- **axes** (*tuple(int)*) – The row to integrate over. None means integration over all value
- **x** (*numeric*) – If a partial integration is performed, this are the value where x will be evaluated.
- **n_axes** (*int*) – the number of total dimensions (old?)
- **draws_per_dim** (*int*) – How many random points to draw per dimensions
- **method** (*str*) – Which integration method to use
- **dtype** (*dtype*) – **|dtype_arg_descr|**
- **mc_sampler** (*callable*) – A function that takes one argument (*n_draws* or similar) and returns random value between 0 and 1.
- **()** (*importance_sampling*) –

Returns the integral

Return type numerical

zfit.core.integration.**normalization_chunked**(*func*, *n_axes*, *batch_size*, *num_batches*, *dtype*, *space*, *x=None*, *shape_after=()*)

zfit.core.integration.**normalization_nograd**(*func*, *n_axes*, *batch_size*, *num_batches*, *dtype*, *space*, *x=None*, *shape_after=()*)

zfit.core.integration.**numeric_integrate**()
Integrate *func* using numerical methods.

interfaces

class zfit.core.interfaces.**ZfitConstraint**
Bases: abc.ABC

value()

class zfit.core.interfaces.**ZfitData**
Bases: *zfit.core.interfaces.ZfitDimensional*

axes
Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

sort_by_axes(*axes*, *allow_superset: bool = False*)

sort_by_obs(*obs*, *allow_superset: bool = False*)

space
Return the *Space* object that defines the dimensionality of the object.

value(*obs: List[str] = None*) → Union[float, tensorflow.python.framework.ops.Tensor]

weights

class zfit.core.interfaces.**ZfitDependentsMixin**
Bases: object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

class zfit.core.interfaces.**ZfitDimensional**
Bases: *zfit.core.interfaces.ZfitObject*

axes
Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

space
Return the *Space* object that defines the dimensionality of the object.

class zfit.core.interfaces.**ZfitFunc**
Bases: *zfit.core.interfaces.ZfitModel*

as_pdf()

axes
Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The *DType* of *Tensor's handled by this 'model*.

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Integrate the function over *limits* (normalized over *norm_range* if not False).

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*) – the limits to normalize over or False to integrate the unnormalized probability
- **name** (*str*) –

Returns the integral value

Return type Tensor

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

params

partial_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Partially integrate the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: int = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*)

Register an analytic integral with the class.

Parameters

- **()** (*limits*) –
- **()** – **|limits_arg_descr|**
- **priority** (*int*) –
- **supports_multiple_limits** (*bool*) –
- **supports_norm_range** (*bool*) –

Returns:

classmethod register_inverse_analytic_integral (*func: Callable*)

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

sample (*n: int, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Sample *n* points within *limits* from the model.

Parameters

- **n** (*int*) – The number of samples to be generated
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns Tensor(n_obs, n_samples)

space

Return the *Space* object that defines the dimensionality of the object.

update_integration_options (**args*, ***kwargs*)

class zfit.core.interfaces.**ZfitFunctorMixin**

Bases: object

get_models () → List[zfit.core.interfaces.ZfitModel]

models

class zfit.core.interfaces.**ZfitLoss**

Bases: *zfit.core.interfaces.ZfitObject*, *zfit.core.interfaces.ZfitDependentsMixin*

add_constraints(*constraints: List[tensorflow.python.framework.ops.Tensor]*)

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

model

name

Name prepended to all ops created by this *model*.

value() → Union[tensorflow.python.framework.ops.Tensor, numpy.array]

class zfit.core.interfaces.**ZfitModel**

Bases: *zfit.core.interfaces.ZfitNumeric*, *zfit.core.interfaces.ZfitDimensional*

axes

Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The *DType* of *Tensor's handled by this 'model*.

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Integrate the function over *limits* (normalized over *norm_range* if not False).

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*) – the limits to normalize over or False to integrate the unnormalized probability
- **name** (*str*) –

Returns the integral value

Return type Tensor

n_obs

Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

params

partial_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Partially integrate the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: int = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*)
Register an analytic integral with the class.

Parameters

- **()** (*limits*) –
- **()** – **|limits_arg_descr|**
- **priority** (*int*) –
- **supports_multiple_limits** (*bool*) –
- **supports_norm_range** (*bool*) –

Returns:

classmethod register_inverse_analytic_integral(*func: Callable*)
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

sample(*n: int, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Sample *n* points within *limits* from the model.

Parameters

- **n** (*int*) – The number of samples to be generated

- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns Tensor(n_obs, n_samples)

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(**args*, ***kwargs*)

class zfit.core.interfaces.**ZfitNumeric**
Bases: *zfit.core.interfaces.ZfitDependentsMixin*, *zfit.core.interfaces.ZfitObject*

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The *DType* of *Tensor*'s handled by this 'model.

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

name
Name prepended to all ops created by this *model*.

params

class zfit.core.interfaces.**ZfitObject**
Bases: abc.ABC

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

name
Name prepended to all ops created by this *model*.

class zfit.core.interfaces.**ZfitPDF**
Bases: *zfit.core.interfaces.ZfitModel*

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

axes
Return the axes.

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]*) → zfit.core.interfaces.ZfitPDF

dtype
The *DType* of *Tensor*'s handled by this 'model.

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

get_yield() → Optional[zfit.core.interfaces.ZfitParameter]

integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Integrate the function over *limits* (normalized over *norm_range* if not False).

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*) – the limits to normalize over or False to integrate the unnormalized probability
- **name** (*str*) –

Returns the integral value

Return type Tensor

is_extended

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[tensorflow.python.framework.ops.Tensor, numpy.array]

obs
Return the observables.

params

partial_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Partially integrate the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'model'*) → Union[float, tensorflow.python.framework.ops.Tensor]

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: int = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*)
Register an analytic integral with the class.

Parameters

- **()** (*limits*) –
- **()** – |limits_arg_descr|
- **priority** (*int*) –
- **supports_multiple_limits** (*bool*) –
- **supports_norm_range** (*bool*) –

Returns:

classmethod register_inverse_analytic_integral (*func: Callable*)
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

sample (*n: int, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Sample *n* points within *limits* from the model.

Parameters

- **n** (*int*) – The number of samples to be generated
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns Tensor(n_obs, n_samples)

set_norm_range ()

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options (**args*, ***kwargs*)

class zfit.core.interfaces.**ZfitParameter**
Bases: *zfit.core.interfaces.ZfitNumeric*

copy (*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The *DType* of *Tensor*'s handled by this 'model*.

floating

get_dependents (*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params (*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

independent

name
Name prepended to all ops created by this *model*.

params

value () → tensorflow.python.framework.ops.Tensor

class zfit.core.interfaces.**ZfitSpace**
Bases: *zfit.core.interfaces.ZfitObject*

area () → float
Return the total area of all the limits and axes. Useful, for example, for MC integration.

axes

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

get_axes(*obs: Union[str, Tuple[str, ...]] = None*, *as_dict: bool = True*)
Return the axes number of the observable *if available* (set by *axes_by_obs*).

Raises `AxesNotUnambiguousError` – In case

get_subspace(*obs: Union[str, Iterable[str], zfit.Space] = None*, *axes=None*, *name=None*) → zfit.core.limits.Space

iter_areas(*rel: bool = False*) → Tuple[float, ...]
Return the areas of each limit.

iter_limits()
Iterate through the limits by returning several observables/(lower, upper)-tuples.

limits
Return the tuple(lower, upper).

lower
Return the lower limits.

n_limits
Return the number of limits.

n_obs
Return the number of observables (axis).

name
Name prepended to all ops created by this *model*.

obs
Return a list of the observable names.

upper
Return the upper limits.

with_autofill_axes(*overwrite: bool*)
Return a *Space* with filled axes corresponding to range(len(n_obs)).

Parameters **overwrite** (*bool*) – If *self.axes* is not None, replace the axes with the autofilled ones. If axes is already set, don't do anything if *overwrite* is False.

Returns *Space*

with_axes(*axes*)
Sort by *obs* and return the new instance.

Parameters **()** (*axes*) –

Returns *Space*

with_limits(*limits*, *name*)
Return a copy of the space with the new *limits* (and the new *name*).

Parameters

- **()** (*limits*) –
- **name** (*str*) –

Returns *Space*

with_obs(*obs*)
Sort by *obs* and return the new instance.

Parameters **()** (*obs*) –

Returns *Space*

limits

NamedSpace and limits

Limits define a certain interval in a specific dimension. This can be used to define, for example, the limits of an integral over several dimensions or a normalization range.

with limits

Therefore a different way of specifying limits is possible, basically by defining chunks of the lower and the upper limits. The shape of a lower resp. upper limit is (n_limits, n_obs).

Example: 1-dim: 1 to 4, 2.-dim: 21 to 24 AND 1.-dim: 6 to 7, 2.-dim 26 to 27 >>> lower = ((1, 21), (6, 26)) >>> upper = ((4, 24), (7, 27)) >>> limits2 = Space(limits=(lower, upper), obs=('obs1', 'obs2')

General form:

lower = ((lower1_dim1, lower1_dim2, lower1_dim3), (lower2_dim1, lower2_dim2, lower2_dim3),...) upper = ((upper1_dim1, upper1_dim2, upper1_dim3), (upper2_dim1, upper2_dim2, upper2_dim3),...)

Using *Space*

`NamedSpace` offers a few useful functions to easier deal with the intervals

Handling areas

For example when doing a MC integration using the expectation value, it is mandatory to know the total area of your intervals. You can retrieve the total area or (if multiple limits (=intervals

are given) the area of each interval.

```
>>> area = limits2.area()
>>> area_1, area_2 = limits2.iter_areas(rel=False)  # if rel is True, return␣
↪the fraction of 1
```

Retrieve the limits

```
>>> lower, upper = limits2.limits
```

which you can now iterate through. For example, to calc an integral (assuming there is a function *integrate* taking the lower and upper limits and returning the function), you can do >>> def integrate(lower_limit, upper_limit): return 42 # dummy function >>> integral = sum(integrate(lower_limit=low, upper_limit=up) for low, up in zip(lower, upper))

class zfit.core.limits.**Any**
Bases: object

class zfit.core.limits.**AnyLower**
Bases: *zfit.core.limits.Any*

class zfit.core.limits.**AnyUpper**
Bases: *zfit.core.limits.Any*

class zfit.core.limits.**Space**(*obs: Union[str, Iterable[str], zfit.Space], limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, name: Optional[str] = 'Space'*)
Bases: *zfit.core.interfaces.ZfitSpace*, *zfit.core.baseobject.BaseObject*

Define a space with the name (*obs*) of the axes (and it's number) and possibly it's limits.

Parameters

- **obs** (*str, List[str,..]*) –
- **()** (*limits*) –
- **name** (*str*) –

ANY = <Any>

ANY_LOWER = <Any Lower Limit>

ANY_UPPER = <Any Upper Limit>

AUTO_FILL = <object object>

add(*other: Union[zfit.Space, Iterable[zfit.Space]]*)
Add the limits of the spaces. Only works for the same obs.

In case the observables are different, the order of the first space is taken.

Parameters **other** (*Space*) –

Returns

Return type *Space*

area() → float
Return the total area of all the limits and axes. Useful, for example, for MC integration.

axes
The axes ("obs with int") the space is defined in.

Returns:

combine(*other: Union[zfit.Space, Iterable[zfit.Space]]*) → zfit.core.interfaces.ZfitSpace
Combine spaces with different obs (but consistent limits).

Parameters **other** (*Space*) –

Returns

Return type *Space*

copy(*name: Optional[str] = None*, ***overwrite_kwargs*) → zfit.Space
Create a new *Space* using the current attributes and overwriting with *overwrite_overwrite_kwargs*.

Parameters

- **name** (*str*) – The new name. If not given, the new instance will be named the same as the current one.
- **()** (***overwrite_kwargs*) –

Returns *Space*

classmethod from_axes(*axes: Union[int, Iterable[int]]*, *limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None*, *name: str = None*) → zfit.Space
Create a space from *axes* instead of from *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **name** (*str*) –

Returns *Space*

get_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, as_dict: bool = False, autofill: bool = False*) → Union[Tuple[int], None, Dict[str, int]]

Return the axes corresponding to the *obs* (or all if None).

Parameters

- **()** (*obs*) –
- **as_dict** (*bool*) – If True, returns a ordered dictionary with {obs: axis}
- **autofill** (*bool*) – If True and the axes are not specified, automatically fill them with the default numbering and return (not setting them).

Returns Tuple, OrderedDict

Raises

- ValueError – if the requested *obs* do not match with the one defined in the range
- AxesNotSpecifiedError – If the axes in this *Space* have not been specified.

get_obs_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*)

get_reorder_indices(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*) → Tuple[int]

Indices that would order *self.obs* as *obs* respectively *self.axes* as *axes*.

Parameters

- **()** (*axes*) –
- **()** –

Returns:

get_subspace(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, name: Optional[str] = None*) → zfit.Space

Create a *Space* consisting of only a subset of the *obs*/*axes* (only one allowed).

Parameters

- **obs** (*str, Tuple[str]*) –
- **axes** (*int, Tuple[int]*) –
- **()** (*name*) –

Returns:

iter_areas(*rel: bool = False*) → Tuple[float, ...]

Return the areas of each interval

Parameters **rel** (*bool*) – If True, return the relative fraction of each interval

Returns

Return type Tuple[float]

iter_limits(*as_tuple: bool = True*) → Union[Tuple[zfit.Space], Tuple[Tuple[Tuple[float]]], Tuple[Tuple[float]]]

Return the limits, either as *Space* objects or as pure limits-tuple.

This makes iterating over limits easier: *for limit in space.iter_limits()* allows to, for example, pass *limit* to a function that can deal with simple limits only or if *as_tuple* is True the *limit* can be directly used to calculate something.

Example

```
for lower, upper in space.iter_limits(as_tuple=True):
    integrals = integrate(lower, upper)  # calculate integral
integral = sum(integrals)
```

Returns

Return type List[*Space*] or List[limit,...]

limit1d
return the tuple(lower, upper).

Returns so `lower, upper = space.limit1d` for a simple, 1 obs limit.

Return type tuple(float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified limits getter for 1 obs, 1 limit only

limit2d
return the tuple(low_obs1, low_obs2, up_obs1, up_obs2).

Returns

so *low_x, low_y, up_x, up_y = space.limit2d* for a single, 2 obs limit. low_x is the lower limit in x, up_x is the upper limit in x etc.

Return type tuple(float, float, float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *limits* for exactly 2 obs, 1 limit

limits
Return the limits.

Returns:

limits1d
return the tuple(low_1, ..., low_n, up_1, ..., up_n).

Returns

so *low_1, low_2, up_1, up_2 = space.limits1d* for several, 1 obs limits. low_1 to up_1 is the first interval, low_2 to up_2 is the second interval etc.

Return type tuple(float, float, ..)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *.limits* for exactly 1 obs, n limits

lower
Return the lower limits.

Returns:

n_limits
The number of different limits.

Returns int >= 1

n_obs
Return the number of observables/axes.

Returns int >= 1

name
The name of the object.

obs
The observables ("axes with str")the space is defined in.

Returns:

obs_axes

reorder_by_indices(*indices: Tuple[int]*)
Return a *Space* reordered by the indices.

Parameters **()** (*indices*) –

upper
Return the upper limits.

Returns:

with_autofill_axes(*overwrite: bool = False*) → zfit.Space
Return a *Space* with filled axes corresponding to range(len(n_obs)).

Parameters **overwrite** (*bool*) – If *self.axes* is not None, replace the axes with the autofilled ones. If axes is already set, don't do anything if *overwrite* is False.

Returns *Space*

with_axes(*axes: Union[int, Iterable[int]]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*axes*) –

Returns *Space*

with_limits(*limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool], name: Optional[str] = None*) → zfit.Space
Return a copy of the space with the new *limits* (and the new *name*).

Parameters

- **()** (*limits*) –
- **name** (*str*) –

Returns *Space*

with_obs(*obs: Union[str, Iterable[str], zfit.Space]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*obs*) –

Returns *Space*

with_obs_axes(*obs_axes: Union[OrderedDict[str, int], Dict[str, int]], ordered: bool = False, allow_subset=False*) → zfit.Space
Return a new *Space* with reordered observables and set the *axes*.

Parameters

- **obs_axes** (*OrderedDict[str, int]*) – An ordered dict with {obs: axes}.
- **ordered** (*bool*) – If True (and the *obs_axes* is an *OrderedDict*), the

- **()** (*allow_subset*) –

Returns

Return type *Space*

zfit.core.limits.**contains_tensor**(*object*)

zfit.core.limits.**convert_to_obs_str**(*obs*)
Convert *obs* to the list of obs, also if it is a *Space*.

zfit.core.limits.**convert_to_space**(*obs: Union[str, Iterable[str], zfit.Space, None] = None, axes: Union[int, Iterable[int], None] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, *, overwrite_limits: bool = False, one_dim_limits_only: bool = True, simple_limits_only: bool = True*) → Union[None, zfit.core.limits.Space, bool]
Convert *limits* to a *Space* object if not already None or False.

Parameters

- **obs** (Union[Tuple[float, float], *Space*]) –
- **()** (*axes*) –
- **()** –
- **overwrite_limits** (*bool*) – If *obs* or *axes* is a *Space* _and_ *limits* are given, return an instance of *Space* with the new limits. If the flag is *False*, the *limits* argument will be ignored if
- **one_dim_limits_only** (*bool*) –
- **simple_limits_only** (*bool*) –

Returns

Return type Union[*Space*, False, None]

Raises OverdefinedError – if *obs* or *axes* is a *Space* and *axes* respectively *obs* is not *None*.

zfit.core.limits.**no_multiple_limits**(*func*)
Decorator: Catch the 'limits' kwargs. If it contains multiple limits, raise MultipleLimitsNotImplementedError.

zfit.core.limits.**no_norm_range**(*func*)
Decorator: Catch the 'norm_range' kwargs. If not None, raise NormRangeNotImplementedError.

zfit.core.limits.**shape_np_tf**(*object*)

zfit.core.limits.**supports**(*, *norm_range: bool = False*, *multiple_limits: bool = False*) → Callable
Decorator: Add (mandatory for some methods) on a method to control what it can handle.

If any of the flags is set to False, it will check the arguments and, in case they match a flag (say if a *norm_range* is passed while the *norm_range* flag is set to *False*), it will raise a corresponding exception (in this example a *NormRangeNotImplementedError*) that will be catched by an earlier function that knows how to handle things.

Parameters

- **norm_range** (*bool*) – If False, no norm_range argument will be passed through resp. will be *None*
- **multiple_limits** (*bool*) – If False, only simple limits are to be expected and no iteration is therefore required.

loss

class zfit.core.loss.**BaseLoss**(*model: Union[zfit.core.interfaces.ZfitModel, Iterable[zfit.core.interfaces.ZfitModel]], data: Union[zfit.Data, Iterable[zfit.Data]], fit_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = <zfit.util.checks.NotSpecified object>, constraints: Iterable[Union[zfit.core.interfaces.ZfitConstraint, Callable]] = None*)

Bases: *zfit.core.dependents.BaseDependentsMixin*, *zfit.core.interfaces.ZfitLoss*, *zfit.util.cache.Cachable*, *zfit.core.baseobject.BaseObject*

A "simultaneous fit" can be performed by giving one or more *model*, *data*, *fit_range* to the loss. The length of each has to match the length of the others.

Parameters

- **model** (*Iterable[ZfitModel]*) – The model or models to evaluate the data on
- **data** (*Iterable[ZfitData]*) – Data to use
- **fit_range** (Iterable[*Space*]) – The fitting range. It's the norm_range for the models (if
- **they** – have a norm_range) and the data_range for the data.
- **constraints** (*Iterable[tf.Tensor]*) – A Tensor representing a loss constraint. Using *zfit.constraint.** allows for easy use of predefined constraints.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.core.loss.**CachedLoss**(*model*, *data*, *fit_range=None*, *constraints=None*)
Bases: *zfit.core.loss.BaseLoss*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.core.loss.**ExtendedUnbinnedNLL**(*model*, *data*, *fit_range=<zfit.util.checks.NotSpecified object>*, *constraints=None*)
Bases: *zfit.core.loss.UnbinnedNLL*

An Unbinned Negative Log Likelihood with an additional poisson term for the

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.core.loss.**SimpleLoss**(*func: Callable*, *dependents: Iterable[zfit.Parameter] = <zfit.util.checks.NotSpecified object>*, *errordef: Optional[float] = None*)
Bases: *zfit.core.loss.BaseLoss*

Loss from a (function returning a) Tensor.

