

---

# **zfit Documentation**

***Release 0.4.0***

**zfit**

**Jan 06, 2020**



---

## Contents

---

<b>1</b>	<b>Getting started with zfit</b>	<b>3</b>
1.1	What did just happen? . . . . .	6
<b>2</b>	<b>zfit introduction</b>	<b>9</b>
2.1	Space, Observable and Range . . . . .	9
2.2	Parameter . . . . .	11
2.3	Building a model . . . . .	12
2.4	Data . . . . .	17
2.5	Loss . . . . .	18
2.6	Minimization . . . . .	20
<b>3</b>	<b>zfit Project</b>	<b>23</b>
3.1	Installation . . . . .	23
3.2	Contributing . . . . .	24
3.3	Upgrade guide . . . . .	25
3.4	Changelog . . . . .	25
3.5	Development Lead . . . . .	29
3.6	Authors . . . . .	29
3.7	Contributors . . . . .	29
<b>4</b>	<b>zfit API documentation</b>	<b>31</b>
4.1	zfit package . . . . .	31
	<b>Python Module Index</b>	<b>537</b>
	<b>Index</b>	<b>539</b>



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}_{-0.0070162654764939734}
```

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

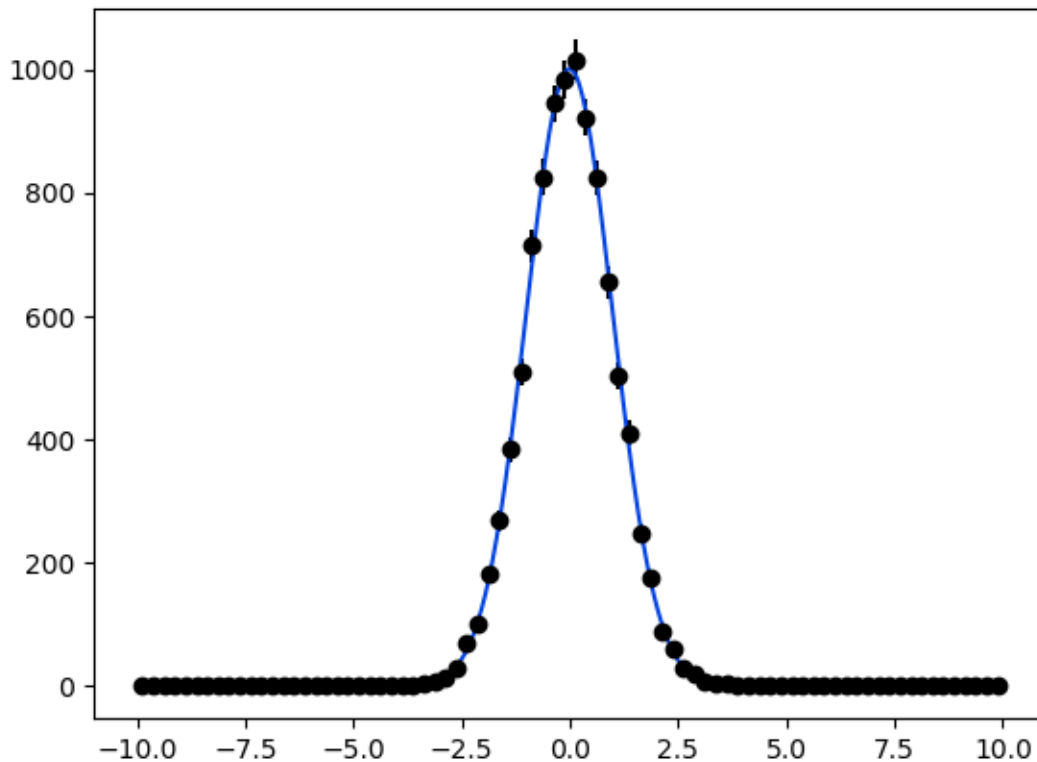
>>> # Printing information on specific parameters, e.g. mu
>>> print("mu={}".format(params[mu]['value']))
mu=0.012464509810750313
```

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: {} \n result: {}".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: {} \n result: {}".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: {} \n result: {}".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
```



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  $\leftrightarrow$  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
->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.  $\int_{-\infty}^{\infty} f(x)dx = 1$ . 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 is 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(pdf=[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(pdf=[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)
...
...     # save 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 accessibility, the data concept implemented 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 functionalities 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 blocks 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
```



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](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 build 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](mailto: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](mailto:zfit@physik.uzh.ch)>

## 3.6 Authors

Jonas Eschle <[Jonas.Eschle@cern.ch](mailto:Jonas.Eschle@cern.ch)>

Albert Puig <[apuignav@gmail.com](mailto:apuignav@gmail.com)>

Rafael Silva Coutinho <[rsilvaco@cern.ch](mailto:rsilvaco@cern.ch)>

Matthieu Marinangeli <[matthieu.marinangeli@cern.ch](mailto:matthieu.marinangeli@cern.ch)>

## 3.7 Contributors

Chris Burr <[christopher.burr@cern.ch](mailto:christopher.burr@cern.ch)>

Abhijit Mathad <[amathad@cern.ch](mailto:amathad@cern.ch)>

Oliver Lantwin <[oliver.lantwin@cern.ch](mailto:oliver.lantwin@cern.ch)>



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:

$$\begin{aligned} \text{num\_prefix\_dims} &= \text{sparse\_delta.indices.ndims} - 1 & \text{batch\_dim} &= \text{num\_prefix\_dims} + 1 \\ \text{'sparse\_delta.updates.shape} &= \text{sparse\_delta.indices.shape} + \text{var.shape}[\text{batch\_dim:}] \end{aligned}$$

where

$$\text{sparse\_delta.updates.shape}[\text{num\_prefix\_dims}] == \text{sparse\_delta.indices.shape}[\text{num\_prefix\_dims}] == \text{var.shape}[\text{num\_prefix\_dims}]$$

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

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

**Parameters** **()** (*cache*) –

**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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

The innermost dimension of *indices* (with length *K*) corresponds to indices into elements (if  $K = P$ ) or slices (if  $K < P$ ) along the *K*<sup>th</sup> 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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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

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

**Parameters** **()** (*caler*) –

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

**Parameters** **()** (*caler*) –

**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\_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`  $\geq 1$

#### **n\_obs**

Return the number of observables/axes.

**Returns** `int`  $\geq 1$

#### **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*)  $\rightarrow$  `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]]*)  $\rightarrow$  `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*)  $\rightarrow$  `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 caught 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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (`catcher:` `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

Register a `catcher` that caches values produces by this instance; a dependent.

**Parameters** `() (catcher)` –

**classmethod register\_inverse\_analytic\_integral** (`func: Callable`)  $\rightarrow$  `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 catchers.

**sample** (`n:` `Union[int, tensorflow.python.framework.ops.Tensor, str]`  $=$  `None`, `limits:` `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`  $=$  `None`, `name: str`  $=$  `'sample'`)  $\rightarrow$  `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, Iter-  
                       able[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, tensor-  
                    flow.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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cache*) –

**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** (*catcher*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** `()` (*catcher*) –

**reset\_cache** (*reseter*: `zfit.util.cache.ZfitCachable`)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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, Iter-
                                     able[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** (*cache*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** **()** (*cache*) –

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

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: zfit.util.cache.ZfitCachable)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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.Distribution,
dist_params, dist_kwargs=None,
name: str = 'Distribution-Constraint', 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** (*catcher*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]]) *Iter-*

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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** (*catcher*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: `zfit.util.cache.ZfitCachable`)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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, It-
                           erable[str], zfit.Space], name: str = None, weights=None)
```

```
classmethod from_tensor (obs: Union[str, Iterable[str], zfit.Space], ten-
                           sor: tensorflow.python.framework.ops.Tensor, weights:
                           Union[tensorflow.python.framework.ops.Tensor, None,
                           numpy.ndarray] = None, name: str = None, dtype: tensor-
                           flow.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,
                           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** (*cache*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])  
Register a *cache* that caches values produces by this instance; a dependent.

**Parameters** () (*cache*) –

**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`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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*) – **lnorm\_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*) – **ldims\_arg\_descr**
- **limits** (*possible*) – **llimits\_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*) – **ldtype\_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](#), [tensorflow.python.framework.ops.Tensor](#)], *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\_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`  $\geq 1$

**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*)  $\rightarrow$  `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]]*)  $\rightarrow$  `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*)  $\rightarrow$  `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]*)  $\rightarrow$  `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*)  $\rightarrow$  `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 caught 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]]) Iter-
    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_ al-
            low_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, Iter-
    able[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, Iter-
    able[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_ al-
      low_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, com-
    plex, 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, depends: 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.

    • depends – 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** (*catcher*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** () (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

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

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

**Parameters** () (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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 convertible to a ZfitParameter

```

zfit.core.operations.multiply_func_func (func1:          zfit.core.interfaces.ZfitFunc,          func2:
                                         zfit.core.interfaces.ZfitFunc, name: str = 'multi-
                                         ply_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 = 'multi-
                                         ply_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** (*catcher*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])

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

**Parameters** **()** (*catcher*) –

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

**Parameters** () (*catcher*) –

**reset\_cache** (*reseter*: zfit.util.cache.ZfitCachable)

**reset\_cache\_self** ()  
Clear the cache of self and all dependent catchers.

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()  
Clear the cache of self and all dependent catchers.

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

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

**Parameters** **()** (*catcher*) –

**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_ al-
          low_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** (*caler*: Union[zfit.core.interfaces.ZfitCachable, Iter-  
able[zfit.core.interfaces.ZfitCachable]])  
Register a *caler* that caches values produces by this instance; a dependent.

**Parameters** () (*caler*) –

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

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter: zfit.util.cache.ZfitCachable*)

**reset\_cache\_self()**

Clear the cache of self and all dependent catchers.

**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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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, col-  
                                         lections=None, validate_shape=True,  
                                         caching_device=None, name=None,  
                                         dtype=None, variable_def=None, im-  
                                         port_scope=None, constraint=None, dis-  
                                         tribute_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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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: tensor-
                                             flow.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** (*catcher*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: zfit.util.cache.ZfitCachable)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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_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\_pdf*s (*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 dictionary.

**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  $\Delta$  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** (*catcher*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])

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

**Parameters** `()` (*catcher*) –

**reset\_cache** (*reseter*: zfit.util.cache.ZfitCachable)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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,
                                                                    equal-
                                                                    i-
                                                                    ties=None,
                                                                    in-
                                                                    equal-
                                                                    i-
                                                                    ties=None,
                                                                    var_to_bounds=None,
                                                                    **op-
                                                                    ti-
                                                                    mizer_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,
                                                                    equali-
                                                                    ties=None,
                                                                    inequali-
                                                                    ties=None,
                                                                    var_to_bounds=None,
                                                                    **opti-
                                                                    mizer_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 com-
ponent 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'`)  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** **()** (*cache*) –

**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_`. 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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** `() (caler)` –

**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  $\exp(\text{lambda} * x)$ .

The function is normalized over a finite range and therefore a pdf. So the PDF is precisely defined as

$$\frac{e^{\lambda \cdot x}}{\int_{\text{lower}}^{\text{upper}} e^{\lambda \cdot x} dx}$$

#### 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'`)  $\rightarrow$  `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `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]`)  $\rightarrow$  `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'`)  $\rightarrow$  `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*) – Prepend 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** (*catcher*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** () (*catcher*) –

**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 catchers.

**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, tensor-
                                                    flow.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, Iter-
                                                    able[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, tensor-
                    flow.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_`. 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*)  $\rightarrow$  *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\_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*)  $\rightarrow$  *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, It-
                                         erable[str], zfit.Space], name: str = 'Gauss')
```

Bases: `zfit.models.dist_tfp.WrapDistribution`

Gaussian or Normal distribution with a mean (mu) and a standartdeviation (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  $[-\infty, \infty]$  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, Iter-
                    able[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, tensor-
                    flow.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**

- (*only\_floating*) – If True, return only the floating parameters.
- (*names*) – 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*) – Prepend 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\_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, tensor-
flow.python.framework.ops.Tensor], sigma:
Union[zfit.core.interfaces.ZfitParameter,
int, float, complex, tensor-
flow.python.framework.ops.Tensor], low:
Union[zfit.core.interfaces.ZfitParameter,
int, float, complex, tensor-
flow.python.framework.ops.Tensor], high:
Union[zfit.core.interfaces.ZfitParameter,
int, float, complex, tensor-
flow.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, tensor-
flow.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_`. 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\_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\_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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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'`)  
 $\rightarrow$  `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'`)  $\rightarrow$  `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'`)  $\rightarrow$  `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** (*caler*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** ( ) (*caler*) –

**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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** () (*cache*) –

**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.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\_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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** () (*cache*) –

**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 *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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cache*) –

**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.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, tensor-
                    flow.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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cache*) –

**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, tensor-
                    flow.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, Iter-
                    able[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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cache*) –

**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\_*. 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*) – Prepend 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\_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*, bool) – 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\_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** (`cache`: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`) *Iter-*

Register a `cache` that caches values produces by this instance; a dependent.

**Parameters** `() (cache)` –

**classmethod register\_inverse\_analytic\_integral** (`func`: `Callable`)  $\rightarrow$  `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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `List[zfit.core.interfaces.ZfitModel]`

**get\_params** (`only_floating: bool = False`, `names: Union[str, List[str], None] = None`)  $\rightarrow$  `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** ()  $\rightarrow$  `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\_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** (`cache`: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`) *Iter-*

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

**Parameters** `() (cache)` –

**classmethod register\_inverse\_analytic\_integral** (`func`: `Callable`)  $\rightarrow$  `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'`)  $\rightarrow$  `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:

$$f(x; \mu, \sigma, \alpha, n) = \begin{cases} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), & \text{for } \frac{x-\mu}{\sigma} \geq -\alpha A \cdot \left(B - \frac{x-\mu}{\sigma}\right)^{-n}, \\ \text{for } \frac{x-\mu}{\sigma} < -\alpha \end{cases}$$



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`)  $\rightarrow$  `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`)  $\rightarrow$  `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*)  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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:

$$f(x; \mu, \sigma, \alpha_L, n_L, \alpha_R, n_R) = \begin{cases} A_L \cdot (B_L - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} < -\alpha_L \exp(-\frac{(x-\mu)^2}{2\sigma^2}), \\ -\alpha_L \leq \text{for } \frac{x-\mu}{\sigma} \leq \alpha_R A_R \cdot (B_R - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} > \alpha_R \end{cases}$$

with

$$A_{L/R} = \left( \frac{n_{L/R}}{|\alpha_{L/R}|} \right)_{L/R}^n \cdot \exp\left(-\frac{|\alpha_{L/R}|^2}{2}\right)$$

$$B_{L/R} = \frac{n_{L/R}}{|\alpha_{L/R}|} - |\alpha_{L/R}|$$

### 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_`. 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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** `() (caler)` –

**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, alphas, nl, alphas, 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

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \\ \text{with } T_0 = 1, T_1 = x$$

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\_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

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \\ \text{with } T_0 = 1, T_1 = 2x$$

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_`. 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**

- (*only\_floating*) – If True, return only the floating parameters.
- (*names*) – 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*) – Prepend 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\_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, Iter-  
                                     able[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, tensor-  
                    flow.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*) – Prepend 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\_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** (*cache*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])

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

**Parameters** **()** (*cache*) –

**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_`. 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\_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** (*cacheder*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cacheder*) –

**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

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \\ \text{with } P_0 = 1, P_1 = x$$

#### 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\_*. 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) – llimits_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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** `() (cache)` –

**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).

$$x_{n+1} = \text{recurrence}(x_n, x_{n-1}, n)$$

**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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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'`)  
 $\rightarrow$  `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'`)  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`) *Iter-*

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

**Parameters** `() (cache)` –

**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: tensor-
                                                  flow.python.framework.ops.Tensor,
                                                  limits: zfit.Space) → tensor-
                                                  flow.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, Iter-
                       able[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, tensor-
                    flow.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\_*. 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*) – Prepend 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*)  $\rightarrow$  *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\_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** (*catcher*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** () (*catcher*) –

**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 catchers.

**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'*)  $\rightarrow$  `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*)  $\rightarrow$  `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\_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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** **()** (*caler*) –

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

**Parameters** () (*cache*) –

**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** (*cache*: *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*

**acquire\_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  $\geq 2$ .
- **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 configured, 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 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*: 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]) → [*tf.Tensor*, *tf.Tensor*]

Calculate using autodiff the gradients of *func()* wrt *params*; also return *func()*.

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:** tuple(*tf.Tensor*, *tf.Tensor*): value and gradient

`zfit.z.math.automatic_value_gradients_hessian` (*func*: Callable, *params*: Iterable[zfit.Parameter]) → [*tf.Tensor*, *tf.Tensor*, *tf.Tensor*]

Calculate using autodiff the gradients and hessian matrix of *func()* wrt *params*; also return *func()*.

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:** 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]` tensor-

## 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]`)  $\rightarrow$  `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*)  $\rightarrow$  `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** (*catcher*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** **()** (*cache*) –

**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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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\_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.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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ( $names=None$ )  $\rightarrow$  `List[zfit.core.interfaces.ZfitModel]`

**get\_params** ( $only\_floating$ : `bool = False`,  $names$ : `Union[str, List[str], None] = None`)  $\rightarrow$  `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\_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.BaseFuncForFunc*

**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'`)  $\rightarrow$  `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`)  $\rightarrow$  `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*)  $\rightarrow$  `List[zfit.core.interfaces.ZfitModel]`

**get\_params** (*only\_floating*: `bool = False`, *names*: `Union[str, List[str], None] = None`)  $\rightarrow$  `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\_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 (catcher: Union[zfit.core.interfaces.ZfitCachable,
    Iterable[zfit.core.interfaces.ZfitCachable]])

```

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

**Parameters** **()** (*catcher*) –

```

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 catchers.

```

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 *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\_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 (catcher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])
    Register a catcher that caches values produces by this instance; a dependent.
    Parameters () (catcher) –
reset_cache (reseter: zfit.util.cache.ZfitCachable)
reset_cache_self ()
    Clear the cache of self and all dependent catchers.
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[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** () (*catcher*) –

**reset\_cache** (*reseter*: `zfit.util.cache.ZfitCachable`)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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** (*catcher*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`) *Iter-*

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

**Parameters** () (*catcher*) –

**reset\_cache** (*reseter*: `zfit.util.cache.ZfitCachable`)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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`)  $\rightarrow$  `zfit.core.interfaces.ZfitObject`

**dtype**

The dtype of the object

**floating**

**get\_dependents** (`only_floating: bool = True`)  $\rightarrow$  `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`)  $\rightarrow$  `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** (`catcher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

Register a `catcher` that caches values produces by this instance; a dependent.

**Parameters** `()` (`catcher`) –

**reset\_cache** (`reseter: zfit.util.cache.ZfitCachable`)

**reset\_cache\_self** ()

Clear the cache of self and all dependent catchers.

**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 loop 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** (*catcher: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter: zfit.util.cache.ZfitCachable*)

**reset\_cache\_self()**

Clear the cache of self and all dependent catchers.

**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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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  $[d_0, \dots, d_{Q-2}, K]$  where  $0 < K \leq P$ .

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

**Parameters** **()** (*catcher*) –

**reset\_cache** (*reseter*: *zfit.util.cache.ZfitCachable*)

**reset\_cache\_self** ()  
 Clear the cache of self and all dependent catchers.

**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` (*acher*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

Parameters () (*acher*) –

`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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** () (*caler*) –

**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='BaseFunc'tor', **kwargs)
```

Bases: `zfit.models.basefunc`tor.`Func`torMixin, `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`)  $\rightarrow$  `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`)  $\rightarrow$  `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*)  $\rightarrow$  `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'`)  $\rightarrow$  `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `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**

- (*only\_floating*) – If True, return only the floating parameters.
- (*names*) – 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\_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  $\exp(\lambda \cdot x)$ .

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\_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, com-
plex, tensorflow.python.framework.ops.Tensor], al-
pha: Union[zfit.core.interfaces.ZfitParameter, int, float,
complex, tensorflow.python.framework.ops.Tensor], n:
Union[zfit.core.interfaces.ZfitParameter, int, float, complex, ten-
sorflow.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:

$$f(x; \mu, \sigma, \alpha, n) = \begin{cases} \exp(-\frac{(x-\mu)^2}{2\sigma^2}), & \text{for } \frac{x-\mu}{\sigma} \geq -\alpha A \cdot (B - \frac{x-\mu}{\sigma})^{-n}, \\ \text{for } \frac{x-\mu}{\sigma} < -\alpha \end{cases}$$

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, Iter-
able[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_`. 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*) – Prepend 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** (*catcher*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*) *Iter-*

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

**Parameters** () (*catcher*) –

**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 catchers.

**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], alphas: 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], alphas: 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:

$$f(x; \mu, \sigma, \alpha_L, n_L, \alpha_R, n_R) = \begin{cases} A_L \cdot (B_L - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} < -\alpha_L \exp(-\frac{(x-\mu)^2}{2\sigma^2}), \\ -\alpha_L \leq \text{for } \frac{x-\mu}{\sigma} \leq \alpha_R A_R \cdot (B_R - \frac{x-\mu}{\sigma})^{-n}, & \text{for } \frac{x-\mu}{\sigma} > \alpha_R \end{cases}$$

with

$$A_{L/R} = \left( \frac{n_{L/R}}{|\alpha_{L/R}|} \right)_{L/R}^n \cdot \exp\left(-\frac{|\alpha_{L/R}|^2}{2}\right)$$

$$B_{L/R} = \frac{n_{L/R}}{|\alpha_{L/R}|} - |\alpha_{L/R}|$$

#### 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\_*. 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*) – Prepend 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) – llimits_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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`) *Iter-*

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

**Parameters** `() (cache)` –

**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  $[-\text{inf}, \text{inf}]$  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\_*. 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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** ()  $\rightarrow$  `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'`)  
 $\rightarrow$  `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'`)  $\rightarrow$  `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'`)  $\rightarrow$  `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** (*caler*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** **()** (*caler*) –

**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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iter-  
able[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*caler*) –

**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`)  $\rightarrow$  `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`)  $\rightarrow$  `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** `()`  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*caler*) –

**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*) – Prepend 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\_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

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \\ \text{with } T_0 = 1, T_1 = x$$

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*) – Prepend 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\_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** (*cache*: Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]])

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

**Parameters** **()** (*cache*) –

**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

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \\ \text{with } P_0 = 1, P_1 = x$$

#### 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'`)  $\rightarrow$  `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `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]`)  $\rightarrow$  `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'`)  $\rightarrow$  `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*) – Prepend 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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iter-  
able[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*caler*) –

**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

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \\ \text{with } T_0 = 1, T_1 = 2x$$

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\_*. 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*) – Prepend 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** (*cache*: `Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]`)

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

**Parameters** `() (cache)` –

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

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

**Parameters** **()** (*cache*) –

**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*) – Prepend 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\_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).

$$x_{n+1} = \text{recurrence}(x_n, x_{n-1}, n)$$

**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\_*. 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*) – Prepend 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** (*cache*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** **()** (*cache*) –

**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'`)  $\rightarrow$  `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `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]`)  $\rightarrow$  `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'`)  $\rightarrow$  `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\_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** (*cacheder*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*cacheder*) –

**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*)  $\rightarrow$  `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*)  $\rightarrow$  `List[zfit.core.interfaces.ZfitModel]`

**get\_params** (*only\_floating: bool = False, names: Union[str, List[str], None] = None*)  $\rightarrow$  `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** ()  $\rightarrow$  `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'*)  $\rightarrow$  `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\_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** (*caler*: *Union[zfit.core.interfaces.ZfitCachable, Iter-  
able[zfit.core.interfaces.ZfitCachable]]*)

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

**Parameters** () (*caler*) –

**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\_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, Iter-  
                       able[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'`)  
 $\rightarrow$  `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'`)  $\rightarrow$  `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'`)  $\rightarrow$  `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** (*cacheder*: *Union*[*zfit.core.interfaces.ZfitCachable*, *Iterable*[*zfit.core.interfaces.ZfitCachable*]])

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

**Parameters** ( ) (*cacheder*) –

**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*) – Prepend 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\_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** (*cache*: *Union[zfit.core.interfaces.ZfitCachable, Iterable[zfit.core.interfaces.ZfitCachable]]*)

Register a *cache* 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*)



### Z

- `zfit`, 31
- `zfit.constraint`, 348
- `zfit.core`, 49
  - `basefunc`, 49
  - `basemodel`, 54
  - `baseobject`, 59
  - `basepdf`, 60
  - `constraint`, 67
  - `data`, 73
  - `dependents`, 83
  - `dimension`, 83
  - `integration`, 85
  - `interfaces`, 88
  - `limits`, 97
  - `loss`, 103
  - `operations`, 108
  - `parameter`, 109
  - `sample`, 138
  - `testing`, 142
- `zfit.data`, 351
- `zfit.func`, 354
- `zfit.loss`, 374
- `zfit.minimize`, 379
- `zfit.minimizers`, 143
  - `base_tf`, 143
  - `baseminimizer`, 143
  - `fitresult`, 144
  - `interface`, 146
  - `minimizer_minuit`, 147
  - `minimizer_tfp`, 149
  - `minimizers_scipy`, 149
  - `optimizers_tf`, 150
  - `tf_external_optimizer`, 150
- `zfit.models`, 154
  - `basefunctor`, 154
  - `basic`, 158
  - `dist_tfp`, 172
  - `functions`, 206
  - `functor`, 232
  - `physics`, 252
  - `polynomials`, 267
  - `special`, 311
- `zfit.param`, 382
- `zfit.pdf`, 396
- `zfit.sample`, 535
- `zfit.settings`, 535
- `zfit.util`, 331
  - `cache`, 331
  - `checks`, 334
  - `container`, 334
  - `diverse`, 335
  - `exception`, 336
  - `execution`, 340
  - `graph`, 340
  - `logging`, 341
  - `temporary`, 341
  - `ztyping`, 342
- `zfit.z`, 342
  - `const`, 342
  - `math`, 343
  - `random`, 345
  - `tools`, 345
  - `wrapping_tf`, 345
  - `zextension`, 346



## Symbols

`__eq__()` (*zfit.core.parameter.TFBaseVariable method*), 128  
`__instancecheck__()` (*zfit.core.parameter.MetaBaseParameter method*), 116  
`__iter__()` (*zfit.Parameter method*), 32  
`__iter__()` (*zfit.core.parameter.Parameter method*), 117  
`__iter__()` (*zfit.core.parameter.TFBaseVariable method*), 128  
`__iter__()` (*zfit.param.Parameter method*), 384  
`__ne__()` (*zfit.Parameter method*), 32  
`__ne__()` (*zfit.core.parameter.Parameter method*), 117  
`__ne__()` (*zfit.core.parameter.TFBaseVariable method*), 128  
`__ne__()` (*zfit.param.Parameter method*), 384  
`__subclasscheck__()` (*zfit.core.parameter.MetaBaseParameter method*), 116

## A

`abs_square()` (in module *zfit.z.zextension*), 346  
`Adam` (class in *zfit.minimize*), 379  
`Adam` (class in *zfit.minimizers.optimizers\_tf*), 150  
`AdamMinimizer` (in module *zfit.minimize*), 379  
`add()` (in module *zfit.core.operations*), 108  
`add()` (*zfit.core.limits.Space method*), 98  
`add()` (*zfit.core.sample.EventSpace method*), 138  
`add()` (*zfit.Space method*), 44  
`add_cache_dependents()` (*zfit.ComplexParameter method*), 42  
`add_cache_dependents()` (*zfit.ComposedParameter method*), 41  
`add_cache_dependents()` (*zfit.constraint.GaussianConstraint method*), 350  
`add_cache_dependents()` (*zfit.constraint.SimpleConstraint method*),

348  
`add_cache_dependents()` (*zfit.core.basefunc.BaseFunc method*), 49  
`add_cache_dependents()` (*zfit.core.basemodel.BaseModel method*), 54  
`add_cache_dependents()` (*zfit.core.baseobject.BaseNumeric method*), 59  
`add_cache_dependents()` (*zfit.core.basepdf.BasePDF method*), 61  
`add_cache_dependents()` (*zfit.core.constraint.BaseConstraint method*), 67  
`add_cache_dependents()` (*zfit.core.constraint.DistributionConstraint method*), 69  
`add_cache_dependents()` (*zfit.core.constraint.GaussianConstraint method*), 70  
`add_cache_dependents()` (*zfit.core.constraint.SimpleConstraint method*), 72  
`add_cache_dependents()` (*zfit.core.data.Data method*), 73  
`add_cache_dependents()` (*zfit.core.data.SampleData method*), 76  
`add_cache_dependents()` (*zfit.core.data.Sampler method*), 79  
`add_cache_dependents()` (*zfit.core.loss.BaseLoss method*), 103  
`add_cache_dependents()` (*zfit.core.loss.CachedLoss method*), 104  
`add_cache_dependents()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 105  
`add_cache_dependents()` (*zfit.core.loss.SimpleLoss method*), 106  
`add_cache_dependents()` (*zfit.core.loss.UnbinnedNLL method*), 107  
`add_cache_dependents()`

[\(zfit.core.parameter.BaseComposedParameter method\), 109](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.BaseZParameter method\), 111](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.ComplexParameter method\), 112](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.ComposedParameter method\), 113](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.ConstantParameter method\), 115](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.Parameter method\), 117](#)  
[add\\_cache\\_dependents \(\) \(zfit.core.parameter.ZfitParameterMixin method\), 136](#)  
[add\\_cache\\_dependents \(\) \(zfit.data.Data method\), 351](#)  
[add\\_cache\\_dependents \(\) \(zfit.func.BaseFunc method\), 354](#)  
[add\\_cache\\_dependents \(\) \(zfit.func.ProdFunc method\), 359](#)  
[add\\_cache\\_dependents \(\) \(zfit.func.SimpleFunc method\), 369](#)  
[add\\_cache\\_dependents \(\) \(zfit.func.SumFunc method\), 364](#)  
[add\\_cache\\_dependents \(\) \(zfit.loss.BaseLoss method\), 377](#)  
[add\\_cache\\_dependents \(\) \(zfit.loss.ExtendedUnbinnedNLL method\), 374](#)  
[add\\_cache\\_dependents \(\) \(zfit.loss.SimpleLoss method\), 378](#)  
[add\\_cache\\_dependents \(\) \(zfit.loss.UnbinnedNLL method\), 375](#)  
[add\\_cache\\_dependents \(\) \(zfit.minimize.Minuit method\), 380](#)  
[add\\_cache\\_dependents \(\) \(zfit.minimizers.minimizer\\_minuit.Minuit method\), 148](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.basefunctor.FunctorMixin method\), 154](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.basic.CustomGaussOLD method\), 158](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.basic.Exponential method\), 165](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.dist\\_tfp.ExponentialTFP method\), 172](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.dist\\_tfp.Gauss method\), 179](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.dist\\_tfp.TruncatedGauss method\), 186](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.dist\\_tfp.Uniform method\), 193](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.dist\\_tfp.WrapDistribution method\), 199](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functions.BaseFunctorFunc method\), 206](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functions.ProdFunc method\), 211](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functions.SimpleFunc method\), 217](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functions.SumFunc method\), 222](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functions.ZFunc method\), 227](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functor.BaseFunctor method\), 232](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functor.ProductPDF method\), 238](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.functor.SumPDF method\), 245](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.physics.CrystalBall method\), 253](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.physics.DoubleCB method\), 260](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.polynomials.Chebyshev method\), 268](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.polynomials.Chebyshev2 method\), 275](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.polynomials.Hermite method\), 282](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.polynomials.Laguerre method\), 289](#)  
[add\\_cache\\_dependents \(\) \(zfit.models.polynomials.Legendre method\), 297](#)



`add_cache_dependents()` (*zfit.models.polynomials.RecursivePolynomial method*), 303  
`add_cache_dependents()` (*zfit.models.special.SimpleFunctorPDF method*), 311  
`add_cache_dependents()` (*zfit.models.special.SimplePDF method*), 318  
`add_cache_dependents()` (*zfit.models.special.ZPDF method*), 324  
`add_cache_dependents()` (*zfit.param.ComplexParameter method*), 395  
`add_cache_dependents()` (*zfit.param.ComposedParameter method*), 394  
`add_cache_dependents()` (*zfit.param.ConstantParameter method*), 382  
`add_cache_dependents()` (*zfit.param.Parameter method*), 384  
`add_cache_dependents()` (*zfit.Parameter method*), 32  
`add_cache_dependents()` (*zfit.pdf.BaseFunctor method*), 403  
`add_cache_dependents()` (*zfit.pdf.BasePDF method*), 396  
`add_cache_dependents()` (*zfit.pdf.Chebyshev method*), 459  
`add_cache_dependents()` (*zfit.pdf.Chebyshev2 method*), 473  
`add_cache_dependents()` (*zfit.pdf.CrystalBall method*), 417  
`add_cache_dependents()` (*zfit.pdf.DoubleCB method*), 425  
`add_cache_dependents()` (*zfit.pdf.Exponential method*), 410  
`add_cache_dependents()` (*zfit.pdf.Gauss method*), 432  
`add_cache_dependents()` (*zfit.pdf.Hermite method*), 481  
`add_cache_dependents()` (*zfit.pdf.Laguerre method*), 488  
`add_cache_dependents()` (*zfit.pdf.Legendre method*), 466  
`add_cache_dependents()` (*zfit.pdf.ProductPDF method*), 501  
`add_cache_dependents()` (*zfit.pdf.RecursivePolynomial method*), 495  
`add_cache_dependents()` (*zfit.pdf.SimpleFunctorPDF method*), 528  
`add_cache_dependents()` (*zfit.pdf.SimplePDF method*), 522  
`add_cache_dependents()` (*zfit.pdf.SumPDF method*), 508  
`add_cache_dependents()` (*zfit.pdf.TruncatedGauss method*), 445  
`add_cache_dependents()` (*zfit.pdf.Uniform method*), 438  
`add_cache_dependents()` (*zfit.pdf.WrapDistribution method*), 452  
`add_cache_dependents()` (*zfit.pdf.ZPDF method*), 515  
`add_cache_dependents()` (*zfit.util.cache.Cachable method*), 332  
`add_cache_dependents()` (*zfit.util.cache.FunctionCacheHolder method*), 333  
`add_cache_dependents()` (*zfit.util.cache.ZfitCachable method*), 334  
`add_constraints()` (*zfit.core.interfaces.ZfitLoss method*), 91  
`add_constraints()` (*zfit.core.loss.BaseLoss method*), 103  
`add_constraints()` (*zfit.core.loss.CachedLoss method*), 104  
`add_constraints()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 105  
`add_constraints()` (*zfit.core.loss.SimpleLoss method*), 106  
`add_constraints()` (*zfit.core.loss.UnbinnedNLL method*), 107  
`add_constraints()` (*zfit.loss.BaseLoss method*), 377  
`add_constraints()` (*zfit.loss.ExtendedUnbinnedNLL method*), 374  
`add_constraints()` (*zfit.loss.SimpleLoss method*), 378  
`add_constraints()` (*zfit.loss.UnbinnedNLL method*), 375  
`add_func_func()` (in module *zfit.core.operations*), 108  
`add_param_func()` (in module *zfit.core.operations*), 108  
`add_param_param()` (in module *zfit.core.operations*), 108  
`add_pdf_pdf()` (in module *zfit.core.operations*), 109  
`add_spaces()` (in module *zfit.core.dimension*), 83  
`aggregation` (*zfit.core.parameter.Parameter attribute*), 117  
`aggregation` (*zfit.core.parameter.TFBaseVariable attribute*), 128  
`aggregation` (*zfit.param.Parameter attribute*), 385  
`aggregation` (*zfit.Parameter attribute*), 32  
`all_parents()` (in module *zfit.util.graph*), 340

AlreadyExtendedPDFError, 336  
 analytic\_integrate()  
     (zfit.core.basefunc.BaseFunc method), 49  
 analytic\_integrate()  
     (zfit.core.basemodel.BaseModel method), 54  
 analytic\_integrate()  
     (zfit.core.basepdf.BasePDF method), 61  
 analytic\_integrate() (zfit.func.BaseFunc method), 354  
 analytic\_integrate() (zfit.func.ProdFunc method), 359  
 analytic\_integrate() (zfit.func.SimpleFunc method), 369  
 analytic\_integrate() (zfit.func.SumFunc method), 364  
 analytic\_integrate()  
     (zfit.models.basefunc.FunctorMixin method), 154  
 analytic\_integrate()  
     (zfit.models.basic.CustomGaussOLD method), 159  
 analytic\_integrate()  
     (zfit.models.basic.Exponential method), 165  
 analytic\_integrate()  
     (zfit.models.dist\_tfp.ExponentialTFP method), 172  
 analytic\_integrate() (zfit.models.dist\_tfp.Gauss method), 179  
 analytic\_integrate()  
     (zfit.models.dist\_tfp.TruncatedGauss method), 186  
 analytic\_integrate()  
     (zfit.models.dist\_tfp.Uniform method), 193  
 analytic\_integrate()  
     (zfit.models.dist\_tfp.WrapDistribution method), 200  
 analytic\_integrate()  
     (zfit.models.functions.BaseFuncorFunc method), 207  
 analytic\_integrate()  
     (zfit.models.functions.ProdFunc method), 212  
 analytic\_integrate()  
     (zfit.models.functions.SimpleFunc method), 217  
 analytic\_integrate()  
     (zfit.models.functions.SumFunc method), 222  
 analytic\_integrate()  
     (zfit.models.functions.ZFunc method), 227  
 analytic\_integrate()  
     (zfit.models.functor.BaseFuncor method), 232  
 analytic\_integrate()  
     (zfit.models.functor.ProductPDF method), 239  
 analytic\_integrate()  
     (zfit.models.functor.SumPDF method), 246  
 analytic\_integrate()  
     (zfit.models.physics.CrystalBall method), 253  
 analytic\_integrate()  
     (zfit.models.physics.DoubleCB method), 261  
 analytic\_integrate()  
     (zfit.models.polynomials.Chebyshev method), 268  
 analytic\_integrate()  
     (zfit.models.polynomials.Chebyshev2 method), 275  
 analytic\_integrate()  
     (zfit.models.polynomials.Hermite method), 282  
 analytic\_integrate()  
     (zfit.models.polynomials.Laguerre method), 289  
 analytic\_integrate()  
     (zfit.models.polynomials.Legendre method), 297  
 analytic\_integrate()  
     (zfit.models.polynomials.RecursivePolynomial method), 304  
 analytic\_integrate()  
     (zfit.models.special.SimpleFuncorPDF method), 311  
 analytic\_integrate()  
     (zfit.models.special.SimplePDF method), 318  
 analytic\_integrate() (zfit.models.special.ZPDF method), 325  
 analytic\_integrate() (zfit.pdf.BaseFuncor method), 403  
 analytic\_integrate() (zfit.pdf.BasePDF method), 397  
 analytic\_integrate() (zfit.pdf.Chebyshev method), 459  
 analytic\_integrate() (zfit.pdf.Chebyshev2 method), 474  
 analytic\_integrate() (zfit.pdf.CrystalBall method), 417  
 analytic\_integrate() (zfit.pdf.DoubleCB method), 425  
 analytic\_integrate() (zfit.pdf.Exponential method), 410  
 analytic\_integrate() (zfit.pdf.Gauss method), 432  
 analytic\_integrate() (zfit.pdf.Hermite method),

- 481  
`analytic_integrate()` (*zfit.pdf.Laguerre method*), 488  
`analytic_integrate()` (*zfit.pdf.Legendre method*), 466  
`analytic_integrate()` (*zfit.pdf.ProductPDF method*), 501  
`analytic_integrate()` (*zfit.pdf.RecursivePolynomial method*), 495  
`analytic_integrate()` (*zfit.pdf.SimpleFunctorPDF method*), 528  
`analytic_integrate()` (*zfit.pdf.SimplePDF method*), 522  
`analytic_integrate()` (*zfit.pdf.SumPDF method*), 509  
`analytic_integrate()` (*zfit.pdf.TruncatedGauss method*), 446  
`analytic_integrate()` (*zfit.pdf.Uniform method*), 439  
`analytic_integrate()` (*zfit.pdf.WrapDistribution method*), 452  
`analytic_integrate()` (*zfit.pdf.ZPDF method*), 515  
AnalyticIntegral (class in *zfit.core.integration*), 85  
Any (class in *zfit.core.limits*), 97  
ANY (*zfit.core.limits.Space* attribute), 98  
ANY (*zfit.core.sample.EventSpace* attribute), 138  
ANY (*zfit.Space* attribute), 44  
ANY\_LOWER (*zfit.core.limits.Space* attribute), 98  
ANY\_LOWER (*zfit.core.sample.EventSpace* attribute), 138  
ANY\_LOWER (*zfit.Space* attribute), 44  
ANY\_UPPER (*zfit.core.limits.Space* attribute), 98  
ANY\_UPPER (*zfit.core.sample.EventSpace* attribute), 138  
ANY\_UPPER (*zfit.Space* attribute), 44  
AnyLower (class in *zfit.core.limits*), 97  
AnyUpper (class in *zfit.core.limits*), 97  
`apply_yield()` (*zfit.core.basepdf.BasePDF method*), 61  
`apply_yield()` (*zfit.models.basic.CustomGaussOLD method*), 159  
`apply_yield()` (*zfit.models.basic.Exponential method*), 166  
`apply_yield()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 173  
`apply_yield()` (*zfit.models.dist\_tfp.Gauss method*), 180  
`apply_yield()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 187  
`apply_yield()` (*zfit.models.dist\_tfp.Uniform method*), 193  
`apply_yield()` (*zfit.models.dist\_tfp.WrapDistribution method*), 200  
`apply_yield()` (*zfit.models.functor.BaseFunctor method*), 232  
`apply_yield()` (*zfit.models.functor.ProductPDF method*), 239  
`apply_yield()` (*zfit.models.functor.SumPDF method*), 246  
`apply_yield()` (*zfit.models.physics.CrystalBall method*), 253  
`apply_yield()` (*zfit.models.physics.DoubleCB method*), 261  
`apply_yield()` (*zfit.models.polynomials.Chebyshev method*), 268  
`apply_yield()` (*zfit.models.polynomials.Chebyshev2 method*), 276  
`apply_yield()` (*zfit.models.polynomials.Hermite method*), 283  
`apply_yield()` (*zfit.models.polynomials.Laguerre method*), 290  
`apply_yield()` (*zfit.models.polynomials.Legendre method*), 297  
`apply_yield()` (*zfit.models.polynomials.RecursivePolynomial method*), 304  
`apply_yield()` (*zfit.models.special.SimpleFunctorPDF method*), 312  
`apply_yield()` (*zfit.models.special.SimplePDF method*), 318  
`apply_yield()` (*zfit.models.special.ZPDF method*), 325  
`apply_yield()` (*zfit.pdf.BaseFunctor method*), 404  
`apply_yield()` (*zfit.pdf.BasePDF method*), 397  
`apply_yield()` (*zfit.pdf.Chebyshev method*), 460  
`apply_yield()` (*zfit.pdf.Chebyshev2 method*), 474  
`apply_yield()` (*zfit.pdf.CrystalBall method*), 418  
`apply_yield()` (*zfit.pdf.DoubleCB method*), 425  
`apply_yield()` (*zfit.pdf.Exponential method*), 410  
`apply_yield()` (*zfit.pdf.Gauss method*), 432  
`apply_yield()` (*zfit.pdf.Hermite method*), 481  
`apply_yield()` (*zfit.pdf.Laguerre method*), 488  
`apply_yield()` (*zfit.pdf.Legendre method*), 467  
`apply_yield()` (*zfit.pdf.ProductPDF method*), 502  
`apply_yield()` (*zfit.pdf.RecursivePolynomial method*), 495  
`apply_yield()` (*zfit.pdf.SimpleFunctorPDF method*), 529  
`apply_yield()` (*zfit.pdf.SimplePDF method*), 522  
`apply_yield()` (*zfit.pdf.SumPDF method*), 509  
`apply_yield()` (*zfit.pdf.TruncatedGauss method*), 446  
`apply_yield()` (*zfit.pdf.Uniform method*), 439  
`apply_yield()` (*zfit.pdf.WrapDistribution method*), 453  
`apply_yield()` (*zfit.pdf.ZPDF method*), 516  
`acquire_cpu()` (*zfit.util.execution.RunManager method*), 340  
`area()` (*zfit.core.interfaces.ZfitSpace method*), 95  
`area()` (*zfit.core.limits.Space method*), 98

`area()` (`zfit.core.sample.EventSpace` method), 138  
`area()` (`zfit.Space` method), 44  
`arg` (`zfit.ComplexParameter` attribute), 42  
`arg` (`zfit.core.parameter.ComplexParameter` attribute), 112  
`arg` (`zfit.param.ComplexParameter` attribute), 395  
`args` (`zfit.minimizers.baseminimizer.FailMinimizeNaN` attribute), 144  
`args` (`zfit.util.exception.AlreadyExtendedPDFError` attribute), 336  
`args` (`zfit.util.exception.AxesNotSpecifiedError` attribute), 336  
`args` (`zfit.util.exception.AxesNotUnambiguousError` attribute), 336  
`args` (`zfit.util.exception.BasePDFSubclassingError` attribute), 336  
`args` (`zfit.util.exception.BreakingAPIChangeError` attribute), 336  
`args` (`zfit.util.exception.ConversionError` attribute), 336  
`args` (`zfit.util.exception.ExtendedPDFError` attribute), 336  
`args` (`zfit.util.exception.IncompatibleError` attribute), 336  
`args` (`zfit.util.exception.IntentionNotUnambiguousError` attribute), 337  
`args` (`zfit.util.exception.LimitsIncompatibleError` attribute), 337  
`args` (`zfit.util.exception.LimitsNotSpecifiedError` attribute), 337  
`args` (`zfit.util.exception.LimitsOverdefinedError` attribute), 337  
`args` (`zfit.util.exception.LimitsUnderdefinedError` attribute), 337  
`args` (`zfit.util.exception.LogicalUndefinedOperationError` attribute), 337  
`args` (`zfit.util.exception.ModelIncompatibleError` attribute), 337  
`args` (`zfit.util.exception.MultipleLimitsNotImplementedError` attribute), 337  
`args` (`zfit.util.exception.NameAlreadyTakenError` attribute), 338  
`args` (`zfit.util.exception.NormRangeNotImplementedError` attribute), 338  
`args` (`zfit.util.exception.NormRangeNotSpecifiedError` attribute), 338  
`args` (`zfit.util.exception.NoSessionSpecifiedError` attribute), 338  
`args` (`zfit.util.exception.NotExtendedPDFError` attribute), 338  
`args` (`zfit.util.exception.NotMinimizedError` attribute), 338  
`args` (`zfit.util.exception.NotSpecifiedError` attribute), 338  
`args` (`zfit.util.exception.ObsIncompatibleError` attribute), 338  
`args` (`zfit.util.exception.ObsNotSpecifiedError` attribute), 339  
`args` (`zfit.util.exception.OverdefinedError` attribute), 339  
`args` (`zfit.util.exception.PDFCompatibilityError` attribute), 339  
`args` (`zfit.util.exception.ShapeIncompatibleError` attribute), 339  
`args` (`zfit.util.exception.SpaceIncompatibleError` attribute), 339  
`args` (`zfit.util.exception.SubclassingError` attribute), 339  
`args` (`zfit.util.exception.UnderdefinedError` attribute), 339  
`args` (`zfit.util.exception.WeightsNotImplementedError` attribute), 339  
`args` (`zfit.util.exception.WorkInProgressError` attribute), 340  
`as_func()` (`zfit.core.basepdf.BasePDF` method), 62  
`as_func()` (`zfit.core.interfaces.ZfitPDF` method), 93  
`as_func()` (`zfit.models.basic.CustomGaussOLD` method), 159  
`as_func()` (`zfit.models.basic.Exponential` method), 166  
`as_func()` (`zfit.models.dist_tfp.ExponentialTFP` method), 173  
`as_func()` (`zfit.models.dist_tfp.Gauss` method), 180  
`as_func()` (`zfit.models.dist_tfp.TruncatedGauss` method), 187  
`as_func()` (`zfit.models.dist_tfp.Uniform` method), 194  
`as_func()` (`zfit.models.dist_tfp.WrapDistribution` method), 200  
`as_func()` (`zfit.models.functor.BaseFunctor` method), 233  
`as_func()` (`zfit.models.functor.ProductPDF` method), 239  
`as_func()` (`zfit.models.functor.SumPDF` method), 246  
`as_func()` (`zfit.models.physics.CrystalBall` method), 254  
`as_func()` (`zfit.models.physics.DoubleCB` method), 261  
`as_func()` (`zfit.models.polynomials.Chebyshev` method), 269  
`as_func()` (`zfit.models.polynomials.Chebyshev2` method), 276  
`as_func()` (`zfit.models.polynomials.Hermite` method), 283  
`as_func()` (`zfit.models.polynomials.Laguerre` method), 290  
`as_func()` (`zfit.models.polynomials.Legendre` method), 297  
`as_func()` (`zfit.models.polynomials.RecursivePolynomial` method), 304  
`as_func()` (`zfit.models.special.SimpleFunctorPDF` method), 312