Parameters

- **func** – Callable that constructs the loss and returns a tensor.
- **dependents** – The dependents (independent *zfit.Parameter*) of the loss. If not given, the dependents are figured out automatically.
- **errordef** – Definition of which change in the loss corresponds to a change of 1 sigma. For example, 1 for Chi squared, 0.5 for negative log-likelihood.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (`cacher`) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.core.loss.**UnbinnedNLL**(*model*, *data*, *fit_range=<zfit.util.checks.NotSpecified object>*, *constraints=None*)
Bases: `zfit.core.loss.BaseLoss`

The Unbinned Negative Log Likelihood.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (`bool`) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (`cacher`) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

operations

zfit.core.operations.**add**(*object1: Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF], object2: Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF]*) → Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF]
Add two objects and return a new object (may depending on the old).

Parameters

- **()** (`object2`) – A ZfitParameter, ZfitFunc or ZfitPDF to add with object2
- **()** – A ZfitParameter, ZfitFunc or ZfitPDF to add with object1

zfit.core.operations.**add_func_func**(*func1: zfit.core.interfaces.ZfitFunc, func2: zfit.core.interfaces.ZfitFunc, name: str = 'add_func_func'*) → SumFunc

zfit.core.operations.**add_param_func**(*param: zfit.core.interfaces.ZfitParameter, func: zfit.core.interfaces.ZfitFunc*) → zfit.core.interfaces.ZfitFunc

zfit.core.operations.**add_param_param**(*param1: zfit.core.interfaces.ZfitParameter, param2: zfit.core.interfaces.ZfitParameter*) → zfit.core.interfaces.ZfitParameter

zfit.core.operations.**add_pdf_pdf**(*pdf1: zfit.core.interfaces.ZfitPDF, pdf2: zfit.core.interfaces.ZfitPDF, name: str = 'add_pdf_pdf'*) → SumPDF

zfit.core.operations.**convert_func_to_pdf**(*func: Union[zfit.core.interfaces.ZfitFunc, Callable], obs=None, name=None*) → zfit.core.interfaces.ZfitPDF

zfit.core.operations.**convert_pdf_to_func**(*pdf: zfit.core.interfaces.ZfitPDF, norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]*) → zfit.core.interfaces.ZfitFunc

zfit.core.operations.**multiply**(*object1: Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF], object2: Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF]*) → Union[zfit.core.interfaces.ZfitParameter, zfit.core.interfaces.ZfitFunction, zfit.core.interfaces.ZfitPDF]

Multiply two objects and return a new object (may depending on the old).

Parameters

- **()** (*object2*) – A ZfitParameter, ZfitFunc or ZfitPDF to multiply with object2
- **()** – A ZfitParameter, ZfitFunc or ZfitPDF to multiply with object1

Raises TypeError – if one of the objects is neither a ZfitFunc, ZfitPDF or convertable to a ZfitParameter

zfit.core.operations.**multiply_func_func**(*func1: zfit.core.interfaces.ZfitFunc, func2: zfit.core.interfaces.ZfitFunc, name: str = 'multiply_func_func'*) → ProdFunc

zfit.core.operations.**multiply_param_func**(*param: zfit.core.interfaces.ZfitParameter, func: zfit.core.interfaces.ZfitFunc*) → zfit.core.interfaces.ZfitFunc

zfit.core.operations.**multiply_param_param**(*param1: zfit.core.interfaces.ZfitParameter, param2: zfit.core.interfaces.ZfitParameter*) → zfit.core.interfaces.ZfitParameter

zfit.core.operations.**multiply_param_pdf**(*param: zfit.core.interfaces.ZfitParameter, pdf: zfit.core.interfaces.ZfitPDF*) → zfit.core.interfaces.ZfitPDF

zfit.core.operations.**multiply_pdf_pdf**(*pdf1: zfit.core.interfaces.ZfitPDF, pdf2: zfit.core.interfaces.ZfitPDF, name: str = 'multiply_pdf_pdf'*) → ProductPDF

parameter

Define Parameter which holds the value.

class zfit.core.parameter.**BaseComposedParameter**(*params, value_fn, name='BaseComposedParameter', **kwargs*)

Bases: *zfit.core.parameter.BaseZParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name

The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.core.parameter.**BaseParameter**
Bases: *zfit.core.interfaces.ZfitParameter*

copy(*deep: bool = False*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The *DType* of *Tensor‘s handled by this ‘model*.

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

independent

name
Name prepended to all ops created by this *model*.

params

value() → tensorflow.python.framework.ops.Tensor

class zfit.core.parameter.**BaseZParameter**(*name*, ***kwargs*)
Bases: *zfit.core.parameter.ZfitParameterMixin*, *zfit.core.parameter.ComposedVariable*, *zfit.core.parameter.BaseParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.core.parameter.**ComplexParameter**(*name, value_fn, dependents, dtype=tf.complex128, **kwargs*)
Bases: *zfit.core.parameter.ComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

arg

assign(*value, use_locking=False, name=None, read_value=True*)

conj

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

static from_cartesian(*name*, *real*, *imag*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

static from_polar(*name*, *mod*, *arg*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

imag

independent

mod

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

real

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.core.parameter.**ComposedParameter**(*name*, *value_fn*, *dependents*, *dtype=tf.float64*, ***kwargs*)
Bases: *zfit.core.parameter.BaseComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name

The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.core.parameter.**ComposedVariable**(*name: str*, *value_fn: Callable*, ***kwargs*)
Bases: object

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

numpy()

read_value()

value()

class zfit.core.parameter.**ConstantParameter**(*name*, *value*, *dtype=tf.float64*)
Bases: *zfit.core.parameter.BaseZParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False*, *names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.core.parameter.**MetaBaseParameter**
Bases: tensorflow.python.ops.variables.VariableMetaclass, abc.ABCMeta

__instancecheck__(*instance*)
Override for isinstance(instance, cls).

__subclasscheck__(*subclass*)
Override for issubclass(subclass, cls).

mro()
Return a type's method resolution order.

register(*subclass*)
Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

class zfit.core.parameter.**Parameter**(*name*, *value*, *lower_limit=None*, *upper_limit=None*, *step_size=None*, *floating=True*, *dtype=tf.float64*, ***kwargs*)
Bases: *zfit.core.parameter.ZfitParameterMixin*, *zfit.core.parameter.TFBaseVariable*, *zfit.core.parameter.BaseParameter*

Class for fit parameters, derived from TF Variable class.

name : name of the parameter, value : starting value lower_limit : lower limit upper_limit : upper limit step_size : step size (set to 0 for fixed parameters)

class SaveSliceInfo(*full_name=None*, *full_shape=None*, *var_offset=None*, *var_shape=None*, *save_slice_info_def=None*, *import_scope=None*)
Bases: object

Information on how to save this Variable as a slice.

Provides internal support for saving variables as slices of a larger variable. This API is not public and is subject to change.

Available properties:

- full_name
- full_shape

- var_offset
- var_shape

Create a *SaveSliceInfo*.

Parameters

- **full_name** – Name of the full variable of which this *Variable* is a slice.
- **full_shape** – Shape of the full variable, as a list of int.
- **var_offset** – Offset of this *Variable* into the full variable, as a list of int.
- **var_shape** – Shape of this *Variable*, as a list of int.
- **save_slice_info_def** – *SaveSliceInfoDef* protocol buffer. If not *None*, recreates the SaveSliceInfo object its contents. *save_slice_info_def* and other arguments are mutually exclusive.
- **import_scope** – Optional *string*. Name scope to add. Only used when initializing from protocol buffer.

spec
Computes the spec string used for saving.

to_proto(*export_scope=None*)
Returns a SaveSliceInfoDef() proto.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Returns A *SaveSliceInfoDef* protocol buffer, or None if the *Variable* is not in the specified name scope.

__iter__()
Dummy method to prevent iteration.

Do not call.

NOTE(mrry): If we register __getitem__ as an overloaded operator, Python will valiantly attempt to iterate over the variable's Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises TypeError – when invoked.

__ne__(*other*)
Compares two variables element-wise for equality.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (bool) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

aggregation

assign(*value*, *use_locking=None*, *name=None*, *read_value=True*)
Assigns a new value to this variable.

Parameters

- **value** – A *Tensor*. The new value for this variable.
- **use_locking** – If *True*, use locking during the assignment.
- **name** – The name to use for the assignment.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_add(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Adds a value to this variable.

Parameters

- **delta** – A *Tensor*. The value to add to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_sub(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Subtracts a value from this variable.

Parameters

- **delta** – A *Tensor*. The value to subtract from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

batch_scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable batch-wise.

Analogous to *batch_gather*. This assumes that this variable and the sparse_delta IndexedSlices have a series of leading dimensions that are the same for all of them, and the updates are performed on the last dimension of indices. In other words, the dimensions should be the following:

num_prefix_dims = sparse_delta.indices.ndims - 1 batch_dim = num_prefix_dims + 1 ‘sparse_delta.updates.shape = sparse_delta.indices.shape + var.shape[

batch_dim:]‘

where

sparse_delta.updates.shape[:num_prefix_dims] == sparse_delta.indices.shape[:num_prefix_dims] == var.shape[:num_prefix_dims]

And the operation performed can be expressed as:

‘var[i_1, . . . , i_n,

sparse_delta.indices[i_1, . . . , i_n, j]] = sparse_delta.updates[i_1, . . . , i_n, j]‘

When sparse_delta.indices is a 1D tensor, this operation is equivalent to *scatter_update*.

To avoid this operation one can looping over the first *ndims* of the variable and using *scatter_update* on the subtensors that result of slicing the first dimension. This is a valid option for *ndims = 1*, but less efficient than this implementation.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

constraint
Returns the constraint function associated with this variable.

Returns The constraint function that was passed to the variable constructor. Can be *None* if no constraint was passed.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

count_up_to(*limit*)
Increments this variable until it reaches *limit*. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Dataset.range instead.

When that Op is run it tries to increment the variable by *1*. If incrementing the variable would bring it above *limit* then the Op raises the exception *OutOfRangeError*.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for *count_up_to(self, limit)*.

Parameters **limit** – value at which incrementing the variable raises an error.

Returns A *Tensor* that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

create
The op responsible for initializing this variable.

device
The device this variable is on.

dtype
The dtype of the object

eval(*session=None*)
Evaluates and returns the value of this variable.

experimental_ref()
Returns a hashable reference object to this Variable.

Warning: Experimental API that could be changed or removed.

The primary usecase for this API is to put variables in a set/dictionary. We can't put variables in a set/dictionary as *variable.__hash__()* is no longer available starting Tensorflow 2.0.

‘‘‘python import tensorflow as tf

x = tf.Variable(5) y = tf.Variable(10) z = tf.Variable(10)

The followings will raise an exception starting 2.0 # TypeError: Variable is unhashable if Variable equality is enabled. variable_set = {x, y, z} variable_dict = {x: ‘five’, y: ‘ten’} ‘‘‘

Instead, we can use *variable.experimental_ref()*.

‘‘‘python variable_set = {x.experimental_ref(),

y.experimental_ref(), z.experimental_ref()}

print(x.experimental_ref() in variable_set) ==> True

variable_dict = {x.experimental_ref(): ‘five’, y.experimental_ref(): ‘ten’, z.experimental_ref(): ‘ten’}

print(variable_dict[y.experimental_ref()]) ==> ten ‘‘‘

Also, the reference object provides *.deref()* function that returns the original Variable.

```
`python x = tf.Variable(5) print(x.experimental_ref().deref()) ==>
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=5> `
```

floating

static from_proto(*variable_def*, *import_scope=None*)
Returns a *Variable* object created from *variable_def*.

gather_nd(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather_nd*.

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_shape()
Alias of *Variable.shape*.

graph
The *Graph* of this variable.

graph_caching_methods = []

handle
The handle by which this variable can be accessed.

has_limits

independent

initial_value
Returns the Tensor used as the initial value for the variable.

initialized_value()
Returns the value of the initialized variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun) contexts.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
`python # Initialize 'v' with a random tensor. v = tf.Variable(tf.
random.truncated_normal([10, 40])) # Use `initialized_value` to
guarantee that `v` has been # initialized before its value is used
to initialize `w`. # The random values are picked only once. w = tf.
Variable(v.initialized_value() * 2.0) `
```

Returns A *Tensor* holding the value of this variable after its initializer has run.

initializer
The op responsible for initializing this variable.

is_initialized(*name=None*)
Checks whether a resource variable has been initialized.

Outputs boolean scalar indicating whether the tensor has been initialized.

Parameters **name** – A name for the operation (optional).

Returns A *Tensor* of type *bool*.

load(*value*, *session=None*)
Load new value into this variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Variable.assign which has equivalent behavior in 2.X.

Writes new value to variable's memory. Doesn't add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See *tf.compat.v1.Session* for more information on launching a graph and on sessions.

‘‘‘python v = tf.Variable([1, 2]) init = tf.compat.v1.global_variables_initializer()

with tf.compat.v1.Session() as sess: sess.run(init) # Usage passing the session explicitly. v.load([2, 3], sess) print(v.eval(sess)) # prints [2 3] # Usage with the default session. The 'with' block # above makes 'sess' the default session. v.load([3, 4], sess) print(v.eval()) # prints [3 4]

‘‘‘

Parameters

- **value** – New variable value

- **session** – The session to use to evaluate this variable. If none, the default session is used.

Raises ValueError – Session is not passed and no default session

lower_limit

name
The name of the object.

numpy()

old_graph_caching_methods = []

op
The op for this variable.

params

randomize(*minval=None*, *maxval=None*, *sampler=<built-in method uniform of numpy.random.mtrand.RandomState object>*)
Update the value with a randomised value between minval and maxval.

Parameters

- **minval** (*Numerical*) –
- **maxval** (*Numerical*) –
- **()** (*sampler*) –

read_value()
Constructs an op which reads the value of this variable.

Should be used when there are multiple reads, or when it is desirable to read the value only after some condition is true.

Returns the read operation.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

scatter_add(*sparse_delta*, *use_locking=False*, *name=None*)
Adds *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be added to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered addition has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_div(*sparse_delta*, *use_locking=False*, *name=None*)
Divide this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to divide this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered division has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_max(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the max of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of max with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered maximization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_min(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the min of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of min with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered minimization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_mul(*sparse_delta*, *use_locking=False*, *name=None*)
Multiply this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to multiply this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered multiplication has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_nd_add(*indices*, *updates*, *name=None*)
Applies sparse addition to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K* = *P*) or slices (if *K* < *P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

'''**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) add = ref.scatter_nd_add(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(add)

'''

The resulting update to ref would look like this:

[1, 13, 3, 14, 14, 6, 7, 20]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_sub(*indices*, *updates*, *name=None*)

Applies sparse subtraction to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K* = *P*) or slices (if *K* < *P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

'''**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_sub(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

'''

The resulting update to ref would look like this:

[1, -9, 3, -6, -6, 6, 7, -4]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_update(*indices*, *updates*, *name=None*)

Applies sparse assignment to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_update(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, 11, 3, 10, 9, 6, 7, 12]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_sub(*sparse_delta*, *use_locking=False*, *name=None*)

Subtracts *tf.IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be subtracted from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

set_shape(*shape*)
Unsupported.

set_value(*value: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)
Set the `Parameter` to *value* (temporarily if used in a context manager).

Parameters **value** (*float*) – The value the parameter will take on.

shape
The shape of this variable.

sparse_read(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather*.

step_size

synchronization

to_proto(*export_scope=None*)
Converts a *ResourceVariable* to a *VariableDef* protocol buffer.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Raises `RuntimeError` – If run in EAGER mode.

Returns A *VariableDef* protocol buffer, or *None* if the *Variable* is not in the specified name scope.

trainable

upper_limit

value()
A cached operation which reads the value of this variable.

class zfit.core.parameter.**TFBaseVariable**(*initial_value=None*, *trainable=None*, *collections=None*, *validate_shape=True*, *caching_device=None*, *name=None*, *dtype=None*, *variable_def=None*, *import_scope=None*, *constraint=None*, *distribute_strategy=None*, *synchronization=None*, *aggregation=None*, *shape=None*)
Bases: `tensorflow.python.ops.resource_variable_ops.ResourceVariable`

Creates a variable.

Parameters

- **initial_value** – A *Tensor*, or Python object convertible to a *Tensor*, which is the initial value for the Variable. Can also be a callable with no argument that returns the

initial value when called. (Note that initializer functions from init_ops.py must first be bound

to a shape before being used here.)

- **trainable** – If *True*, the default, also adds the variable to the graph collection *GraphKeys.TRAINABLE_VARIABLES*. This collection is used as the default list of variables to use by the *Optimizer* classes. Defaults to *True*, unless *synchronization* is set to *ON_READ*, in which case it defaults to *False*.
- **collections** – List of graph collections keys. The new variable is added to these collections. Defaults to *[GraphKeys.GLOBAL_VARIABLES]*.
- **validate_shape** – Ignored. Provided for compatibility with tf.Variable.
- **caching_device** – Optional device string or function describing where the Variable should be cached for reading. Defaults to the Variable's device. If not *None*, caches on another device. Typical use is to cache on the device where the Ops using the Variable reside, to deduplicate copying through *Switch* and other conditional statements.
- **name** – Optional name for the variable. Defaults to *'Variable'* and gets uniquified automatically.
- **dtype** – If set, initial_value will be converted to the given type. If None, either the datatype will be kept (if initial_value is a Tensor) or float32 will be used (if it is a Python object convertible to a Tensor).
- **variable_def** – *VariableDef* protocol buffer. If not None, recreates the *ResourceVariable* object with its contents. *variable_def* and other arguments (except for import_scope) are mutually exclusive.
- **import_scope** – Optional *string*. Name scope to add to the ResourceVariable. Only used when *variable_def* is provided.
- **constraint** – An optional projection function to be applied to the variable after being updated by an *Optimizer* (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected Tensor representing the value of the variable and return the Tensor for the projected value (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
- **distribute_strategy** – The tf.distribute.Strategy this variable is being created inside of.
- **synchronization** – Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class *tf.VariableSynchronization*. By default the synchronization is set to *AUTO* and the current *DistributionStrategy* chooses when to synchronize.
- **aggregation** – Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class *tf.VariableAggregation*.
- **shape** – (optional) The shape of this variable. If None, the shape of *initial_value* will be used. When setting this argument to *tf.TensorShape(None)* (representing an unspecified shape), the variable can be assigned with values of different shapes.

Raises `ValueError` – If the initial value is not specified, or does not have a shape and *validate_shape* is *True*.

@compatibility(eager) When Eager Execution is enabled, the default for the *collections* argument is *None*, which signifies that this *Variable* will not be added to any collections. @end_compatibility

class SaveSliceInfo(*full_name=None*, *full_shape=None*, *var_offset=None*, *var_shape=None*, *save_slice_info_def=None*, *import_scope=None*)

Bases: object

Information on how to save this Variable as a slice.

Provides internal support for saving variables as slices of a larger variable. This API is not public and is subject to change.

Available properties:

- full_name
- full_shape
- var_offset
- var_shape

Create a *SaveSliceInfo*.

Parameters

- **full_name** – Name of the full variable of which this *Variable* is a slice.
- **full_shape** – Shape of the full variable, as a list of int.
- **var_offset** – Offset of this *Variable* into the full variable, as a list of int.
- **var_shape** – Shape of this *Variable*, as a list of int.
- **save_slice_info_def** – *SaveSliceInfoDef* protocol buffer. If not *None*, recreates the SaveSliceInfo object its contents. *save_slice_info_def* and other arguments are mutually exclusive.
- **import_scope** – Optional *string*. Name scope to add. Only used when initializing from protocol buffer.

spec

Computes the spec string used for saving.

to_proto(*export_scope=None*)

Returns a SaveSliceInfoDef() proto.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Returns A *SaveSliceInfoDef* protocol buffer, or None if the *Variable* is not in the specified name scope.

__eq__(*other*)

Compares two variables element-wise for equality.

__iter__()

Dummy method to prevent iteration.

Do not call.

NOTE(mrry): If we register __getitem__ as an overloaded operator, Python will valiantly attempt to iterate over the variable's Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises TypeError – when invoked.

__ne__(*other*)

Compares two variables element-wise for equality.

aggregation

assign(*value*, *use_locking=None*, *name=None*, *read_value=True*)
Assigns a new value to this variable.

Parameters

- **value** – A *Tensor*. The new value for this variable.
- **use_locking** – If *True*, use locking during the assignment.
- **name** – The name to use for the assignment.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_add(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Adds a value to this variable.

Parameters

- **delta** – A *Tensor*. The value to add to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_sub(*delta*, *use_locking=None*, *name=None*, *read_value=True*)
Subtracts a value from this variable.

Parameters

- **delta** – A *Tensor*. The value to subtract from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

batch_scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable batch-wise.

Analogous to *batch_gather*. This assumes that this variable and the sparse_delta IndexedSlices have a series of leading dimensions that are the same for all of them, and the updates are performed on the last dimension of indices. In other words, the dimensions should be the following:

num_prefix_dims = sparse_delta.indices.ndims - 1 batch_dim = num_prefix_dims + 1 ‘sparse_delta.updates.shape = sparse_delta.indices.shape + var.shape[

batch_dim:]‘

where

sparse_delta.updates.shape[:num_prefix_dims] == sparse_delta.indices.shape[:num_prefix_dims] == var.shape[:num_prefix_dims]

And the operation performed can be expressed as:

‘**var[i_1, . . . , i_n,**

sparse_delta.indices[i_1, . . . , i_n, j]] = sparse_delta.updates[i_1, . . . , i_n, j]‘

When sparse_delta.indices is a 1D tensor, this operation is equivalent to *scatter_update*.

To avoid this operation one can looping over the first *ndims* of the variable and using *scatter_update* on the subtensors that result of slicing the first dimension. This is a valid option for *ndims = 1*, but less efficient than this implementation.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

constraint
Returns the constraint function associated with this variable.

Returns The constraint function that was passed to the variable constructor. Can be *None* if no constraint was passed.

count_up_to(*limit*)
Increments this variable until it reaches *limit*. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Dataset.range instead.

When that Op is run it tries to increment the variable by *1*. If incrementing the variable would bring it above *limit* then the Op raises the exception *OutOfRangeError*.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for *count_up_to(self, limit)*.

Parameters **limit** – value at which incrementing the variable raises an error.

Returns A *Tensor* that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

create
The op responsible for initializing this variable.

device
The device this variable is on.

dtype
The dtype of this variable.

eval(*session=None*)
Evaluates and returns the value of this variable.

experimental_ref()
Returns a hashable reference object to this Variable.

Warning: Experimental API that could be changed or removed.

The primary usecase for this API is to put variables in a set/dictionary. We can't put variables in a set/dictionary as *variable.__hash__()* is no longer available starting Tensorflow 2.0.

‘‘‘python import tensorflow as tf

x = tf.Variable(5) y = tf.Variable(10) z = tf.Variable(10)

The followings will raise an exception starting 2.0 # TypeError: Variable is unhashable if Variable equality is enabled. variable_set = {x, y, z} variable_dict = {x: ‘five’, y: ‘ten’} ‘‘‘

Instead, we can use *variable.experimental_ref()*.

‘‘‘python variable_set = {x.experimental_ref(),

y.experimental_ref(), z.experimental_ref()}

print(x.experimental_ref() in variable_set) ==> True

variable_dict = {x.experimental_ref(): ‘five’, y.experimental_ref(): ‘ten’, z.experimental_ref(): ‘ten’}

print(variable_dict[y.experimental_ref()]) ==> ten ‘‘‘

Also, the reference object provides *.deref()* function that returns the original Variable.