- `as_func()` (*zfit.models.special.SimplePDF* method), 319
- `as_func()` (*zfit.models.special.ZPDF* method), 325
- `as_func()` (*zfit.pdf.BaseFunctor* method), 404
- `as_func()` (*zfit.pdf.BasePDF* method), 397
- `as_func()` (*zfit.pdf.Chebyshev* method), 460
- `as_func()` (*zfit.pdf.Chebyshev2* method), 474
- `as_func()` (*zfit.pdf.CrystalBall* method), 418
- `as_func()` (*zfit.pdf.DoubleCB* method), 425
- `as_func()` (*zfit.pdf.Exponential* method), 411
- `as_func()` (*zfit.pdf.Gauss* method), 432
- `as_func()` (*zfit.pdf.Hermite* method), 481
- `as_func()` (*zfit.pdf.Laguerre* method), 488
- `as_func()` (*zfit.pdf.Legendre* method), 467
- `as_func()` (*zfit.pdf.ProductPDF* method), 502
- `as_func()` (*zfit.pdf.RecursivePolynomial* method), 495
- `as_func()` (*zfit.pdf.SimpleFunctorPDF* method), 529
- `as_func()` (*zfit.pdf.SimplePDF* method), 522
- `as_func()` (*zfit.pdf.SumPDF* method), 509
- `as_func()` (*zfit.pdf.TruncatedGauss* method), 446
- `as_func()` (*zfit.pdf.Uniform* method), 439
- `as_func()` (*zfit.pdf.WrapDistribution* method), 453
- `as_func()` (*zfit.pdf.ZPDF* method), 516
- `as_pdf()` (*zfit.core.basefunc.BaseFunc* method), 49
- `as_pdf()` (*zfit.core.interfaces.ZfitFunc* method), 89
- `as_pdf()` (*zfit.func.BaseFunc* method), 355
- `as_pdf()` (*zfit.func.ProdFunc* method), 360
- `as_pdf()` (*zfit.func.SimpleFunc* method), 370
- `as_pdf()` (*zfit.func.SumFunc* method), 365
- `as_pdf()` (*zfit.models.functions.BaseFunctorFunc* method), 207
- `as_pdf()` (*zfit.models.functions.ProdFunc* method), 212
- `as_pdf()` (*zfit.models.functions.SimpleFunc* method), 217
- `as_pdf()` (*zfit.models.functions.SumFunc* method), 222
- `as_pdf()` (*zfit.models.functions.ZFunc* method), 227
- `assign()` (*zfit.ComplexParameter* method), 43
- `assign()` (*zfit.ComposedParameter* method), 41
- `assign()` (*zfit.core.parameter.BaseComposedParameter* method), 110
- `assign()` (*zfit.core.parameter.BaseZParameter* method), 111
- `assign()` (*zfit.core.parameter.ComplexParameter* method), 112
- `assign()` (*zfit.core.parameter.ComposedParameter* method), 114
- `assign()` (*zfit.core.parameter.ComposedVariable* method), 115
- `assign()` (*zfit.core.parameter.ConstantParameter* method), 115
- `assign()` (*zfit.core.parameter.Parameter* method), 117
- `assign()` (*zfit.core.parameter.TFBaseVariable* method), 128
- `assign()` (*zfit.core.parameter.ZfitBaseVariable* method), 136
- `assign()` (*zfit.param.ComplexParameter* method), 395
- `assign()` (*zfit.param.ComposedParameter* method), 394
- `assign()` (*zfit.param.ConstantParameter* method), 383
- `assign()` (*zfit.param.Parameter* method), 385
- `assign()` (*zfit.Parameter* method), 32
- `assign_add()` (*zfit.core.parameter.Parameter* method), 118
- `assign_add()` (*zfit.core.parameter.TFBaseVariable* method), 129
- `assign_add()` (*zfit.param.Parameter* method), 385
- `assign_add()` (*zfit.Parameter* method), 33
- `assign_sub()` (*zfit.core.parameter.Parameter* method), 118
- `assign_sub()` (*zfit.core.parameter.TFBaseVariable* method), 129
- `assign_sub()` (*zfit.param.Parameter* method), 385
- `assign_sub()` (*zfit.Parameter* method), 33
- `AUTO_FILL` (*zfit.core.limits.Space* attribute), 98
- `AUTO_FILL` (*zfit.core.sample.EventSpace* attribute), 138
- `AUTO_FILL` (*zfit.Space* attribute), 44
- `auto_integrate()` (in module *zfit.core.integration*), 87
- `autodiff_gradient()` (in module *zfit.z.math*), 343
- `autodiff_hessian()` (in module *zfit.z.math*), 343
- `autodiff_value_gradients()` (in module *zfit.z.math*), 343
- `automatic_value_gradients_hessian()` (in module *zfit.z.math*), 343
- `axes` (*zfit.core.basefunc.BaseFunc* attribute), 50
- `axes` (*zfit.core.basemodel.BaseModel* attribute), 55
- `axes` (*zfit.core.basepdf.BasePDF* attribute), 62
- `axes` (*zfit.core.data.Data* attribute), 73
- `axes` (*zfit.core.data.SampleData* attribute), 76
- `axes` (*zfit.core.data.Sampler* attribute), 80
- `axes` (*zfit.core.dimension.BaseDimensional* attribute), 83
- `axes` (*zfit.core.integration.PartialIntegralSampleData* attribute), 86
- `axes` (*zfit.core.interfaces.ZfitData* attribute), 88
- `axes` (*zfit.core.interfaces.ZfitDimensional* attribute), 88
- `axes` (*zfit.core.interfaces.ZfitFunc* attribute), 89
- `axes` (*zfit.core.interfaces.ZfitModel* attribute), 91
- `axes` (*zfit.core.interfaces.ZfitPDF* attribute), 93
- `axes` (*zfit.core.interfaces.ZfitSpace* attribute), 95
- `axes` (*zfit.core.limits.Space* attribute), 98
- `axes` (*zfit.core.sample.EventSpace* attribute), 138
- `axes` (*zfit.data.Data* attribute), 351
- `axes` (*zfit.func.BaseFunc* attribute), 355
- `axes` (*zfit.func.ProdFunc* attribute), 360
- `axes` (*zfit.func.SimpleFunc* attribute), 370
- `axes` (*zfit.func.SumFunc* attribute), 365

axes (*zfit.models.basefunctor.FunctorMixin* attribute), 154

axes (*zfit.models.basic.CustomGaussOLD* attribute), 159

axes (*zfit.models.basic.Exponential* attribute), 166

axes (*zfit.models.dist\_tfp.ExponentialTFP* attribute), 173

axes (*zfit.models.dist\_tfp.Gauss* attribute), 180

axes (*zfit.models.dist\_tfp.TruncatedGauss* attribute), 187

axes (*zfit.models.dist\_tfp.Uniform* attribute), 194

axes (*zfit.models.dist\_tfp.WrapDistribution* attribute), 200

axes (*zfit.models.functions.BaseFunctorFunc* attribute), 207

axes (*zfit.models.functions.ProdFunc* attribute), 212

axes (*zfit.models.functions.SimpleFunc* attribute), 217

axes (*zfit.models.functions.SumFunc* attribute), 222

axes (*zfit.models.functions.ZFunc* attribute), 227

axes (*zfit.models.functor.BaseFunctor* attribute), 233

axes (*zfit.models.functor.ProductPDF* attribute), 239

axes (*zfit.models.functor.SumPDF* attribute), 246

axes (*zfit.models.physics.CrystalBall* attribute), 254

axes (*zfit.models.physics.DoubleCB* attribute), 261

axes (*zfit.models.polynomials.Chebyshev* attribute), 269

axes (*zfit.models.polynomials.Chebyshev2* attribute), 276

axes (*zfit.models.polynomials.Hermite* attribute), 283

axes (*zfit.models.polynomials.Laguerre* attribute), 290

axes (*zfit.models.polynomials.Legendre* attribute), 297

axes (*zfit.models.polynomials.RecursivePolynomial* attribute), 304

axes (*zfit.models.special.SimpleFunctorPDF* attribute), 312

axes (*zfit.models.special.SimplePDF* attribute), 319

axes (*zfit.models.special.ZPDF* attribute), 325

axes (*zfit.pdf.BaseFunctor* attribute), 404

axes (*zfit.pdf.BasePDF* attribute), 397

axes (*zfit.pdf.Chebyshev* attribute), 460

axes (*zfit.pdf.Chebyshev2* attribute), 474

axes (*zfit.pdf.CrystalBall* attribute), 418

axes (*zfit.pdf.DoubleCB* attribute), 426

axes (*zfit.pdf.Exponential* attribute), 411

axes (*zfit.pdf.Gauss* attribute), 432

axes (*zfit.pdf.Hermite* attribute), 481

axes (*zfit.pdf.Laguerre* attribute), 489

axes (*zfit.pdf.Legendre* attribute), 467

axes (*zfit.pdf.ProductPDF* attribute), 502

axes (*zfit.pdf.RecursivePolynomial* attribute), 496

axes (*zfit.pdf.SimpleFunctorPDF* attribute), 529

axes (*zfit.pdf.SimplePDF* attribute), 522

axes (*zfit.pdf.SumPDF* attribute), 509

axes (*zfit.pdf.TruncatedGauss* attribute), 446

axes (*zfit.pdf.Uniform* attribute), 439

axes (*zfit.pdf.WrapDistribution* attribute), 453

axes (*zfit.pdf.ZPDF* attribute), 516

axes (*zfit.Space* attribute), 44

AxesNotSpecifiedError, 336

AxesNotUnambiguousError, 336

## B

BaseComposedParameter (class in *zfit.core.parameter*), 109

BaseConstraint (class in *zfit.core.constraint*), 67

BaseDependentsMixin (class in *zfit.core.dependents*), 83

BaseDimensional (class in *zfit.core.dimension*), 83

BaseFunc (class in *zfit.core.basefunc*), 49

BaseFunc (class in *zfit.func*), 354

BaseFunctor (class in *zfit.models.functor*), 232

BaseFunctor (class in *zfit.pdf*), 403

BaseFunctorFunc (class in *zfit.models.functions*), 206

BaseLoss (class in *zfit.core.loss*), 103

BaseLoss (class in *zfit.loss*), 376

BaseMinimizer (class in *zfit.minimizers.baseminimizer*), 143

BaseModel (class in *zfit.core.basemodel*), 54

BaseNumeric (class in *zfit.core.baseobject*), 59

BaseObject (class in *zfit.core.baseobject*), 60

BaseParameter (class in *zfit.core.parameter*), 111

BasePDF (class in *zfit.core.basepdf*), 61

BasePDF (class in *zfit.pdf*), 396

BasePDFSubclassingError, 336

BaseStrategy (class in *zfit.minimizers.baseminimizer*), 144

BaseZParameter (class in *zfit.core.parameter*), 111

batch\_scatter\_update () (*zfit.core.parameter.Parameter* method), 118

batch\_scatter\_update () (*zfit.core.parameter.TFBaseVariable* method), 129

batch\_scatter\_update () (*zfit.param.Parameter* method), 386

batch\_scatter\_update () (*zfit.Parameter* method), 33

BFGS (class in *zfit.minimize*), 381

BFGS (class in *zfit.minimizers.minimizer\_tfp*), 149

BreakingAPIChangeError, 336

## C

Cachable (class in *zfit.util.cache*), 332

CachedLoss (class in *zfit.core.loss*), 104

Chebyshev (class in *zfit.models.polynomials*), 267

Chebyshev (class in *zfit.pdf*), 459

Chebyshev2 (class in *zfit.models.polynomials*), 274

Chebyshev2 (class in *zfit.pdf*), 473

`chebyshev2_shape()` (in module `zfit.models.polynomials`), 310  
`chebyshev_shape()` (in module `zfit.models.polynomials`), 310  
`check_numerics()` (in module `zfit.z.wrapping_tf`), 345  
`check_wrapped_functions_registered()` (`zfit.z.zextension.FunctionWrapperRegistry` class method), 346  
`chunked_average()` (in module `zfit.core.integration`), 87  
`chunksize` (`zfit.util.execution.RunManager` attribute), 340  
`clear()` (`zfit.util.container.DotDict` method), 334  
`combine()` (`zfit.core.limits.Space` method), 98  
`combine()` (`zfit.core.sample.EventSpace` method), 138  
`combine()` (`zfit.Space` method), 44  
`combine_spaces()` (in module `zfit.core.dimension`), 84  
`common_obs()` (in module `zfit.core.dimension`), 84  
`complex()` (in module `zfit.z.wrapping_tf`), 346  
`ComplexParameter` (class in `zfit`), 42  
`ComplexParameter` (class in `zfit.core.parameter`), 112  
`ComplexParameter` (class in `zfit.param`), 395  
`ComposedParameter` (class in `zfit`), 41  
`ComposedParameter` (class in `zfit.core.parameter`), 113  
`ComposedParameter` (class in `zfit.param`), 394  
`ComposedVariable` (class in `zfit.core.parameter`), 115  
`conj` (`zfit.ComplexParameter` attribute), 43  
`conj` (`zfit.core.parameter.ComplexParameter` attribute), 112  
`conj` (`zfit.param.ComplexParameter` attribute), 395  
`constant()` (in module `zfit.z.zextension`), 346  
`ConstantParameter` (class in `zfit.core.parameter`), 115  
`ConstantParameter` (class in `zfit.param`), 382  
`constraint` (`zfit.core.parameter.Parameter` attribute), 119  
`constraint` (`zfit.core.parameter.TFBaseVariable` attribute), 130  
`constraint` (`zfit.param.Parameter` attribute), 386  
`constraint` (`zfit.Parameter` attribute), 34  
`constraints` (`zfit.core.loss.BaseLoss` attribute), 103  
`constraints` (`zfit.core.loss.CachedLoss` attribute), 104  
`constraints` (`zfit.core.loss.ExtendedUnbinnedNLL` attribute), 105  
`constraints` (`zfit.core.loss.SimpleLoss` attribute), 106  
`constraints` (`zfit.core.loss.UnbinnedNLL` attribute), 107  
`constraints` (`zfit.loss.BaseLoss` attribute), 377  
`constraints` (`zfit.loss.ExtendedUnbinnedNLL` attribute), 374  
`constraints` (`zfit.loss.SimpleLoss` attribute), 378  
`constraints` (`zfit.loss.UnbinnedNLL` attribute), 375  
`contains_tensor()` (in module `zfit.core.limits`), 102  
`converged` (`zfit.minimizers.fitresult.FitResult` attribute), 145  
`ConversionError`, 336  
`convert_coeffs_dict_to_list()` (in module `zfit.models.polynomials`), 310  
`convert_func_to_pdf()` (in module `zfit.core.operations`), 109  
`convert_pdf_to_func()` (in module `zfit.core.operations`), 109  
`convert_sort_space()` (`zfit.core.basefunc.BaseFunc` method), 50  
`convert_sort_space()` (`zfit.core.basemodel.BaseModel` method), 55  
`convert_sort_space()` (`zfit.core.basepdf.BasePDF` method), 62  
`convert_sort_space()` (`zfit.core.data.Data` method), 73  
`convert_sort_space()` (`zfit.core.data.SampleData` method), 77  
`convert_sort_space()` (`zfit.core.data.Sampler` method), 80  
`convert_sort_space()` (`zfit.data.Data` method), 351  
`convert_sort_space()` (`zfit.func.BaseFunc` method), 355  
`convert_sort_space()` (`zfit.func.ProdFunc` method), 360  
`convert_sort_space()` (`zfit.func.SimpleFunc` method), 370  
`convert_sort_space()` (`zfit.func.SumFunc` method), 365  
`convert_sort_space()` (`zfit.models.basefunc.FunctorMixin` method), 154  
`convert_sort_space()` (`zfit.models.basic.CustomGaussOLD` method), 159  
`convert_sort_space()` (`zfit.models.basic.Exponential` method), 166  
`convert_sort_space()` (`zfit.models.dist_tfp.ExponentialTFP` method), 173  
`convert_sort_space()` (`zfit.models.dist_tfp.Gauss` method), 180  
`convert_sort_space()` (`zfit.models.dist_tfp.TruncatedGauss` method), 187  
`convert_sort_space()`

`(zfit.models.dist_tfp.Uniform method)`, 194  
`convert_sort_space()`  
`(zfit.models.dist_tfp.WrapDistribution method)`, 200  
`convert_sort_space()`  
`(zfit.models.functions.BaseFuncFunc method)`, 207  
`convert_sort_space()`  
`(zfit.models.functions.ProdFunc method)`, 212  
`convert_sort_space()`  
`(zfit.models.functions.SimpleFunc method)`, 217  
`convert_sort_space()`  
`(zfit.models.functions.SumFunc method)`, 222  
`convert_sort_space()`  
`(zfit.models.functions.ZFunc method)`, 227  
`convert_sort_space()`  
`(zfit.models.functor.BaseFuncFunc method)`, 233  
`convert_sort_space()`  
`(zfit.models.functor.ProductPDF method)`, 239  
`convert_sort_space()`  
`(zfit.models.functor.SumPDF method)`, 246  
`convert_sort_space()`  
`(zfit.models.physics.CrystalBall method)`, 254  
`convert_sort_space()`  
`(zfit.models.physics.DoubleCB method)`, 261  
`convert_sort_space()`  
`(zfit.models.polynomials.Chebyshev method)`, 269  
`convert_sort_space()`  
`(zfit.models.polynomials.Chebyshev2 method)`, 276  
`convert_sort_space()`  
`(zfit.models.polynomials.Hermite method)`, 283  
`convert_sort_space()`  
`(zfit.models.polynomials.Laguerre method)`, 290  
`convert_sort_space()`  
`(zfit.models.polynomials.Legendre method)`, 297  
`convert_sort_space()`  
`(zfit.models.polynomials.RecursivePolynomial method)`, 304  
`convert_sort_space()`  
`(zfit.models.special.SimpleFuncPDF method)`, 312  
`convert_sort_space()`  
`(zfit.models.special.SimplePDF method)`, 319  
`convert_sort_space()` `(zfit.models.special.ZPDF method)`, 325  
`convert_sort_space()` `(zfit.pdf.BaseFuncFunc method)`, 404  
`convert_sort_space()` `(zfit.pdf.BasePDF method)`, 397  
`convert_sort_space()` `(zfit.pdf.Chebyshev method)`, 460  
`convert_sort_space()` `(zfit.pdf.Chebyshev2 method)`, 474  
`convert_sort_space()` `(zfit.pdf.CrystalBall method)`, 418  
`convert_sort_space()` `(zfit.pdf.DoubleCB method)`, 426  
`convert_sort_space()` `(zfit.pdf.Exponential method)`, 411  
`convert_sort_space()` `(zfit.pdf.Gauss method)`, 432  
`convert_sort_space()` `(zfit.pdf.Hermite method)`, 482  
`convert_sort_space()` `(zfit.pdf.Laguerre method)`, 489  
`convert_sort_space()` `(zfit.pdf.Legendre method)`, 467  
`convert_sort_space()` `(zfit.pdf.ProductPDF method)`, 502  
`convert_sort_space()` `(zfit.pdf.RecursivePolynomial method)`, 496  
`convert_sort_space()` `(zfit.pdf.SimpleFuncPDF method)`, 529  
`convert_sort_space()` `(zfit.pdf.SimplePDF method)`, 523  
`convert_sort_space()` `(zfit.pdf.SumPDF method)`, 509  
`convert_sort_space()` `(zfit.pdf.TruncatedGauss method)`, 446  
`convert_sort_space()` `(zfit.pdf.Uniform method)`, 439  
`convert_sort_space()` `(zfit.pdf.WrapDistribution method)`, 453  
`convert_sort_space()` `(zfit.pdf.ZPDF method)`, 516  
`convert_to_container()` `(in module zfit.util.container)`, 335  
`convert_to_obs_str()` `(in module zfit.core.limits)`, 102  
`convert_to_parameter()` `(in module zfit)`, 44  
`convert_to_parameter()` `(in module zfit.core.parameter)`, 137  
`convert_to_parameter()` `(in module zfit.param)`, 396  
`convert_to_space()` `(in module zfit)`, 48



`convert_to_space()` (in module `zfit.core.limits`), 102  
`convert_to_tensor()` (in module `zfit.z.zextension`), 346  
`copy()` (`zfit.ComplexParameter` method), 43  
`copy()` (`zfit.ComposedParameter` method), 41  
`copy()` (`zfit.constraint.GaussianConstraint` method), 350  
`copy()` (`zfit.constraint.SimpleConstraint` method), 348  
`copy()` (`zfit.core.basefunc.BaseFunc` method), 50  
`copy()` (`zfit.core.basemodel.BaseModel` method), 55  
`copy()` (`zfit.core.baseobject.BaseNumeric` method), 59  
`copy()` (`zfit.core.baseobject.BaseObject` method), 60  
`copy()` (`zfit.core.basepdf.BasePDF` method), 62  
`copy()` (`zfit.core.constraint.BaseConstraint` method), 68  
`copy()` (`zfit.core.constraint.DistributionConstraint` method), 69  
`copy()` (`zfit.core.constraint.GaussianConstraint` method), 70  
`copy()` (`zfit.core.constraint.SimpleConstraint` method), 72  
`copy()` (`zfit.core.data.Data` method), 74  
`copy()` (`zfit.core.data.SampleData` method), 77  
`copy()` (`zfit.core.data.Sampler` method), 80  
`copy()` (`zfit.core.dimension.BaseDimensional` method), 83  
`copy()` (`zfit.core.integration.PartialIntegralSampleData` method), 86  
`copy()` (`zfit.core.interfaces.ZfitData` method), 88  
`copy()` (`zfit.core.interfaces.ZfitDimensional` method), 88  
`copy()` (`zfit.core.interfaces.ZfitFunc` method), 89  
`copy()` (`zfit.core.interfaces.ZfitLoss` method), 91  
`copy()` (`zfit.core.interfaces.ZfitModel` method), 91  
`copy()` (`zfit.core.interfaces.ZfitNumeric` method), 93  
`copy()` (`zfit.core.interfaces.ZfitObject` method), 93  
`copy()` (`zfit.core.interfaces.ZfitParameter` method), 95  
`copy()` (`zfit.core.interfaces.ZfitPDF` method), 93  
`copy()` (`zfit.core.interfaces.ZfitSpace` method), 96  
`copy()` (`zfit.core.limits.Space` method), 98  
`copy()` (`zfit.core.loss.BaseLoss` method), 103  
`copy()` (`zfit.core.loss.CachedLoss` method), 104  
`copy()` (`zfit.core.loss.ExtendedUnbinnedNLL` method), 105  
`copy()` (`zfit.core.loss.SimpleLoss` method), 106  
`copy()` (`zfit.core.loss.UnbinnedNLL` method), 107  
`copy()` (`zfit.core.parameter.BaseComposedParameter` method), 110  
`copy()` (`zfit.core.parameter.BaseParameter` method), 111  
`copy()` (`zfit.core.parameter.BaseZParameter` method), 111  
`copy()` (`zfit.core.parameter.ComplexParameter` method), 112  
`copy()` (`zfit.core.parameter.ComposedParameter` method), 114  
`copy()` (`zfit.core.parameter.ConstantParameter` method), 115  
`copy()` (`zfit.core.parameter.Parameter` method), 119  
`copy()` (`zfit.core.parameter.ZfitParameterMixin` method), 137  
`copy()` (`zfit.core.sample.EventSpace` method), 138  
`copy()` (`zfit.data.Data` method), 352  
`copy()` (`zfit.func.BaseFunc` method), 355  
`copy()` (`zfit.func.ProdFunc` method), 360  
`copy()` (`zfit.func.SimpleFunc` method), 370  
`copy()` (`zfit.func.SumFunc` method), 365  
`copy()` (`zfit.loss.BaseLoss` method), 377  
`copy()` (`zfit.loss.ExtendedUnbinnedNLL` method), 374  
`copy()` (`zfit.loss.SimpleLoss` method), 378  
`copy()` (`zfit.loss.UnbinnedNLL` method), 375  
`copy()` (`zfit.minimize.Adam` method), 379  
`copy()` (`zfit.minimize.BFGS` method), 382  
`copy()` (`zfit.minimize.Minuit` method), 380  
`copy()` (`zfit.minimize.Scipy` method), 381  
`copy()` (`zfit.minimize.WrapOptimizer` method), 379  
`copy()` (`zfit.minimizers.base_tf.WrapOptimizer` method), 143  
`copy()` (`zfit.minimizers.baseminimizer.BaseMinimizer` method), 143  
`copy()` (`zfit.minimizers.minimizer_minuit.Minuit` method), 148  
`copy()` (`zfit.minimizers.minimizer_tfp.BFGS` method), 149  
`copy()` (`zfit.minimizers.minimizers_scipy.Scipy` method), 149  
`copy()` (`zfit.minimizers.optimizers_tf.Adam` method), 150  
`copy()` (`zfit.models.basefunctor.FunctorMixin` method), 155  
`copy()` (`zfit.models.basic.CustomGaussOLD` method), 160  
`copy()` (`zfit.models.basic.Exponential` method), 166  
`copy()` (`zfit.models.dist_tfp.ExponentialTFP` method), 173  
`copy()` (`zfit.models.dist_tfp.Gauss` method), 180  
`copy()` (`zfit.models.dist_tfp.TruncatedGauss` method), 187  
`copy()` (`zfit.models.dist_tfp.Uniform` method), 194  
`copy()` (`zfit.models.dist_tfp.WrapDistribution` method), 201  
`copy()` (`zfit.models.functions.BaseFunctorFunc` method), 207  
`copy()` (`zfit.models.functions.ProdFunc` method), 212  
`copy()` (`zfit.models.functions.SimpleFunc` method), 218  
`copy()` (`zfit.models.functions.SumFunc` method), 223  
`copy()` (`zfit.models.functions.ZFunc` method), 228  
`copy()` (`zfit.models.functor.BaseFunctor` method), 233

- `copy()` (`zfit.models.functor.ProductPDF` method), 240
- `copy()` (`zfit.models.functor.SumPDF` method), 247
- `copy()` (`zfit.models.physics.CrystalBall` method), 254
- `copy()` (`zfit.models.physics.DoubleCB` method), 262
- `copy()` (`zfit.models.polynomials.Chebyshev` method), 269
- `copy()` (`zfit.models.polynomials.Chebyshev2` method), 276
- `copy()` (`zfit.models.polynomials.Hermite` method), 283
- `copy()` (`zfit.models.polynomials.Laguerre` method), 290
- `copy()` (`zfit.models.polynomials.Legendre` method), 298
- `copy()` (`zfit.models.polynomials.RecursivePolynomial` method), 305
- `copy()` (`zfit.models.special.SimpleFunctorPDF` method), 312
- `copy()` (`zfit.models.special.SimplePDF` method), 319
- `copy()` (`zfit.models.special.ZPDF` method), 326
- `copy()` (`zfit.param.ComplexParameter` method), 395
- `copy()` (`zfit.param.ComposedParameter` method), 394
- `copy()` (`zfit.param.ConstantParameter` method), 383
- `copy()` (`zfit.param.Parameter` method), 386
- `copy()` (`zfit.Parameter` method), 34
- `copy()` (`zfit.pdf.BaseFunctor` method), 404
- `copy()` (`zfit.pdf.BasePDF` method), 398
- `copy()` (`zfit.pdf.Chebyshev` method), 460
- `copy()` (`zfit.pdf.Chebyshev2` method), 475
- `copy()` (`zfit.pdf.CrystalBall` method), 418
- `copy()` (`zfit.pdf.DoubleCB` method), 426
- `copy()` (`zfit.pdf.Exponential` method), 411
- `copy()` (`zfit.pdf.Gauss` method), 433
- `copy()` (`zfit.pdf.Hermite` method), 482
- `copy()` (`zfit.pdf.Laguerre` method), 489
- `copy()` (`zfit.pdf.Legendre` method), 467
- `copy()` (`zfit.pdf.ProductPDF` method), 502
- `copy()` (`zfit.pdf.RecursivePolynomial` method), 496
- `copy()` (`zfit.pdf.SimpleFunctorPDF` method), 529
- `copy()` (`zfit.pdf.SimplePDF` method), 523
- `copy()` (`zfit.pdf.SumPDF` method), 509
- `copy()` (`zfit.pdf.TruncatedGauss` method), 447
- `copy()` (`zfit.pdf.Uniform` method), 440
- `copy()` (`zfit.pdf.WrapDistribution` method), 453
- `copy()` (`zfit.pdf.ZPDF` method), 516
- `copy()` (`zfit.Space` method), 44
- `copy()` (`zfit.util.container.DotDict` method), 334
- `count_up_to()` (`zfit.core.parameter.Parameter` method), 119
- `count_up_to()` (`zfit.core.parameter.TFBaseVariable` method), 130
- `count_up_to()` (`zfit.param.Parameter` method), 386
- `count_up_to()` (`zfit.Parameter` method), 34
- `counts_multinomial()` (in module `zfit.z.random`), 345
- `covariance()` (`zfit.minimizers.fitresult.FitResult` method), 145
- `create` (`zfit.core.parameter.Parameter` attribute), 119
- `create` (`zfit.core.parameter.TFBaseVariable` attribute), 130
- `create` (`zfit.param.Parameter` attribute), 387
- `create` (`zfit.Parameter` attribute), 34
- `create_extended()` (`zfit.core.basepdf.BasePDF` method), 62
- `create_extended()` (`zfit.core.interfaces.ZfitPDF` method), 93
- `create_extended()` (`zfit.models.basic.CustomGaussOLD` method), 160
- `create_extended()` (`zfit.models.basic.Exponential` method), 167
- `create_extended()` (`zfit.models.dist_tfp.ExponentialTFP` method), 173
- `create_extended()` (`zfit.models.dist_tfp.Gauss` method), 180
- `create_extended()` (`zfit.models.dist_tfp.TruncatedGauss` method), 187
- `create_extended()` (`zfit.models.dist_tfp.Uniform` method), 194
- `create_extended()` (`zfit.models.dist_tfp.WrapDistribution` method), 201
- `create_extended()` (`zfit.models.functor.BaseFunctor` method), 233
- `create_extended()` (`zfit.models.functor.ProductPDF` method), 240
- `create_extended()` (`zfit.models.functor.SumPDF` method), 247
- `create_extended()` (`zfit.models.physics.CrystalBall` method), 254
- `create_extended()` (`zfit.models.physics.DoubleCB` method), 262
- `create_extended()` (`zfit.models.polynomials.Chebyshev` method), 269
- `create_extended()` (`zfit.models.polynomials.Chebyshev2` method), 276
- `create_extended()` (`zfit.models.polynomials.Hermite` method), 283
- `create_extended()` (`zfit.models.polynomials.Laguerre` method), 291
- `create_extended()` (`zfit.models.polynomials.Legendre` method),

- 298  
`create_extended()`  
 (`zfit.models.polynomials.RecursivePolynomial`  
*method*), 305  
`create_extended()`  
 (`zfit.models.special.SimpleFunctorPDF`  
*method*), 313  
`create_extended()` (`zfit.models.special.SimplePDF`  
*method*), 319  
`create_extended()` (`zfit.models.special.ZPDF`  
*method*), 326  
`create_extended()` (`zfit.pdf.BaseFunctor` *method*),  
 404  
`create_extended()` (`zfit.pdf.BasePDF` *method*), 398  
`create_extended()` (`zfit.pdf.Chebyshev` *method*),  
 460  
`create_extended()` (`zfit.pdf.Chebyshev2` *method*),  
 475  
`create_extended()` (`zfit.pdf.CrystalBall` *method*),  
 419  
`create_extended()` (`zfit.pdf.DoubleCB` *method*),  
 426  
`create_extended()` (`zfit.pdf.Exponential` *method*),  
 411  
`create_extended()` (`zfit.pdf.Gauss` *method*), 433  
`create_extended()` (`zfit.pdf.Hermite` *method*), 482  
`create_extended()` (`zfit.pdf.Laguerre` *method*), 489  
`create_extended()` (`zfit.pdf.Legendre` *method*), 468  
`create_extended()` (`zfit.pdf.ProductPDF` *method*),  
 503  
`create_extended()` (`zfit.pdf.RecursivePolynomial`  
*method*), 496  
`create_extended()` (`zfit.pdf.SimpleFunctorPDF`  
*method*), 529  
`create_extended()` (`zfit.pdf.SimplePDF` *method*),  
 523  
`create_extended()` (`zfit.pdf.SumPDF` *method*), 510  
`create_extended()` (`zfit.pdf.TruncatedGauss`  
*method*), 447  
`create_extended()` (`zfit.pdf.Uniform` *method*), 440  
`create_extended()` (`zfit.pdf.WrapDistribution`  
*method*), 453  
`create_extended()` (`zfit.pdf.ZPDF` *method*), 516  
`create_immutable()`  
 (`zfit.util.cache.FunctionCacheHolder` *method*),  
 333  
`create_limits()` (`zfit.core.sample.EventSpace`  
*method*), 139  
`create_poly()` (in module `zfit.models.polynomials`),  
 310  
`create_projection_pdf()`  
 (`zfit.core.basepdf.BasePDF` *method*), 62  
`create_projection_pdf()`  
 (`zfit.models.basic.CustomGaussOLD` *method*),  
 160  
`create_projection_pdf()`  
 (`zfit.models.basic.Exponential` *method*), 167  
`create_projection_pdf()`  
 (`zfit.models.dist_tfp.ExponentialTFP` *method*),  
 174  
`create_projection_pdf()`  
 (`zfit.models.dist_tfp.Gauss` *method*), 181  
`create_projection_pdf()`  
 (`zfit.models.dist_tfp.TruncatedGauss` *method*),  
 188  
`create_projection_pdf()`  
 (`zfit.models.dist_tfp.Uniform` *method*), 194  
`create_projection_pdf()`  
 (`zfit.models.dist_tfp.WrapDistribution` *method*),  
 201  
`create_projection_pdf()`  
 (`zfit.models.functor.BaseFunctor` *method*),  
 233  
`create_projection_pdf()`  
 (`zfit.models.functor.ProductPDF` *method*),  
 240  
`create_projection_pdf()`  
 (`zfit.models.functor.SumPDF` *method*), 247  
`create_projection_pdf()`  
 (`zfit.models.physics.CrystalBall` *method*),  
 254  
`create_projection_pdf()`  
 (`zfit.models.physics.DoubleCB` *method*),  
 262  
`create_projection_pdf()`  
 (`zfit.models.polynomials.Chebyshev` *method*),  
 269  
`create_projection_pdf()`  
 (`zfit.models.polynomials.Chebyshev2` *method*),  
 277  
`create_projection_pdf()`  
 (`zfit.models.polynomials.Hermite` *method*),  
 284  
`create_projection_pdf()`  
 (`zfit.models.polynomials.Laguerre` *method*),  
 291  
`create_projection_pdf()`  
 (`zfit.models.polynomials.Legendre` *method*),  
 298  
`create_projection_pdf()`  
 (`zfit.models.polynomials.RecursivePolynomial`  
*method*), 305  
`create_projection_pdf()`  
 (`zfit.models.special.SimpleFunctorPDF`  
*method*), 313  
`create_projection_pdf()`  
 (`zfit.models.special.SimplePDF` *method*),  
 319

```

create_projection_pdf()
    (zfit.models.special.ZPDF method), 326
create_projection_pdf() (zfit.pdf.BaseFunc
    method), 405
create_projection_pdf() (zfit.pdf.BasePDF
    method), 398
create_projection_pdf() (zfit.pdf.Chebyshev
    method), 461
create_projection_pdf() (zfit.pdf.Chebyshev2
    method), 475
create_projection_pdf() (zfit.pdf.CrystalBall
    method), 419
create_projection_pdf() (zfit.pdf.DoubleCB
    method), 426
create_projection_pdf() (zfit.pdf.Exponential
    method), 411
create_projection_pdf() (zfit.pdf.Gauss
    method), 433
create_projection_pdf() (zfit.pdf.Hermite
    method), 482
create_projection_pdf() (zfit.pdf.Laguerre
    method), 489
create_projection_pdf() (zfit.pdf.Legendre
    method), 468
create_projection_pdf() (zfit.pdf.ProductPDF
    method), 503
create_projection_pdf()
    (zfit.pdf.RecursivePolynomial method), 496
create_projection_pdf()
    (zfit.pdf.SimpleFuncPDF method), 530
create_projection_pdf() (zfit.pdf.SimplePDF
    method), 523
create_projection_pdf() (zfit.pdf.SumPDF
    method), 510
create_projection_pdf()
    (zfit.pdf.TruncatedGauss method), 447
create_projection_pdf() (zfit.pdf.Uniform
    method), 440
create_projection_pdf()
    (zfit.pdf.WrapDistribution method), 454
create_projection_pdf() (zfit.pdf.ZPDF
    method), 517
create_sampler() (zfit.core.basefunc.BaseFunc
    method), 50
create_sampler() (zfit.core.basemodel.BaseModel
    method), 55
create_sampler() (zfit.core.basepdf.BasePDF
    method), 62
create_sampler() (zfit.func.BaseFunc method), 355
create_sampler() (zfit.func.ProdFunc method), 360
create_sampler() (zfit.func.SimpleFunc method),
    370
create_sampler() (zfit.func.SumFunc method), 365
create_sampler() (zfit.models.basefunc.Func
    method), 155
create_sampler() (zfit.models.basic.CustomGaussOLD
    method), 160
create_sampler() (zfit.models.basic.Exponential
    method), 167
create_sampler() (zfit.models.dist_tfp.ExponentialTFP
    method), 174
create_sampler() (zfit.models.dist_tfp.Gauss
    method), 181
create_sampler() (zfit.models.dist_tfp.TruncatedGauss
    method), 188
create_sampler() (zfit.models.dist_tfp.Uniform
    method), 195
create_sampler() (zfit.models.dist_tfp.WrapDistribution
    method), 201
create_sampler() (zfit.models.functions.BaseFunc
    method), 207
create_sampler() (zfit.models.functions.ProdFunc
    method), 212
create_sampler() (zfit.models.functions.SimpleFunc
    method), 218
create_sampler() (zfit.models.functions.SumFunc
    method), 223
create_sampler() (zfit.models.functions.ZFunc
    method), 228
create_sampler() (zfit.models.func
    method), 233
create_sampler() (zfit.models.func
    method), 240
create_sampler() (zfit.models.func
    method), 247
create_sampler() (zfit.models.physics.CrystalBall
    method), 255
create_sampler() (zfit.models.physics.DoubleCB
    method), 262
create_sampler() (zfit.models.polynomials.Chebyshev
    method), 269
create_sampler() (zfit.models.polynomials.Chebyshev2
    method), 277
create_sampler() (zfit.models.polynomials.Hermite
    method), 284
create_sampler() (zfit.models.polynomials.Laguerre
    method), 291
create_sampler() (zfit.models.polynomials.Legendre
    method), 298
create_sampler() (zfit.models.polynomials.RecursivePolynomial
    method), 305
create_sampler() (zfit.models.special.SimpleFuncPDF
    method), 313
create_sampler() (zfit.models.special.SimplePDF
    method), 320
create_sampler() (zfit.models.special.ZPDF
    method), 326
create_sampler() (zfit.pdf.BaseFunc method),

```



- 405  
 create\_sampler() (*zfit.pdf.BasePDF method*), 398  
 create\_sampler() (*zfit.pdf.Chebyshev method*), 461  
 create\_sampler() (*zfit.pdf.Chebyshev2 method*), 475  
 create\_sampler() (*zfit.pdf.CrystalBall method*), 419  
 create\_sampler() (*zfit.pdf.DoubleCB method*), 426  
 create\_sampler() (*zfit.pdf.Exponential method*), 412  
 create\_sampler() (*zfit.pdf.Gauss method*), 433  
 create\_sampler() (*zfit.pdf.Hermite method*), 482  
 create\_sampler() (*zfit.pdf.Laguerre method*), 489  
 create\_sampler() (*zfit.pdf.Legendre method*), 468  
 create\_sampler() (*zfit.pdf.ProductPDF method*), 503  
 create\_sampler() (*zfit.pdf.RecursivePolynomial method*), 496  
 create\_sampler() (*zfit.pdf.SimpleFunctorPDF method*), 530  
 create\_sampler() (*zfit.pdf.SimplePDF method*), 523  
 create\_sampler() (*zfit.pdf.SumPDF method*), 510  
 create\_sampler() (*zfit.pdf.TruncatedGauss method*), 447  
 create\_sampler() (*zfit.pdf.Uniform method*), 440  
 create\_sampler() (*zfit.pdf.WrapDistribution method*), 454  
 create\_sampler() (*zfit.pdf.ZPDF method*), 517  
 CrystalBall (*class in zfit.models.physics*), 252  
 CrystalBall (*class in zfit.pdf*), 416  
 crystalball\_func() (*in module zfit.models.physics*), 267  
 crystalball\_integral() (*in module zfit.models.physics*), 267  
 CustomGaussOLD (*class in zfit.models.basic*), 158
- ## D
- Data (*class in zfit.core.data*), 73  
 Data (*class in zfit.data*), 351  
 data (*zfit.core.interfaces.ZfitLoss attribute*), 91  
 data (*zfit.core.loss.BaseLoss attribute*), 103  
 data (*zfit.core.loss.CachedLoss attribute*), 104  
 data (*zfit.core.loss.ExtendedUnbinnedNLL attribute*), 105  
 data (*zfit.core.loss.SimpleLoss attribute*), 106  
 data (*zfit.core.loss.UnbinnedNLL attribute*), 107  
 data (*zfit.loss.BaseLoss attribute*), 377  
 data (*zfit.loss.ExtendedUnbinnedNLL attribute*), 375  
 data (*zfit.loss.SimpleLoss attribute*), 378  
 data (*zfit.loss.UnbinnedNLL attribute*), 376  
 data\_range (*zfit.core.data.Data attribute*), 74  
 data\_range (*zfit.core.data.SampleData attribute*), 77  
 data\_range (*zfit.core.data.Sampler attribute*), 80  
 data\_range (*zfit.data.Data attribute*), 352  
 DefaultStrategy (*class in zfit.minimizers.baseminimizer*), 144  
 degree (*zfit.models.polynomials.Chebyshev attribute*), 270  
 degree (*zfit.models.polynomials.Chebyshev2 attribute*), 277  
 degree (*zfit.models.polynomials.Hermite attribute*), 284  
 degree (*zfit.models.polynomials.Laguerre attribute*), 292  
 degree (*zfit.models.polynomials.Legendre attribute*), 299  
 degree (*zfit.models.polynomials.RecursivePolynomial attribute*), 306  
 degree (*zfit.pdf.Chebyshev attribute*), 461  
 degree (*zfit.pdf.Chebyshev2 attribute*), 476  
 degree (*zfit.pdf.Hermite attribute*), 483  
 degree (*zfit.pdf.Laguerre attribute*), 490  
 degree (*zfit.pdf.Legendre attribute*), 469  
 degree (*zfit.pdf.RecursivePolynomial attribute*), 497  
 device (*zfit.core.parameter.Parameter attribute*), 119  
 device (*zfit.core.parameter.TFBaseVariable attribute*), 130  
 device (*zfit.param.Parameter attribute*), 387  
 device (*zfit.Parameter attribute*), 34  
 dict\_to\_matrix() (*in module zfit.minimizers.fitresult*), 146  
 distribution (*zfit.constraint.GaussianConstraint attribute*), 350  
 distribution (*zfit.core.constraint.DistributionConstraint attribute*), 69  
 distribution (*zfit.core.constraint.GaussianConstraint attribute*), 71  
 distribution (*zfit.models.dist\_tfp.ExponentialTFP attribute*), 174  
 distribution (*zfit.models.dist\_tfp.Gauss attribute*), 181  
 distribution (*zfit.models.dist\_tfp.TruncatedGauss attribute*), 188  
 distribution (*zfit.models.dist\_tfp.Uniform attribute*), 195  
 distribution (*zfit.models.dist\_tfp.WrapDistribution attribute*), 202  
 distribution (*zfit.pdf.Gauss attribute*), 434  
 distribution (*zfit.pdf.TruncatedGauss attribute*), 448  
 distribution (*zfit.pdf.Uniform attribute*), 441  
 distribution (*zfit.pdf.WrapDistribution attribute*), 454  
 DistributionConstraint (*class in zfit.core.constraint*), 69  
 do\_recurrence() (*in module zfit.models.polynomials*), 310  
 DotDict (*class in zfit.util.container*), 334

`double_crystalball_func()` (in module `zfit.models.physics`), 267  
`double_crystalball_integral()` (in module `zfit.models.physics`), 267  
`DoubleCB` (class in `zfit.models.physics`), 259  
`DoubleCB` (class in `zfit.pdf`), 424  
`dtype` (`zfit.ComplexParameter` attribute), 43  
`dtype` (`zfit.ComposedParameter` attribute), 41  
`dtype` (`zfit.constraint.GaussianConstraint` attribute), 350  
`dtype` (`zfit.constraint.SimpleConstraint` attribute), 348  
`dtype` (`zfit.core.basefunc.BaseFunc` attribute), 50  
`dtype` (`zfit.core.basemodel.BaseModel` attribute), 56  
`dtype` (`zfit.core.baseobject.BaseNumeric` attribute), 59  
`dtype` (`zfit.core.basepdf.BasePDF` attribute), 63  
`dtype` (`zfit.core.constraint.BaseConstraint` attribute), 68  
`dtype` (`zfit.core.constraint.DistributionConstraint` attribute), 69  
`dtype` (`zfit.core.constraint.GaussianConstraint` attribute), 71  
`dtype` (`zfit.core.constraint.SimpleConstraint` attribute), 72  
`dtype` (`zfit.core.data.Data` attribute), 74  
`dtype` (`zfit.core.data.SampleData` attribute), 77  
`dtype` (`zfit.core.data.Sampler` attribute), 80  
`dtype` (`zfit.core.interfaces.ZfitFunc` attribute), 89  
`dtype` (`zfit.core.interfaces.ZfitModel` attribute), 91  
`dtype` (`zfit.core.interfaces.ZfitNumeric` attribute), 93  
`dtype` (`zfit.core.interfaces.ZfitParameter` attribute), 95  
`dtype` (`zfit.core.interfaces.ZfitPDF` attribute), 93  
`dtype` (`zfit.core.parameter.BaseComposedParameter` attribute), 110  
`dtype` (`zfit.core.parameter.BaseParameter` attribute), 111  
`dtype` (`zfit.core.parameter.BaseZParameter` attribute), 111  
`dtype` (`zfit.core.parameter.ComplexParameter` attribute), 113  
`dtype` (`zfit.core.parameter.ComposedParameter` attribute), 114  
`dtype` (`zfit.core.parameter.ConstantParameter` attribute), 115  
`dtype` (`zfit.core.parameter.Parameter` attribute), 119  
`dtype` (`zfit.core.parameter.TFBaseVariable` attribute), 130  
`dtype` (`zfit.core.parameter.ZfitBaseVariable` attribute), 136  
`dtype` (`zfit.core.parameter.ZfitParameterMixin` attribute), 137  
`dtype` (`zfit.data.Data` attribute), 352  
`dtype` (`zfit.func.BaseFunc` attribute), 356  
`dtype` (`zfit.func.ProdFunc` attribute), 361  
`dtype` (`zfit.func.SimpleFunc` attribute), 371  
`dtype` (`zfit.func.SumFunc` attribute), 366  
`dtype` (`zfit.models.basefunc.FunctorMixin` attribute), 155  
`dtype` (`zfit.models.basic.CustomGaussOLD` attribute), 161  
`dtype` (`zfit.models.basic.Exponential` attribute), 168  
`dtype` (`zfit.models.dist_tfp.ExponentialTFP` attribute), 174  
`dtype` (`zfit.models.dist_tfp.Gauss` attribute), 181  
`dtype` (`zfit.models.dist_tfp.TruncatedGauss` attribute), 188  
`dtype` (`zfit.models.dist_tfp.Uniform` attribute), 195  
`dtype` (`zfit.models.dist_tfp.WrapDistribution` attribute), 202  
`dtype` (`zfit.models.functions.BaseFunctorFunc` attribute), 208  
`dtype` (`zfit.models.functions.ProdFunc` attribute), 213  
`dtype` (`zfit.models.functions.SimpleFunc` attribute), 218  
`dtype` (`zfit.models.functions.SumFunc` attribute), 223  
`dtype` (`zfit.models.functions.ZFunc` attribute), 228  
`dtype` (`zfit.models.functor.BaseFunctor` attribute), 234  
`dtype` (`zfit.models.functor.ProductPDF` attribute), 241  
`dtype` (`zfit.models.functor.SumPDF` attribute), 248  
`dtype` (`zfit.models.physics.CrystalBall` attribute), 255  
`dtype` (`zfit.models.physics.DoubleCB` attribute), 263  
`dtype` (`zfit.models.polynomials.Chebyshev` attribute), 270  
`dtype` (`zfit.models.polynomials.Chebyshev2` attribute), 277  
`dtype` (`zfit.models.polynomials.Hermite` attribute), 285  
`dtype` (`zfit.models.polynomials.Laguerre` attribute), 292  
`dtype` (`zfit.models.polynomials.Legendre` attribute), 299  
`dtype` (`zfit.models.polynomials.RecursivePolynomial` attribute), 306  
`dtype` (`zfit.models.special.SimpleFunctorPDF` attribute), 313  
`dtype` (`zfit.models.special.SimplePDF` attribute), 320  
`dtype` (`zfit.models.special.ZPDF` attribute), 327  
`dtype` (`zfit.param.ComplexParameter` attribute), 395  
`dtype` (`zfit.param.ComposedParameter` attribute), 394  
`dtype` (`zfit.param.ConstantParameter` attribute), 383  
`dtype` (`zfit.param.Parameter` attribute), 387  
`dtype` (`zfit.Parameter` attribute), 34  
`dtype` (`zfit.pdf.BaseFunctor` attribute), 405  
`dtype` (`zfit.pdf.BasePDF` attribute), 399  
`dtype` (`zfit.pdf.Chebyshev` attribute), 462  
`dtype` (`zfit.pdf.Chebyshev2` attribute), 476  
`dtype` (`zfit.pdf.CrystalBall` attribute), 420  
`dtype` (`zfit.pdf.DoubleCB` attribute), 427  
`dtype` (`zfit.pdf.Exponential` attribute), 412  
`dtype` (`zfit.pdf.Gauss` attribute), 434  
`dtype` (`zfit.pdf.Hermite` attribute), 483  
`dtype` (`zfit.pdf.Laguerre` attribute), 490  
`dtype` (`zfit.pdf.Legendre` attribute), 469  
`dtype` (`zfit.pdf.ProductPDF` attribute), 504

dtype (*zfit.pdf.RecursivePolynomial* attribute), 497  
 dtype (*zfit.pdf.SimpleFunctorPDF* attribute), 530  
 dtype (*zfit.pdf.SimplePDF* attribute), 524  
 dtype (*zfit.pdf.SumPDF* attribute), 511  
 dtype (*zfit.pdf.TruncatedGauss* attribute), 448  
 dtype (*zfit.pdf.Uniform* attribute), 441  
 dtype (*zfit.pdf.WrapDistribution* attribute), 454  
 dtype (*zfit.pdf.ZPDF* attribute), 517

## E

edm (*zfit.minimizers.fitresult.FitResult* attribute), 145  
 error() (*zfit.minimizers.fitresult.FitResult* method), 145  
 error() (*zfit.minimizers.interface.ZfitResult* method), 146  
 errordef (*zfit.core.interfaces.ZfitLoss* attribute), 91  
 errordef (*zfit.core.loss.BaseLoss* attribute), 103  
 errordef (*zfit.core.loss.CachedLoss* attribute), 104  
 errordef (*zfit.core.loss.ExtendedUnbinnedNLL* attribute), 105  
 errordef (*zfit.core.loss.SimpleLoss* attribute), 106  
 errordef (*zfit.core.loss.UnbinnedNLL* attribute), 107  
 errordef (*zfit.loss.BaseLoss* attribute), 377  
 errordef (*zfit.loss.ExtendedUnbinnedNLL* attribute), 375  
 errordef (*zfit.loss.SimpleLoss* attribute), 378  
 errordef (*zfit.loss.UnbinnedNLL* attribute), 376  
 eval() (*zfit.core.parameter.Parameter* method), 119  
 eval() (*zfit.core.parameter.TFBaseVariable* method), 130  
 eval() (*zfit.param.Parameter* method), 387  
 eval() (*zfit.Parameter* method), 34  
 EventSpace (class in *zfit.core.sample*), 138  
 exp() (in module *zfit.z.wrapping\_tf*), 346  
 experimental\_ref()  
     (*zfit.core.parameter.Parameter* method), 119  
 experimental\_ref()  
     (*zfit.core.parameter.TFBaseVariable* method), 130  
 experimental\_ref() (*zfit.param.Parameter* method), 387  
 experimental\_ref() (*zfit.Parameter* method), 34  
 Exponential (class in *zfit.models.basic*), 165  
 Exponential (class in *zfit.pdf*), 410  
 ExponentialTFP (class in *zfit.models.dist\_tfp*), 172  
 extended\_sampling() (in module *zfit.core.sample*), 142  
 ExtendedPDFError, 336  
 ExtendedUnbinnedNLL (class in *zfit.core.loss*), 105  
 ExtendedUnbinnedNLL (class in *zfit.loss*), 374  
 ExternalOptimizerInterface (class in *zfit.minimizers.tf\_external\_optimizer*), 150

extract\_extended\_pdfs() (in module *zfit.core.sample*), 142

## F

factory (*zfit.core.sample.EventSpace* attribute), 139  
 FailMinimizeNaN, 144  
 feed\_function() (in module *zfit.core.data*), 83  
 feed\_function\_for\_partial\_run() (in module *zfit.core.data*), 83  
 fetch\_function() (in module *zfit.core.data*), 83  
 fit\_range (*zfit.core.interfaces.ZfitLoss* attribute), 91  
 fit\_range (*zfit.core.loss.BaseLoss* attribute), 103  
 fit\_range (*zfit.core.loss.CachedLoss* attribute), 104  
 fit\_range (*zfit.core.loss.ExtendedUnbinnedNLL* attribute), 105  
 fit\_range (*zfit.core.loss.SimpleLoss* attribute), 107  
 fit\_range (*zfit.core.loss.UnbinnedNLL* attribute), 108  
 fit\_range (*zfit.loss.BaseLoss* attribute), 377  
 fit\_range (*zfit.loss.ExtendedUnbinnedNLL* attribute), 375  
 fit\_range (*zfit.loss.SimpleLoss* attribute), 378  
 fit\_range (*zfit.loss.UnbinnedNLL* attribute), 376  
 FitResult (class in *zfit.minimizers.fitresult*), 144  
 floating (*zfit.ComplexParameter* attribute), 43  
 floating (*zfit.ComposedParameter* attribute), 41  
 floating (*zfit.core.interfaces.ZfitParameter* attribute), 95  
 floating (*zfit.core.parameter.BaseComposedParameter* attribute), 110  
 floating (*zfit.core.parameter.BaseParameter* attribute), 111  
 floating (*zfit.core.parameter.BaseZParameter* attribute), 111  
 floating (*zfit.core.parameter.ComplexParameter* attribute), 113  
 floating (*zfit.core.parameter.ComposedParameter* attribute), 114  
 floating (*zfit.core.parameter.ConstantParameter* attribute), 115  
 floating (*zfit.core.parameter.Parameter* attribute), 120  
 floating (*zfit.core.parameter.ZfitParameterMixin* attribute), 137  
 floating (*zfit.param.ComplexParameter* attribute), 395  
 floating (*zfit.param.ComposedParameter* attribute), 394  
 floating (*zfit.param.ConstantParameter* attribute), 383  
 floating (*zfit.param.Parameter* attribute), 387  
 floating (*zfit.Parameter* attribute), 35  
 fmin (*zfit.minimizers.fitresult.FitResult* attribute), 146  
 fmin (*zfit.minimizers.interface.ZfitResult* attribute), 147  
 fracs (*zfit.models.functor.SumPDF* attribute), 248