```
`python x = tf.Variable(5) print(x.experimental_ref().deref()) ==>
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=5> `
```

static from_proto(*variable_def*, *import_scope=None*)
Returns a *Variable* object created from *variable_def*.

gather_nd(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather_nd*.

get_shape()
Alias of *Variable.shape*.

graph
The *Graph* of this variable.

handle
The handle by which this variable can be accessed.

initial_value
Returns the Tensor used as the initial value for the variable.

initialized_value()
Returns the value of the initialized variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun) contexts.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
`python # Initialize 'v' with a random tensor. v = tf.Variable(tf.
random.truncated_normal([10, 40])) # Use `initialized_value` to
guarantee that `v` has been # initialized before its value is used
to initialize `w`. # The random values are picked only once. w = tf.
Variable(v.initialized_value() * 2.0) `
```

Returns A *Tensor* holding the value of this variable after its initializer has run.

initializer
The op responsible for initializing this variable.

is_initialized(*name=None*)
Checks whether a resource variable has been initialized.

Outputs boolean scalar indicating whether the tensor has been initialized.

Parameters **name** – A name for the operation (optional).

Returns A *Tensor* of type *bool*.

load(*value*, *session=None*)
Load new value into this variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Variable.assign which has equivalent behavior in 2.X.

Writes new value to variable's memory. Doesn't add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See *tf.compat.v1.Session* for more information on launching a graph and on sessions.

‘‘‘python v = tf.Variable([1, 2]) init = tf.compat.v1.global_variables_initializer()

with tf.compat.v1.Session() as sess: sess.run(init) # Usage passing the session explicitly. v.load([2, 3], sess) print(v.eval(sess)) # prints [2 3] # Usage with the default session. The ‘with’ block # above makes ‘sess’ the default session. v.load([3, 4], sess) print(v.eval()) # prints [3 4]

‘‘‘

Parameters

- **value** – New variable value
- **session** – The session to use to evaluate this variable. If none, the default session is used.

Raises `ValueError` – Session is not passed and no default session

name
The name of the handle for this variable.

numpy()

op
The op for this variable.

read_value()
Constructs an op which reads the value of this variable.

Should be used when there are multiple reads, or when it is desirable to read the value only after some condition is true.

Returns the read operation.

scatter_add(*sparse_delta*, *use_locking=False*, *name=None*)
Adds *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be added to this variable.
- **use_locking** – If *True*, use locking during the operation.

- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered addition has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_div(*sparse_delta*, *use_locking=False*, *name=None*)
Divide this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to divide this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered division has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_max(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the max of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of max with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered maximization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_min(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the min of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of min with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered minimization has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_mul(*sparse_delta*, *use_locking=False*, *name=None*)
Multiply this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to multiply this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered multiplication has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_nd_add(*indices*, *updates*, *name=None*)

Applies sparse addition to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) add = ref.scatter_nd_add(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(add)

‘‘‘

The resulting update to ref would look like this:

[1, 13, 3, 14, 14, 6, 7, 20]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_sub(*indices*, *updates*, *name=None*)

Applies sparse subtraction to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_sub(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, -9, 3, -6, -6, 6, 7, -4]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_update(*indices*, *updates*, *name=None*)

Applies sparse assignment to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K‘th dimension of ‘ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_update(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, 11, 3, 10, 9, 6, 7, 12]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_sub(*sparse_delta*, *use_locking=False*, *name=None*)

Subtracts *tf.IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be subtracted from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

set_shape(*shape*)
Unsupported.

shape
The shape of this variable.

sparse_read(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather*.

synchronization

to_proto(*export_scope=None*)
Converts a *ResourceVariable* to a *VariableDef* protocol buffer.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Raises `RuntimeError` – If run in EAGER mode.

Returns A *VariableDef* protocol buffer, or *None* if the *Variable* is not in the specified name scope.

trainable

value()
A cached operation which reads the value of this variable.

class zfit.core.parameter.**ZfitBaseVariable**(*variable: tensorflow.python.ops.variables.Variable*, ***kwargs*)
Bases: `object`

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

dtype

value()

class zfit.core.parameter.**ZfitParameterMixin**(*name*, ***kwargs*)
Bases: *zfit.core.baseobject.BaseNumeric*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name

The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

zfit.core.parameter.**convert_to_parameter**(*value, name=None, prefer_floating=False, dependents=None, graph_mode=False*) → zfit.core.interfaces.ZfitParameter

Convert a *numerical* to a fixed/floating parameter or return if already a parameter.

Parameters

- **()** (*name*) –
- **()** –
- **prefer_floating** – If True, create a Parameter instead of a FixedParameter _if possible_.

zfit.core.parameter.**get_auto_number**()

zfit.core.parameter.**register_tensor_conversion**(*convertable*, *overload_operators=True*, *priority=1*)

sample

class zfit.core.sample.**EventSpace**(*obs: Union[str, Iterable[str], zfit.Space], limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool], factory=None, name: Optional[str] = 'Space'*)

Bases: *zfit.core.limits.Space*

EXPERIMENTAL SPACE CLASS!

ANY = <Any>

ANY_LOWER = <Any Lower Limit>

ANY_UPPER = <Any Upper Limit>

AUTO_FILL = <object object>

add(*other: Union[zfit.Space, Iterable[zfit.Space]]*)
Add the limits of the spaces. Only works for the same obs.

In case the observables are different, the order of the first space is taken.

Parameters **other** (*Space*) –

Returns

Return type *Space*

area() → float
Return the total area of all the limits and axes. Useful, for example, for MC integration.

axes
The axes ("obs with int") the space is defined in.

Returns:

combine(*other: Union[zfit.Space, Iterable[zfit.Space]]*)
Combine spaces with different obs (but consistent limits).

Parameters **other** (*Space*) –

Returns

Return type *Space*

copy(*name: Optional[str] = None*, ***overwrite_kwargs*) → zfit.Space
Create a new *Space* using the current attributes and overwriting with *overwrite_overwrite_kwargs*.

Parameters

- **name** (*str*) – The new name. If not given, the new instance will be named the same as the current one.

- **()** (***overwrite_kwargs*) –

Returns *Space*

create_limits(*n*)

factory

classmethod from_axes(*axes: Union[int, Iterable[int]], limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, name: str = None*) → zfit.Space

Create a space from *axes* instead of from *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **name** (*str*) –

Returns *Space*

get_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, as_dict: bool = False, autofill: bool = False*) → Union[Tuple[int], None, Dict[str, int]]

Return the axes corresponding to the *obs* (or all if None).

Parameters

- **()** (*obs*) –
- **as_dict** (*bool*) – If True, returns a ordered dictionary with {obs: axis}
- **autofill** (*bool*) – If True and the axes are not specified, automatically fill them with the default numbering and return (not setting them).

Returns Tuple, OrderedDict

Raises

- ValueError – if the requested *obs* do not match with the one defined in the range
- AxesNotSpecifiedError – If the axes in this *Space* have not been specified.

get_obs_axes(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*)

get_reorder_indices(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None*) → Tuple[int]

Indices that would order *self.obs* as *obs* respectively *self.axes* as *axes*.

Parameters

- **()** (*axes*) –
- **()** –

Returns:

get_subspace(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, name: Optional[str] = None*) → zfit.Space

Create a *Space* consisting of only a subset of the *obs*/*axes* (only one allowed).

Parameters

- **obs** (*str, Tuple[str]*) –
- **axes** (*int, Tuple[int]*) –

- **()** (*name*) –

Returns:

is_generator

iter_areas(*rel: bool = False*) → Tuple[float, ...]
Return the areas of each interval

Parameters **rel** (*bool*) – If True, return the relative fraction of each interval

Returns

Return type Tuple[float]

iter_limits(*as_tuple: bool = True*) → Union[Tuple[zfit.Space], Tuple[Tuple[Tuple[float]]], Tuple[Tuple[float]]]
Return the limits, either as *Space* objects or as pure limits-tuple.

This makes iterating over limits easier: *for limit in space.iter_limits()* allows to, for example, pass *limit* to a function that can deal with simple limits only or if *as_tuple* is True the *limit* can be directly used to calculate something.

Example

```
for lower, upper in space.iter_limits(as_tuple=True):
    integrals = integrate(lower, upper)  # calculate integral
integral = sum(integrals)
```

Returns

Return type List[*Space*] or List[limit,...]

limit1d
return the tuple(lower, upper).

Returns so `lower, upper = space.limit1d` for a simple, 1 obs limit.

Return type tuple(float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified limits getter for 1 obs, 1 limit only

limit2d
return the tuple(low_obs1, low_obs2, up_obs1, up_obs2).

Returns

so *low_x, low_y, up_x, up_y = space.limit2d* for a single, 2 obs limit. low_x is the lower limit in x, up_x is the upper limit in x etc.

Return type tuple(float, float, float, float)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *limits* for exactly 2 obs, 1 limit

limits
Return the limits.

Returns:

limits1d
return the tuple(low_1, . . . , low_n, up_1, . . . , up_n).

Returns

so *low_1, low_2, up_1, up_2 = space.limits1d* for several, 1 obs limits. low_1 to up_1 is the first interval, low_2 to up_2 is the second interval etc.

Return type tuple(float, float, ..)

Raises `RuntimeError` – if the conditions (n_obs or n_limits) are not satisfied.

Type Simplified *.limits* for exactly 1 obs, n limits

lower
Return the lower limits.

Returns:

n_limits
The number of different limits.

Returns int >= 1

n_obs
Return the number of observables/axes.

Returns int >= 1

name
The name of the object.

obs
The observables ("axes with str")the space is defined in.

Returns:

obs_axes

reorder_by_indices(*indices: Tuple[int]*)
Return a *Space* reordered by the indices.

Parameters **()** (*indices*) –

upper
Return the upper limits.

Returns:

with_autofill_axes(*overwrite: bool = False*) → zfit.Space
Return a *Space* with filled axes corresponding to range(len(n_obs)).

Parameters **overwrite** (*bool*) – If *self.axes* is not None, replace the axes with the autofilled ones. If axes is already set, don't do anything if *overwrite* is False.

Returns *Space*

with_axes(*axes: Union[int, Iterable[int]]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*axes*) –

Returns *Space*

with_limits(*limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool], name: Optional[str] = None*) → zfit.Space
Return a copy of the space with the new *limits* (and the new *name*).

Parameters

- **()** (*limits*) –
- **name** (*str*) –

Returns *Space*

with_obs(*obs: Union[str, Iterable[str], zfit.Space]*) → zfit.Space
Sort by *obs* and return the new instance.

Parameters **()** (*obs*) –

Returns *Space*

with_obs_axes(*obs_axes: Union[OrderedDict[str, int], Dict[str, int]], ordered: bool = False, allow_subset=False*) → zfit.Space
Return a new *Space* with reordered observables and set the *axes*.

Parameters

- **obs_axes** (*OrderedDict[str, int]*) – An ordered dict with {obs: axes}.
- **ordered** (*bool*) – If True (and the *obs_axes* is an *OrderedDict*), the
- **()** (*allow_subset*) –

Returns

Return type *Space*

class zfit.core.sample.**UniformSampleAndWeights**
Bases: object

zfit.core.sample.**extended_sampling**(*pdfs: Union[Iterable[zfit.core.interfaces.ZfitPDF], zfit.core.interfaces.ZfitPDF], limits: zfit.core.limits.Space*) → tensorflow.python.framework.ops.Tensor
Create a sample from extended pdfs by sampling poissonian using the yield.

Parameters

- **pdfs** (*iterable[ZfitPDF]*) –
- **limits** (*Space*) –

Returns

Return type Union[tf.Tensor]

zfit.core.sample.**extract_extended_pdfs**(*pdfs: Union[Iterable[zfit.core.interfaces.ZfitPDF], zfit.core.interfaces.ZfitPDF]*) → List[zfit.core.interfaces.ZfitPDF]
Return all extended pdfs that are daughters.

Parameters **pdfs** (*Iterable[pdfs]*) –

Returns

Return type List[pdfs]

testing

Module for testing of the zfit components. Contains a singleton instance to register new PDFs and let them be tested.

zfit.core.testing.**setup_function**()

zfit.core.testing.**teardown_function**()

minimizers

Submodules

base_tf

class zfit.minimizers.base_tf.**WrapOptimizer**(*optimizer*, *tolerance=None*, *verbosity=None*, *name=None*, ***kwargs*)

Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss*, *params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)

Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

baseminimizer

Definition of minimizers, wrappers etc.

class zfit.minimizers.baseminimizer.**BaseMinimizer**(*name*, *tolerance*, *verbosity*, *minimizer_options*, *strategy=None*, ***kwargs*)

Bases: *zfit.minimizers.interface.ZfitMinimizer*

Minimizer for loss functions.

Additional *minimizer_options* (given as **kwargs) can be accessed and changed via the attribute (dict) *minimizer.minimizer_options*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

class zfit.minimizers.baseminimizer.**BaseStrategy**
Bases: *zfit.minimizers.baseminimizer.ZfitStrategy*

minimize_nan(*loss*, *params*, *minimizer*, *loss_value=None*, *gradient_values=None*)

class zfit.minimizers.baseminimizer.**DefaultStrategy**
Bases: *zfit.minimizers.baseminimizer.BaseStrategy*

minimize_nan(*loss*, *params*, *minimizer*, *loss_value=None*, *gradient_values=None*)

exception zfit.minimizers.baseminimizer.**FailMinimizeNaN**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class zfit.minimizers.baseminimizer.**ToyStrategyFail**
Bases: *zfit.minimizers.baseminimizer.BaseStrategy*

minimize_nan(*loss*, *params*, *minimizer*, *loss_value=None*, *gradient_values=None*)

class zfit.minimizers.baseminimizer.**ZfitStrategy**
Bases: abc.ABC

minimize_nan(*loss*, *loss_value*, *params*)

zfit.minimizers.baseminimizer.**print_gradients**(*params*, *values*, *gradients*, *loss=None*)

zfit.minimizers.baseminimizer.**print_params**(*params*, *values*, *loss=None*)

fitresult

class zfit.minimizers.fitresult.**FitResult**(*params: Dict[zfit.core.interfaces.ZfitParameter, float], edm: float, fmin: float, status: int, converged: bool, info: dict, loss: zfit.core.interfaces.ZfitLoss, minimizer: zfit.minimizers.interface.ZfitMinimizer*)
Bases: *zfit.minimizers.interface.ZfitResult*

Create a *FitResult* from a minimization. Store parameter values, minimization infos and calculate errors.

Any errors calculated are saved under *self.params* dictionary with {parameter: {error_name1: {'low': value 'high': value or similar}}

Parameters **params** (OrderedDict[*Parameter*, float]) – Result of the fit where each

:param *Parameter* key has the value: from the minimum found by the minimizer. :param edm: The estimated distance to minimum, estimated by the minimizer (if available) :type edm: Union[int, float] :param fmin: The minimum of the function found by the minimizer :type fmin: Union[numpy.float64, float] :param status: A status code (if available) :type status: int :param converged: Whether the fit has successfully converged or not. :type converged: bool :param info: Additional information (if available) like *number of function calls* and the

original minimizer return message.

Parameters

- **loss** (*Union[ZfitLoss]*) – The loss function that was minimized. Contains also the pdf, data etc.
- **minimizer** (*ZfitMinimizer*) – Minimizer that was used to obtain this *FitResult* and will be used to calculate certain errors. If the minimizer is state-based (like "iminuit"), then this is a copy and the state of other *FitResults* or of the *actual* minimizer that performed the minimization won't be altered.

converged

covariance(*params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*, *as_dict: bool = False*)

Calculate the covariance matrix for *params*.

Parameters

- **params** (list(*Parameter*)) – The parameters to calculate the covariance matrix. If *params* is *None*, use all *floating* parameters.
- **as_dict** (*bool*) – Default *False*. If *True* then returns a dictionnary.

Returns 2D *numpy.array* of shape (N, N); *dict'(param1, param2) -> covariance if 'as_dict == True*.

edm

The estimated distance to the minimum.

Returns numeric

error(*params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*, *method: Union[str, Callable] = 'minuit_minos'*, *error_name: str = None*, *sigma: float = 1.0*) → collections.OrderedDict

Calculate and set for *params* the asymmetric error using the set error method.

Parameters

- **params** (list(*Parameter* or str)) – The parameters or their names to calculate the errors. If *params* is *None*, use all *floating* parameters.
- **method** (*str or Callable*) – The method to use to calculate the errors. Valid choices are {'minuit_minos'} or a Callable.
- **sigma** (*float*) – Errors are calculated with respect to *sigma* std deviations. The definition of 1 sigma depends on the loss function and is defined there.

  For example, the negative log-likelihood (without the factor of 2) has a correspondents of Δ NLL of 1 corresponds to 1 std deviation.
- **error_name** (*str*) – The name for the error in the dictionary.