- `fracs` (`zfit.pdf.SumPDF` attribute), 511
  - `from_axes()` (`zfit.core.limits.Space` class method), 98
  - `from_axes()` (`zfit.core.sample.EventSpace` class method), 139
  - `from_axes()` (`zfit.Space` class method), 45
  - `from_cartesian()` (`zfit.ComplexParameter` static method), 43
  - `from_cartesian()` (`zfit.core.parameter.ComplexParameter` static method), 113
  - `from_cartesian()` (`zfit.param.ComplexParameter` static method), 395
  - `from_numpy()` (`zfit.core.data.Data` class method), 74
  - `from_numpy()` (`zfit.core.data.SampleData` class method), 77
  - `from_numpy()` (`zfit.core.data.Sampler` class method), 80
  - `from_numpy()` (`zfit.data.Data` class method), 352
  - `from_pandas()` (`zfit.core.data.Data` class method), 74
  - `from_pandas()` (`zfit.core.data.SampleData` class method), 77
  - `from_pandas()` (`zfit.core.data.Sampler` class method), 80
  - `from_pandas()` (`zfit.data.Data` class method), 352
  - `from_polar()` (`zfit.ComplexParameter` static method), 43
  - `from_polar()` (`zfit.core.parameter.ComplexParameter` static method), 113
  - `from_polar()` (`zfit.param.ComplexParameter` static method), 395
  - `from_proto()` (`zfit.core.parameter.Parameter` static method), 120
  - `from_proto()` (`zfit.core.parameter.TFBaseVariable` static method), 131
  - `from_proto()` (`zfit.param.Parameter` static method), 387
  - `from_proto()` (`zfit.Parameter` static method), 35
  - `from_root()` (`zfit.core.data.Data` class method), 74
  - `from_root()` (`zfit.core.data.SampleData` class method), 77
  - `from_root()` (`zfit.core.data.Sampler` class method), 81
  - `from_root()` (`zfit.data.Data` class method), 352
  - `from_root_iter()` (`zfit.core.data.Data` class method), 75
  - `from_root_iter()` (`zfit.core.data.SampleData` class method), 78
  - `from_root_iter()` (`zfit.core.data.Sampler` class method), 81
  - `from_root_iter()` (`zfit.data.Data` class method), 353
  - `from_sample()` (`zfit.core.data.SampleData` class method), 78
  - `from_sample()` (`zfit.core.data.Sampler` class method), 81
  - `from_tensor()` (`zfit.core.data.Data` class method), 75
  - `from_tensor()` (`zfit.core.data.LightDataset` class method), 76
  - `from_tensor()` (`zfit.core.data.SampleData` class method), 78
  - `from_tensor()` (`zfit.core.data.Sampler` class method), 81
  - `from_tensor()` (`zfit.data.Data` class method), 353
  - `fromkeys()` (`zfit.util.container.DotDict` method), 334
  - `func()` (`zfit.core.basefunc.BaseFunc` method), 51
  - `func()` (`zfit.core.interfaces.ZfitFunc` method), 89
  - `func()` (`zfit.func.BaseFunc` method), 356
  - `func()` (`zfit.func.ProdFunc` method), 361
  - `func()` (`zfit.func.SimpleFunc` method), 371
  - `func()` (`zfit.func.SumFunc` method), 366
  - `func()` (`zfit.models.functions.BaseFunc` method), 208
  - `func()` (`zfit.models.functions.ProdFunc` method), 213
  - `func()` (`zfit.models.functions.SimpleFunc` method), 218
  - `func()` (`zfit.models.functions.SumFunc` method), 223
  - `func()` (`zfit.models.functions.ZFunc` method), 228
  - `func_integral_chebyshev1()` (in module `zfit.models.polynomials`), 310
  - `func_integral_chebyshev2()` (in module `zfit.models.polynomials`), 310
  - `func_integral_hermite()` (in module `zfit.models.polynomials`), 310
  - `func_integral_laguerre()` (in module `zfit.models.polynomials`), 310
  - `FunctionCacheHolder` (class in `zfit.util.cache`), 332
  - `FunctionWrapperRegistry` (class in `zfit.z.zextension`), 346
  - `FunctorMixin` (class in `zfit.models.basefunctor`), 154
- ## G
- `gather_nd()` (`zfit.core.parameter.Parameter` method), 120
  - `gather_nd()` (`zfit.core.parameter.TFBaseVariable` method), 131
  - `gather_nd()` (`zfit.param.Parameter` method), 387
  - `gather_nd()` (`zfit.Parameter` method), 35
  - `Gauss` (class in `zfit.models.dist_tfp`), 178
  - `Gauss` (class in `zfit.pdf`), 431
  - `gauss_2d()` (in module `zfit.util.diverse`), 335
  - `gauss_4d()` (in module `zfit.util.diverse`), 335
  - `GaussianConstraint` (class in `zfit.constraint`), 349
  - `GaussianConstraint` (class in `zfit.core.constraint`), 70
  - `GaussianMixture2D` (class in `zfit.util.diverse`), 335
  - `GaussianMixture4D` (class in `zfit.util.diverse`), 335
  - `generalized_laguerre_polys_factory()` (in module `zfit.models.polynomials`), 310
  - `generalized_laguerre_recurrence_factory()` (in module `zfit.models.polynomials`), 310



`generalized_laguerre_shape_factory()` (in module `zfit.models.polynomials`), 310  
`get()` (`zfit.util.container.DotDict` method), 334  
`get_auto_number()` (in module `zfit.core.parameter`), 138  
`get_axes()` (`zfit.core.interfaces.ZfitSpace` method), 96  
`get_axes()` (`zfit.core.limits.Space` method), 99  
`get_axes()` (`zfit.core.sample.EventSpace` method), 139  
`get_axes()` (`zfit.Space` method), 45  
`get_cache_counting()` (`zfit.core.data.SampleData` class method), 78  
`get_cache_counting()` (`zfit.core.data.Sampler` class method), 81  
`get_dependents()` (`zfit.ComplexParameter` method), 43  
`get_dependents()` (`zfit.ComposedParameter` method), 42  
`get_dependents()` (`zfit.constraint.GaussianConstraint` method), 350  
`get_dependents()` (`zfit.constraint.SimpleConstraint` method), 348  
`get_dependents()` (`zfit.core.basefunc.BaseFunc` method), 51  
`get_dependents()` (`zfit.core.basemodel.BaseModel` method), 56  
`get_dependents()` (`zfit.core.baseobject.BaseNumeric` method), 59  
`get_dependents()` (`zfit.core.basepdf.BasePDF` method), 63  
`get_dependents()` (`zfit.core.constraint.BaseConstraint` method), 68  
`get_dependents()` (`zfit.core.constraint.DistributionConstraint` method), 69  
`get_dependents()` (`zfit.core.constraint.GaussianConstraint` method), 71  
`get_dependents()` (`zfit.core.constraint.SimpleConstraint` method), 72  
`get_dependents()` (`zfit.core.dependents.BaseDependentsMixin` method), 83  
`get_dependents()` (`zfit.core.interfaces.ZfitDependentsMixin` method), 88  
`get_dependents()` (`zfit.core.interfaces.ZfitFunc` method), 89  
`get_dependents()` (`zfit.core.interfaces.ZfitLoss` method), 91  
`get_dependents()` (`zfit.core.interfaces.ZfitModel` method), 91  
`get_dependents()` (`zfit.core.interfaces.ZfitNumeric` method), 93  
`get_dependents()` (`zfit.core.interfaces.ZfitParameter` method), 95  
`get_dependents()` (`zfit.core.interfaces.ZfitPDF` method), 93  
`get_dependents()` (`zfit.core.loss.BaseLoss` method), 103  
`get_dependents()` (`zfit.core.loss.CachedLoss` method), 104  
`get_dependents()` (`zfit.core.loss.ExtendedUnbinnedNLL` method), 105  
`get_dependents()` (`zfit.core.loss.SimpleLoss` method), 107  
`get_dependents()` (`zfit.core.loss.UnbinnedNLL` method), 108  
`get_dependents()` (`zfit.core.parameter.BaseComposedParameter` method), 110  
`get_dependents()` (`zfit.core.parameter.BaseParameter` method), 111  
`get_dependents()` (`zfit.core.parameter.BaseZParameter` method), 111  
`get_dependents()` (`zfit.core.parameter.ComplexParameter` method), 113  
`get_dependents()` (`zfit.core.parameter.ComposedParameter` method), 114  
`get_dependents()` (`zfit.core.parameter.ConstantParameter` method), 115  
`get_dependents()` (`zfit.core.parameter.Parameter` method), 120  
`get_dependents()` (`zfit.core.parameter.ZfitParameterMixin` method), 137  
`get_dependents()` (`zfit.func.BaseFunc` method), 356  
`get_dependents()` (`zfit.func.ProdFunc` method), 361  
`get_dependents()` (`zfit.func.SimpleFunc` method), 371  
`get_dependents()` (`zfit.func.SumFunc` method), 366  
`get_dependents()` (`zfit.loss.BaseLoss` method), 377  
`get_dependents()` (`zfit.loss.ExtendedUnbinnedNLL` method), 375  
`get_dependents()` (`zfit.loss.SimpleLoss` method), 378  
`get_dependents()` (`zfit.loss.UnbinnedNLL` method), 376  
`get_dependents()` (`zfit.models.basefunc.FunctorMixin` method), 155  
`get_dependents()` (`zfit.models.basic.CustomGaussOLD` method), 161  
`get_dependents()` (`zfit.models.basic.Exponential` method), 168  
`get_dependents()` (`zfit.models.dist_tfp.ExponentialTFP` method), 174  
`get_dependents()` (`zfit.models.dist_tfp.Gauss` method), 181  
`get_dependents()` (`zfit.models.dist_tfp.TruncatedGauss` method), 188  
`get_dependents()` (`zfit.models.dist_tfp.Uniform` method), 195  
`get_dependents()` (`zfit.models.dist_tfp.WrapDistribution` method), 202

`get_dependents()` (`zfit.models.functions.BaseFunc` `method`), 208  
`get_dependents()` (`zfit.models.functions.ProdFunc` `method`), 213  
`get_dependents()` (`zfit.models.functions.SimpleFunc` `method`), 218  
`get_dependents()` (`zfit.models.functions.SumFunc` `method`), 223  
`get_dependents()` (`zfit.models.functions.ZFunc` `method`), 228  
`get_dependents()` (`zfit.models.functor.BaseFunc` `method`), 234  
`get_dependents()` (`zfit.models.functor.ProductPDF` `method`), 241  
`get_dependents()` (`zfit.models.functor.SumPDF` `method`), 248  
`get_dependents()` (`zfit.models.physics.CrystalBall` `method`), 255  
`get_dependents()` (`zfit.models.physics.DoubleCB` `method`), 263  
`get_dependents()` (`zfit.models.polynomials.Chebyshev` `method`), 270  
`get_dependents()` (`zfit.models.polynomials.Chebyshev2` `method`), 277  
`get_dependents()` (`zfit.models.polynomials.Hermite` `method`), 285  
`get_dependents()` (`zfit.models.polynomials.Laguerre` `method`), 292  
`get_dependents()` (`zfit.models.polynomials.Legendre` `method`), 299  
`get_dependents()` (`zfit.models.polynomials.RecursivePolynomial` `method`), 306  
`get_dependents()` (`zfit.models.special.SimpleFuncPDF` `method`), 314  
`get_dependents()` (`zfit.models.special.SimplePDF` `method`), 320  
`get_dependents()` (`zfit.models.special.ZPDF` `method`), 327  
`get_dependents()` (`zfit.param.ComplexParameter` `method`), 395  
`get_dependents()` (`zfit.param.ComposedParameter` `method`), 394  
`get_dependents()` (`zfit.param.ConstantParameter` `method`), 383  
`get_dependents()` (`zfit.param.Parameter` `method`), 387  
`get_dependents()` (`zfit.Parameter` `method`), 35  
`get_dependents()` (`zfit.pdf.BaseFunc` `method`), 405  
`get_dependents()` (`zfit.pdf.BasePDF` `method`), 399  
`get_dependents()` (`zfit.pdf.Chebyshev` `method`), 462  
`get_dependents()` (`zfit.pdf.Chebyshev2` `method`), 476  
`get_dependents()` (`zfit.pdf.CrystalBall` `method`), 420  
`get_dependents()` (`zfit.pdf.DoubleCB` `method`), 427  
`get_dependents()` (`zfit.pdf.Exponential` `method`), 412  
`get_dependents()` (`zfit.pdf.Gauss` `method`), 434  
`get_dependents()` (`zfit.pdf.Hermite` `method`), 483  
`get_dependents()` (`zfit.pdf.Laguerre` `method`), 490  
`get_dependents()` (`zfit.pdf.Legendre` `method`), 469  
`get_dependents()` (`zfit.pdf.ProductPDF` `method`), 504  
`get_dependents()` (`zfit.pdf.RecursivePolynomial` `method`), 497  
`get_dependents()` (`zfit.pdf.SimpleFuncPDF` `method`), 530  
`get_dependents()` (`zfit.pdf.SimplePDF` `method`), 524  
`get_dependents()` (`zfit.pdf.SumPDF` `method`), 511  
`get_dependents()` (`zfit.pdf.TruncatedGauss` `method`), 448  
`get_dependents()` (`zfit.pdf.Uniform` `method`), 441  
`get_dependents()` (`zfit.pdf.WrapDistribution` `method`), 454  
`get_dependents()` (`zfit.pdf.ZPDF` `method`), 517  
`get_dependents_auto()` (in module `zfit.util.graph`), 340  
`get_iteration()` (`zfit.core.data.Data` `method`), 75  
`get_iteration()` (`zfit.core.data.SampleData` `method`), 78  
`get_iteration()` (`zfit.core.data.Sampler` `method`), 81  
`get_iteration()` (`zfit.data.Data` `method`), 353  
`get_logger()` (in module `zfit.util.logging`), 341  
`get_max_axes()` (`zfit.core.integration.AnalyticIntegral` `method`), 85  
`get_max_integral()` (`zfit.core.integration.AnalyticIntegral` `method`), 85  
`get_models()` (`zfit.core.interfaces.ZfitFunc` `method`), 90  
`get_models()` (`zfit.func.ProdFunc` `method`), 361  
`get_models()` (`zfit.func.SumFunc` `method`), 366  
`get_models()` (`zfit.models.basefunc.Func` `method`), 155  
`get_models()` (`zfit.models.functions.BaseFunc` `method`), 208  
`get_models()` (`zfit.models.functions.ProdFunc` `method`), 213  
`get_models()` (`zfit.models.functions.SumFunc` `method`), 224  
`get_models()` (`zfit.models.functor.BaseFunc` `method`), 234  
`get_models()` (`zfit.models.functor.ProductPDF` `method`), 241  
`get_models()` (`zfit.models.functor.SumPDF` `method`), 241

- 248
- `get_models()` (*zfit.models.special.SimpleFunctorPDF method*), 314
- `get_models()` (*zfit.pdf.BaseFunctor method*), 405
- `get_models()` (*zfit.pdf.ProductPDF method*), 504
- `get_models()` (*zfit.pdf.SimpleFunctorPDF method*), 531
- `get_models()` (*zfit.pdf.SumPDF method*), 511
- `get_obs_axes()` (*zfit.core.limits.Space method*), 99
- `get_obs_axes()` (*zfit.core.sample.EventSpace method*), 139
- `get_obs_axes()` (*zfit.Space method*), 45
- `get_params()` (*zfit.ComplexParameter method*), 43
- `get_params()` (*zfit.ComposedParameter method*), 42
- `get_params()` (*zfit.constraint.GaussianConstraint method*), 350
- `get_params()` (*zfit.constraint.SimpleConstraint method*), 349
- `get_params()` (*zfit.core.basefunc.BaseFunc method*), 51
- `get_params()` (*zfit.core.basemodel.BaseModel method*), 56
- `get_params()` (*zfit.core.baseobject.BaseNumeric method*), 59
- `get_params()` (*zfit.core.basepdf.BasePDF method*), 63
- `get_params()` (*zfit.core.constraint.BaseConstraint method*), 68
- `get_params()` (*zfit.core.constraint.DistributionConstraint method*), 69
- `get_params()` (*zfit.core.constraint.GaussianConstraint method*), 71
- `get_params()` (*zfit.core.constraint.SimpleConstraint method*), 72
- `get_params()` (*zfit.core.interfaces.ZfitFunc method*), 89
- `get_params()` (*zfit.core.interfaces.ZfitModel method*), 91
- `get_params()` (*zfit.core.interfaces.ZfitNumeric method*), 93
- `get_params()` (*zfit.core.interfaces.ZfitParameter method*), 95
- `get_params()` (*zfit.core.interfaces.ZfitPDF method*), 93
- `get_params()` (*zfit.core.parameter.BaseComposedParameter method*), 110
- `get_params()` (*zfit.core.parameter.BaseParameter method*), 111
- `get_params()` (*zfit.core.parameter.BaseZParameter method*), 111
- `get_params()` (*zfit.core.parameter.ComplexParameter method*), 113
- `get_params()` (*zfit.core.parameter.ComposedParameter method*), 114
- `get_params()` (*zfit.core.parameter.ConstantParameter method*), 115
- `get_params()` (*zfit.core.parameter.Parameter method*), 120
- `get_params()` (*zfit.core.parameter.ZfitParameterMixin method*), 137
- `get_params()` (*zfit.func.BaseFunc method*), 356
- `get_params()` (*zfit.func.ProdFunc method*), 361
- `get_params()` (*zfit.func.SimpleFunc method*), 371
- `get_params()` (*zfit.func.SumFunc method*), 366
- `get_params()` (*zfit.models.basefunctor.FunctorMixin method*), 155
- `get_params()` (*zfit.models.basic.CustomGaussOLD method*), 161
- `get_params()` (*zfit.models.basic.Exponential method*), 168
- `get_params()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 175
- `get_params()` (*zfit.models.dist\_tfp.Gauss method*), 181
- `get_params()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 189
- `get_params()` (*zfit.models.dist\_tfp.Uniform method*), 195
- `get_params()` (*zfit.models.dist\_tfp.WrapDistribution method*), 202
- `get_params()` (*zfit.models.functions.BaseFunctorFunc method*), 208
- `get_params()` (*zfit.models.functions.ProdFunc method*), 213
- `get_params()` (*zfit.models.functions.SimpleFunc method*), 219
- `get_params()` (*zfit.models.functions.SumFunc method*), 224
- `get_params()` (*zfit.models.functions.ZFunc method*), 229
- `get_params()` (*zfit.models.functor.BaseFunctor method*), 234
- `get_params()` (*zfit.models.functor.ProductPDF method*), 241
- `get_params()` (*zfit.models.functor.SumPDF method*), 248
- `get_params()` (*zfit.models.physics.CrystalBall method*), 255
- `get_params()` (*zfit.models.physics.DoubleCB method*), 263
- `get_params()` (*zfit.models.polynomials.Chebyshev method*), 270
- `get_params()` (*zfit.models.polynomials.Chebyshev2 method*), 277
- `get_params()` (*zfit.models.polynomials.Hermite method*), 285
- `get_params()` (*zfit.models.polynomials.Laguerre method*), 292

`get_params()` (*zfit.models.polynomials.Legendre method*), 299  
`get_params()` (*zfit.models.polynomials.RecursivePolynomial method*), 306  
`get_params()` (*zfit.models.special.SimpleFunctorPDF method*), 314  
`get_params()` (*zfit.models.special.SimplePDF method*), 320  
`get_params()` (*zfit.models.special.ZPDF method*), 327  
`get_params()` (*zfit.param.ComplexParameter method*), 395  
`get_params()` (*zfit.param.ComposedParameter method*), 394  
`get_params()` (*zfit.param.ConstantParameter method*), 383  
`get_params()` (*zfit.param.Parameter method*), 387  
`get_params()` (*zfit.Parameter method*), 35  
`get_params()` (*zfit.pdf.BaseFunctor method*), 406  
`get_params()` (*zfit.pdf.BasePDF method*), 399  
`get_params()` (*zfit.pdf.Chebyshev method*), 462  
`get_params()` (*zfit.pdf.Chebyshev2 method*), 476  
`get_params()` (*zfit.pdf.CrystalBall method*), 420  
`get_params()` (*zfit.pdf.DoubleCB method*), 427  
`get_params()` (*zfit.pdf.Exponential method*), 412  
`get_params()` (*zfit.pdf.Gauss method*), 434  
`get_params()` (*zfit.pdf.Hermite method*), 483  
`get_params()` (*zfit.pdf.Laguerre method*), 490  
`get_params()` (*zfit.pdf.Legendre method*), 469  
`get_params()` (*zfit.pdf.ProductPDF method*), 504  
`get_params()` (*zfit.pdf.RecursivePolynomial method*), 497  
`get_params()` (*zfit.pdf.SimpleFunctorPDF method*), 531  
`get_params()` (*zfit.pdf.SimplePDF method*), 524  
`get_params()` (*zfit.pdf.SumPDF method*), 511  
`get_params()` (*zfit.pdf.TruncatedGauss method*), 448  
`get_params()` (*zfit.pdf.Uniform method*), 441  
`get_params()` (*zfit.pdf.WrapDistribution method*), 455  
`get_params()` (*zfit.pdf.ZPDF method*), 518  
`get_reorder_indices()` (*zfit.core.limits.Space method*), 99  
`get_reorder_indices()` (*zfit.core.sample.EventSpace method*), 139  
`get_reorder_indices()` (*zfit.Space method*), 45  
`get_same_obs` (*in module zfit.core.dimension*), 84  
`get_shape()` (*zfit.core.parameter.Parameter method*), 120  
`get_shape()` (*zfit.core.parameter.TFBaseVariable method*), 131  
`get_shape()` (*zfit.param.Parameter method*), 388  
`get_shape()` (*zfit.Parameter method*), 35  
`get_subspace()` (*zfit.core.interfaces.ZfitSpace method*), 96  
`get_subspace()` (*zfit.core.limits.Space method*), 99  
`get_subspace()` (*zfit.core.sample.EventSpace method*), 139  
`get_subspace()` (*zfit.Space method*), 45  
`get_verbosity()` (*in module zfit.settings*), 535  
`get_yield()` (*zfit.core.basepdf.BasePDF method*), 63  
`get_yield()` (*zfit.core.interfaces.ZfitPDF method*), 93  
`get_yield()` (*zfit.models.basic.CustomGaussOLD method*), 161  
`get_yield()` (*zfit.models.basic.Exponential method*), 168  
`get_yield()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 175  
`get_yield()` (*zfit.models.dist\_tfp.Gauss method*), 182  
`get_yield()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 189  
`get_yield()` (*zfit.models.dist\_tfp.Uniform method*), 196  
`get_yield()` (*zfit.models.dist\_tfp.WrapDistribution method*), 202  
`get_yield()` (*zfit.models.functor.BaseFunctor method*), 234  
`get_yield()` (*zfit.models.functor.ProductPDF method*), 241  
`get_yield()` (*zfit.models.functor.SumPDF method*), 248  
`get_yield()` (*zfit.models.physics.CrystalBall method*), 256  
`get_yield()` (*zfit.models.physics.DoubleCB method*), 263  
`get_yield()` (*zfit.models.polynomials.Chebyshev method*), 270  
`get_yield()` (*zfit.models.polynomials.Chebyshev2 method*), 278  
`get_yield()` (*zfit.models.polynomials.Hermite method*), 285  
`get_yield()` (*zfit.models.polynomials.Laguerre method*), 292  
`get_yield()` (*zfit.models.polynomials.Legendre method*), 299  
`get_yield()` (*zfit.models.polynomials.RecursivePolynomial method*), 306  
`get_yield()` (*zfit.models.special.SimpleFunctorPDF method*), 314  
`get_yield()` (*zfit.models.special.SimplePDF method*), 320  
`get_yield()` (*zfit.models.special.ZPDF method*), 327  
`get_yield()` (*zfit.pdf.BaseFunctor method*), 406  
`get_yield()` (*zfit.pdf.BasePDF method*), 399  
`get_yield()` (*zfit.pdf.Chebyshev method*), 462  
`get_yield()` (*zfit.pdf.Chebyshev2 method*), 476  
`get_yield()` (*zfit.pdf.CrystalBall method*), 420  
`get_yield()` (*zfit.pdf.DoubleCB method*), 427



- `get_yield()` (*zfit.pdf.Exponential method*), 413
- `get_yield()` (*zfit.pdf.Gauss method*), 434
- `get_yield()` (*zfit.pdf.Hermite method*), 483
- `get_yield()` (*zfit.pdf.Laguerre method*), 490
- `get_yield()` (*zfit.pdf.Legendre method*), 469
- `get_yield()` (*zfit.pdf.ProductPDF method*), 504
- `get_yield()` (*zfit.pdf.RecursivePolynomial method*), 497
- `get_yield()` (*zfit.pdf.SimpleFuncPDF method*), 531
- `get_yield()` (*zfit.pdf.SimplePDF method*), 524
- `get_yield()` (*zfit.pdf.SumPDF method*), 511
- `get_yield()` (*zfit.pdf.TruncatedGauss method*), 448
- `get_yield()` (*zfit.pdf.Uniform method*), 441
- `get_yield()` (*zfit.pdf.WrapDistribution method*), 455
- `get_yield()` (*zfit.pdf.ZPDF method*), 518
- `gradients()` (*zfit.core.basefunc.BaseFunc method*), 51
- `gradients()` (*zfit.core.basemodel.BaseModel method*), 56
- `gradients()` (*zfit.core.basepdf.BasePDF method*), 64
- `gradients()` (*zfit.core.interfaces.ZfitLoss method*), 91
- `gradients()` (*zfit.core.loss.BaseLoss method*), 103
- `gradients()` (*zfit.core.loss.CachedLoss method*), 104
- `gradients()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 105
- `gradients()` (*zfit.core.loss.SimpleLoss method*), 107
- `gradients()` (*zfit.core.loss.UnbinnedNLL method*), 108
- `gradients()` (*zfit.func.BaseFunc method*), 356
- `gradients()` (*zfit.func.ProdFunc method*), 361
- `gradients()` (*zfit.func.SimpleFunc method*), 371
- `gradients()` (*zfit.func.SumFunc method*), 366
- `gradients()` (*zfit.loss.BaseLoss method*), 377
- `gradients()` (*zfit.loss.ExtendedUnbinnedNLL method*), 375
- `gradients()` (*zfit.loss.SimpleLoss method*), 378
- `gradients()` (*zfit.loss.UnbinnedNLL method*), 376
- `gradients()` (*zfit.models.basefunc.FuncMixin method*), 156
- `gradients()` (*zfit.models.basic.CustomGaussOLD method*), 161
- `gradients()` (*zfit.models.basic.Exponential method*), 168
- `gradients()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 175
- `gradients()` (*zfit.models.dist\_tfp.Gauss method*), 182
- `gradients()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 189
- `gradients()` (*zfit.models.dist\_tfp.Uniform method*), 196
- `gradients()` (*zfit.models.dist\_tfp.WrapDistribution method*), 202
- `gradients()` (*zfit.models.functions.BaseFuncPDF method*), 209
- `gradients()` (*zfit.models.functions.ProdFunc method*), 214
- `gradients()` (*zfit.models.functions.SimpleFunc method*), 219
- `gradients()` (*zfit.models.functions.SumFunc method*), 224
- `gradients()` (*zfit.models.functions.ZFunc method*), 229
- `gradients()` (*zfit.models.func.BaseFunc method*), 235
- `gradients()` (*zfit.models.func.ProductPDF method*), 241
- `gradients()` (*zfit.models.func.SumPDF method*), 248
- `gradients()` (*zfit.models.physics.CrystalBall method*), 256
- `gradients()` (*zfit.models.physics.DoubleCB method*), 263
- `gradients()` (*zfit.models.polynomials.Chebyshev method*), 271
- `gradients()` (*zfit.models.polynomials.Chebyshev2 method*), 278
- `gradients()` (*zfit.models.polynomials.Hermite method*), 285
- `gradients()` (*zfit.models.polynomials.Laguerre method*), 292
- `gradients()` (*zfit.models.polynomials.Legendre method*), 299
- `gradients()` (*zfit.models.polynomials.RecursivePolynomial method*), 306
- `gradients()` (*zfit.models.special.SimpleFuncPDF method*), 314
- `gradients()` (*zfit.models.special.SimplePDF method*), 321
- `gradients()` (*zfit.models.special.ZPDF method*), 327
- `gradients()` (*zfit.pdf.BaseFunc method*), 406
- `gradients()` (*zfit.pdf.BasePDF method*), 399
- `gradients()` (*zfit.pdf.Chebyshev method*), 462
- `gradients()` (*zfit.pdf.Chebyshev2 method*), 476
- `gradients()` (*zfit.pdf.CrystalBall method*), 420
- `gradients()` (*zfit.pdf.DoubleCB method*), 427
- `gradients()` (*zfit.pdf.Exponential method*), 413
- `gradients()` (*zfit.pdf.Gauss method*), 434
- `gradients()` (*zfit.pdf.Hermite method*), 483
- `gradients()` (*zfit.pdf.Laguerre method*), 491
- `gradients()` (*zfit.pdf.Legendre method*), 469
- `gradients()` (*zfit.pdf.ProductPDF method*), 504
- `gradients()` (*zfit.pdf.RecursivePolynomial method*), 497
- `gradients()` (*zfit.pdf.SimpleFuncPDF method*), 531
- `gradients()` (*zfit.pdf.SimplePDF method*), 524