Returns

A *OrderedDict* containing as keys the parameter names and as value a *dict* which contains (next to probably more things) two keys 'lower' and 'upper', holding the calculated errors. Example: result['par1']['upper'] -> the asymmetric upper error of 'par1'

Return type *OrderedDict*

fmin
Function value at the minimum.

Returns numeric

hesse(*params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*, *method: Union[str, Callable] = 'minuit_hesse'*, *error_name: Optional[str] = None*, *sigma=1.0*) → collections.OrderedDict
Calculate for *params* the symmetric error using the Hessian matrix.

Parameters

- **params** (list(*Parameter*)) – The parameters to calculate the Hessian symmetric error. If None, use all parameters.
- **method** (*str*) – the method to calculate the hessian. Can be {'minuit'} or a callable.
- **error_name** (*str*) – The name for the error in the dictionary.

Returns

Result of the hessian (symmetric) error as dict with each parameter holding the error dict {'error': sym_error}.

So given param_a (from zfit.Parameter(.)) *error_a = result.hesse(params=param_a)[param_a]['error']* error_a is the hessian error.

Return type OrderedDict

info

loss

minimizer

params

status

zfit.minimizers.fitresult.**dict_to_matrix**(*params*, *matrix_dict*)

interface

class zfit.minimizers.interface.**ZfitMinimizer**
Bases: object

Define the minimizer interface.

minimize(*loss*, *params=None*)

step(*loss*, *params=None*)

tolerance

class zfit.minimizers.interface.**ZfitResult**
Bases: object

error(*params*, *method*, *sigma*)
Calculate and set for *params* the asymmetric error using the set error method.

Parameters

- **params** (list(*zfit.FitParameters* or str)) – The parameters or their names to calculate the errors. If *params* is *None*, use all *floating* parameters.
- **method** (*str or Callable*) – The method to use to calculate the errors. Valid choices are {'minuit_minos'} or a Callable.

Returns

A *OrderedDict* containing as keys the parameter names and as value a *dict* which contains (next to probably more things) two keys 'lower' and 'upper', holding the calculated errors. Example: result['par1']['upper'] -> the asymmetric upper error of 'par1'

Return type *OrderedDict*

fmin

hesse(*params*, *method*)
Calculate for *params* the symmetric error using the Hessian matrix.

Parameters

- **params** (list(*zfit.FitParameters*)) – The parameters to calculate the Hessian symmetric error. If None, use all parameters.
- **method** (*str*) – the method to calculate the hessian. Can be {'minuit'} or a callable.

Returns

Result of the hessian (symmetric) error as dict with each parameter holding the error dict {'error': sym_error}.

So given param_a (from zfit.Parameter(.)) *error_a = result.hesse(params=param_a)[param_a]['error']* error_a is the hessian error.

Return type OrderedDict

loss

minimizer

params

minimizer_minuit

class zfit.minimizers.minimizer_minuit.**Minuit**(*strategy: zfit.minimizers.baseminimizer.ZfitStrategy = None*, *minimize_strategy: int = 1*, *tolerance: float = None*, *verbosity: int = 5*, *name: str = None*, *ncall: int = 10000*, ***minimizer_options*)
Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*, *zfit.util.cache.Cachable*

Parameters

- **()** (*ncall*) – A ZfitStrategy object that defines the behavior of
- **minimizer in certain situations.** (*the*) –
- **()** – A number used by minuit to define the strategy

- **()** – Internal numerical tolerance
- **()** – Regulates how much will be printed during minimization. Values between 0 and 10 are valid.
- **()** – Name of the minimizer
- **()** – Maximum number of minimization steps.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy()

graph_caching_methods = []

minimize(*loss: zfit.core.interfaces.ZfitLoss, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)

Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

minimizer_tfp

class zfit.minimizers.minimizer_tfp.**BFGS**(*strategy: zfit.minimizers.baseminimizer.ZfitStrategy = None*, *tolerance: float = 1e-05*, *verbosity: int = 5*, *name: str = 'BFGS_TFP'*, *options: Mapping[KT, VT_co] = None*)

Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

Parameters

- **strategy** (ZfitStrategy) – Strategy that handles NaN and more (to come, experimental)
- **tolerance** (*float*) – Difference between the function value that suffices to stop minimization
- **verbosity** – The higher, the more is printed. Between 1 and 10 typically
- **name** – Name of the Minimizer
- **options** – A *dict* containing the options given to the minimization function, overriding the default

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)

Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

minimizers_scipy

class zfit.minimizers.minimizers_scipy.**Scipy**(*minimizer='L-BFGS-B'*, *tolerance=None*, *verbosity=5*, *name=None*, ***minimizer_options*)

Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

optimizers_tf

class zfit.minimizers.optimizers_tf.**Adam**(*tolerance=None, learning_rate=0.2, beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False, name='Adam', **kwargs*)
Bases: *zfit.minimizers.base_tf.WrapOptimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult
Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

tf_external_optimizer

TensorFlow interface for third-party optimizers.

class zfit.minimizers.tf_external_optimizer.**ExternalOptimizerInterface**(*loss*, *var_list=None*, *equalities=None*, *inequalities=None*, *var_to_bounds=None*, ***optimizer_kwargs*)

Bases: object

Base class for interfaces with external optimization algorithms.

Subclass this and implement *_minimize* in order to wrap a new optimization algorithm.

ExternalOptimizerInterface should not be instantiated directly; instead use e.g. *ScipyOptimizerInterface*.

Initialize a new interface instance.

Parameters

- **loss** – A scalar *Tensor* to be minimized.
- **var_list** – Optional *list* of *Variable* objects to update to minimize *loss*. Defaults to the list of variables collected in the graph under the key *GraphKeys.TRAINABLE_VARIABLES*.
- **equalities** – Optional *list* of equality constraint scalar ‘Tensor‘s to be held equal to zero.
- **inequalities** – Optional *list* of inequality constraint scalar ‘Tensor‘s to be held non-negative.
- **var_to_bounds** – Optional *dict* where each key is an optimization *Variable* and each corresponding value is a length-2 tuple of *(low, high)* bounds. Although enforcing this kind of simple constraint could be accomplished with the *inequalities* arg, not all optimization algorithms support general inequality constraints, e.g. L-BFGS-B. Both *low* and *high* can either be numbers or anything convertible to a NumPy array that can be broadcast to the shape of *var* (using *np.broadcast_to*). To indicate that there is no bound, use *None* (or *+/- np.infty*). For example, if *var* is a 2x3 matrix, then any of the following corresponding *bounds* could be supplied: * *(0, np.infty)*: Each element of *var* held positive. * *(-np.infty, [1, 2])*: First column less than 1, second column less than 2.
  - *(-np.infty, [[1], [2], [3]])*: First row less than 1, second row less than 2, etc.
  - *(-np.infty, [[1, 2, 3], [4, 5, 6]])*: Entry *var[0, 0]* less than 1, *var[0, 1]* less than 2, etc.
- ****optimizer_kwargs** – Other subclass-specific keyword arguments.

minimize(*feed_dict=None*, *fetches=None*, *step_callback=None*, *loss_callback=None*, ***run_kwargs*)

Minimize a scalar *Tensor*.

Variables subject to optimization are updated in-place at the end of optimization.

Note that this method does *not* just return a minimization *Op*, unlike *Optimizer.minimize()*; instead it actually performs minimization by executing commands to control a *Session*.

Parameters

- **session** – A *Session* instance.
- **feed_dict** – A feed dict to be passed to calls to *session.run*.
- **fetches** – A list of *Tensor's to fetch and supply to 'loss_callback* as positional arguments.
- **step_callback** – A function to be called at each optimization step; arguments are the current values of all optimization variables flattened into a single vector.
- **loss_callback** – A function to be called every time the loss and gradients are computed, with evaluated fetches supplied as positional arguments.
- ****run_kwargs** – kwargs to pass to *session.run*.

class zfit.minimizers.tf_external_optimizer.**ScipyOptimizerInterface**(*loss*, *var_list=None*, *equalities=None*, *inequalities=None*, *var_to_bounds=None*, ***optimizer_kwargs*)

Bases: *zfit.minimizers.tf_external_optimizer.ExternalOptimizerInterface*

Wrapper allowing *scipy.optimize.minimize* to operate a *tf.compat.v1.Session*.

Example:

‘‘‘python vector = tf.Variable([7., 7.], ‘vector’)

Make vector norm as small as possible. loss = tf.reduce_sum(tf.square(vector))

optimizer = ScipyOptimizerInterface(loss, options={‘maxiter’: 100})

with tf.compat.v1.Session() as session: optimizer.minimize(session)

The value of vector should now be [0., 0.]. ‘‘‘

Example with simple bound constraints:

‘‘‘python vector = tf.Variable([7., 7.], ‘vector’)

Make vector norm as small as possible. loss = tf.reduce_sum(tf.square(vector))

optimizer = ScipyOptimizerInterface(loss, var_to_bounds={vector: ([1, 2], np.infty)})

with tf.compat.v1.Session() as session: optimizer.minimize(session)

The value of vector should now be [1., 2.]. ‘‘‘

Example with more complicated constraints:

‘‘‘python vector = tf.Variable([7., 7.], ‘vector’)

Make vector norm as small as possible. loss = tf.reduce_sum(tf.square(vector)) # Ensure the vector's y component is = 1. equalities = [vector[1] - 1.] # Ensure the vector's x component is >= 1. inequalities = [vector[0] - 1.]

Our default SciPy optimization algorithm, L-BFGS-B, does not support # general constraints. Thus we use SLSQP instead. optimizer = ScipyOptimizerInterface(

loss, equalities=equalities, inequalities=inequalities, method='SLSQP')

with tf.compat.v1.Session() as session: optimizer.minimize(session)

The value of vector should now be [1., 1.]. ```

Initialize a new interface instance.

Parameters

- **loss** – A scalar *Tensor* to be minimized.
- **var_list** – Optional *list* of *Variable* objects to update to minimize *loss*. Defaults to the list of variables collected in the graph under the key *GraphKeys.TRAINABLE_VARIABLES*.
- **equalities** – Optional *list* of equality constraint scalar `Tensor`s to be held equal to zero.
- **inequalities** – Optional *list* of inequality constraint scalar `Tensor`s to be held non-negative.
- **var_to_bounds** – Optional *dict* where each key is an optimization *Variable* and each corresponding value is a length-2 tuple of *(low, high)* bounds. Although enforcing this kind of simple constraint could be accomplished with the *inequalities* arg, not all optimization algorithms support general inequality constraints, e.g. L-BFGS-B. Both *low* and *high* can either be numbers or anything convertible to a NumPy array that can be broadcast to the shape of *var* (using *np.broadcast_to*). To indicate that there is no bound, use *None* (or *+/- np.infty*). For example, if *var* is a 2x3 matrix, then any of the following corresponding *bounds* could be supplied: * *(0, np.infty)*: Each element of *var* held positive. * *(-np.infty, [1, 2])*: First column less than 1, second column less than 2.
  - *(-np.infty, [[1], [2], [3]])*: First row less than 1, second row less than 2, etc.
  - *(-np.infty, [[1, 2, 3], [4, 5, 6]])*: Entry *var[0, 0]* less than 1, *var[0, 1]* less than 2, etc.
- ****optimizer_kwargs** – Other subclass-specific keyword arguments.

minimize(*feed_dict=None*, *fetches=None*, *step_callback=None*, *loss_callback=None*, ***run_kwargs*)

Minimize a scalar *Tensor*.

Variables subject to optimization are updated in-place at the end of optimization.

Note that this method does *not* just return a minimization *Op*, unlike *Optimizer.minimize()*; instead it actually performs minimization by executing commands to control a *Session*.

Parameters

- **session** – A *Session* instance.
- **feed_dict** – A feed dict to be passed to calls to *session.run*.
- **fetches** – A list of *Tensor`s to fetch and supply to `loss_callback* as positional arguments.
- **step_callback** – A function to be called at each optimization step; arguments are the current values of all optimization variables flattened into a single vector.
- **loss_callback** – A function to be called every time the loss and gradients are computed, with evaluated fetches supplied as positional arguments.

- ****run_kwargs** – kwargs to pass to *session.run*.

models

Submodules

basefunctor

class zfit.models.basefunctor.**FunctorMixin**(*models*, *obs*, ***kwargs*)

Bases: *zfit.core.interfaces.ZfitFunctorMixin*, *zfit.core.basemodel.BaseModel*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]*, *norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None*, *name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None*, *axes: Union[int, Iterable[int]] = None*, *limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None*, *limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None*, *fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True*, *name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False*, *names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated

- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()
Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options (*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

basic

Basic PDFs are provided here. Gauss, exponential... that can be used together with Functors to build larger models.

class zfit.models.basic.**CustomGaussOLD** (*mu, sigma, obs, name='Gauss'*)
Bases: *zfit.core.basepdf.BasePDF*

add_cache_dependents (*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.

- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.basic.**Exponential**(*lambda_*, *obs: Union[str, Iterable[str], zfit.Space]*, *name: str = 'Exponential'*, ***kwargs*)

Bases: *zfit.core.basepdf.BasePDF*

Exponential function exp(lambda * x).

The function is normalized over a finite range and therefore a pdf. So the PDF is precisely defined as $\frac{e^{\lambda \cdot x}}{\int_{lower}^{upper} e^{\lambda \cdot x} dx}$

Parameters

- **lambda** (*Parameter*) – Accessed as parameter "lambda".
- **obs** (*Space*) – The *Space* the pdf is defined in.
- **name** (*str*) – Name of the pdf.
- **dtype** (*DType*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for

- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

dist_tfp

A rich selection of analytically implemented Distributions (models) are available in TensorFlow Probability. While their API is slightly different from the zfit models, it is similar enough to be easily wrapped.

Therefore a convenient wrapper as well as a lot of implementations are provided.

class zfit.models.dist_tfp.**ExponentialTFP**(*tau: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Exponential'*)
Bases: *zfit.models.dist_tfp.WrapDistribution*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func**(*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()**(*limits*) – **|limits_arg_descr|**
- **priority**(*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits**(*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range**(*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()**(*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()**(*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.dist_tfp.**Gauss**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Gauss'*)

Bases: *zfit.models.dist_tfp.WrapDistribution*

Gaussian or Normal distribution with a mean (mu) and a standartdevation (sigma).

The gaussian shape is defined as

$$f(x \mid \mu, \sigma^2) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

with the normalization over [-inf, inf] of

$$\frac{1}{\sqrt{2\pi\sigma^2}}$$

The normalization changes for different normalization ranges

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, `Parameter`) –

- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func**(*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()**(*limits*) – **|limits_arg_descr|**
- **priority**(*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits**(*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range**(*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()**(*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()**(*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.dist_tfp.**TruncatedGauss**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], low: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], high: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'TruncatedGauss'*)

Bases: *zfit.models.dist_tfp.WrapDistribution*

Gaussian distribution that is 0 outside of *low*, *high*. Equivalent to the product of Gauss and Uniform.

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **low** (*Parameter*) – Below this value, the pdf is zero.
- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func**(*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()**(*limits*) – **|limits_arg_descr|**
- **priority**(*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits**(*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range**(*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()**(*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()**(*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.dist_tfp.**Uniform**(*low: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], high: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Uniform'*)

Bases: *zfit.models.dist_tfp.WrapDistribution*

Uniform distribution which is constant between *low*, *high* and zero outside.

Parameters

- **low** (*Parameter*) – Below this value, the pdf is zero.
- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func (*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space (*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.dist_tfp.**WrapDistribution**(*distribution*, *dist_params*, *obs*, *params=None*, *dist_kwargs=None*, *dtype=tf.float64*, *name=None*, ***kwargs*)

Bases: *zfit.core.basepdf.BasePDF*

Baseclass to wrap tensorflow-probability distributions automatically.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

distribution

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.

- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

zfit.models.dist_tfp.**tfd_analytic_sample** (*n: int, dist: tensorflow_probability.python.distributions.distribution.Distribution, limits: Union[str, Iterable[str], zfit.Space]*)
Sample analytically with a *tfd.Distribution* within the limits. No preprocessing.

Parameters

- **n** – Number of samples to get
- **dist** – Distribution to sample from
- **limits** – Limits to sample from within

Returns The sampled data with the number of samples and the number of observables.

Return type *tf.Tensor* (n, n_obs)

functions

class zfit.models.functions.**BaseFunctorFunc** (*funcs*, *name='BaseFunctorFunc'*, *params=None*, ***kwargs*)
Bases: *zfit.models.basefunctor.FunctorMixin*, *zfit.core.basefunc.BaseFunc*

add_cache_dependents (*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functions.**ProdFunc**(*funcs: Iterable[zfit.core.interfaces.ZfitFunc], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumFunc', **kwargs*)
Bases: *zfit.models.functions.BaseFunctorFunc*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functions.**SimpleFunc**(*obs: Union[str, Iterable[str], zfit.Space], func: Callable, name: str = 'Function', **params*)
Bases: *zfit.core.basefunc.BaseFunc*

Create a simple function out of of *func* with the observables *obs* depending on *parameters*.

Parameters

- **func** (*function*) –

- **obs** (*Union[str, Tuple[str]]*) –
- **name** (*str*) –
- **()** (***params*) – The parameters as keyword arguments. E.g. *mu=Parameter(...)*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents (*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params (*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients (*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate (***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, `Space`) – the limits to integrate over
- **norm_range** (tuple, `Space`, False) – the limits to normalize over
- **name** (`str`) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate (***kwargs*)

partial_integrate (***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.

- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functions.**SumFunc**(*funcs: Iterable[zfit.core.interfaces.ZfitFunc], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumFunc', **kwargs*)

Bases: *zfit.models.functions.BaseFunctorFunc*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents (*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models (*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params (*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients (*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate (***kwargs*)

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate (***kwargs*)

partial_integrate (***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.

- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functions.**ZFunc**(*obs: Union[str, Iterable[str], zfit.Space], name: str = 'ZFunc', **params*)

Bases: *zfit.core.basemodel.SimpleModelSubclassMixin*, *zfit.core.basefunc.BaseFunc*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]

The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents (*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params (*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients (*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate (***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, `Space`) – the limits to integrate over
- **norm_range** (tuple, `Space`, False) – the limits to normalize over
- **name** (`str`) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate (***kwargs*)

partial_integrate (***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.

- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

functor

Functors are functions that take typically one or more other PDF. Prominent examples are a sum, convolution etc.

A FunctorBase class is provided to make handling the models easier.

Their implementation is often non-trivial.

class zfit.models.functor.**BaseFunctor**(*pdfs*, *name='BaseFunctor'*, ***kwargs*)
Bases: *zfit.models.basefunctor.FunctorMixin*, *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]
If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]

Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended

Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

pdfs_extended

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functor.**ProductPDF**(*pdfs: List[zfit.core.interfaces.ZfitPDF], obs: Union[str, Iterable[str], zfit.Space] = None, name='ProductPDF'*)

Bases: *zfit.models.functor.BaseFunctor*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.

- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a `Parameter` is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

pdfs_extended

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.

- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.functor.**SumPDF**(*pdfs: List[zfit.core.interfaces.ZfitPDF], fracs: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumPDF'*)
Bases: *zfit.models.functor.BaseFunctor*

Create the sum of the *pdfs* with *fracs* as coefficients.

Parameters

- **pdfs** (*pdf*) – The pdfs to add.
- **fracs** (*iterable*) – coefficients for the linear combination of the pdfs. If pdfs are extended, this throws an error.
  - len(frac) == len(basic) - 1 results in the interpretation of a non-extended pdf. The last coefficient will equal to 1 - sum(frac)
  - len(frac) == len(pdf) each pdf in *pdfs* will become an extended pdf with the given yield.
- **name** (*str*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –

- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –

- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If

fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.

- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- `NotExtendedPDFError` – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

fracs

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

pdfs_extended

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.

- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

physics

class zfit.models.physics.**CrystalBall**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alpha: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], n: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'CrystalBall', dtype: Type[CT_co] = tf.float64*)
Bases: *zfit.core.basepdf.BasePDF*

'Crystal Ball shaped PDF'__. A combination of a Gaussian with an powerlaw tail.