`gradients()` (*zfit.pdf.SumPDF method*), 511  
`gradients()` (*zfit.pdf.TruncatedGauss method*), 448  
`gradients()` (*zfit.pdf.Uniform method*), 441  
`gradients()` (*zfit.pdf.WrapDistribution method*), 455  
`gradients()` (*zfit.pdf.ZPDF method*), 518  
`graph` (*zfit.core.parameter.Parameter attribute*), 120  
`graph` (*zfit.core.parameter.TFBaseVariable attribute*), 131  
`graph` (*zfit.param.Parameter attribute*), 388  
`graph` (*zfit.Parameter attribute*), 35  
`graph_caching_methods` (*zfit.ComplexParameter attribute*), 43  
`graph_caching_methods` (*zfit.ComposedParameter attribute*), 42  
`graph_caching_methods` (*zfit.constraint.GaussianConstraint attribute*), 350  
`graph_caching_methods` (*zfit.constraint.SimpleConstraint attribute*), 349  
`graph_caching_methods` (*zfit.core.basefunc.BaseFunc attribute*), 51  
`graph_caching_methods` (*zfit.core.basemodel.BaseModel attribute*), 56  
`graph_caching_methods` (*zfit.core.baseobject.BaseNumeric attribute*), 60  
`graph_caching_methods` (*zfit.core.basepdf.BasePDF attribute*), 64  
`graph_caching_methods` (*zfit.core.constraint.BaseConstraint attribute*), 68  
`graph_caching_methods` (*zfit.core.constraint.DistributionConstraint attribute*), 69  
`graph_caching_methods` (*zfit.core.constraint.GaussianConstraint attribute*), 71  
`graph_caching_methods` (*zfit.core.constraint.SimpleConstraint attribute*), 72  
`graph_caching_methods` (*zfit.core.data.Data attribute*), 75  
`graph_caching_methods` (*zfit.core.data.SampleData attribute*), 78  
`graph_caching_methods` (*zfit.core.data.Sampler attribute*), 81  
`graph_caching_methods` (*zfit.core.loss.BaseLoss attribute*), 103  
`graph_caching_methods` (*zfit.core.loss.CachedLoss attribute*), 104  
`graph_caching_methods` (*zfit.core.loss.ExtendedUnbinnedNLL attribute*), 105  
`graph_caching_methods` (*zfit.core.loss.SimpleLoss attribute*), 107  
`graph_caching_methods` (*zfit.core.loss.UnbinnedNLL attribute*), 108  
`graph_caching_methods` (*zfit.core.parameter.BaseComposedParameter attribute*), 110  
`graph_caching_methods` (*zfit.core.parameter.BaseZParameter attribute*), 112  
`graph_caching_methods` (*zfit.core.parameter.ComplexParameter attribute*), 113  
`graph_caching_methods` (*zfit.core.parameter.ComposedParameter attribute*), 114  
`graph_caching_methods` (*zfit.core.parameter.ConstantParameter attribute*), 115  
`graph_caching_methods` (*zfit.core.parameter.Parameter attribute*), 120  
`graph_caching_methods` (*zfit.core.parameter.ZfitParameterMixin attribute*), 137  
`graph_caching_methods` (*zfit.data.Data attribute*), 353  
`graph_caching_methods` (*zfit.func.BaseFunc attribute*), 356  
`graph_caching_methods` (*zfit.func.ProdFunc attribute*), 361  
`graph_caching_methods` (*zfit.func.SimpleFunc attribute*), 371  
`graph_caching_methods` (*zfit.func.SumFunc attribute*), 366  
`graph_caching_methods` (*zfit.loss.BaseLoss attribute*), 377  
`graph_caching_methods` (*zfit.loss.ExtendedUnbinnedNLL attribute*), 375  
`graph_caching_methods` (*zfit.loss.SimpleLoss attribute*), 378  
`graph_caching_methods` (*zfit.loss.UnbinnedNLL attribute*), 376  
`graph_caching_methods` (*zfit.minimize.Minuit attribute*), 380  
`graph_caching_methods` (*zfit.minimizers.minimizer\_minuit.Minuit attribute*), 148  
`graph_caching_methods` (*zfit.models.basefunctor.FunctorMixin attribute*), 156  
`graph_caching_methods`

- ([zfit.models.basic.CustomGaussOLD attribute](#)), 285
- [161](#)
- [graph\\_caching\\_methods](#) ([zfit.models.basic.Exponential attribute](#)), 168
- [graph\\_caching\\_methods](#) ([zfit.models.dist\\_tfp.ExponentialTFP attribute](#)), 175
- [graph\\_caching\\_methods](#) ([zfit.models.dist\\_tfp.Gauss attribute](#)), 182
- [graph\\_caching\\_methods](#) ([zfit.models.dist\\_tfp.TruncatedGauss attribute](#)), 189
- [graph\\_caching\\_methods](#) ([zfit.models.dist\\_tfp.Uniform attribute](#)), 196
- [graph\\_caching\\_methods](#) ([zfit.models.dist\\_tfp.WrapDistribution attribute](#)), 202
- [graph\\_caching\\_methods](#) ([zfit.models.functions.BaseFuncторFunc attribute](#)), 209
- [graph\\_caching\\_methods](#) ([zfit.models.functions.ProdFunc attribute](#)), 214
- [graph\\_caching\\_methods](#) ([zfit.models.functions.SimpleFunc attribute](#)), 219
- [graph\\_caching\\_methods](#) ([zfit.models.functions.SumFunc attribute](#)), 224
- [graph\\_caching\\_methods](#) ([zfit.models.functions.ZFunc attribute](#)), 229
- [graph\\_caching\\_methods](#) ([zfit.models.functor.BaseFuncтор attribute](#)), 235
- [graph\\_caching\\_methods](#) ([zfit.models.functor.ProductPDF attribute](#)), 241
- [graph\\_caching\\_methods](#) ([zfit.models.functor.SumPDF attribute](#)), 248
- [graph\\_caching\\_methods](#) ([zfit.models.physics.CrystalBall attribute](#)), 256
- [graph\\_caching\\_methods](#) ([zfit.models.physics.DoubleCB attribute](#)), 263
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.Chebyshev attribute](#)), 271
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.Chebyshev2 attribute](#)), 278
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.Hermite attribute](#)), 285
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.Laguerre attribute](#)), 292
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.Legendre attribute](#)), 299
- [graph\\_caching\\_methods](#) ([zfit.models.polynomials.RecursivePolynomial attribute](#)), 306
- [graph\\_caching\\_methods](#) ([zfit.models.special.SimpleFuncторPDF attribute](#)), 314
- [graph\\_caching\\_methods](#) ([zfit.models.special.SimplePDF attribute](#)), 321
- [graph\\_caching\\_methods](#) ([zfit.models.special.ZPDF attribute](#)), 327
- [graph\\_caching\\_methods](#) ([zfit.param.ComplexParameter attribute](#)), 396
- [graph\\_caching\\_methods](#) ([zfit.param.ComposedParameter attribute](#)), 394
- [graph\\_caching\\_methods](#) ([zfit.param.ConstantParameter attribute](#)), 383
- [graph\\_caching\\_methods](#) ([zfit.param.Parameter attribute](#)), 388
- [graph\\_caching\\_methods](#) ([zfit.Parameter attribute](#)), 35
- [graph\\_caching\\_methods](#) ([zfit.pdf.BaseFuncтор attribute](#)), 406
- [graph\\_caching\\_methods](#) ([zfit.pdf.BasePDF attribute](#)), 399
- [graph\\_caching\\_methods](#) ([zfit.pdf.Chebyshev attribute](#)), 462
- [graph\\_caching\\_methods](#) ([zfit.pdf.Chebyshev2 attribute](#)), 476
- [graph\\_caching\\_methods](#) ([zfit.pdf.CrystalBall attribute](#)), 420
- [graph\\_caching\\_methods](#) ([zfit.pdf.DoubleCB attribute](#)), 427
- [graph\\_caching\\_methods](#) ([zfit.pdf.Exponential attribute](#)), 413
- [graph\\_caching\\_methods](#) ([zfit.pdf.Gauss attribute](#)), 434
- [graph\\_caching\\_methods](#) ([zfit.pdf.Hermite attribute](#)), 483
- [graph\\_caching\\_methods](#) ([zfit.pdf.Laguerre attribute](#)), 491
- [graph\\_caching\\_methods](#) ([zfit.pdf.Legendre attribute](#)), 469
- [graph\\_caching\\_methods](#) ([zfit.pdf.ProductPDF attribute](#)), 469

tribute), 504  
graph\_caching\_methods  
(zfit.pdf.RecursivePolynomial attribute), 498  
graph\_caching\_methods  
(zfit.pdf.SimpleFunctorPDF attribute), 531  
graph\_caching\_methods (zfit.pdf.SimplePDF attribute), 524  
graph\_caching\_methods (zfit.pdf.SumPDF attribute), 511  
graph\_caching\_methods (zfit.pdf.TruncatedGauss attribute), 448  
graph\_caching\_methods (zfit.pdf.Uniform attribute), 441  
graph\_caching\_methods  
(zfit.pdf.WrapDistribution attribute), 455  
graph\_caching\_methods (zfit.pdf.ZPDF attribute), 518  
graph\_caching\_methods (zfit.util.cache.Cachable attribute), 332  
graph\_caching\_methods  
(zfit.util.cache.FunctionCacheHolder attribute), 333

## H

handle (zfit.core.parameter.Parameter attribute), 120  
handle (zfit.core.parameter.TFBaseVariable attribute), 131  
handle (zfit.param.Parameter attribute), 388  
handle (zfit.Parameter attribute), 35  
has\_limits (zfit.core.parameter.Parameter attribute), 121  
has\_limits (zfit.param.Parameter attribute), 388  
has\_limits (zfit.Parameter attribute), 35  
Hermite (class in zfit.models.polynomials), 281  
Hermite (class in zfit.pdf), 480  
hermite\_shape() (in module zfit.models.polynomials), 311  
hesse() (zfit.minimizers.fitresult.FitResult method), 146  
hesse() (zfit.minimizers.interface.ZfitResult method), 147

## I

imag (zfit.ComplexParameter attribute), 43  
imag (zfit.core.parameter.ComplexParameter attribute), 113  
imag (zfit.param.ComplexParameter attribute), 396  
IncompatibleError, 336  
independent (zfit.ComplexParameter attribute), 43  
independent (zfit.ComposedParameter attribute), 42  
independent (zfit.core.interfaces.ZfitParameter attribute), 95

independent (zfit.core.parameter.BaseComposedParameter attribute), 110  
independent (zfit.core.parameter.BaseParameter attribute), 111  
independent (zfit.core.parameter.BaseZParameter attribute), 112  
independent (zfit.core.parameter.ComplexParameter attribute), 113  
independent (zfit.core.parameter.ComposedParameter attribute), 114  
independent (zfit.core.parameter.ConstantParameter attribute), 115  
independent (zfit.core.parameter.Parameter attribute), 121  
independent (zfit.param.ComplexParameter attribute), 396  
independent (zfit.param.ComposedParameter attribute), 394  
independent (zfit.param.ConstantParameter attribute), 383  
independent (zfit.param.Parameter attribute), 388  
independent (zfit.Parameter attribute), 35  
info (zfit.minimizers.fitresult.FitResult attribute), 146  
initial\_value (zfit.core.parameter.Parameter attribute), 121  
initial\_value (zfit.core.parameter.TFBaseVariable attribute), 131  
initial\_value (zfit.param.Parameter attribute), 388  
initial\_value (zfit.Parameter attribute), 35  
initialize() (zfit.core.data.Data method), 75  
initialize() (zfit.core.data.SampleData method), 78  
initialize() (zfit.core.data.Sampler method), 82  
initialize() (zfit.data.Data method), 353  
initialized\_value()  
(zfit.core.parameter.Parameter method), 121  
initialized\_value()  
(zfit.core.parameter.TFBaseVariable method), 131  
initialized\_value() (zfit.param.Parameter method), 388  
initialized\_value() (zfit.Parameter method), 35  
initializer (zfit.core.parameter.Parameter attribute), 121  
initializer (zfit.core.parameter.TFBaseVariable attribute), 132  
initializer (zfit.param.Parameter attribute), 388  
initializer (zfit.Parameter attribute), 36  
Integral (class in zfit.core.integration), 86  
integrate() (zfit.core.basefunc.BaseFunc method), 51  
integrate() (zfit.core.basemodel.BaseModel method), 56



- `integrate()` (*zfit.core.basepdf.BasePDF method*), 64
- `integrate()` (*zfit.core.integration.AnalyticIntegral method*), 85
- `integrate()` (*zfit.core.interfaces.ZfitFunc method*), 89
- `integrate()` (*zfit.core.interfaces.ZfitModel method*), 91
- `integrate()` (*zfit.core.interfaces.ZfitPDF method*), 93
- `integrate()` (*zfit.func.BaseFunc method*), 356
- `integrate()` (*zfit.func.ProdFunc method*), 361
- `integrate()` (*zfit.func.SimpleFunc method*), 372
- `integrate()` (*zfit.func.SumFunc method*), 366
- `integrate()` (*zfit.models.basefunctor.FunctorMixin method*), 156
- `integrate()` (*zfit.models.basic.CustomGaussOLD method*), 161
- `integrate()` (*zfit.models.basic.Exponential method*), 168
- `integrate()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 175
- `integrate()` (*zfit.models.dist\_tfp.Gauss method*), 182
- `integrate()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 189
- `integrate()` (*zfit.models.dist\_tfp.Uniform method*), 196
- `integrate()` (*zfit.models.dist\_tfp.WrapDistribution method*), 202
- `integrate()` (*zfit.models.functions.BaseFunctorFunc method*), 209
- `integrate()` (*zfit.models.functions.ProdFunc method*), 214
- `integrate()` (*zfit.models.functions.SimpleFunc method*), 219
- `integrate()` (*zfit.models.functions.SumFunc method*), 224
- `integrate()` (*zfit.models.functions.ZFunc method*), 229
- `integrate()` (*zfit.models.functor.BaseFunctor method*), 235
- `integrate()` (*zfit.models.functor.ProductPDF method*), 241
- `integrate()` (*zfit.models.functor.SumPDF method*), 248
- `integrate()` (*zfit.models.physics.CrystalBall method*), 256
- `integrate()` (*zfit.models.physics.DoubleCB method*), 263
- `integrate()` (*zfit.models.polynomials.Chebyshev method*), 271
- `integrate()` (*zfit.models.polynomials.Chebyshev2 method*), 278
- `integrate()` (*zfit.models.polynomials.Hermite method*), 285
- `integrate()` (*zfit.models.polynomials.Laguerre method*), 292
- `integrate()` (*zfit.models.polynomials.Legendre method*), 299
- `integrate()` (*zfit.models.polynomials.RecursivePolynomial method*), 306
- `integrate()` (*zfit.models.special.SimpleFunctorPDF method*), 314
- `integrate()` (*zfit.models.special.SimplePDF method*), 321
- `integrate()` (*zfit.models.special.ZPDF method*), 327
- `integrate()` (*zfit.pdf.BaseFunctor method*), 406
- `integrate()` (*zfit.pdf.BasePDF method*), 399
- `integrate()` (*zfit.pdf.Chebyshev method*), 462
- `integrate()` (*zfit.pdf.Chebyshev2 method*), 476
- `integrate()` (*zfit.pdf.CrystalBall method*), 420
- `integrate()` (*zfit.pdf.DoubleCB method*), 427
- `integrate()` (*zfit.pdf.Exponential method*), 413
- `integrate()` (*zfit.pdf.Gauss method*), 434
- `integrate()` (*zfit.pdf.Hermite method*), 484
- `integrate()` (*zfit.pdf.Laguerre method*), 491
- `integrate()` (*zfit.pdf.Legendre method*), 469
- `integrate()` (*zfit.pdf.ProductPDF method*), 504
- `integrate()` (*zfit.pdf.RecursivePolynomial method*), 498
- `integrate()` (*zfit.pdf.SimpleFunctorPDF method*), 531
- `integrate()` (*zfit.pdf.SimplePDF method*), 524
- `integrate()` (*zfit.pdf.SumPDF method*), 511
- `integrate()` (*zfit.pdf.TruncatedGauss method*), 448
- `integrate()` (*zfit.pdf.Uniform method*), 441
- `integrate()` (*zfit.pdf.WrapDistribution method*), 455
- `integrate()` (*zfit.pdf.ZPDF method*), 518
- `Integration` (class in *zfit.core.integration*), 86
- `IntentionNotUnambiguousError`, 337
- `interpolate()` (in module *zfit.z.math*), 344
- `invalidates_cache()` (in module *zfit.util.cache*), 334
- `is_combinable()` (in module *zfit.core.dimension*), 84
- `is_container()` (in module *zfit.util.container*), 335
- `is_extended` (*zfit.core.basepdf.BasePDF attribute*), 64
- `is_extended` (*zfit.core.interfaces.ZfitPDF attribute*), 94
- `is_extended` (*zfit.models.basic.CustomGaussOLD attribute*), 161
- `is_extended` (*zfit.models.basic.Exponential attribute*), 168
- `is_extended` (*zfit.models.dist\_tfp.ExponentialTFP attribute*), 175
- `is_extended` (*zfit.models.dist\_tfp.Gauss attribute*), 182
- `is_extended` (*zfit.models.dist\_tfp.TruncatedGauss attribute*), 189

- `is_extended` (`zfit.models.dist_tfp.Uniform` attribute), 196
  - `is_extended` (`zfit.models.dist_tfp.WrapDistribution` attribute), 202
  - `is_extended` (`zfit.models.functor.BaseFunctor` attribute), 235
  - `is_extended` (`zfit.models.functor.ProductPDF` attribute), 241
  - `is_extended` (`zfit.models.functor.SumPDF` attribute), 248
  - `is_extended` (`zfit.models.physics.CrystalBall` attribute), 256
  - `is_extended` (`zfit.models.physics.DoubleCB` attribute), 263
  - `is_extended` (`zfit.models.polynomials.Chebyshev` attribute), 271
  - `is_extended` (`zfit.models.polynomials.Chebyshev2` attribute), 278
  - `is_extended` (`zfit.models.polynomials.Hermite` attribute), 285
  - `is_extended` (`zfit.models.polynomials.Laguerre` attribute), 292
  - `is_extended` (`zfit.models.polynomials.Legendre` attribute), 300
  - `is_extended` (`zfit.models.polynomials.RecursivePolynomial` attribute), 306
  - `is_extended` (`zfit.models.special.SimpleFunctorPDF` attribute), 314
  - `is_extended` (`zfit.models.special.SimplePDF` attribute), 321
  - `is_extended` (`zfit.models.special.ZPDF` attribute), 327
  - `is_extended` (`zfit.pdf.BaseFunctor` attribute), 406
  - `is_extended` (`zfit.pdf.BasePDF` attribute), 399
  - `is_extended` (`zfit.pdf.Chebyshev` attribute), 462
  - `is_extended` (`zfit.pdf.Chebyshev2` attribute), 477
  - `is_extended` (`zfit.pdf.CrystalBall` attribute), 420
  - `is_extended` (`zfit.pdf.DoubleCB` attribute), 428
  - `is_extended` (`zfit.pdf.Exponential` attribute), 413
  - `is_extended` (`zfit.pdf.Gauss` attribute), 434
  - `is_extended` (`zfit.pdf.Hermite` attribute), 484
  - `is_extended` (`zfit.pdf.Laguerre` attribute), 491
  - `is_extended` (`zfit.pdf.Legendre` attribute), 469
  - `is_extended` (`zfit.pdf.ProductPDF` attribute), 504
  - `is_extended` (`zfit.pdf.RecursivePolynomial` attribute), 498
  - `is_extended` (`zfit.pdf.SimpleFunctorPDF` attribute), 531
  - `is_extended` (`zfit.pdf.SimplePDF` attribute), 524
  - `is_extended` (`zfit.pdf.SumPDF` attribute), 511
  - `is_extended` (`zfit.pdf.TruncatedGauss` attribute), 448
  - `is_extended` (`zfit.pdf.Uniform` attribute), 441
  - `is_extended` (`zfit.pdf.WrapDistribution` attribute), 455
  - `is_extended` (`zfit.pdf.ZPDF` attribute), 518
  - `is_generator` (`zfit.core.sample.EventSpace` attribute), 140
  - `is_initialized()` (`zfit.core.parameter.Parameter` method), 121
  - `is_initialized()` (`zfit.core.parameter.TFBaseVariable` method), 132
  - `is_initialized()` (`zfit.param.Parameter` method), 388
  - `is_initialized()` (`zfit.Parameter` method), 36
  - `IS_TENSOR` (`zfit.util.cache.FunctionCacheHolder` attribute), 333
  - `items()` (`zfit.util.container.DotDict` method), 334
  - `iter_areas()` (`zfit.core.interfaces.ZfitSpace` method), 96
  - `iter_areas()` (`zfit.core.limits.Space` method), 99
  - `iter_areas()` (`zfit.core.sample.EventSpace` method), 140
  - `iter_areas()` (`zfit.Space` method), 46
  - `iter_limits()` (`zfit.core.interfaces.ZfitSpace` method), 96
  - `iter_limits()` (`zfit.core.limits.Space` method), 99
  - `iter_limits()` (`zfit.core.sample.EventSpace` method), 140
  - `iter_limits()` (`zfit.Space` method), 46
  - `iterator` (`zfit.core.data.Data` attribute), 75
  - `iterator` (`zfit.core.data.SampleData` attribute), 78
  - `iterator` (`zfit.core.data.Sampler` attribute), 82
  - `iterator` (`zfit.data.Data` attribute), 353
- ## K
- `keys()` (`zfit.util.container.DotDict` method), 335
- ## L
- `Laguerre` (class in `zfit.models.polynomials`), 289
  - `Laguerre` (class in `zfit.pdf`), 487
  - `laguerre_shape()` (in module `zfit.models.polynomials`), 311
  - `laguerre_shape_alpha_minusone()` (in module `zfit.models.polynomials`), 311
  - `Legendre` (class in `zfit.models.polynomials`), 296
  - `Legendre` (class in `zfit.pdf`), 466
  - `legendre_integral()` (in module `zfit.models.polynomials`), 311
  - `legendre_shape()` (in module `zfit.models.polynomials`), 311
  - `LightDataset` (class in `zfit.core.data`), 76
  - `limit1d` (`zfit.core.limits.Space` attribute), 100
  - `limit1d` (`zfit.core.sample.EventSpace` attribute), 140
  - `limit1d` (`zfit.Space` attribute), 46
  - `limit2d` (`zfit.core.limits.Space` attribute), 100
  - `limit2d` (`zfit.core.sample.EventSpace` attribute), 140
  - `limit2d` (`zfit.Space` attribute), 46
  - `limits` (`zfit.core.interfaces.ZfitSpace` attribute), 96

- `limits` (`zfit.core.limits.Space` attribute), 100
  - `limits` (`zfit.core.sample.EventSpace` attribute), 140
  - `limits` (`zfit.Space` attribute), 46
  - `limits1d` (`zfit.core.limits.Space` attribute), 100
  - `limits1d` (`zfit.core.sample.EventSpace` attribute), 140
  - `limits1d` (`zfit.Space` attribute), 46
  - `limits_consistent()` (in module `zfit.core.dimension`), 84
  - `limits_overlap()` (in module `zfit.core.dimension`), 85
  - `LimitsIncompatibleError`, 337
  - `LimitsNotSpecifiedError`, 337
  - `LimitsOverdefinedError`, 337
  - `LimitsUnderdefinedError`, 337
  - `load()` (`zfit.core.parameter.Parameter` method), 121
  - `load()` (`zfit.core.parameter.TFBaseVariable` method), 132
  - `load()` (`zfit.param.Parameter` method), 388
  - `load()` (`zfit.Parameter` method), 36
  - `log()` (in module `zfit.z.wrapping_tf`), 346
  - `log_pdf()` (`zfit.core.basepdf.BasePDF` method), 64
  - `log_pdf()` (`zfit.models.basic.CustomGaussOLD` method), 161
  - `log_pdf()` (`zfit.models.basic.Exponential` method), 168
  - `log_pdf()` (`zfit.models.dist_tfp.ExponentialTFP` method), 175
  - `log_pdf()` (`zfit.models.dist_tfp.Gauss` method), 182
  - `log_pdf()` (`zfit.models.dist_tfp.TruncatedGauss` method), 189
  - `log_pdf()` (`zfit.models.dist_tfp.Uniform` method), 196
  - `log_pdf()` (`zfit.models.dist_tfp.WrapDistribution` method), 203
  - `log_pdf()` (`zfit.models.functor.BaseFunctor` method), 235
  - `log_pdf()` (`zfit.models.functor.ProductPDF` method), 241
  - `log_pdf()` (`zfit.models.functor.SumPDF` method), 248
  - `log_pdf()` (`zfit.models.physics.CrystalBall` method), 256
  - `log_pdf()` (`zfit.models.physics.DoubleCB` method), 263
  - `log_pdf()` (`zfit.models.polynomials.Chebyshev` method), 271
  - `log_pdf()` (`zfit.models.polynomials.Chebyshev2` method), 278
  - `log_pdf()` (`zfit.models.polynomials.Hermite` method), 285
  - `log_pdf()` (`zfit.models.polynomials.Laguerre` method), 292
  - `log_pdf()` (`zfit.models.polynomials.Legendre` method), 300
  - `log_pdf()` (`zfit.models.polynomials.RecursivePolynomial` method), 306
  - `log_pdf()` (`zfit.models.special.SimpleFunctorPDF` method), 314
  - `log_pdf()` (`zfit.models.special.SimplePDF` method), 321
  - `log_pdf()` (`zfit.models.special.ZPDF` method), 327
  - `log_pdf()` (`zfit.pdf.BaseFunctor` method), 406
  - `log_pdf()` (`zfit.pdf.BasePDF` method), 399
  - `log_pdf()` (`zfit.pdf.Chebyshev` method), 462
  - `log_pdf()` (`zfit.pdf.Chebyshev2` method), 477
  - `log_pdf()` (`zfit.pdf.CrystalBall` method), 420
  - `log_pdf()` (`zfit.pdf.DoubleCB` method), 428
  - `log_pdf()` (`zfit.pdf.Exponential` method), 413
  - `log_pdf()` (`zfit.pdf.Gauss` method), 435
  - `log_pdf()` (`zfit.pdf.Hermite` method), 484
  - `log_pdf()` (`zfit.pdf.Laguerre` method), 491
  - `log_pdf()` (`zfit.pdf.Legendre` method), 469
  - `log_pdf()` (`zfit.pdf.ProductPDF` method), 504
  - `log_pdf()` (`zfit.pdf.RecursivePolynomial` method), 498
  - `log_pdf()` (`zfit.pdf.SimpleFunctorPDF` method), 531
  - `log_pdf()` (`zfit.pdf.SimplePDF` method), 525
  - `log_pdf()` (`zfit.pdf.SumPDF` method), 511
  - `log_pdf()` (`zfit.pdf.TruncatedGauss` method), 448
  - `log_pdf()` (`zfit.pdf.Uniform` method), 441
  - `log_pdf()` (`zfit.pdf.WrapDistribution` method), 455
  - `log_pdf()` (`zfit.pdf.ZPDF` method), 518
  - `LogicalUndefinedOperationError`, 337
  - `loss` (`zfit.minimizers.fitresult.FitResult` attribute), 146
  - `loss` (`zfit.minimizers.interface.ZfitResult` attribute), 147
  - `lower` (`zfit.core.interfaces.ZfitSpace` attribute), 96
  - `lower` (`zfit.core.limits.Space` attribute), 100
  - `lower` (`zfit.core.sample.EventSpace` attribute), 141
  - `lower` (`zfit.Space` attribute), 47
  - `lower_limit` (`zfit.core.parameter.Parameter` attribute), 122
  - `lower_limit` (`zfit.param.Parameter` attribute), 389
  - `lower_limit` (`zfit.Parameter` attribute), 36
- ## M
- `mc_integrate()` (in module `zfit.core.integration`), 87
  - `MetaBaseParameter` (class in `zfit.core.parameter`), 116
  - `minimize()` (`zfit.minimize.Adam` method), 379
  - `minimize()` (`zfit.minimize.BFGS` method), 382
  - `minimize()` (`zfit.minimize.Minuit` method), 380
  - `minimize()` (`zfit.minimize.Scipy` method), 381
  - `minimize()` (`zfit.minimize.WrapOptimizer` method), 379
  - `minimize()` (`zfit.minimizers.base_tf.WrapOptimizer` method), 143
  - `minimize()` (`zfit.minimizers.baseminimizer.BaseMinimizer` method), 143
  - `minimize()` (`zfit.minimizers.interface.ZfitMinimizer` method), 146

`minimize()` (`zfit.minimizers.minimizer_minuit.Minuit` attribute), 148  
`minimize()` (`zfit.minimizers.minimizer_tfp.BFGS` attribute), 149  
`minimize()` (`zfit.minimizers.minimizers_scipy.Scipy` attribute), 149  
`minimize()` (`zfit.minimizers.optimizers_tf.Adam` attribute), 150  
`minimize()` (`zfit.minimizers.tf_external_optimizer.ExternalOptimizer` attribute), 151  
`minimize()` (`zfit.minimizers.tf_external_optimizer.ScipyOptimizer` attribute), 153  
`minimize_nan()` (`zfit.minimizers.baseminimizer.BaseStrategy` attribute), 144  
`minimize_nan()` (`zfit.minimizers.baseminimizer.DefaultStrategy` attribute), 144  
`minimize_nan()` (`zfit.minimizers.baseminimizer.ToyStrategy` attribute), 144  
`minimize_nan()` (`zfit.minimizers.baseminimizer.ZfitStrategy` attribute), 144  
`minimizer` (`zfit.minimizers.fitresult.FitResult` attribute), 146  
`minimizer` (`zfit.minimizers.interface.ZfitResult` attribute), 147  
`Minuit` (class in `zfit.minimize`), 380  
`Minuit` (class in `zfit.minimizers.minimizer_minuit`), 147  
`MinuitMinimizer` (in module `zfit.minimize`), 379  
`mod` (`zfit.ComplexParameter` attribute), 43  
`mod` (`zfit.core.parameter.ComplexParameter` attribute), 113  
`mod` (`zfit.param.ComplexParameter` attribute), 396  
`model` (`zfit.core.interfaces.ZfitLoss` attribute), 91  
`model` (`zfit.core.loss.BaseLoss` attribute), 104  
`model` (`zfit.core.loss.CachedLoss` attribute), 104  
`model` (`zfit.core.loss.ExtendedUnbinnedNLL` attribute), 106  
`model` (`zfit.core.loss.SimpleLoss` attribute), 107  
`model` (`zfit.core.loss.UnbinnedNLL` attribute), 108  
`model` (`zfit.loss.BaseLoss` attribute), 377  
`model` (`zfit.loss.ExtendedUnbinnedNLL` attribute), 375  
`model` (`zfit.loss.SimpleLoss` attribute), 378  
`model` (`zfit.loss.UnbinnedNLL` attribute), 376  
`model()` (`zfit.util.diverse.GaussianMixture2D` method), 335  
`model()` (`zfit.util.diverse.GaussianMixture4D` method), 335  
`ModelIncompatibleError`, 337  
`models` (`zfit.core.interfaces.ZfitFuncMixin` attribute), 90  
`models` (`zfit.func.ProdFunc` attribute), 361  
`models` (`zfit.func.SumFunc` attribute), 366  
`models` (`zfit.models.basefunc.FunctorMixin` attribute), 156  
`models` (`zfit.models.functions.BaseFunc` attribute), 209  
`models` (`zfit.models.functions.ProdFunc` attribute), 214  
`models` (`zfit.models.functions.SumFunc` attribute), 224  
`models` (`zfit.models.functor.BaseFunc` attribute), 235  
`models` (`zfit.models.functor.ProductPDF` attribute), 242  
`models` (`zfit.models.functor.SumPDF` attribute), 249  
`models` (`zfit.models.special.SimpleFuncPDF` attribute), 314  
`models` (`zfit.pdf.ProductPDF` attribute), 505  
`models` (`zfit.pdf.SimpleFuncPDF` attribute), 531  
`models` (`zfit.pdf.SumPDF` attribute), 512  
`multiply()` (`zfit.core.parameter.MetaBaseParameter` attribute), 116  
`multiply()` (in module `zfit.core.operations`), 109  
`multiply_func_func()` (in module `zfit.core.operations`), 109  
`multiply_param_func()` (in module `zfit.core.operations`), 109  
`multiply_param_param()` (in module `zfit.core.operations`), 109  
`multiply_param_pdf()` (in module `zfit.core.operations`), 109  
`multiply_pdf_pdf()` (in module `zfit.core.operations`), 109  
`multivariate_gauss()` (in module `zfit.util.diverse`), 335

## N

`n_cpu` (`zfit.util.execution.RunManager` attribute), 340  
`n_limits` (`zfit.core.interfaces.ZfitSpace` attribute), 96  
`n_limits` (`zfit.core.limits.Space` attribute), 100  
`n_limits` (`zfit.core.sample.EventSpace` attribute), 141  
`n_limits` (`zfit.Space` attribute), 47  
`n_obs` (`zfit.core.basefunc.BaseFunc` attribute), 51  
`n_obs` (`zfit.core.basemodel.BaseModel` attribute), 56  
`n_obs` (`zfit.core.basepdf.BasePDF` attribute), 64  
`n_obs` (`zfit.core.data.Data` attribute), 75  
`n_obs` (`zfit.core.data.SampleData` attribute), 78  
`n_obs` (`zfit.core.data.Sampler` attribute), 82  
`n_obs` (`zfit.core.dimension.BaseDimensional` attribute), 83  
`n_obs` (`zfit.core.integration.PartialIntegralSampleData` attribute), 87  
`n_obs` (`zfit.core.interfaces.ZfitData` attribute), 88  
`n_obs` (`zfit.core.interfaces.ZfitDimensional` attribute), 88  
`n_obs` (`zfit.core.interfaces.ZfitFunc` attribute), 89  
`n_obs` (`zfit.core.interfaces.ZfitModel` attribute), 91  
`n_obs` (`zfit.core.interfaces.ZfitPDF` attribute), 94  
`n_obs` (`zfit.core.interfaces.ZfitSpace` attribute), 96  
`n_obs` (`zfit.core.limits.Space` attribute), 100  
`n_obs` (`zfit.core.sample.EventSpace` attribute), 141  
`n_obs` (`zfit.data.Data` attribute), 353



- `n_obs` (`zfit.func.BaseFunc` attribute), 356
- `n_obs` (`zfit.func.ProdFunc` attribute), 361
- `n_obs` (`zfit.func.SimpleFunc` attribute), 372
- `n_obs` (`zfit.func.SumFunc` attribute), 366
- `n_obs` (`zfit.models.basefunc.FunctorMixin` attribute), 156
- `n_obs` (`zfit.models.basic.CustomGaussOLD` attribute), 162
- `n_obs` (`zfit.models.basic.Exponential` attribute), 169
- `n_obs` (`zfit.models.dist_tfp.ExponentialTFP` attribute), 175
- `n_obs` (`zfit.models.dist_tfp.Gauss` attribute), 182
- `n_obs` (`zfit.models.dist_tfp.TruncatedGauss` attribute), 189
- `n_obs` (`zfit.models.dist_tfp.Uniform` attribute), 196
- `n_obs` (`zfit.models.dist_tfp.WrapDistribution` attribute), 203
- `n_obs` (`zfit.models.functions.BaseFunctorFunc` attribute), 209
- `n_obs` (`zfit.models.functions.ProdFunc` attribute), 214
- `n_obs` (`zfit.models.functions.SimpleFunc` attribute), 219
- `n_obs` (`zfit.models.functions.SumFunc` attribute), 224
- `n_obs` (`zfit.models.functions.ZFunc` attribute), 229
- `n_obs` (`zfit.models.functor.BaseFunctor` attribute), 235
- `n_obs` (`zfit.models.functor.ProductPDF` attribute), 242
- `n_obs` (`zfit.models.functor.SumPDF` attribute), 249
- `n_obs` (`zfit.models.physics.CrystalBall` attribute), 256
- `n_obs` (`zfit.models.physics.DoubleCB` attribute), 264
- `n_obs` (`zfit.models.polynomials.Chebyshev` attribute), 271
- `n_obs` (`zfit.models.polynomials.Chebyshev2` attribute), 278
- `n_obs` (`zfit.models.polynomials.Hermite` attribute), 285
- `n_obs` (`zfit.models.polynomials.Laguerre` attribute), 293
- `n_obs` (`zfit.models.polynomials.Legendre` attribute), 300
- `n_obs` (`zfit.models.polynomials.RecursivePolynomial` attribute), 307
- `n_obs` (`zfit.models.special.SimpleFunctorPDF` attribute), 314
- `n_obs` (`zfit.models.special.SimplePDF` attribute), 321
- `n_obs` (`zfit.models.special.ZPDF` attribute), 328
- `n_obs` (`zfit.pdf.BaseFunctor` attribute), 406
- `n_obs` (`zfit.pdf.BasePDF` attribute), 400
- `n_obs` (`zfit.pdf.Chebyshev` attribute), 462
- `n_obs` (`zfit.pdf.Chebyshev2` attribute), 477
- `n_obs` (`zfit.pdf.CrystalBall` attribute), 421
- `n_obs` (`zfit.pdf.DoubleCB` attribute), 428
- `n_obs` (`zfit.pdf.Exponential` attribute), 413
- `n_obs` (`zfit.pdf.Gauss` attribute), 435
- `n_obs` (`zfit.pdf.Hermite` attribute), 484
- `n_obs` (`zfit.pdf.Laguerre` attribute), 491
- `n_obs` (`zfit.pdf.Legendre` attribute), 470
- `n_obs` (`zfit.pdf.ProductPDF` attribute), 505
- `n_obs` (`zfit.pdf.RecursivePolynomial` attribute), 498
- `n_obs` (`zfit.pdf.SimpleFunctorPDF` attribute), 531
- `n_obs` (`zfit.pdf.SimplePDF` attribute), 525
- `n_obs` (`zfit.pdf.SumPDF` attribute), 512
- `n_obs` (`zfit.pdf.TruncatedGauss` attribute), 449
- `n_obs` (`zfit.pdf.Uniform` attribute), 442
- `n_obs` (`zfit.pdf.WrapDistribution` attribute), 455
- `n_obs` (`zfit.pdf.ZPDF` attribute), 518
- `n_obs` (`zfit.Space` attribute), 47
- `n_samples` (`zfit.core.data.Sampler` attribute), 82
- `name` (`zfit.ComplexParameter` attribute), 43
- `name` (`zfit.ComposedParameter` attribute), 42
- `name` (`zfit.constraint.GaussianConstraint` attribute), 350
- `name` (`zfit.constraint.SimpleConstraint` attribute), 349
- `name` (`zfit.core.basefunc.BaseFunc` attribute), 51
- `name` (`zfit.core.basemodel.BaseModel` attribute), 56
- `name` (`zfit.core.baseobject.BaseNumeric` attribute), 60
- `name` (`zfit.core.baseobject.BaseObject` attribute), 60
- `name` (`zfit.core.basepdf.BasePDF` attribute), 64
- `name` (`zfit.core.constraint.BaseConstraint` attribute), 68
- `name` (`zfit.core.constraint.DistributionConstraint` attribute), 69
- `name` (`zfit.core.constraint.GaussianConstraint` attribute), 71
- `name` (`zfit.core.constraint.SimpleConstraint` attribute), 72
- `name` (`zfit.core.data.Data` attribute), 75
- `name` (`zfit.core.data.SampleData` attribute), 78
- `name` (`zfit.core.data.Sampler` attribute), 82
- `name` (`zfit.core.dimension.BaseDimensional` attribute), 83
- `name` (`zfit.core.integration.PartialIntegralSampleData` attribute), 87
- `name` (`zfit.core.interfaces.ZfitData` attribute), 88
- `name` (`zfit.core.interfaces.ZfitDimensional` attribute), 88
- `name` (`zfit.core.interfaces.ZfitFunc` attribute), 89
- `name` (`zfit.core.interfaces.ZfitLoss` attribute), 91
- `name` (`zfit.core.interfaces.ZfitModel` attribute), 91
- `name` (`zfit.core.interfaces.ZfitNumeric` attribute), 93
- `name` (`zfit.core.interfaces.ZfitObject` attribute), 93
- `name` (`zfit.core.interfaces.ZfitParameter` attribute), 95
- `name` (`zfit.core.interfaces.ZfitPDF` attribute), 94
- `name` (`zfit.core.interfaces.ZfitSpace` attribute), 96
- `name` (`zfit.core.limits.Space` attribute), 101
- `name` (`zfit.core.loss.BaseLoss` attribute), 104
- `name` (`zfit.core.loss.CachedLoss` attribute), 105
- `name` (`zfit.core.loss.ExtendedUnbinnedNLL` attribute), 106
- `name` (`zfit.core.loss.SimpleLoss` attribute), 107
- `name` (`zfit.core.loss.UnbinnedNLL` attribute), 108
- `name` (`zfit.core.parameter.BaseComposedParameter` attribute), 110
- `name` (`zfit.core.parameter.BaseParameter` attribute), 111
- `name` (`zfit.core.parameter.BaseZParameter` attribute), 112

- name (*zfit.core.parameter.ComplexParameter* attribute), 113
- name (*zfit.core.parameter.ComposedParameter* attribute), 114
- name (*zfit.core.parameter.ConstantParameter* attribute), 115
- name (*zfit.core.parameter.Parameter* attribute), 122
- name (*zfit.core.parameter.TFBaseVariable* attribute), 132
- name (*zfit.core.parameter.ZfitParameterMixin* attribute), 137
- name (*zfit.core.sample.EventSpace* attribute), 141
- name (*zfit.data.Data* attribute), 353
- name (*zfit.func.BaseFunc* attribute), 356
- name (*zfit.func.ProdFunc* attribute), 361
- name (*zfit.func.SimpleFunc* attribute), 372
- name (*zfit.func.SumFunc* attribute), 366
- name (*zfit.loss.BaseLoss* attribute), 377
- name (*zfit.loss.ExtendedUnbinnedNLL* attribute), 375
- name (*zfit.loss.SimpleLoss* attribute), 378
- name (*zfit.loss.UnbinnedNLL* attribute), 376
- name (*zfit.models.basefunctor.FunctorMixin* attribute), 156
- name (*zfit.models.basic.CustomGaussOLD* attribute), 162
- name (*zfit.models.basic.Exponential* attribute), 169
- name (*zfit.models.dist\_tfp.ExponentialTFP* attribute), 175
- name (*zfit.models.dist\_tfp.Gauss* attribute), 182
- name (*zfit.models.dist\_tfp.TruncatedGauss* attribute), 189
- name (*zfit.models.dist\_tfp.Uniform* attribute), 196
- name (*zfit.models.dist\_tfp.WrapDistribution* attribute), 203
- name (*zfit.models.functions.BaseFunctorFunc* attribute), 209
- name (*zfit.models.functions.ProdFunc* attribute), 214
- name (*zfit.models.functions.SimpleFunc* attribute), 219
- name (*zfit.models.functions.SumFunc* attribute), 224
- name (*zfit.models.functions.ZFunc* attribute), 229
- name (*zfit.models.functor.BaseFunctor* attribute), 235
- name (*zfit.models.functor.ProductPDF* attribute), 242
- name (*zfit.models.functor.SumPDF* attribute), 249
- name (*zfit.models.physics.CrystalBall* attribute), 256
- name (*zfit.models.physics.DoubleCB* attribute), 264
- name (*zfit.models.polynomials.Chebyshev* attribute), 271
- name (*zfit.models.polynomials.Chebyshev2* attribute), 278
- name (*zfit.models.polynomials.Hermite* attribute), 285
- name (*zfit.models.polynomials.Laguerre* attribute), 293
- name (*zfit.models.polynomials.Legendre* attribute), 300
- name (*zfit.models.polynomials.RecursivePolynomial* attribute), 307
- name (*zfit.models.special.SimpleFunctorPDF* attribute), 315
- name (*zfit.models.special.SimplePDF* attribute), 321
- name (*zfit.models.special.ZPDF* attribute), 328
- name (*zfit.param.ComplexParameter* attribute), 396
- name (*zfit.param.ComposedParameter* attribute), 394
- name (*zfit.param.ConstantParameter* attribute), 383
- name (*zfit.param.Parameter* attribute), 389
- name (*zfit.Parameter* attribute), 36
- name (*zfit.pdf.BaseFunctor* attribute), 406
- name (*zfit.pdf.BasePDF* attribute), 400
- name (*zfit.pdf.Chebyshev* attribute), 462
- name (*zfit.pdf.Chebyshev2* attribute), 477
- name (*zfit.pdf.CrystalBall* attribute), 421
- name (*zfit.pdf.DoubleCB* attribute), 428
- name (*zfit.pdf.Exponential* attribute), 413
- name (*zfit.pdf.Gauss* attribute), 435
- name (*zfit.pdf.Hermite* attribute), 484
- name (*zfit.pdf.Laguerre* attribute), 491
- name (*zfit.pdf.Legendre* attribute), 470
- name (*zfit.pdf.ProductPDF* attribute), 505
- name (*zfit.pdf.RecursivePolynomial* attribute), 498
- name (*zfit.pdf.SimpleFunctorPDF* attribute), 531
- name (*zfit.pdf.SimplePDF* attribute), 525
- name (*zfit.pdf.SumPDF* attribute), 512
- name (*zfit.pdf.TruncatedGauss* attribute), 449
- name (*zfit.pdf.Uniform* attribute), 442
- name (*zfit.pdf.WrapDistribution* attribute), 455
- name (*zfit.pdf.ZPDF* attribute), 518
- name (*zfit.Space* attribute), 47
- NameAlreadyTakenError, 338
- nevents (*zfit.core.data.Data* attribute), 75
- nevents (*zfit.core.data.SampleData* attribute), 78
- nevents (*zfit.core.data.Sampler* attribute), 82
- nevents (*zfit.data.Data* attribute), 353
- nll\_gaussian() (in module *zfit.constraint*), 348
- no\_multiple\_limits() (in module *zfit.core.limits*), 102
- no\_norm\_range() (in module *zfit.core.limits*), 102
- norm\_range (*zfit.core.basepdf.BasePDF* attribute), 64
- norm\_range (*zfit.models.basic.CustomGaussOLD* attribute), 162
- norm\_range (*zfit.models.basic.Exponential* attribute), 169
- norm\_range (*zfit.models.dist\_tfp.ExponentialTFP* attribute), 175
- norm\_range (*zfit.models.dist\_tfp.Gauss* attribute), 182
- norm\_range (*zfit.models.dist\_tfp.TruncatedGauss* attribute), 189
- norm\_range (*zfit.models.dist\_tfp.Uniform* attribute), 196
- norm\_range (*zfit.models.dist\_tfp.WrapDistribution* attribute), 203

`norm_range` (`zfit.models.functor.BaseFunctor` attribute), 235  
`norm_range` (`zfit.models.functor.ProductPDF` attribute), 242  
`norm_range` (`zfit.models.functor.SumPDF` attribute), 249  
`norm_range` (`zfit.models.physics.CrystalBall` attribute), 256  
`norm_range` (`zfit.models.physics.DoubleCB` attribute), 264  
`norm_range` (`zfit.models.polynomials.Chebyshev` attribute), 271  
`norm_range` (`zfit.models.polynomials.Chebyshev2` attribute), 278  
`norm_range` (`zfit.models.polynomials.Hermite` attribute), 286  
`norm_range` (`zfit.models.polynomials.Laguerre` attribute), 293  
`norm_range` (`zfit.models.polynomials.Legendre` attribute), 300  
`norm_range` (`zfit.models.polynomials.RecursivePolynomial` attribute), 307  
`norm_range` (`zfit.models.special.SimpleFunctorPDF` attribute), 315  
`norm_range` (`zfit.models.special.SimplePDF` attribute), 321  
`norm_range` (`zfit.models.special.ZPDF` attribute), 328  
`norm_range` (`zfit.pdf.BaseFunctor` attribute), 406  
`norm_range` (`zfit.pdf.BasePDF` attribute), 400  
`norm_range` (`zfit.pdf.Chebyshev` attribute), 463  
`norm_range` (`zfit.pdf.Chebyshev2` attribute), 477  
`norm_range` (`zfit.pdf.CrystalBall` attribute), 421  
`norm_range` (`zfit.pdf.DoubleCB` attribute), 428  
`norm_range` (`zfit.pdf.Exponential` attribute), 413  
`norm_range` (`zfit.pdf.Gauss` attribute), 435  
`norm_range` (`zfit.pdf.Hermite` attribute), 484  
`norm_range` (`zfit.pdf.Laguerre` attribute), 491  
`norm_range` (`zfit.pdf.Legendre` attribute), 470  
`norm_range` (`zfit.pdf.ProductPDF` attribute), 505  
`norm_range` (`zfit.pdf.RecursivePolynomial` attribute), 498  
`norm_range` (`zfit.pdf.SimpleFunctorPDF` attribute), 532  
`norm_range` (`zfit.pdf.SimplePDF` attribute), 525  
`norm_range` (`zfit.pdf.SumPDF` attribute), 512  
`norm_range` (`zfit.pdf.TruncatedGauss` attribute), 449  
`norm_range` (`zfit.pdf.Uniform` attribute), 442  
`norm_range` (`zfit.pdf.WrapDistribution` attribute), 455  
`norm_range` (`zfit.pdf.ZPDF` attribute), 518  
`normalization()` (`zfit.core.basepdf.BasePDF` method), 64  
`normalization()` (`zfit.core.interfaces.ZfitPDF` method), 94  
`normalization()` (`zfit.models.basic.CustomGaussOLD` method), 162  
`normalization()` (`zfit.models.basic.Exponential` method), 169  
`normalization()` (`zfit.models.dist_tfp.ExponentialTFP` method), 175  
`normalization()` (`zfit.models.dist_tfp.Gauss` method), 182  
`normalization()` (`zfit.models.dist_tfp.TruncatedGauss` method), 189  
`normalization()` (`zfit.models.dist_tfp.Uniform` method), 196  
`normalization()` (`zfit.models.dist_tfp.WrapDistribution` method), 203  
`normalization()` (`zfit.models.functor.BaseFunctor` method), 235  
`normalization()` (`zfit.models.functor.ProductPDF` method), 242  
`normalization()` (`zfit.models.functor.SumPDF` method), 249  
`normalization()` (`zfit.models.physics.CrystalBall` method), 256  
`normalization()` (`zfit.models.physics.DoubleCB` method), 264  
`normalization()` (`zfit.models.polynomials.Chebyshev` method), 271  
`normalization()` (`zfit.models.polynomials.Chebyshev2` method), 278  
`normalization()` (`zfit.models.polynomials.Hermite` method), 286  
`normalization()` (`zfit.models.polynomials.Laguerre` method), 293  
`normalization()` (`zfit.models.polynomials.Legendre` method), 300  
`normalization()` (`zfit.models.polynomials.RecursivePolynomial` method), 307  
`normalization()` (`zfit.models.special.SimpleFunctorPDF` method), 315  
`normalization()` (`zfit.models.special.SimplePDF` method), 321  
`normalization()` (`zfit.models.special.ZPDF` method), 328  
`normalization()` (`zfit.pdf.BaseFunctor` method), 407  
`normalization()` (`zfit.pdf.BasePDF` method), 400  
`normalization()` (`zfit.pdf.Chebyshev` method), 463  
`normalization()` (`zfit.pdf.Chebyshev2` method), 477  
`normalization()` (`zfit.pdf.CrystalBall` method), 421  
`normalization()` (`zfit.pdf.DoubleCB` method), 428  
`normalization()` (`zfit.pdf.Exponential` method), 413  
`normalization()` (`zfit.pdf.Gauss` method), 435  
`normalization()` (`zfit.pdf.Hermite` method), 484  
`normalization()` (`zfit.pdf.Laguerre` method), 491  
`normalization()` (`zfit.pdf.Legendre` method), 470  
`normalization()` (`zfit.pdf.ProductPDF` method),