The function is defined as follows:

$$f(x;\mu,\sigma,\alpha,n) = \begin{cases} \exp(-\frac{(x-\mu)^2}{2\sigma^2}), & \text{for} \frac{x-\mu}{\sigma} \geqslant -\alpha A \cdot (B - \frac{x-\mu}{\sigma})^{-n}, \\ \text{for } \frac{x-\mu}{\sigma} < -\alpha \end{cases}$$

with

$$A = \left(\frac{n}{|\alpha|}\right)^n \cdot \exp\left(-\frac{|\alpha|^2}{2}\right)$$
$$B = \frac{n}{|\alpha|} - |\alpha|$$

Parameters

- **mu** (*zfit.Parameter*) – The mean of the gaussian
- **sigma** (*zfit.Parameter*) – Standard deviation of the gaussian
- **alpha** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail
- **n** (*zfit.Parameter*) – Exponent of the powertail
- **obs** (*Space*) –
- **name** (*str*) –
- **dtype** (*tf.DType*) –

__CBShape_

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over

- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()

Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.physics.**DoubleCB**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alphal: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], nl: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alphar: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], nr: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'DoubleCB', dtype: Type[CT_co] = tf.float64*)

Bases: *zfit.core.basepdf.BasePDF*

'Double sided Crystal Ball shaped PDF'__. A combination of two CB using the **mu** (not a frac). on each side.

The function is defined as follows:

$$f(x;\mu,\sigma,\alpha_L,n_L,\alpha_R,n_R) = \begin{cases} A_L \cdot (B_L - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} < -\alpha_L \\ \exp(-\frac{(x-\mu)^2}{2\sigma^2}), & -\alpha_L \leqslant \text{for} \frac{x-\mu}{\sigma} \leqslant \alpha_R \\ A_R \cdot (B_R - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} > \alpha_R \end{cases}$$

with

$$A_{L/R} = \left(\frac{n_{L/R}}{|\alpha_{L/R}|}\right)^n_{L/R} \cdot \exp\left(-\frac{|\alpha_{L/R}|^2}{2}\right)$$

$$B_{L/R} = \frac{n_{L/R}}{|\alpha_{L/R}|} - |\alpha_{L/R}|$$

Parameters

- **mu** (*zfit.Parameter*) – The mean of the gaussian
- **sigma** (*zfit.Parameter*) – Standard deviation of the gaussian
- **alphal** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the left
- **side** –
- **nl** (*zfit.Parameter*) – Exponent of the powertail on the left side
- **alphar** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the right
- **side** –
- **nr** (*zfit.Parameter*) – Exponent of the powertail on the right side
- **obs** (*Space*) –
- **name** (*str*) –
- **dtype** (*tf.DType*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.

- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a `Parameter` is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

zfit.models.physics.**crystalball_func**(*x*, *mu*, *sigma*, *alpha*, *n*)

zfit.models.physics.**crystalball_integral**(*limits*, *params*, *model*)

zfit.models.physics.**double_crystalball_func**(*x*, *mu*, *sigma*, *alphal*, *nl*, *alphar*, *nr*)

zfit.models.physics.**double_crystalball_integral**(*limits*, *params*, *model*)

polynomials

Recurrent polynomials.

class zfit.models.polynomials.**Chebyshev**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None*, *name: str = 'Chebyshev'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Chebyshev (first kind) polynomials of order len(coeffs), coeffs are scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$
$$with T_0 = 1 T_1 = x$$

Notice that T_1 is x as opposed to 2x in Chebyshev polynomials of the second kind.

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –

- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree

degree of the polynomial, starting from 0.

Type int

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.polynomials.**Chebyshev2**(*obs, coeffs: list, apply_scaling: bool = True, coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, name: str = 'Chebyshev2'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Chebyshev (second kind) polynomials of order len(coeffs), coeffs are scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$
$$with T_0 = 1 T_1 = 2x$$

Notice that T_1 is 2x as opposed to x in Chebyshev polynomials of the first kind.

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).

- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree

degree of the polynomial, starting from 0.

Type int

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.polynomials.**Hermite**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None*, *name: str = 'Hermite'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Hermite polynomials (for physics) of order len(coeffs), with coeffs as scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

with P_0 = 1 P_1 = 2x

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree

degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated

- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()
Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional tf.Tensor containing the unnormalized pdf.

Return type tf.Tensor

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.polynomials.**Laguerre**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None*, *name: str = 'Laguerre'*)
Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Laguerre polynomials of order len(coeffs), the coeffs are overall scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$(n+1)L_{n+1}(x) = (2n+1+lpha-x)L_n(x) - (n+lpha)L_{n-1}(x)$$

with P_0 = 1 P_1 = 1 - x

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.

- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.polynomials.**Legendre**(*obs: Union[str, Iterable[str], zfit.Space], coeffs: List[Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]], apply_scaling: bool = True, coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, name: str = 'Legendre'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Legendre polynomials of order len(coeffs), the coeffs are overall scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$
$$with P_0 = 1 P_1 = x$$

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.

- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.

- limits (*Space*): the limits to integrate over.
- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.polynomials.**RecursivePolynomial** (*obs, coeffs: list, apply_scaling: bool = True, coeff0: Optional[tensorflow.python.framework.ops.Tensor] = None, name: str = 'Polynomial'*)
Bases: *zfit.core.basepdf.BasePDF*

1D polynomial generated via three-term recurrence.

Base class to create 1 dimensional recursive polynomials that can be rescaled. Overwrite _poly_func.

Parameters

- **coeffs** (*list*) – Coefficients for each polynomial. Used to calculate the degree.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).

$$x_{n+1} = recurrence(x_n, x_{n-1}, n)$$

add_cache_dependents (*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –

- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –

- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If

fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.

- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- NotExtendedPDFError – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

zfit.models.polynomials.**chebyshev2_shape**(*x*, *coeffs*)

zfit.models.polynomials.**chebyshev_shape**(*x*, *coeffs*)

zfit.models.polynomials.**convert_coeffs_dict_to_list**(*coeffs: Mapping[KT, VT_co]*) → List[T]

zfit.models.polynomials.**create_poly**(*x*, *polys*, *coeffs*, *recurrence*)

zfit.models.polynomials.**do_recurrence**(*x*, *polys*, *degree*, *recurrence*)

zfit.models.polynomials.**func_integral_chebyshev1**(*limits*, *norm_range*, *params*, *model*)

zfit.models.polynomials.**func_integral_chebyshev2**(*limits*, *norm_range*, *params*, *model*)

zfit.models.polynomials.**func_integral_hermite**(*limits*, *norm_range*, *params*, *model*)

zfit.models.polynomials.**func_integral_laguerre**(*limits, norm_range, params: Dict[KT, VT], model*)

The integral of the simple laguerre polynomials.

Defined as $\int L_n = (-1)L_{n+1}^{(-1)}$ with $L^{(lpha)}$ the generalized Laguerre polynom.

Parameters

- **limits** –
- **norm_range** –
- **params** –
- **model** –

Returns:

zfit.models.polynomials.**generalized_laguerre_polys_factory**(*alpha=0.0*)

zfit.models.polynomials.**generalized_laguerre_recurrence_factory**(*alpha=0.0*)

zfit.models.polynomials.**generalized_laguerre_shape_factory**(*alpha=0.0*)

zfit.models.polynomials.**hermite_shape**(*x*, *coeffs*)

zfit.models.polynomials.**laguerre_shape**(*x*, *coeffs*)

zfit.models.polynomials.**laguerre_shape_alpha_minusone**(*x*, *coeffs*)

zfit.models.polynomials.**legendre_integral**(*limits: zfit.Space, norm_range: zfit.Space, params: List[zfit.Parameter], model: zfit.models.polynomials.RecursivePolynomial*)

Recursive integral of Legendre polynomials

zfit.models.polynomials.**legendre_shape**(*x*, *coeffs*)

zfit.models.polynomials.**rescale_minus_plus_one**(*x: tensorflow.python.framework.ops.Tensor*, *limits: zfit.Space*) → tensorflow.python.framework.ops.Tensor

Rescale and shift *x* as *limits* were rescaled and shifted to be in (-1, 1). Useful for orthogonal polynomials.

Parameters

- **x** – Array like data
- **limits** – 1-D limits

Returns the rescaled tensor.

Return type tf.Tensor

special

Special PDFs are provided in this module. One example is a normal function *Function* that allows to simply define a non-normalizable function.

class zfit.models.special.**SimpleFunctorPDF**(*obs*, *pdfs*, *func*, *name='SimpleFunctorPDF'*, ***params*)

Bases: *zfit.models.functor.BaseFunctor*, *zfit.models.special.SimplePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

pdfs_extended

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for

- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.special.**SimplePDF**(*obs*, *func*, *name='SimplePDF'*, ***params*)
Bases: *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]
If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func (*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space (*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.models.special.**ZPDF**(*obs: Union[str, Iterable[str], zfit.Space], name: str = 'ZPDF', **params*)

Bases: *zfit.core.basemodel.SimpleModelSubclassMixin*, *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.

- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – |limits_arg_descr|
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

zfit.models.special.**raise_error_if_norm_range**(*func*)

util

Submodules

cache

Module for caching.

The basic concept of caching in Zfit builds on a "cacher", that caches a certain value and that is dependent of "cache_dependents". By implementing *ZfitCachable*, an object will be able to play both roles. And most importantly, it has a *_cache* dict, that contains all the cache.

Basic principle

A "cacher" adds any dependents that it may comes across with *add_cache_dependents*. For example, for a loss this would be all pdfs and data. Since *Space* is immutable, there is no need to add this as a dependent. This leads to the "cache_dependent" to register the "cacher" and to remember it.

In case, any "cache_dependent" changes in a way the cache of itself (and any "cacher") is invalid, which is done in the simplest case by decorating a method with @*invalidates_cache*, the "cache_dependent":

- clears it's own cache with *reset_cache_self* and
- "clears" any "cacher"s cache with *reset_cache(reseter=self)*, telling the "cacher" that it should reset the cache. This is also where more fine-grained control (depending on which "cache_dependent" calls *reset_cache*) can be brought into play.

Example with a pdf that caches the normalization:

```
class Parameter(Cachable):
    def load(new_value):  # does not require to build a new graph
        # do something

    @invalidates_cache
    def change_limits(new_limits):  # requires to build a new graph (as an example)
        # do something

# create param1, param2 from `Parameter`

class MyPDF(Cachable):
    def __init__(self, param1, param2):
        self.add_cache_dependents([param1, param2])

    def cached_func(...):
        if self._cache.get('my_name') is None:
            result = ...  # calculations here
            self._cache['my_name']
        else:
            result = self._cache['my_name']
        return result
```

class zfit.util.cache.**Cachable**(**args*, ***kwargs*)

Bases: *zfit.util.cache.ZfitCachable*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

graph_caching_methods = []

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

class zfit.util.cache.**FunctionCacheHolder**(*func*, *wrapped_func*, *cachables: Union[zfit.util.cache.ZfitCachable, object, Iterable[Union[zfit.util.cache.ZfitCachable, object]]] = None*, *cachables_mapping=None*)

Bases: *zfit.util.cache.Cachable*

tf.function decorated function holder with caching dependencies on inputs.

A *tf.function* creates a new graph for every signature that is encountered. It automatically caches them but thereby assumes that Python objects are immutable. Any mutation won't be detected. Therefore, an extra wrapper is needed. The input signature is compared with firstly checking whether the function is the same and then doing an equal comparison of the arguments (maybe too costly?).

The *FunctionCacheHolder* holds the

- original python function which serves as the hash of the object
- wrapped python function, *wrapped_func*
- the (keyword-)arguments

If any of the keyword arguments changes in a way that the graph cache is invalid, this holder will have *is_valid* set to False and the *wrapped_func* cannot be used anymore, instead a new *tf.function* should be created as a call to the *wrapped_func* with the given arguments will result in an outdated graph.

Parameters

- **func** (*function*) – Python function that serves as a hash of the holder. Notice that equality is different defined.
- **wrapped_func** (*tf.function wrapped*) – Wrapped *func* with *tf.function*. The holder signals via *is_valid* whether this function is still valid to be used.
- **cachables** – objects that are cached. If they change, the cache is invalidated
- **cachables_mapping** (*Mapping*) – keyword arguments to the function. If the values change, the cache is invalidated.

IS_TENSOR = <object object>

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

create_immutable(*args*, *kwargs*)

Create a tuple of the args and kwargs by combining them as args + kwargs.keys() + kwargs.values()‘

Parameters

- **args** – list like
- **kwargs** – dict-like

Returns tuple

graph_caching_methods = []

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

class zfit.util.cache.**ZfitCachable**
Bases: object

add_cache_dependents(*cache_dependents*, *allow_non_cachable*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

register_cacher(*cacher: zfit.util.cache.ZfitCachable*)

reset_cache(*reseter*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

zfit.util.cache.**invalidates_cache**(*func*)

checks

class zfit.util.checks.**NotSpecified**
Bases: object

class zfit.util.checks.**ZfitNotImplemented**
Bases: object

container

class zfit.util.container.**DotDict**
Bases: dict

dot.notation access to dictionary attributes

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

zfit.util.container.**convert_to_container**(*value: Any*, *container: Callable = <class 'list'>*, *non_containers=None*, *convert_none=False*) → Union[None, Iterable[T_co]]

Convert *value* into a *container* storing *value* if *value* is not yet a python container.

Parameters

- **value** (*object*) –
- **container** (*callable*) – Converts a tuple to a container.
- **non_containers** (*Optional[List[Container]]*) – Types that do not count as a container. Has to be a list of types. As an example, if *non_containers* is [list, tuple] and the value is [5, 3] (-> a list with two entries),this won't be converted to the *container* but end up as (if the container is e.g. a tuple): ([5, 3],) (a tuple with one entry).

Returns:

zfit.util.container.**is_container**(*obj*)

Check if *object* is a list or a tuple.

Parameters **()** (*obj*) –

Returns True if it is a *container*, otherwise False

Return type bool

diverse

class zfit.util.diverse.**GaussianMixture2D**(*prefix*, *n*, *x_range*, *y_range*)

Bases: object

model(*x*)

class zfit.util.diverse.**GaussianMixture4D**(*prefix*, *n*, *ranges*)

Bases: object

model(*x*)

zfit.util.diverse.**gauss_2d**(*x*, *norm*, *xmean*, *ymean*, *xsigma*, *ysigma*, *corr*)

zfit.util.diverse.**gauss_4d**(*x*, *params*)

zfit.util.diverse.**multivariate_gauss**(*x*, *norm*, *mean*, *inv_cov*)

exception

exception zfit.util.exception.**AlreadyExtendedPDFError**
Bases: *zfit.util.exception.ExtendedPDFError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**AxesNotSpecifiedError**
Bases: *zfit.util.exception.NotSpecifiedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**AxesNotUnambiguousError**
Bases: *zfit.util.exception.IntentionNotUnambiguousError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**BasePDFSubclassingError**
Bases: *zfit.util.exception.SubclassingError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**BreakingAPIChangeError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ConversionError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ExtendedPDFError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**IncompatibleError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**IntentionNotUnambiguousError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**LimitsIncompatibleError**
Bases: *zfit.util.exception.IncompatibleError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**LimitsNotSpecifiedError**
Bases: *zfit.util.exception.NotSpecifiedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**LimitsOverdefinedError**
Bases: *zfit.util.exception.OverdefinedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**LimitsUnderdefinedError**
Bases: *zfit.util.exception.UnderdefinedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**LogicalUndefinedOperationError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ModelIncompatibleError**
Bases: *zfit.util.exception.IncompatibleError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**MultipleLimitsNotImplementedError**
Bases: Exception

Indicates that a function does not support several limits in a *Space*.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NameAlreadyTakenError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NoSessionSpecifiedError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NormRangeNotImplementedError**
Bases: Exception

Indicates that a function does not support the normalization range argument *norm_range*.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NormRangeNotSpecifiedError**
Bases: *zfit.util.exception.NotSpecifiedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NotExtendedPDFError**
Bases: *zfit.util.exception.ExtendedPDFError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NotMinimizedError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**NotSpecifiedError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ObsIncompatibleError**
Bases: *zfit.util.exception.IncompatibleError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ObsNotSpecifiedError**
Bases: *zfit.util.exception.NotSpecifiedError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**OverdefinedError**
Bases: *zfit.util.exception.IntentionNotUnambiguousError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**PDFCompatibilityError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**ShapeIncompatibleError**
Bases: *zfit.util.exception.IncompatibleError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**SpaceIncompatibleError**
Bases: *zfit.util.exception.IncompatibleError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**SubclassingError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**UnderdefinedError**
Bases: *zfit.util.exception.IntentionNotUnambiguousError*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**WeightsNotImplementedError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.**WorkInProgressError**
Bases: Exception

Only for developing purpose! Does not serve as a 'real' Exception.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

execution

class zfit.util.execution.**RunManager**(*n_cpu='auto'*)
Bases: object

Handle the resources and runtime specific options. The *run* method is equivalent to *sess.run*

aquire_cpu(*max_cpu: int = -1*) → List[str]

chunksize

n_cpu

set_cpus_explicit(*intra: int*, *inter: int*) → None
Set the number of threads (cpus) used for inter-op and intra-op parallelism

Parameters

- **intra** – Number of threads used to perform an operation. For larger operations, e.g. large Tensors, this is usually beneficial to have >= 2.
- **inter** – Parallelization on the level of ops. This is beneficial, if many operations can be computed independently in parallel.

set_n_cpu(*n_cpu: Union[str, int] = 'auto'*, *strict: bool = False*) → None
Set the number of cpus to be used by zfit. For more control, use *set_cpus_explicit*.

Parameters

- **n_cpu** – Number of cpus, will be the number for inter-op parallelism
- **strict** – If strict, sets intra parallelism to 1

graph

zfit.util.graph.**all_parents**(*op*, *current_obs=None*)

zfit.util.graph.**get_dependents_auto**(*tensor: tensorflow.python.framework.ops.Tensor*, *candidates: List[tensorflow.python.framework.ops.Tensor]*) → List[tensorflow.python.framework.ops.Tensor]
Return the nodes in *candidates* that *tensor* depends on.

Parameters

- **()** (*candidates*) –
- **()** –

logging

This module controls the zfit logging.

The base logger for zfit is called *zfit*, and all loggers created by this module have the form *zfit.XX*, where *XX* is their name.

By default, time, name of the logger and message with the default colorlog color scheme are printed.

zfit.util.logging.**get_logger**(*name*, *stdout_level=None*, *file_level=None*, *file_name=None*)

Get and configure logger.

This logger has two handlers:

- A stdout handler is always configured with *colorlog*.
- A file handler is configured if *file_name* is given. Once it has been configure, it is not

necessary to give it to modify its properties.

Once the logger has been created, *get_logger* can be called again to modify its log levels, independently for the stream and file handlers.

Note: If the logger name doesn't start with "zfit", it is automatically added.

Note: Default logging level at first instantiation is WARNING.

Parameters

- **name** (*str*) – Name of the logger.
- **stdout_level** (*int, optional*) – Logging level for the stream handler. Defaults to *logging.WARNING*.
- **file_level** (*int, optional*) – Logging level for the file handler. Defaults to *logging.WARNING*.
- **file_name** (*str, optional*) – File to log to. If not given, no file logging is performed.

Returns The requested logger.

Return type *logging.Logger*

Raise: ValueError if *file_level* has been specified without having configured the output file.

temporary

class zfit.util.temporary.**TemporarilySet**(*value: Any*, *setter: Callable*, *getter: Callable*, *setter_args=None*, *setter_kwargs=None*, *getter_args=None*, *getter_kwargs=None*)

Bases: object

Temporarily set *value* with *setter* and reset to the old value after leaving the context.

This class can be used to have a setter that can permanently set a value *as well as* just for the time inside a context manager. The usage is as follows:

```
>>> class SimpleX:
>>>     def __init__(self):
>>>          self.x = None
>>>     def _set_x(self, x):
>>>         self.x = x
>>>     def get_x(self):
>>>         return self.x
>>>     def set_x(self, x):
>>>         return TemporarilySet(value=x, setter=self._set_x,␣
↪getter=self.get_x)
```

```
>>> simple_x = SimpleX()
Now we can either set x permanently
>>> simple_x.set_x(42)
>>> print(simple.x)
42
```

or temporarily >>> with simple_x.set_x(13) as value: >>> print("Value from contextmanager:", value) >>> print("simple_x.get_x():", simple_x.get_x()) 13 13

and is afterwards unset again >>> print(simple.x) 42

Parameters

- **value** (*Any*) – The value to be (temporarily) set (and returned if a context manager is applied).
- **setter** (*Callable*) – The setter function with a signature that is compatible to the call: *setter(value, *setter_args, **setter_kwargs)*
- **getter** (*Callable*) – The getter function with a signature that is compatible to the call: *getter(*getter_args, **getter_kwargs)*
- **setter_args** (*List*) – A list of arguments given to the setter
- **setter_kwargs** (*Dict*) – A dict of keyword-arguments given to the setter
- **getter_args** (*List*) – A list of arguments given to the getter
- **getter_kwargs** (*Dict*) – A dict of keyword-arguments given to the getter

ztyping

z

z is a zfit TensorFlow version, that wraps TF while adding some conveniences, basically using a different default dtype (*zfit.ztypes*). In addition, it expands TensorFlow by adding a few convenient functions helping to deal with 'NaN's and similar.

Some function are already wrapped, others are not. Best practice is to use *z* whenever possible and *tf* for the rest.

Submodules

const

math

zfit.z.math.**autodiff_gradient**(*func: Callable, params: Iterable[zfit.Parameter]*) → tensorflow.python.framework.ops.Tensor

Calculate using autodiff the gradients of *func()* wrt *params*.

Automatic differentiation (autodiff) is a way of retreiving the derivative of x wrt y. It works by consecutively applying the chain rule. All that is needed is that every operation knows its own derivative. TensorFlow implements this and anything using *tf.** operations only can use this technique.

Args: func (Callable): Function without arguments that depends on *params* params (ZfitParameter): Parameters that *func* implicitly depends on and with respect to which the

derivatives will be taken.

Returns: *tf.Tensor*: gradient

zfit.z.math.**autodiff_hessian**(*func: Callable, params: Iterable[zfit.Parameter]*) → tensorflow.python.framework.ops.Tensor

Calculate using autodiff the hessian matrix of *func()* wrt *params*.

Automatic differentiation (autodiff) is a way of retrieving the derivative of x wrt y. It works by consecutively applying the chain rule. All that is needed is that every operation knows its own derivative. TensorFlow implements this and anything using *tf.** operations only can use this technique.

Args: func (Callable): Function without arguments that depends on *params* params (ZfitParameter): Parameters that *func* implicitly depends on and with respect to which the

derivatives will be taken.

Returns: *tf.Tensor*: hessian matrix

zfit.z.math.**autodiff_value_gradients**(*func: Callable, params: Iterable[zfit.Parameter]*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Calculate using autodiff the gradients of *func()* wrt *params*; also return *func()*.

Automatic differentiation (autodiff) is a way of retreiving the derivative of x wrt y. It works by consecutively applying the chain rule. All that is needed is that every operation knows its own derivative. TensorFlow implements this and anything using *tf.** operations only can use this technique.

Args: func (Callable): Function without arguments that depends on *params* params (ZfitParameter): Parameters that *func* implicitly depends on and with respect to which the

derivatives will be taken.

Returns: tuple(*tf.Tensor*, *tf.Tensor*): value and gradient

zfit.z.math.**automatic_value_gradients_hessian**(*func: Callable, params: Iterable[zfit.Parameter]*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Calculate using autodiff the gradients and hessian matrix of *func()* wrt *params*; also return *func()*.

Automatic differentiation (autodiff) is a way of retreiving the derivative of x wrt y. It works by consecutively applying the chain rule. All that is needed is that every operation knows its own derivative. TensorFlow implements this and anything using *tf.** operations only can use this technique.

Args: func (Callable): Function without arguments that depends on *params* params (ZfitParameter): Parameters that *func* implicitly depends on and with respect to which the

derivatives will be taken.

Returns: tuple(*tf.Tensor*, *tf.Tensor*, *tf.Tensor*): value, gradient and hessian matrix

zfit.z.math.**interpolate**(*t*, *c*)

Multilinear interpolation on a rectangular grid of arbitrary number of dimensions.

Parameters

- **t** (*tf.Tensor*) – Grid (of rank N)
- **c** (*tf.Tensor*) – Tensor of coordinates for which the interpolation is performed

Returns 1D tensor of interpolated value

Return type tf.Tensor

zfit.z.math.**numerical_gradient**(*func: Callable, params: Iterable[zfit.Parameter]*) → tensorflow.python.framework.ops.Tensor

Calculate numerically the gradients of func() with respect to *params*.

Parameters

- **func** (*Callable*) – Function without arguments that depends on *params*
- **params** (ZfitParameter) – Parameters that *func* implicitly depends on and with respect to which the derivatives will be taken.

Returns gradients

Return type *tf.Tensor*

zfit.z.math.**numerical_hessian**(*func: Callable, params: Iterable[zfit.Parameter]*) → tensorflow.python.framework.ops.Tensor

Calculate numerically the hessian matrix of func() with respect to *params*.

Parameters

- **func** (*Callable*) – Function without arguments that depends on *params*
- **params** (ZfitParameter) – Parameters that *func* implicitly depends on and with respect to which the derivatives will be taken.

Returns hessian matrix

Return type *tf.Tensor*

zfit.z.math.**numerical_value_gradients**(*func: Callable, params: Iterable[zfit.Parameter]*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Calculate numerically the gradients of *func()* with respect to *params*, also returns the value of *func()*.

Parameters

- **func** (*Callable*) – Function without arguments that depends on *params*
- **params** (ZfitParameter) – Parameters that *func* implicitly depends on and with respect to which the derivatives will be taken.

Returns value, gradient

Return type tuple(*tf.Tensor*, *tf.Tensor*)

zfit.z.math.**numerical_value_gradients_hessian**(*func: Callable, params: Iterable[zfit.Parameter]*) → [<class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>, <class 'tensorflow.python.framework.ops.Tensor'>]

Calculate numerically the gradients and hessian matrix of *func()* wrt *params*; also return *func()*.

Parameters

- **func** (*Callable*) – Function without arguments that depends on *params*
- **params** (ZfitParameter) – Parameters that *func* implicitly depends on and with respect to which the derivatives will be taken.

Returns value, gradient and hessian matrix

Return type tuple(*tf.Tensor*, *tf.Tensor*, *tf.Tensor*)

zfit.z.math.**poly_complex**(**args*, *real_x=False*)

Complex polynomial with the last arg being x.

Parameters

- ***args** (*tf.Tensor or equ.*) – Coefficients of the polynomial
- **real_x** (*bool*) – If True, x is assumed to be real.

Returns

Return type tf.Tensor

random

zfit.z.random.**counts_multinomial**(*total_count: Union[int, tensorflow.python.framework.ops.Tensor], probs: Iterable[Union[float, tensorflow.python.framework.ops.Tensor]] = None, logits: Iterable[Union[float, tensorflow.python.framework.ops.Tensor]] = None, dtype=tf.int32*) → tensorflow.python.framework.ops.Tensor

Get the number of counts for different classes with given probs/logits.

Parameters

- **total_count** (*int*) – The total number of draws.
- **probs** – Length k (number of classes) object where the k-1th entry contains the probability to get a single draw from the class k. Have to be from [0, 1] and sum up to 1.
- **logits** – Same as probs but from [-inf, inf] (will be transformet to [0, 1])

Returns py:class.‘tf.Tensor‘: shape (k,) tensor containing the number of draws.

tools

wrapping_tf

zfit.z.wrapping_tf.**check_numerics**(*tensor: Any*, *message: Any*, *name: Any = None*)

Check whether a tensor is finite and not NaN. Extends TF by accepting complex types as well.

Parameters

- **(** (*tensor*) – py:class:~'tensorflow.python.framework.ops.Tensor'):
- **message** (*str*) –
- **name** (*Union[None, None, None]*) –

Returns

Return type tensorflow.python.framework.ops.Tensor

zfit.z.wrapping_tf.**complex**(*real*, *imag*, *name=None*)

zfit.z.wrapping_tf.**exp**(*x*, *name=None*)

zfit.z.wrapping_tf.**log**(*x*, *name=None*)

zfit.z.wrapping_tf.**pow**(*x*, *y*, *name=None*)

zfit.z.wrapping_tf.**random_normal**(*shape*, *mean=0.0*, *stddev=1.0*, *dtype=tf.float64*, *seed=None*, *name=None*)

zfit.z.wrapping_tf.**random_poisson**(*lam: Any*, *shape: Any*, *dtype: tensorflow.python.framework.dtypes.DType = tf.float64*, *seed: Any = None*, *name: Any = None*)

zfit.z.wrapping_tf.**random_uniform**(*shape*, *minval=0*, *maxval=None*, *dtype=tf.float64*, *seed=None*, *name=None*)

zfit.z.wrapping_tf.**sqrt**(*x*, *name=None*)

zfit.z.wrapping_tf.**square**(*x*, *name=None*)

zextension

class zfit.z.zextension.**FunctionWrapperRegistry**(***kwargs_user*)

Bases: object

tf.function-like decorator with additional cache-invalidation functionality.

Parameters ****kwargs_user** – arguments to *tf.function*

classmethod check_wrapped_functions_registered()

registries = [<zfit.z.zextension.FunctionWrapperRegistry object>]

reset(***kwargs_user*)

wrapped_functions = []

zfit.z.zextension.**abs_square**(*x*)

zfit.z.zextension.**constant**(*value*, *dtype=tf.float64*, *shape=None*, *name='Const'*, *verify_shape=None*)

zfit.z.zextension.**convert_to_tensor**(*value*, *dtype=None*, *name=None*, *preferred_dtype=None*)

zfit.z.zextension.**nth_pow**(*x*, *n*, *name=None*)

Calculate the nth power of the complex Tensor x.

Parameters

- **x** (*tf.Tensor, complex*) –
- **n** (*int >= 0*) – Power

- **name** (*str*) – No effect, for API compatibility with tf.pow

zfit.z.zextension.**run_no_nan**(*func*, *x*)

zfit.z.zextension.**safe_where**(*condition: tensorflow.python.framework.ops.Tensor*, *func: Callable*, *safe_func: Callable*, *values: tensorflow.python.framework.ops.Tensor*, *value_safer: Callable = <function ones_like_v2>*) → tensorflow.python.framework.ops.Tensor

Like `tf.where()` but fixes gradient *NaN* if func produces *NaN* with certain *values*.

Parameters

- **condition** (`tf.Tensor`) – Same argument as to `tf.where()`, a boolean `tf.Tensor`
- **func** (*Callable*) – Function taking *values* as argument and returning the tensor _in case condition is **True**_. Equivalent *x* of `tf.where()` but as function.
- **safe_func** (*Callable*) – Function taking *values* as argument and returning the tensor _in case the condition is **False**_, Equivalent *y* of `tf.where()` but as function.
- **values** (`tf.Tensor`) – Values to be evaluated either by *func* or *safe_func* depending on *condition*.
- **value_safer** (*Callable*) – Function taking *values* as arguments and returns "safe" values that won't cause troubles when given to'func' or by taking the gradient with respect to *func(value_safer(values))*.

Returns

Return type `tf.Tensor`

zfit.z.zextension.**stack_x**(*values*, *axis: int = -1*, *name: str = 'stack_x'*)

zfit.z.zextension.**to_complex**(*number*, *dtype=tf.complex128*)

zfit.z.zextension.**to_real**(*x*, *dtype=tf.float64*)

zfit.z.zextension.**unstack_x**(*value: Any*, *num: Any = None*, *axis: int = -1*, *always_list: bool = False*, *name: str = 'unstack_x'*)

Unstack a Data object and return a list of (or a single) tensors in the right order.

Parameters

- **()** (*value*) –
- **num** (*Union[]*) –
- **axis** (*int*) –
- **always_list** (*bool*) – If True, also return a list if only one element.
- **name** (*str*) –

Returns

Return type Union[List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor, None]

4.1.2 Submodules

constraint

zfit.constraint.**nll_gaussian**(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], mu: Union[int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*) → tensorflow.python.framework.ops.Tensor

Return negative log likelihood graph for gaussian constraints on a list of parameters.

Parameters

- **params** (*list(zfit.Parameter)*) – The parameters to constraint
- **mu** (*numerical, list(numerical)*) – The central value of the constraint
- **sigma** (*numerical, list(numerical) or array/tensor*) – The standard deviations or covariance matrix of the constraint. Can either be a single value, a list of values, an array or a tensor

Returns the constraint object

Return type *GaussianConstraint*

Raises ShapeIncompatibleError – if params, mu and sigma don't have the same size

class zfit.constraint.**SimpleConstraint**(*func: Callable, params: Optional[Dict[str, zfit.core.interfaces.ZfitParameter]], sampler: Callable = None*)

Bases: *zfit.core.constraint.BaseConstraint*

Constraint from a (function returning a) Tensor.

The parameters are named "param_{i}" with i starting from 0 and corresponding to the index of params.

Parameters

- **func** – Callable that constructs the constraint and returns a tensor.
- **dependents** – The dependents (independent *zfit.Parameter*) of the loss. If not given, the dependents are figured out automatically.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name
The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n*)
Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

class zfit.constraint.**GaussianConstraint**(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], mu: Union[int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)
Bases: *zfit.core.constraint.DistributionConstraint*

Gaussian constraints on a list of parameters.

Parameters

- **params** (*list(zfit.Parameter)*) – The parameters to constraint
- **mu** (*numerical, list(numerical)*) – The central value of the constraint

- **sigma** (*numerical, list(numerical) or array/tensor*) – The standard deviations or covariance matrix of the constraint. Can either be a single value, a list of values, an array or a tensor

Raises ShapeIncompatibleError – if params, mu and sigma don't have incompatible shapes

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

name

The name of the object.

old_graph_caching_methods = []

params

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n*)
Sample *n* points from the probability density function for the constrained parameters.

Parameters **n** (*int, tf.Tensor*) – The number of samples to be generated.

Returns n_samples)

Return type Dict(*Parameter*

value()

data

class zfit.data.**Data**(*dataset: Union[tensorflow.python.data.ops.dataset_ops.DatasetV2, LightDataset], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = None, weights=None, iterator_feed_dict: Dict[KT, VT] = None, dtype: tensorflow.python.framework.dtypes.DType = None*)
Bases: *zfit.util.cache.Cachable*, *zfit.core.interfaces.ZfitData*, *zfit.core.dimension.BaseDimensional*, *zfit.core.baseobject.BaseObject*

Create a data holder from a *dataset* used to feed into *models*.

Parameters

- **()** (*dtype*) – A dataset storing the actual values
- **()** – Observables where the data is defined in
- **()** – Name of the *Data*
- **()** –
- **()** –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –

- **()** –
- **()** –

Returns:

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data_range

dtype

classmethod from_numpy(*obs: Union[str, Iterable[str], zfit.Space], array: numpy.ndarray, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create *Data* from a *np.array*.

Parameters

- **()** (*obs*) –
- **array** (*numpy.ndarray*) –
- **name** (*str*) –

Returns

Return type zfit.Data

classmethod from_pandas(*df: pandas.core.frame.DataFrame, obs: Union[str, Iterable[str], zfit.Space] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*)

Create a *Data* from a pandas DataFrame. If *obs* is *None*, columns are used as obs.

Parameters

- **df** (*pandas.DataFrame*) –
- **weights** (*tf.Tensor, None, np.ndarray, str]*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents).
- **obs** (*zfit.Space*) –
- **name** (*str*) –

classmethod from_root(*path: str, treepath: str, branches: List[str] = None, branches_alias: Dict[KT, VT] = None, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray, str] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None, root_dir_options=None*) → zfit.core.data.Data

Create a *Data* from a ROOT file. Arguments are passed to *uproot*.

Parameters

- **path** (*str*) –
- **treepath** (*str*) –
- **branches** (*List[str]]*) –
- **branches_alias** (*dict*) – A mapping from the *branches* (as keys) to the actual *observables* (as values). This allows to have different *observable* names, independent of the branch name in the file.

- **weights** (*tf.Tensor, None, np.ndarray, str*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents). Can be a column of the ROOT file by using a string corresponding to a column.
- **name** (*str*) –
- **()** (*root_dir_options*) –

Returns

Return type *zfit.Data*

classmethod from_root_iter(*path*, *treepath*, *branches=None*, *entrysteps=None*, *name=None*, ***kwargs*)

classmethod from_tensor(*obs: Union[str, Iterable[str], zfit.Space], tensor: tensorflow.python.framework.ops.Tensor, weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray] = None, name: str = None, dtype: tensorflow.python.framework.dtypes.DType = None*) → zfit.core.data.Data

Create a *Data* from a *tf.Tensor*. *Value* simply returns the tensor (in the right order).

Parameters

- **obs** (*Union[str, List[str]*) –
- **tensor** (*tf.Tensor*) –
- **name** (*str*) –

Returns

Return type zfit.core.Data

get_iteration()

graph_caching_methods = []

initialize()

iterator

n_obs

Return the number of observables.

name

The name of the object.

nevents

numpy()

obs

Return the observables.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

set_data_range(*data_range*)

set_weights(*weights: Union[tensorflow.python.framework.ops.Tensor, None, numpy.ndarray]*)
Set (temporarily) the weights of the dataset.

Parameters **weights** (*tf.Tensor*, np.ndarray, None) –

sort_by_axes(*axes: Union[int, Iterable[int]], allow_superset: bool = False*)

sort_by_obs(*obs: Union[str, Iterable[str], zfit.Space], allow_superset: bool = False*)

space
Return the *Space* object that defines the dimensionality of the object.

to_pandas(*obs: Union[str, Iterable[str], zfit.Space] = None*)
Create a *pd.DataFrame* from *obs* as columns and return it.

Parameters **()** (*obs*) – The observables to use as columns. If *None*, all observables are used.

Returns:

unstack_x(*obs: Union[str, Iterable[str], zfit.Space] = None, always_list: bool = False*)
Return the unstacked data: a list of tensors or a single Tensor.

Parameters

- **()** (*obs*) – which observables to return
- **always_list** (*bool*) – If True, always return a list (also if length 1)

Returns List(tf.Tensor)

value(*obs: Union[str, Iterable[str], zfit.Space] = None*)

weights

func

class zfit.func.**BaseFunc**(*obs=None, dtype: Type[CT_co] = tf.float64, name: str = 'BaseFunc', params: Any = None*)
Bases: *zfit.core.basemodel.BaseModel*, *zfit.core.interfaces.ZfitFunc*

TODO(docs): explain subclassing

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a Sampler that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.

- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]
The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.func.**ProdFunc**(*funcs: Iterable[zfit.core.interfaces.ZfitFunc], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumFunc', **kwargs*)
Bases: *zfit.models.functions.BaseFunctorFunc*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF
Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]
The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.func.**SumFunc**(*funcs: Iterable[zfit.core.interfaces.ZfitFunc], obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumFunc', **kwargs*)
Bases: *zfit.models.functions.BaseFunctorFunc*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF
Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]
The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.func.**SimpleFunc**(*obs: Union[str, Iterable[str], zfit.Space], func: Callable, name: str = 'Function', **params*)
Bases: *zfit.core.basefunc.BaseFunc*

Create a simple function out of of *func* with the observables *obs* depending on *parameters*.

Parameters

- **func** (*function*) –
- **obs** (*Union[str, Tuple[str]]*) –
- **name** (*str*) –
- **()** (***params*) – The parameters as keyword arguments. E.g. *mu=Parameter(...)*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

as_pdf() → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

Returns a PDF with the current function as the unnormalized probability.

Return type *ZfitPDF*

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_params*)

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a Sampler that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

func(*x: Union[float, tensorflow.python.framework.ops.Tensor], name: str = 'value'*) → Union[float, tensorflow.python.framework.ops.Tensor]
The function evaluated at *x*.

Parameters

- **x** (*Data*) –
- **name** (*str*) –

Returns # TODO(Mayou36): or dataset? Update: rather not, what would obs be?

Return type tf.Tensor

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

n_obs
Return the number of observables.

name
The name of the object.

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()

Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

space
Return the *Space* object that defines the dimensionality of the object.

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

loss

class zfit.loss.**ExtendedUnbinnedNLL**(*model*, *data*, *fit_range=<zfit.util.checks.NotSpecified object>*, *constraints=None*)
Bases: *zfit.core.loss.UnbinnedNLL*

An Unbinned Negative Log Likelihood with an additional poisson term for the

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name
Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (`cacher`) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.loss.**UnbinnedNLL**(*model*, *data*, *fit_range=<zfit.util.checks.NotSpecified object>*, *constraints=None*)
Bases: `zfit.core.loss.BaseLoss`

The Unbinned Negative Log Likelihood.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (`bool`) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name

Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (`cacher`) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.loss.**BaseLoss**(*model: Union[zfit.core.interfaces.ZfitModel, Iterable[zfit.core.interfaces.ZfitModel]], data: Union[zfit.Data, Iterable[zfit.Data]], fit_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = <zfit.util.checks.NotSpecified object>, constraints: Iterable[Union[zfit.core.interfaces.ZfitConstraint, Callable]] = None*)

Bases: `zfit.core.dependents.BaseDependentsMixin`, `zfit.core.interfaces.ZfitLoss`, `zfit.util.cache.Cachable`, `zfit.core.baseobject.BaseObject`

A "simultaneous fit" can be performed by giving one or more *model*, *data*, *fit_range* to the loss. The length of each has to match the length of the others.

Parameters

- **model** (`Iterable[ZfitModel]`) – The model or models to evaluate the data on
- **data** (`Iterable[ZfitData]`) – Data to use
- **fit_range** (Iterable[`Space`]) – The fitting range. It's the norm_range for the models (if
- **they** – have a norm_range) and the data_range for the data.

- **constraints** (*Iterable[tf.Tensor]*) – A Tensor representing a loss constraint. Using *zfit.constraint.** allows for easy use of predefined constraints.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name

Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

class zfit.loss.**SimpleLoss**(*func: Callable, dependents: Iterable[zfit.Parameter] = <zfit.util.checks.NotSpecified object>, errordef: Optional[float] = None*)

Bases: *zfit.core.loss.BaseLoss*

Loss from a (function returning a) Tensor.

Parameters

- **func** – Callable that constructs the loss and returns a tensor.
- **dependents** – The dependents (independent *zfit.Parameter*) of the loss. If not given, the dependents are figured out automatically.
- **errordef** – Definition of which change in the loss corresponds to a change of 1 sigma. For example, 1 for Chi squared, 0.5 for negative log-likelihood.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

add_constraints(*constraints*)

constraints

copy(*deep: bool = False, name: str = None, **overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

gradients(*params: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor] = None*) → List[tensorflow.python.framework.ops.Tensor]

graph_caching_methods = []

model

name

Name prepended to all ops created by this *model*.

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

value_gradients(*params*)

value_gradients_hessian(*params*)

minimize

zfit.minimize.**MinuitMinimizer**
alias of *zfit.minimizers.minimizer_minuit.Minuit*

zfit.minimize.**ScipyMinimizer**
alias of *zfit.minimizers.minimizers_scipy.Scipy*

zfit.minimize.**AdamMinimizer**
alias of *zfit.minimizers.optimizers_tf.Adam*

class zfit.minimize.**WrapOptimizer**(*optimizer*, *tolerance=None*, *verbosity=None*, *name=None*, ***kwargs*)
Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult
Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

class zfit.minimize.**Adam**(*tolerance=None*, *learning_rate=0.2*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, *use_locking=False*, *name='Adam'*, ***kwargs*)
Bases: *zfit.minimizers.base_tf.WrapOptimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult
Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

class zfit.minimize.**Minuit**(*strategy: zfit.minimizers.baseminimizer.ZfitStrategy = None, minimize_strategy: int = 1, tolerance: float = None, verbosity: int = 5, name: str = None, ncall: int = 10000, **minimizer_options*)
Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*, *zfit.util.cache.Cachable*

Parameters

- **()** (*ncall*) – A ZfitStrategy object that defines the behavior of
- **minimizer in certain situations.** (*the*) –
- **()** – A number used by minuit to define the strategy
- **()** – Internal numerical tolerance
- **()** – Regulates how much will be printed during minimization. Values between 0 and 10 are valid.
- **()** – Name of the minimizer
- **()** – Maximum number of minimization steps.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

copy()

graph_caching_methods = []

minimize(*loss: zfit.core.interfaces.ZfitLoss, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult
Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

old_graph_caching_methods = []

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

class zfit.minimize.**Scipy**(*minimizer='L-BFGS-B', tolerance=None, verbosity=5, name=None, **minimizer_options*)
Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult
Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)
Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

class zfit.minimize.**BFGS**(*strategy: zfit.minimizers.baseminimizer.ZfitStrategy = None*, *tolerance: float = 1e-05*, *verbosity: int = 5*, *name: str = 'BFGS_TFP'*, *options: Mapping[KT, VT_co] = None*)

Bases: *zfit.minimizers.baseminimizer.BaseMinimizer*

Parameters

- **strategy** (ZfitStrategy) – Strategy that handles NaN and more (to come, experimental)
- **tolerance** (*float*) – Difference between the function value that suffices to stop minimization
- **verbosity** – The higher, the more is printed. Between 1 and 10 typically
- **name** – Name of the Minimizer
- **options** – A *dict* containing the options given to the minimization function, overriding the default

copy()

minimize(*loss: zfit.core.interfaces.ZfitLoss*, *params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*) → zfit.minimizers.fitresult.FitResult

Fully minimize the *loss* with respect to *params*.

Parameters

- **loss** (ZfitLoss) – Loss to be minimized.
- **params** (list(*zfit.Parameter*) – The parameters with respect to which to minimize the *loss*. If *None*, the parameters will be taken from the *loss*.

Returns The fit result.

Return type *FitResult*

step(*loss, params: Union[Iterable[zfit.core.interfaces.ZfitParameter], None, Iterable[str]] = None*)

Perform a single step in the minimization (if implemented).

Parameters **()** (*params*) –

Returns:

Raises NotImplementedError – if the *step* method is not implemented in the minimizer.

tolerance

param

class zfit.param.**ConstantParameter**(*name*, *value*, *dtype=tf.float64*)

Bases: *zfit.core.parameter.BaseZParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.param.**Parameter**(*name*, *value*, *lower_limit=None*, *upper_limit=None*, *step_size=None*, *floating=True*, *dtype=tf.float64*, ***kwargs*)
Bases: *zfit.core.parameter.ZfitParameterMixin*, *zfit.core.parameter.TFBaseVariable*, *zfit.core.parameter.BaseParameter*

Class for fit parameters, derived from TF Variable class.

name : name of the parameter, value : starting value lower_limit : lower limit upper_limit : upper limit step_size : step size (set to 0 for fixed parameters)

class SaveSliceInfo(*full_name=None*, *full_shape=None*, *var_offset=None*, *var_shape=None*, *save_slice_info_def=None*, *import_scope=None*)

Bases: object

Information on how to save this Variable as a slice.

Provides internal support for saving variables as slices of a larger variable. This API is not public and is subject to change.

Available properties:

- full_name
- full_shape
- var_offset
- var_shape

Create a *SaveSliceInfo*.

Parameters

- **full_name** – Name of the full variable of which this *Variable* is a slice.
- **full_shape** – Shape of the full variable, as a list of int.
- **var_offset** – Offset of this *Variable* into the full variable, as a list of int.
- **var_shape** – Shape of this *Variable*, as a list of int.
- **save_slice_info_def** – *SaveSliceInfoDef* protocol buffer. If not *None*, recreates the SaveSliceInfo object its contents. *save_slice_info_def* and other arguments are mutually exclusive.
- **import_scope** – Optional *string*. Name scope to add. Only used when initializing from protocol buffer.

spec

Computes the spec string used for saving.

to_proto(*export_scope=None*)

Returns a SaveSliceInfoDef() proto.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Returns A *SaveSliceInfoDef* protocol buffer, or None if the *Variable* is not in the specified name scope.

__iter__()

Dummy method to prevent iteration.

Do not call.

NOTE(mrry): If we register __getitem__ as an overloaded operator, Python will valiantly attempt to iterate over the variable's Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises TypeError – when invoked.

__ne__(*other*)

Compares two variables element-wise for equality.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

aggregation

assign(*value*, *use_locking=None*, *name=None*, *read_value=True*)

Assigns a new value to this variable.

Parameters

- **value** – A *Tensor*. The new value for this variable.
- **use_locking** – If *True*, use locking during the assignment.
- **name** – The name to use for the assignment.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_add(*delta*, *use_locking=None*, *name=None*, *read_value=True*)

Adds a value to this variable.

Parameters

- **delta** – A *Tensor*. The value to add to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

assign_sub(*delta*, *use_locking=None*, *name=None*, *read_value=True*)

Subtracts a value from this variable.

Parameters

- **delta** – A *Tensor*. The value to subtract from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – The name to use for the operation.
- **read_value** – A *bool*. Whether to read and return the new value of the variable or not.

Returns If *read_value* is *True*, this method will return the new value of the variable after the assignment has completed. Otherwise, when in graph mode it will return the *Operation* that does the assignment, and when in eager mode it will return *None*.

batch_scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable batch-wise.

Analogous to *batch_gather*. This assumes that this variable and the sparse_delta IndexedSlices have a series of leading dimensions that are the same for all of them, and the updates are performed on the last dimension of indices. In other words, the dimensions should be the following:

num_prefix_dims = sparse_delta.indices.ndims - 1 batch_dim = num_prefix_dims + 1 ‘sparse_delta.updates.shape = sparse_delta.indices.shape + var.shape[

batch_dim:]‘

where

sparse_delta.updates.shape[:num_prefix_dims] == sparse_delta.indices.shape[:num_prefix_dims] == var.shape[:num_prefix_dims]

And the operation performed can be expressed as:

‘var[i_1, . . . , i_n,

sparse_delta.indices[i_1, . . . , i_n, j]] = sparse_delta.updates[i_1, . . . , i_n, j]‘

When sparse_delta.indices is a 1D tensor, this operation is equivalent to *scatter_update*.

To avoid this operation one can looping over the first *ndims* of the variable and using *scatter_update* on the subtensors that result of slicing the first dimension. This is a valid option for *ndims = 1*, but less efficient than this implementation.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

constraint
Returns the constraint function associated with this variable.

Returns The constraint function that was passed to the variable constructor. Can be *None* if no constraint was passed.

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

count_up_to(*limit*)
Increments this variable until it reaches *limit*. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Dataset.range instead.

When that Op is run it tries to increment the variable by *1*. If incrementing the variable would bring it above *limit* then the Op raises the exception *OutOfRangeError*.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for *count_up_to(self, limit)*.

Parameters **limit** – value at which incrementing the variable raises an error.

Returns A *Tensor* that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

create
The op responsible for initializing this variable.

device
The device this variable is on.

dtype
The dtype of the object

eval(*session=None*)
Evaluates and returns the value of this variable.

experimental_ref()
Returns a hashable reference object to this Variable.

Warning: Experimental API that could be changed or removed.

The primary usecase for this API is to put variables in a set/dictionary. We can't put variables in a set/dictionary as *variable.__hash__()* is no longer available starting Tensorflow 2.0.

“‘python import tensorflow as tf

x = tf.Variable(5) y = tf.Variable(10) z = tf.Variable(10)

The followings will raise an exception starting 2.0 # TypeError: Variable is unhashable if Variable equality is enabled. variable_set = {x, y, z} variable_dict = {x: ‘five’, y: ‘ten’} “‘

Instead, we can use *variable.experimental_ref()*.

“‘python variable_set = {x.experimental_ref(),

y.experimental_ref(), z.experimental_ref()}

print(x.experimental_ref() in variable_set) ==> True

variable_dict = {x.experimental_ref(): ‘five’, y.experimental_ref(): ‘ten’, z.experimental_ref(): ‘ten’}

print(variable_dict[y.experimental_ref()]) ==> ten “‘

Also, the reference object provides *.deref()* function that returns the original Variable.

```
`python x = tf.Variable(5) print(x.experimental_ref().deref()) ==>
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=5> `
```

floating

static from_proto(*variable_def*, *import_scope=None*)
Returns a *Variable* object created from *variable_def*.

gather_nd(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather_nd*.

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_shape()
Alias of *Variable.shape*.

graph
The *Graph* of this variable.

graph_caching_methods = []

handle
The handle by which this variable can be accessed.

has_limits

independent

initial_value
Returns the Tensor used as the initial value for the variable.

initialized_value()
Returns the value of the initialized variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun) contexts.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
`python # Initialize 'v' with a random tensor. v = tf.Variable(tf.
random.truncated_normal([10, 40])) # Use `initialized_value` to
guarantee that `v` has been # initialized before its value is used
to initialize `w`. # The random values are picked only once. w = tf.
Variable(v.initialized_value() * 2.0) `
```

Returns A *Tensor* holding the value of this variable after its initializer has run.

initializer
The op responsible for initializing this variable.

is_initialized(*name=None*)
Checks whether a resource variable has been initialized.

Outputs boolean scalar indicating whether the tensor has been initialized.

Parameters **name** – A name for the operation (optional).

Returns A *Tensor* of type *bool*.

load(*value*, *session=None*)
Load new value into this variable. (deprecated)

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Prefer Variable.assign which has equivalent behavior in 2.X.

Writes new value to variable's memory. Doesn't add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See *tf.compat.v1.Session* for more information on launching a graph and on sessions.

‘‘‘python v = tf.Variable([1, 2]) init = tf.compat.v1.global_variables_initializer()

with tf.compat.v1.Session() as sess: sess.run(init) # Usage passing the session explicitly. v.load([2, 3], sess) print(v.eval(sess)) # prints [2 3] # Usage with the default session. The ‘with’ block # above makes ‘sess’ the default session. v.load([3, 4], sess) print(v.eval()) # prints [3 4]

‘‘‘

Parameters

- **value** – New variable value
- **session** – The session to use to evaluate this variable. If none, the default session is used.

Raises `ValueError` – Session is not passed and no default session

lower_limit

name
The name of the object.

numpy()

old_graph_caching_methods = []

op
The op for this variable.

params

randomize(*minval=None*, *maxval=None*, *sampler=<built-in method uniform of numpy.random.mtrand.RandomState object>*)
Update the value with a randomised value between minval and maxval.

Parameters

- **minval** (*Numerical*) –
- **maxval** (*Numerical*) –
- **()** (*sampler*) –

read_value()
Constructs an op which reads the value of this variable.

Should be used when there are multiple reads, or when it is desirable to read the value only after some condition is true.

Returns the read operation.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

scatter_add(*sparse_delta*, *use_locking=False*, *name=None*)
Adds *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be added to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered addition has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_div(*sparse_delta*, *use_locking=False*, *name=None*)
Divide this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to divide this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered division has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_max(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the max of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of max with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered maximization has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_min(*sparse_delta*, *use_locking=False*, *name=None*)
Updates this variable with the min of *tf.IndexedSlices* and itself.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to use as an argument of min with this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered minimization has completed.

Raises TypeError – if *sparse_delta* is not an *IndexedSlices*.

scatter_mul(*sparse_delta*, *use_locking=False*, *name=None*)
Multiply this variable by *tf.IndexedSlices*.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to multiply this variable by.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered multiplication has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_nd_add(*indices*, *updates*, *name=None*)

Applies sparse addition to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) add = ref.scatter_nd_add(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(add)

‘‘‘

The resulting update to ref would look like this:

[1, 13, 3, 14, 14, 6, 7, 20]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_sub(*indices*, *updates*, *name=None*)

Applies sparse subtraction to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K'th dimension of 'ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

```
` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `
```

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_sub(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, -9, 3, -6, -6, 6, 7, -4]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_nd_update(*indices*, *updates*, *name=None*)

Applies sparse assignment to individual values or slices in a Variable.

ref is a *Tensor* with rank *P* and *indices* is a *Tensor* of rank *Q*.

indices must be integer tensor, containing indices into *ref*. It must be shape *[d_0, ..., d_{Q-2}, K]* where *0 < K <= P*.

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if *K = P*) or slices (if *K < P*) along the *K‘th dimension of ‘ref*.

updates is *Tensor* of rank *Q-1+P-K* with shape:

` [d_0, ..., d_{Q-2}, ref.shape[K], ..., ref.shape[P-1]]. `

For example, say we want to add 4 scattered elements to a rank-1 tensor to 8 elements. In Python, that update would look like this:

‘‘‘**python** ref = tf.Variable([1, 2, 3, 4, 5, 6, 7, 8]) indices = tf.constant([[4], [3], [1] ,[7]]) updates = tf.constant([9, 10, 11, 12]) op = ref.scatter_nd_update(indices, updates) with tf.compat.v1.Session() as sess:

print sess.run(op)

‘‘‘

The resulting update to ref would look like this:

[1, 11, 3, 10, 9, 6, 7, 12]

See *tf.scatter_nd* for more details about how to make updates to slices.

Parameters

- **indices** – The indices to be used in the operation.
- **updates** – The values to be used in the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

scatter_sub(*sparse_delta*, *use_locking=False*, *name=None*)
Subtracts *tf.IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be subtracted from this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update(*sparse_delta*, *use_locking=False*, *name=None*)
Assigns *tf.IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *tf.IndexedSlices* to be assigned to this variable.
- **use_locking** – If *True*, use locking during the operation.
- **name** – the name of the operation.

Returns A *Tensor* that will hold the new value of this variable after the scattered subtraction has completed.

Raises `TypeError` – if *sparse_delta* is not an *IndexedSlices*.

set_shape(*shape*)
Unsupported.

set_value(*value: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)
Set the *Parameter* to *value* (temporarily if used in a context manager).

Parameters **value** (*float*) – The value the parameter will take on.

shape
The shape of this variable.

sparse_read(*indices*, *name=None*)
Reads the value of this variable sparsely, using *gather*.

step_size

synchronization

to_proto(*export_scope=None*)
Converts a *ResourceVariable* to a *VariableDef* protocol buffer.

Parameters **export_scope** – Optional *string*. Name scope to remove.

Raises `RuntimeError` – If run in EAGER mode.

Returns A *VariableDef* protocol buffer, or *None* if the *Variable* is not in the specified name scope.

trainable

upper_limit

value()
A cached operation which reads the value of this variable.

class zfit.param.**ComposedParameter**(*name*, *value_fn*, *dependents*, *dtype=tf.float64*, ***kwargs*)
Bases: *zfit.core.parameter.BaseComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

independent

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

class zfit.param.**ComplexParameter**(*name*, *value_fn*, *dependents*, *dtype=tf.complex128*, ***kwargs*)
Bases: *zfit.core.parameter.ComposedParameter*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

arg

assign(*value*, *use_locking=False*, *name=None*, *read_value=True*)

conj

copy(*deep: bool = False*, *name: str = None*, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype
The dtype of the object

floating

static from_cartesian(*name*, *real*, *imag*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

static from_polar(*name*, *mod*, *arg*, *dtype=tf.complex128*, *floating=True*, ***kwargs*)

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

graph_caching_methods = []

imag

independent

mod

name
The name of the object.

numpy()

old_graph_caching_methods = []

params

read_value()

real

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

value()

zfit.param.**convert_to_parameter**(*value*, *name=None*, *prefer_floating=False*, *dependents=None*, *graph_mode=False*) → zfit.core.interfaces.ZfitParameter
Convert a *numerical* to a fixed/floating parameter or return if already a parameter.

Parameters

- **()** (*name*) –
- **()** –
- **prefer_floating** – If True, create a Parameter instead of a FixedParameter _if **possible**_.

pdf

class zfit.pdf.**BasePDF**(*obs: Union[str, Iterable[str], zfit.Space], params: Dict[str, zfit.core.interfaces.ZfitParameter] = None, dtype: Type[CT_co] = tf.float64, name: str = 'BasePDF', **kwargs*)
Bases: *zfit.core.interfaces.ZfitPDF*, *zfit.core.basemodel.BaseModel*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –

- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –

- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If

fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.

- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- `NotExtendedPDFError` – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**BaseFunctor**(*pdfs*, *name='BaseFunctor'*, ***kwargs*)

Bases: *zfit.models.basefunctor.FunctorMixin*, *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]

Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended

Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models

Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

pdfs_extended

classmethod register_additional_repr (***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional tf.Tensor containing the unnormalized pdf.

Return type tf.Tensor

update_integration_options(*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Exponential**(*lambda_, obs: Union[str, Iterable[str], zfit.Space], name: str = 'Exponential', **kwargs*)

Bases: *zfit.core.basepdf.BasePDF*

Exponential function exp(lambda * x).

The function is normalized over a finite range and therefore a pdf. So the PDF is precisely defined as $\frac{e^{\lambda \cdot x}}{\int_{lower}^{upper} e^{\lambda \cdot x} dx}$

Parameters

- **lambda** (*Parameter*) – Accessed as parameter "lambda".
- **obs** (*Space*) – The *Space* the pdf is defined in.
- **name** (*str*) – Name of the pdf.
- **dtype** (*DType*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over

- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()

Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**CrystalBall**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alpha: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], n: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'CrystalBall', dtype: Type[CT_co] = tf.float64*)

Bases: *zfit.core.basepdf.BasePDF*

'Crystal Ball shaped PDF'__. A combination of a Gaussian with an powerlaw tail.

The function is defined as follows:

$$f(x;\mu,\sigma,\alpha,n) = \begin{cases} \exp(-\frac{(x-\mu)^2}{2\sigma^2}), & \text{for} \frac{x-\mu}{\sigma} \geqslant -\alpha \\ A \cdot (B - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} < -\alpha \end{cases}$$

with

$$A = \left(\frac{n}{|\alpha|}\right)^n \cdot \exp\left(-\frac{|\alpha|^2}{2}\right)$$

$$B = \frac{n}{|\alpha|} - |\alpha|$$

Parameters

- **mu** (*zfit.Parameter*) – The mean of the gaussian
- **sigma** (*zfit.Parameter*) – Standard deviation of the gaussian
- **alpha** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail
- **n** (*zfit.Parameter*) – Exponent of the powertail
- **obs** (*Space*) –
- **name** (*str*) –
- **dtype** (*tf.DType*) –

__CBShape_

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises *TypeError* – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for

- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**DoubleCB**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alphal: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], nl: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], alphar: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], nr: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'DoubleCB', dtype: Type[CT_co] = tf.float64*)

Bases: *zfit.core.basepdf.BasePDF*

'Double sided Crystal Ball shaped PDF'__. A combination of two CB using the **mu** (not a frac). on each side.

The function is defined as follows:

$$f(x;\mu,\sigma,\alpha_L,n_L,\alpha_R,n_R) = \begin{cases} A_L \cdot (B_L - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} < -\alpha_L \exp(-\frac{(x-\mu)^2}{2\sigma^2}), \\ -\alpha_L \leqslant \text{for} \frac{x-\mu}{\sigma} \leqslant \alpha_R A_R \cdot (B_R - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} > \alpha_R \end{cases}$$

with

$$A_{L/R} = \left(\frac{n_{L/R}}{|\alpha_{L/R}|}\right)^n_{L/R} \cdot \exp\left(-\frac{|\alpha_{L/R}|^2}{2}\right)$$

$$B_{L/R} = \frac{n_{L/R}}{|\alpha_{L/R}|} - |\alpha_{L/R}|$$

Parameters

- **mu** (*zfit.Parameter*) – The mean of the gaussian
- **sigma** (*zfit.Parameter*) – Standard deviation of the gaussian
- **alphal** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the left
- **side** –
- **nl** (*zfit.Parameter*) – Exponent of the powertail on the left side
- **alphar** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the right
- **side** –

- **nr** (*zfit.Parameter*) – Exponent of the powertail on the right side
- **obs** (*Space*) –
- **name** (*str*) –
- **dtype** (*tf.DType*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.

- limits (*Space*): the limits to integrate over.
- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Gauss** (*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Gauss'*)
Bases: *zfit.models.dist_tfp.WrapDistribution*

Gaussian or Normal distribution with a mean (mu) and a standartdevation (sigma).

The gaussian shape is defined as

$$f(x \mid \mu, \sigma^2) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

with the normalization over [-inf, inf] of

$$\frac{1}{\sqrt{2\pi\sigma^2}}$$

The normalization changes for different normalization ranges

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **obs** (*Space*) – Observables and normalization range the pdf is defined in

- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

distribution

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.

- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Uniform**(*low: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], high: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Uniform'*)
Bases: *zfit.models.dist_tfp.WrapDistribution*

Uniform distribution which is constant between *low*, *high* and zero outside.

Parameters

- **low** (*Parameter*) – Below this value, the pdf is zero.
- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –

- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –

- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If

fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.

- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- `NotExtendedPDFError` – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**TruncatedGauss**(*mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], low: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], high: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'TruncatedGauss'*)

Bases: *zfit.models.dist_tfp.WrapDistribution*

Gaussian distribution that is 0 outside of *low*, *high*. Equivalent to the product of Gauss and Uniform.

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **low** (*Parameter*) – Below this value, the pdf is zero.
- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –

- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –

- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If

fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.

- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- `NotExtendedPDFError` – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**WrapDistribution**(*distribution*, *dist_params*, *obs*, *params=None*, *dist_kwargs=None*, *dtype=tf.float64*, *name=None*, ***kwargs*)

Bases: *zfit.core.basepdf.BasePDF*

Baseclass to wrap tensorflow-probability distributions automatically.

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

distribution

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional tf.Tensor containing the unnormalized pdf.

Return type tf.Tensor

update_integration_options(*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Chebyshev**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None*, *name: str = 'Chebyshev'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Chebyshev (first kind) polynomials of order len(coeffs), coeffs are scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$
$$with T_0 = 1 T_1 = x$$

Notice that T_1 is x as opposed to 2x in Chebyshev polynomials of the second kind.

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*, *allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]*, *norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None*, *name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over

- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree

degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (`numerical`) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (`str`) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated

- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional tf.Tensor containing the unnormalized pdf.

Return type tf.Tensor

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Legendre**(*obs: Union[str, Iterable[str], zfit.Space], coeffs: List[Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]], apply_scaling: bool = True, coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, name: str = 'Legendre'*)
Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Legendre polynomials of order len(coeffs), the coeffs are overall scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$
$$with P_0 = 1 P_1 = x$$

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.

- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Chebyshev2**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None*, *name: str = 'Chebyshev2'*)

Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Chebyshev (second kind) polynomials of order len(coeffs), coeffs are scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order_** of the polynomial is

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$
$$with T_0 = 1 T_1 = 2x$$

Notice that T_1 is 2x as opposed to x in Chebyshev polynomials of the first kind.

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.

- limits (*Space*): the limits to integrate over.
- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Hermite**(*obs, coeffs: list, apply_scaling: bool = True, coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, name: str = 'Hermite'*)
Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Hermite polynomials (for physics) of order len(coeffs), with coeffs as scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

with P_0 = 1 P_1 = 2x

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).

- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:

- **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
- limits (*Space*): the limits to integrate over.
- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.

- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None, mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**Laguerre**(*obs, coeffs: list, apply_scaling: bool = True, coeff0: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, name: str = 'Laguerre'*)
Bases: *zfit.models.polynomials.RecursivePolynomial*

Linear combination of Laguerre polynomials of order len(coeffs), the coeffs are overall scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$(n+1)L_{n+1}(x) = (2n+1+lpha-x)L_n(x) - (n+lpha)L_{n-1}(x)$$

with P_0 = 1 P_1 = 1 - x

Parameters

- **obs** – The default space the PDF is defined in.

- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func (*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)
Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space (*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

degree

degree of the polynomial, starting from 0.

Type int

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**RecursivePolynomial**(*obs*, *coeffs: list*, *apply_scaling: bool = True*, *coeff0: Optional[tensorflow.python.framework.ops.Tensor] = None*, *name: str = 'Polynomial'*)

Bases: *zfit.core.basepdf.BasePDF*

1D polynomial generated via three-term recurrence.

Base class to create 1 dimensional recursive polynomials that can be rescaled. Overwrite _poly_func.

Parameters

- **coeffs** (*list*) – Coefficients for each polynomial. Used to calculate the degree.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).

$$x_{n+1} = recurrence(x_n, x_{n-1}, n)$$

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

degree
degree of the polynomial, starting from 0.

Type int

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over

- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options (*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**ProductPDF** (*pdfs: List[zfit.core.interfaces.ZfitPDF], obs: Union[str, Iterable[str], zfit.Space] = None, name='ProductPDF'*)
Bases: *zfit.models.functor.BaseFunctor*

add_cache_dependents (*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (`ZfitCachable`) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (`numerical`) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (`str`) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

pdfs_extended

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**SumPDF**(*pdfs: List[zfit.core.interfaces.ZfitPDF], fracs: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None, obs: Union[str, Iterable[str], zfit.Space] = None, name: str = 'SumPDF'*)

Bases: *zfit.models.functor.BaseFunctor*

Create the sum of the *pdfs* with *fracs* as coefficients.

Parameters

- **pdfs** (*pdf*) – The pdfs to add.
- **fracs** (*iterable*) – coefficients for the linear combination of the pdfs. If pdfs are extended, this throws an error.
  - len(frac) == len(basic) - 1 results in the interpretation of a non-extended pdf. The last coefficient will equal to 1 - sum(frac)
  - len(frac) == len(pdf) each pdf in *pdfs* will become an extended pdf with the given yield.
- **name** (*str*) –

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended (*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~‘zfit.core.data.Sampler‘

Raises

- `NotExtendedPDFError` – if ‘extended’ is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype
The dtype of the object

fracs

get_dependents(*only_floating: bool = True) -> OrderedSet(['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (`bool`) – If *True*, only return floating `Parameter`

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (`names`) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (`numerical`) – *float* or *double Tensor*.

- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

pdfs_extended

classmethod register_additional_repr(***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**

- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**ZPDF**(*obs: Union[str, Iterable[str], zfit.Space], name: str = 'ZPDF', **params*)

Bases: *zfit.core.basemodel.SimpleModelSubclassMixin*, *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (*ZfitCachable*) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises `TypeError` – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- `NotImplementedError` – If no analytical integral is available (for this limits).
- `NormRangeNotImplementedError` – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to `Space` and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)

Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]

Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]

Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended

Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs

Return the number of observables.

name

The name of the object.

norm_range

Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

classmethod register_additional_repr (***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)
Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()
Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space

Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**SimplePDF**(*obs*, *func*, *name='SimplePDF'*, ***params*)

Bases: *zfit.core.basepdf.BasePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)

Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes
Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]
Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF
Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF
Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF
Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler
Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- NotExtendedPDFError – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

dtype
The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent *Parameter* that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating *Parameter*

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type *Parameter*

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, *Space*) – *Space* to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated
- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf(***kwargs*)

classmethod register_additional_repr(***kwargs*)
Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an**(*any*) –
- **or callable method of self.**(*attribute*) –

classmethod register_analytic_integral(*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None
Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.

- **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
- params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
- model (*ZfitModel*):The model that is being integrated.

- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher(*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral(*func: Callable*) → None

Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache(*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self()

Clear the cache of self and all dependent cachers.

sample(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData

Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- `NotExtendedPDFError` – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

set_norm_range(*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

class zfit.pdf.**SimpleFunctorPDF**(*obs*, *pdfs*, *func*, *name='SimpleFunctorPDF'*, ***params*)
Bases: *zfit.models.functor.BaseFunctor*, *zfit.models.special.SimplePDF*

add_cache_dependents(*cache_dependents: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]], allow_non_cachable: bool = True*)
Add dependents that render the cache invalid if they change.

Parameters

- **cache_dependents** (ZfitCachable) –
- **allow_non_cachable** (*bool*) – If *True*, allow *cache_dependents* to be non-cachables. If *False*, any *cache_dependents* that is not a *ZfitCachable* will raise an error.

Raises TypeError – if one of the *cache_dependents* is not a *ZfitCachable* _and_ *allow_non_cachable* if *False*.

analytic_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'analytic_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over

- **norm_range** (tuple, *Space*, *False*) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

- NotImplementedError – If no analytical integral is available (for this limits).
- NormRangeNotImplementedError – if the *norm_range* argument is not supported. This means that no analytical normalization is available, explicitly: the **analytical** integral over the limits = norm_range is not available.

apply_yield(*value: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = False, log: bool = False*) → Union[float, tensorflow.python.framework.ops.Tensor]

If a norm_range is given, the value will be multiplied by the yield.

Parameters

- **value** (*numerical*) –
- **()** (*norm_range*) –
- **log** (*bool*) –

Returns numerical

as_func(*norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

convert_sort_space(*obs: Union[str, Iterable[str], zfit.Space] = None, axes: Union[int, Iterable[int]] = None, limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None*) → Optional[zfit.core.limits.Space]

Convert the inputs (using eventually *obs*, *axes*) to *Space* and sort them according to own *obs*.

Parameters

- **()** (*limits*) –
- **()** –
- **()** –

Returns:

copy(***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

Note: the copy model may continue to depend on the original initialization arguments.

Parameters ****override_parameters** – String/value dictionary of initialization arguments to override with new value.

Returns

A new instance of *type(self)* initialized from the union of self.parameters and override_parameters, i.e., *dict(self.parameters, **override_parameters)*.

Return type model

create_extended(*yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended'*) → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

- **yield** (numeric, *Parameter*) –
- **name_addition** (*str*) –

Returns *ZfitPDF*

create_projection_pdf(*limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

The new projection pdf is still fully dependent on the pdf it was created with.

Parameters **limits_to_integrate** (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

create_sampler(*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, fixed_params: Union[bool, List[zfit.core.interfaces.ZfitParameter], Tuple[zfit.core.interfaces.ZfitParameter]] = True, name: str = 'create_sampler'*) → zfit.core.data.Sampler

Create a `Sampler` that acts as *Data* but can be resampled, also with changed parameters and n.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **()** (*name*) – From which space to sample.
- **()** – A list of *Parameters* that will be fixed during several *resample* calls. If True, all are fixed, if False, all are floating. If a *Parameter* is not fixed and its value gets updated (e.g. by a *Parameter.set_value()* call), this will be reflected in *resample*. If fixed, the Parameter will still have the same value as the *Sampler* has been created with when it resamples.
- **()** –

Returns py:class:~'zfit.core.data.Sampler'

Raises

- `NotExtendedPDFError` – if 'extended' is chosen (implicitly by default or explicitly) as an option for *n* but the pdf itself is not extended.
- `ValueError` – if n is an invalid string option.
- `InvalidArgumentError` – if n is not specified and pdf is not extended.

dtype

The dtype of the object

get_dependents(*only_floating: bool = True) -> OrderedSet['z', 'f', 'i', 't', '.', 'P', 'a', 'r', 'm', 'e']*)
Return a set of all independent `Parameter` that this object depends on.

Parameters **only_floating** (*bool*) – If *True*, only return floating `Parameter`

get_models(*names=None*) → List[zfit.core.interfaces.ZfitModel]

get_params(*only_floating: bool = False, names: Union[str, List[str], None] = None*) → List[zfit.core.interfaces.ZfitParameter]
Return the parameters. If it is empty, automatically return all floating variables.

Parameters

- **()** (*names*) – If True, return only the floating parameters.
- **()** – The names of the parameters to return.

Returns

Return type list(*ZfitParameters*)

get_yield() → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

gradients(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]] = None*)

graph_caching_methods = []

integrate(***kwargs*)

is_extended
Flag to tell whether the model is extended or not.

Returns

Return type bool

log_pdf(*x: Union[float, tensorflow.python.framework.ops.Tensor], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'log_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Log probability density function normalized over *norm_range*.

Parameters

- **x** (*numerical*) – *float* or *double Tensor*.
- **norm_range** (tuple, `Space`) – `Space` to normalize over
- **name** (*str*) – Prepended to names of ops created by this function.

Returns a *Tensor* of type *self.dtype*.

Return type log_pdf

models
Return the models of this *Functor*. Can be *pdfs* or *funcs*.

n_obs
Return the number of observables.

name
The name of the object.

norm_range
Return the current normalization range. If None and the ‘obs‘have limits, they are returned.

Returns The current normalization range

Return type *Space* or None

normalization(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], name: str = 'normalization'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Return the normalization of the function (usually the integral over *limits*).

Parameters

- **limits** (tuple, *Space*) – The limits on where to normalize over
- **name** (*str*) –

Returns the normalization value

Return type Tensor

numeric_integrate(*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Numerical integration over the model.

Parameters

- **limits** (tuple, *Space*) – the limits to integrate over
- **norm_range** (tuple, *Space*, False) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs
Return the observables.

old_graph_caching_methods = []

params

partial_analytic_integrate(***kwargs*)

partial_integrate(***kwargs*)

partial_numeric_integrate(*x: Union[float, tensorflow.python.framework.ops.Tensor], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_numeric_integrate'*) → Union[float, tensorflow.python.framework.ops.Tensor]
Force numerical partial integration of the function over the *limits* and evaluate it at *x*.

Dimension of *limits* and *x* have to add up to the full dimension and be therefore equal to the dimensions of *norm_range* (if not False)

Parameters

- **x** (*numerical*) – The value at which the partially integrated function will be evaluated

- **limits** (tuple, *Space*) – the limits to integrate over. Can contain only some axes
- **norm_range** (tuple, *Space*, False) – the limits to normalize over. Has to have all axes
- **name** (*str*) –

Returns the value of the partially integrated function evaluated at *x*.

Return type Tensor

pdf (***kwargs*)

pdfs_extended

classmethod register_additional_repr (***kwargs*)

Register an additional attribute to add to the repr.

Parameters

- **keyword argument. The value has to be gettable from the instance (has to be an** (*any*) –
- **or callable method of self.** (*attribute*) –

classmethod register_analytic_integral (*func: Callable, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, priority: Union[int, float] = 50, *, supports_norm_range: bool = False, supports_multiple_limits: bool = False*) → None

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:
  - **x (*ZfitData*, None): the data for the remaining axes in a partial** integral. If it is not a partial integral, this will be None.
  - limits (*Space*): the limits to integrate over.
  - **norm_range (*Space*, None): Normalization range of the integral.** If not *supports_supports_norm_range*, this will be None.
  - params (Dict[param_name, `zfit.Parameters`]): The parameters of the model.
  - model (*ZfitModel*):The model that is being integrated.
- **()** (*limits*) – **|limits_arg_descr|**
- **priority** (*int*) – Priority of the function. If multiple functions cover the same space, the one with the highest priority will be used.
- **supports_multiple_limits** (*bool*) – If *True*, the *limits* given to the integration function can have multiple limits. If *False*, only simple limits will pass through and multiple limits will be auto-handled.
- **supports_norm_range** (*bool*) – If *True*, *norm_range* argument to the function may not be *None*. If *False*, *norm_range* will always be *None* and care is taken of the normalization automatically.

register_cacher (*cacher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cacher* that caches values produces by this instance; a dependent.

Parameters **()** (*cacher*) –

classmethod register_inverse_analytic_integral (*func: Callable*) → None
Register an inverse analytical integral, the inverse (unnormalized) cdf.

Parameters **()** (*func*) –

reset_cache (*reseter: zfit.util.cache.ZfitCachable*)

reset_cache_self ()
Clear the cache of self and all dependent cachers.

sample (*n: Union[int, tensorflow.python.framework.ops.Tensor, str] = None, limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'sample'*) → zfit.core.data.SampleData
Sample *n* points within *limits* from the model.

If *limits* is not specified, *space* is used (if the space contains limits). If *n* is None and the model is an extended pdf, 'extended' is used by default.

Parameters

- **n** (*int, tf.Tensor, str*) – The number of samples to be generated. Can be a Tensor that will be or a valid string. Currently implemented:
  - 'extended': samples *poisson(yield)* from each pdf that is extended.
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns SampleData(n_obs, n_samples)

Raises

- NotExtendedPDFError – if 'extended' is (implicitly by default or explicitly) chosen as an option for *n* but the pdf itself is not extended.
- ValueError – if n is an invalid string option.
- InvalidArgumentError – if n is not specified and pdf is not extended.

set_norm_range (*norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)
Set the normalization range (temporarily if used with contextmanager).

Parameters **norm_range** (tuple, *Space*) –

space
Return the *Space* object that defines the dimensionality of the object.

unnormalized_pdf (*x: Union[float, tensorflow.python.framework.ops.Tensor], component_norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'unnormalized_pdf'*) → Union[float, tensorflow.python.framework.ops.Tensor]
PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

- **x** (*numerical*) – The value, have to be convertible to a Tensor
- **component_norm_range** (*Space*) – The normalization range for the components. Needed for
- **composition** (*certain*) – pdfs.
- **name** (*str*) –

Returns 1-dimensional `tf.Tensor` containing the unnormalized pdf.

Return type `tf.Tensor`

update_integration_options(*draws_per_dim=None*, *mc_sampler=None*)
Set the integration options.

Parameters

- **draws_per_dim** (*int*) – The draws for MC integration to do
- **()** (*mc_sampler*) –

sample

zfit.sample.**poisson**(*n=None*, *pdfs: Iterable[zfit.core.interfaces.ZfitPDF] = None*)

settings

zfit.settings.**get_verbosity**()

zfit.settings.**set_seed**(*seed*)
Set random seed for numpy

zfit.settings.**set_verbosity**(*verbosity*)

Python Module Index

Z

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z