- 505
- `normalization()` (*zfit.pdf.RecursivePolynomial method*), 498
- `normalization()` (*zfit.pdf.SimpleFunctorPDF method*), 532
- `normalization()` (*zfit.pdf.SimplePDF method*), 525
- `normalization()` (*zfit.pdf.SumPDF method*), 512
- `normalization()` (*zfit.pdf.TruncatedGauss method*), 449
- `normalization()` (*zfit.pdf.Uniform method*), 442
- `normalization()` (*zfit.pdf.WrapDistribution method*), 456
- `normalization()` (*zfit.pdf.ZPDF method*), 518
- `normalization_chunked()` (in module *zfit.core.integration*), 88
- `normalization_nograd()` (in module *zfit.core.integration*), 88
- `NormRangeNotImplementedError`, 338
- `NormRangeNotSpecifiedError`, 338
- `NoSessionSpecifiedError`, 338
- `NotExtendedPDFError`, 338
- `NotMinimizedError`, 338
- `NotSpecified` (class in *zfit.util.checks*), 334
- `NotSpecifiedError`, 338
- `nth_pow()` (in module *zfit.z.zextension*), 346
- `numeric_integrate()` (in module *zfit.core.integration*), 88
- `numeric_integrate()` (*zfit.core.basefunc.BaseFunc method*), 51
- `numeric_integrate()` (*zfit.core.basemodel.BaseModel method*), 56
- `numeric_integrate()` (*zfit.core.basepdf.BasePDF method*), 64
- `numeric_integrate()` (*zfit.func.BaseFunc method*), 357
- `numeric_integrate()` (*zfit.func.ProdFunc method*), 361
- `numeric_integrate()` (*zfit.func.SimpleFunc method*), 372
- `numeric_integrate()` (*zfit.func.SumFunc method*), 366
- `numeric_integrate()` (*zfit.models.basefunctor.FunctorMixin method*), 156
- `numeric_integrate()` (*zfit.models.basic.CustomGaussOLD method*), 162
- `numeric_integrate()` (*zfit.models.basic.Exponential method*), 169
- `numeric_integrate()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 176
- `numeric_integrate()` (*zfit.models.dist\_tfp.Gauss method*), 183
- `numeric_integrate()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 190
- `numeric_integrate()` (*zfit.models.dist\_tfp.Uniform method*), 196
- `numeric_integrate()` (*zfit.models.dist\_tfp.WrapDistribution method*), 203
- `numeric_integrate()` (*zfit.models.functions.BaseFunctorFunc method*), 209
- `numeric_integrate()` (*zfit.models.functions.ProdFunc method*), 214
- `numeric_integrate()` (*zfit.models.functions.SimpleFunc method*), 219
- `numeric_integrate()` (*zfit.models.functions.SumFunc method*), 224
- `numeric_integrate()` (*zfit.models.functions.ZFunc method*), 229
- `numeric_integrate()` (*zfit.models.functor.BaseFunctor method*), 235
- `numeric_integrate()` (*zfit.models.functor.ProductPDF method*), 242
- `numeric_integrate()` (*zfit.models.functor.SumPDF method*), 249
- `numeric_integrate()` (*zfit.models.physics.CrystalBall method*), 257
- `numeric_integrate()` (*zfit.models.physics.DoubleCB method*), 264
- `numeric_integrate()` (*zfit.models.polynomials.Chebyshev method*), 271
- `numeric_integrate()` (*zfit.models.polynomials.Chebyshev2 method*), 279
- `numeric_integrate()` (*zfit.models.polynomials.Hermite method*), 286
- `numeric_integrate()` (*zfit.models.polynomials.Laguerre method*), 293
- `numeric_integrate()` (*zfit.models.polynomials.Legendre method*), 300
- `numeric_integrate()` (*zfit.models.polynomials.RecursivePolynomial method*), 183



[method](#)), 307  
[numeric\\_integrate\(\)](#) ([zfit.models.special.SimpleFunctorPDF method](#)), 315  
[numeric\\_integrate\(\)](#) ([zfit.models.special.SimplePDF method](#)), 321  
[numeric\\_integrate\(\)](#) ([zfit.models.special.ZPDF method](#)), 328  
[numeric\\_integrate\(\)](#) ([zfit.pdf.BaseFunctor method](#)), 407  
[numeric\\_integrate\(\)](#) ([zfit.pdf.BasePDF method](#)), 400  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Chebyshev method](#)), 463  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Chebyshev2 method](#)), 477  
[numeric\\_integrate\(\)](#) ([zfit.pdf.CrystalBall method](#)), 421  
[numeric\\_integrate\(\)](#) ([zfit.pdf.DoubleCB method](#)), 428  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Exponential method](#)), 414  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Gauss method](#)), 435  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Hermite method](#)), 484  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Laguerre method](#)), 491  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Legendre method](#)), 470  
[numeric\\_integrate\(\)](#) ([zfit.pdf.ProductPDF method](#)), 505  
[numeric\\_integrate\(\)](#) ([zfit.pdf.RecursivePolynomial method](#)), 498  
[numeric\\_integrate\(\)](#) ([zfit.pdf.SimpleFunctorPDF method](#)), 532  
[numeric\\_integrate\(\)](#) ([zfit.pdf.SimplePDF method](#)), 525  
[numeric\\_integrate\(\)](#) ([zfit.pdf.SumPDF method](#)), 512  
[numeric\\_integrate\(\)](#) ([zfit.pdf.TruncatedGauss method](#)), 449  
[numeric\\_integrate\(\)](#) ([zfit.pdf.Uniform method](#)), 442  
[numeric\\_integrate\(\)](#) ([zfit.pdf.WrapDistribution method](#)), 456  
[numeric\\_integrate\(\)](#) ([zfit.pdf.ZPDF method](#)), 519  
[numerical\\_gradient\(\)](#) (in module [zfit.z.math](#)), 344  
[numerical\\_hessian\(\)](#) (in module [zfit.z.math](#)), 344  
[numerical\\_value\\_gradients\(\)](#) (in module [zfit.z.math](#)), 344  
[numerical\\_value\\_gradients\\_hessian\(\)](#) (in module [zfit.z.math](#)), 344  
[numpy\(\)](#) ([zfit.ComplexParameter method](#)), 43

[numpy\(\)](#) ([zfit.ComposedParameter method](#)), 42  
[numpy\(\)](#) ([zfit.core.data.Data method](#)), 75  
[numpy\(\)](#) ([zfit.core.data.SampleData method](#)), 78  
[numpy\(\)](#) ([zfit.core.data.Sampler method](#)), 82  
[numpy\(\)](#) ([zfit.core.parameter.BaseComposedParameter method](#)), 110  
[numpy\(\)](#) ([zfit.core.parameter.BaseZParameter method](#)), 112  
[numpy\(\)](#) ([zfit.core.parameter.ComplexParameter method](#)), 113  
[numpy\(\)](#) ([zfit.core.parameter.ComposedParameter method](#)), 114  
[numpy\(\)](#) ([zfit.core.parameter.ComposedVariable method](#)), 115  
[numpy\(\)](#) ([zfit.core.parameter.ConstantParameter method](#)), 116  
[numpy\(\)](#) ([zfit.core.parameter.Parameter method](#)), 122  
[numpy\(\)](#) ([zfit.core.parameter.TFBaseVariable method](#)), 132  
[numpy\(\)](#) ([zfit.data.Data method](#)), 353  
[numpy\(\)](#) ([zfit.param.ComplexParameter method](#)), 396  
[numpy\(\)](#) ([zfit.param.ComposedParameter method](#)), 394  
[numpy\(\)](#) ([zfit.param.ConstantParameter method](#)), 383  
[numpy\(\)](#) ([zfit.param.Parameter method](#)), 389  
[numpy\(\)](#) ([zfit.Parameter method](#)), 36

## O

[obs](#) ([zfit.core.basefunc.BaseFunc attribute](#)), 51  
[obs](#) ([zfit.core.basemodel.BaseModel attribute](#)), 56  
[obs](#) ([zfit.core.basepdf.BasePDF attribute](#)), 65  
[obs](#) ([zfit.core.data.Data attribute](#)), 75  
[obs](#) ([zfit.core.data.SampleData attribute](#)), 79  
[obs](#) ([zfit.core.data.Sampler attribute](#)), 82  
[obs](#) ([zfit.core.dimension.BaseDimensional attribute](#)), 83  
[obs](#) ([zfit.core.integration.PartialIntegralSampleData attribute](#)), 87  
[obs](#) ([zfit.core.interfaces.ZfitData attribute](#)), 88  
[obs](#) ([zfit.core.interfaces.ZfitDimensional attribute](#)), 89  
[obs](#) ([zfit.core.interfaces.ZfitFunc attribute](#)), 89  
[obs](#) ([zfit.core.interfaces.ZfitModel attribute](#)), 92  
[obs](#) ([zfit.core.interfaces.ZfitPDF attribute](#)), 94  
[obs](#) ([zfit.core.interfaces.ZfitSpace attribute](#)), 96  
[obs](#) ([zfit.core.limits.Space attribute](#)), 101  
[obs](#) ([zfit.core.sample.EventSpace attribute](#)), 141  
[obs](#) ([zfit.data.Data attribute](#)), 353  
[obs](#) ([zfit.func.BaseFunc attribute](#)), 357  
[obs](#) ([zfit.func.ProdFunc attribute](#)), 362  
[obs](#) ([zfit.func.SimpleFunc attribute](#)), 372  
[obs](#) ([zfit.func.SumFunc attribute](#)), 367  
[obs](#) ([zfit.models.basefunctor.FunctorMixin attribute](#)), 156  
[obs](#) ([zfit.models.basic.CustomGaussOLD attribute](#)), 162  
[obs](#) ([zfit.models.basic.Exponential attribute](#)), 169  
[obs](#) ([zfit.models.dist\\_tfp.ExponentialTFP attribute](#)), 176

- obs (*zfit.models.dist\_tfp.Gauss* attribute), 183
- obs (*zfit.models.dist\_tfp.TruncatedGauss* attribute), 190
- obs (*zfit.models.dist\_tfp.Uniform* attribute), 197
- obs (*zfit.models.dist\_tfp.WrapDistribution* attribute), 203
- obs (*zfit.models.functions.BaseFuncorFunc* attribute), 209
- obs (*zfit.models.functions.ProdFunc* attribute), 214
- obs (*zfit.models.functions.SimpleFunc* attribute), 219
- obs (*zfit.models.functions.SumFunc* attribute), 224
- obs (*zfit.models.functions.ZFunc* attribute), 229
- obs (*zfit.models.functor.BaseFuncor* attribute), 236
- obs (*zfit.models.functor.ProductPDF* attribute), 242
- obs (*zfit.models.functor.SumPDF* attribute), 249
- obs (*zfit.models.physics.CrystalBall* attribute), 257
- obs (*zfit.models.physics.DoubleCB* attribute), 264
- obs (*zfit.models.polynomials.Chebyshev* attribute), 272
- obs (*zfit.models.polynomials.Chebyshev2* attribute), 279
- obs (*zfit.models.polynomials.Hermite* attribute), 286
- obs (*zfit.models.polynomials.Laguerre* attribute), 293
- obs (*zfit.models.polynomials.Legendre* attribute), 301
- obs (*zfit.models.polynomials.RecursivePolynomial* attribute), 307
- obs (*zfit.models.special.SimpleFuncorPDF* attribute), 315
- obs (*zfit.models.special.SimplePDF* attribute), 322
- obs (*zfit.models.special.ZPDF* attribute), 328
- obs (*zfit.pdf.BaseFuncor* attribute), 407
- obs (*zfit.pdf.BasePDF* attribute), 400
- obs (*zfit.pdf.Chebyshev* attribute), 463
- obs (*zfit.pdf.Chebyshev2* attribute), 478
- obs (*zfit.pdf.CrystalBall* attribute), 421
- obs (*zfit.pdf.DoubleCB* attribute), 429
- obs (*zfit.pdf.Exponential* attribute), 414
- obs (*zfit.pdf.Gauss* attribute), 435
- obs (*zfit.pdf.Hermite* attribute), 485
- obs (*zfit.pdf.Laguerre* attribute), 492
- obs (*zfit.pdf.Legendre* attribute), 470
- obs (*zfit.pdf.ProductPDF* attribute), 505
- obs (*zfit.pdf.RecursivePolynomial* attribute), 499
- obs (*zfit.pdf.SimpleFuncorPDF* attribute), 532
- obs (*zfit.pdf.SimplePDF* attribute), 525
- obs (*zfit.pdf.SumPDF* attribute), 512
- obs (*zfit.pdf.TruncatedGauss* attribute), 449
- obs (*zfit.pdf.Uniform* attribute), 442
- obs (*zfit.pdf.WrapDistribution* attribute), 456
- obs (*zfit.pdf.ZPDF* attribute), 519
- obs (*zfit.Space* attribute), 47
- obs\_axes (*zfit.core.limits.Space* attribute), 101
- obs\_axes (*zfit.core.sample.EventSpace* attribute), 141
- obs\_axes (*zfit.Space* attribute), 47
- ObsIncompatibleError, 338
- ObsNotSpecifiedError, 339
- old\_graph\_caching\_methods (*zfit.ComplexParameter* attribute), 43
- old\_graph\_caching\_methods (*zfit.ComposedParameter* attribute), 42
- old\_graph\_caching\_methods (*zfit.constraint.GaussianConstraint* attribute), 350
- old\_graph\_caching\_methods (*zfit.constraint.SimpleConstraint* attribute), 349
- old\_graph\_caching\_methods (*zfit.core.basefunc.BaseFunc* attribute), 52
- old\_graph\_caching\_methods (*zfit.core.basemodel.BaseModel* attribute), 56
- old\_graph\_caching\_methods (*zfit.core.baseobject.BaseNumeric* attribute), 60
- old\_graph\_caching\_methods (*zfit.core.basepdf.BasePDF* attribute), 65
- old\_graph\_caching\_methods (*zfit.core.constraint.BaseConstraint* attribute), 68
- old\_graph\_caching\_methods (*zfit.core.constraint.DistributionConstraint* attribute), 69
- old\_graph\_caching\_methods (*zfit.core.constraint.GaussianConstraint* attribute), 71
- old\_graph\_caching\_methods (*zfit.core.constraint.SimpleConstraint* attribute), 72
- old\_graph\_caching\_methods (*zfit.core.data.Data* attribute), 75
- old\_graph\_caching\_methods (*zfit.core.data.SampleData* attribute), 79
- old\_graph\_caching\_methods (*zfit.core.data.Sampler* attribute), 82
- old\_graph\_caching\_methods (*zfit.core.loss.BaseLoss* attribute), 104
- old\_graph\_caching\_methods (*zfit.core.loss.CachedLoss* attribute), 105
- old\_graph\_caching\_methods (*zfit.core.loss.ExtendedUnbinnedNLL* attribute), 106
- old\_graph\_caching\_methods (*zfit.core.loss.SimpleLoss* attribute), 107
- old\_graph\_caching\_methods (*zfit.core.loss.UnbinnedNLL* attribute), 108
- old\_graph\_caching\_methods (*zfit.core.parameter.BaseComposedParameter* attribute), 110
- old\_graph\_caching\_methods (*zfit.core.parameter.BaseZParameter* attribute),

- 112
- `old_graph_caching_methods`  
(`zfit.core.parameter.ComplexParameter` attribute), 113
- `old_graph_caching_methods`  
(`zfit.core.parameter.ComposedParameter` attribute), 114
- `old_graph_caching_methods`  
(`zfit.core.parameter.ConstantParameter` attribute), 116
- `old_graph_caching_methods`  
(`zfit.core.parameter.Parameter` attribute), 122
- `old_graph_caching_methods`  
(`zfit.core.parameter.ZfitParameterMixin` attribute), 137
- `old_graph_caching_methods` (`zfit.data.Data` attribute), 353
- `old_graph_caching_methods` (`zfit.func.BaseFunc` attribute), 357
- `old_graph_caching_methods` (`zfit.func.ProdFunc` attribute), 362
- `old_graph_caching_methods`  
(`zfit.func.SimpleFunc` attribute), 372
- `old_graph_caching_methods` (`zfit.func.SumFunc` attribute), 367
- `old_graph_caching_methods` (`zfit.loss.BaseLoss` attribute), 377
- `old_graph_caching_methods`  
(`zfit.loss.ExtendedUnbinnedNLL` attribute), 375
- `old_graph_caching_methods`  
(`zfit.loss.SimpleLoss` attribute), 378
- `old_graph_caching_methods`  
(`zfit.loss.UnbinnedNLL` attribute), 376
- `old_graph_caching_methods`  
(`zfit.minimize.Minuit` attribute), 381
- `old_graph_caching_methods`  
(`zfit.minimizers.minimizer_minuit.Minuit` attribute), 148
- `old_graph_caching_methods`  
(`zfit.models.basefunctor.FunctorMixin` attribute), 156
- `old_graph_caching_methods`  
(`zfit.models.basic.CustomGaussOLD` attribute), 162
- `old_graph_caching_methods`  
(`zfit.models.basic.Exponential` attribute), 169
- `old_graph_caching_methods`  
(`zfit.models.dist_tfp.ExponentialTFP` attribute), 176
- `old_graph_caching_methods`  
(`zfit.models.dist_tfp.Gauss` attribute), 183
- `old_graph_caching_methods`  
(`zfit.models.dist_tfp.TruncatedGauss` attribute), 190
- `old_graph_caching_methods`  
(`zfit.models.dist_tfp.Uniform` attribute), 197
- `old_graph_caching_methods`  
(`zfit.models.dist_tfp.WrapDistribution` attribute), 204
- `old_graph_caching_methods`  
(`zfit.models.functions.BaseFunctorFunc` attribute), 209
- `old_graph_caching_methods`  
(`zfit.models.functions.ProdFunc` attribute), 214
- `old_graph_caching_methods`  
(`zfit.models.functions.SimpleFunc` attribute), 219
- `old_graph_caching_methods`  
(`zfit.models.functions.SumFunc` attribute), 224
- `old_graph_caching_methods`  
(`zfit.models.functions.ZFunc` attribute), 229
- `old_graph_caching_methods`  
(`zfit.models.functor.BaseFunctor` attribute), 236
- `old_graph_caching_methods`  
(`zfit.models.functor.ProductPDF` attribute), 242
- `old_graph_caching_methods`  
(`zfit.models.functor.SumPDF` attribute), 249
- `old_graph_caching_methods`  
(`zfit.models.physics.CrystalBall` attribute), 257
- `old_graph_caching_methods`  
(`zfit.models.physics.DoubleCB` attribute), 264
- `old_graph_caching_methods`  
(`zfit.models.polynomials.Chebyshev` attribute), 272
- `old_graph_caching_methods`  
(`zfit.models.polynomials.Chebyshev2` attribute), 279
- `old_graph_caching_methods`  
(`zfit.models.polynomials.Hermite` attribute), 286
- `old_graph_caching_methods`  
(`zfit.models.polynomials.Laguerre` attribute), 293
- `old_graph_caching_methods`  
(`zfit.models.polynomials.Legendre` attribute), 301
- `old_graph_caching_methods`  
(`zfit.models.polynomials.RecursivePolynomial` attribute), 307

old\_graph\_caching\_methods  
(*zfit.models.special.SimpleFunctorPDF* attribute), 315

old\_graph\_caching\_methods  
(*zfit.models.special.SimplePDF* attribute), 322

old\_graph\_caching\_methods  
(*zfit.models.special.ZPDF* attribute), 328

old\_graph\_caching\_methods  
(*zfit.param.ComplexParameter* attribute), 396

old\_graph\_caching\_methods  
(*zfit.param.ComposedParameter* attribute), 394

old\_graph\_caching\_methods  
(*zfit.param.ConstantParameter* attribute), 383

old\_graph\_caching\_methods  
(*zfit.param.Parameter* attribute), 389

old\_graph\_caching\_methods (*zfit.Parameter* attribute), 36

old\_graph\_caching\_methods  
(*zfit.pdf.BaseFunctor* attribute), 407

old\_graph\_caching\_methods (*zfit.pdf.BasePDF* attribute), 400

old\_graph\_caching\_methods (*zfit.pdf.Chebyshev* attribute), 463

old\_graph\_caching\_methods  
(*zfit.pdf.Chebyshev2* attribute), 478

old\_graph\_caching\_methods  
(*zfit.pdf.CrystalBall* attribute), 421

old\_graph\_caching\_methods (*zfit.pdf.DoubleCB* attribute), 429

old\_graph\_caching\_methods  
(*zfit.pdf.Exponential* attribute), 414

old\_graph\_caching\_methods (*zfit.pdf.Gauss* attribute), 436

old\_graph\_caching\_methods (*zfit.pdf.Hermite* attribute), 485

old\_graph\_caching\_methods (*zfit.pdf.Laguerre* attribute), 492

old\_graph\_caching\_methods (*zfit.pdf.Legendre* attribute), 470

old\_graph\_caching\_methods  
(*zfit.pdf.ProductPDF* attribute), 505

old\_graph\_caching\_methods  
(*zfit.pdf.RecursivePolynomial* attribute), 499

old\_graph\_caching\_methods  
(*zfit.pdf.SimpleFunctorPDF* attribute), 532

old\_graph\_caching\_methods  
(*zfit.pdf.SimplePDF* attribute), 526

old\_graph\_caching\_methods (*zfit.pdf.SumPDF* attribute), 512

old\_graph\_caching\_methods  
(*zfit.pdf.TruncatedGauss* attribute), 449

old\_graph\_caching\_methods (*zfit.pdf.Uniform* attribute), 442

old\_graph\_caching\_methods  
(*zfit.pdf.WrapDistribution* attribute), 456

old\_graph\_caching\_methods (*zfit.pdf.ZPDF* attribute), 519

old\_graph\_caching\_methods  
(*zfit.util.cache.Cachable* attribute), 332

old\_graph\_caching\_methods  
(*zfit.util.cache.FunctionCacheHolder* attribute), 333

op (*zfit.core.parameter.Parameter* attribute), 122

op (*zfit.core.parameter.TFBaseVariable* attribute), 132

op (*zfit.param.Parameter* attribute), 389

op (*zfit.Parameter* attribute), 37

OverdefinedError, 339

## P

Parameter (class in *zfit*), 31

Parameter (class in *zfit.core.parameter*), 116

Parameter (class in *zfit.param*), 383

Parameter.SaveSliceInfo (class in *zfit*), 31

Parameter.SaveSliceInfo (class in *zfit.core.parameter*), 116

Parameter.SaveSliceInfo (class in *zfit.param*), 384

params (*zfit.ComplexParameter* attribute), 43

params (*zfit.ComposedParameter* attribute), 42

params (*zfit.constraint.GaussianConstraint* attribute), 350

params (*zfit.constraint.SimpleConstraint* attribute), 349

params (*zfit.core.basefunc.BaseFunc* attribute), 52

params (*zfit.core.basemodel.BaseModel* attribute), 56

params (*zfit.core.baseobject.BaseNumeric* attribute), 60

params (*zfit.core.basepdf.BasePDF* attribute), 65

params (*zfit.core.constraint.BaseConstraint* attribute), 68

params (*zfit.core.constraint.DistributionConstraint* attribute), 70

params (*zfit.core.constraint.GaussianConstraint* attribute), 71

params (*zfit.core.constraint.SimpleConstraint* attribute), 72

params (*zfit.core.interfaces.ZfitFunc* attribute), 89

params (*zfit.core.interfaces.ZfitModel* attribute), 92

params (*zfit.core.interfaces.ZfitNumeric* attribute), 93

params (*zfit.core.interfaces.ZfitParameter* attribute), 95

params (*zfit.core.interfaces.ZfitPDF* attribute), 94

params (*zfit.core.parameter.BaseComposedParameter* attribute), 110

params (*zfit.core.parameter.BaseParameter* attribute), 111



- params (*zfit.core.parameter.BaseZParameter* attribute), 112
- params (*zfit.core.parameter.ComplexParameter* attribute), 113
- params (*zfit.core.parameter.ComposedParameter* attribute), 114
- params (*zfit.core.parameter.ConstantParameter* attribute), 116
- params (*zfit.core.parameter.Parameter* attribute), 122
- params (*zfit.core.parameter.ZfitParameterMixin* attribute), 137
- params (*zfit.func.BaseFunc* attribute), 357
- params (*zfit.func.ProdFunc* attribute), 362
- params (*zfit.func.SimpleFunc* attribute), 372
- params (*zfit.func.SumFunc* attribute), 367
- params (*zfit.minimizers.fitresult.FitResult* attribute), 146
- params (*zfit.minimizers.interface.ZfitResult* attribute), 147
- params (*zfit.models.basefunc.FunctorMixin* attribute), 156
- params (*zfit.models.basic.CustomGaussOLD* attribute), 162
- params (*zfit.models.basic.Exponential* attribute), 169
- params (*zfit.models.dist\_tfp.ExponentialTFP* attribute), 176
- params (*zfit.models.dist\_tfp.Gauss* attribute), 183
- params (*zfit.models.dist\_tfp.TruncatedGauss* attribute), 190
- params (*zfit.models.dist\_tfp.Uniform* attribute), 197
- params (*zfit.models.dist\_tfp.WrapDistribution* attribute), 204
- params (*zfit.models.functions.BaseFunctorFunc* attribute), 209
- params (*zfit.models.functions.ProdFunc* attribute), 214
- params (*zfit.models.functions.SimpleFunc* attribute), 219
- params (*zfit.models.functions.SumFunc* attribute), 224
- params (*zfit.models.functions.ZFunc* attribute), 229
- params (*zfit.models.functor.BaseFunctor* attribute), 236
- params (*zfit.models.functor.ProductPDF* attribute), 242
- params (*zfit.models.functor.SumPDF* attribute), 249
- params (*zfit.models.physics.CrystalBall* attribute), 257
- params (*zfit.models.physics.DoubleCB* attribute), 264
- params (*zfit.models.polynomials.Chebyshev* attribute), 272
- params (*zfit.models.polynomials.Chebyshev2* attribute), 279
- params (*zfit.models.polynomials.Hermite* attribute), 286
- params (*zfit.models.polynomials.Laguerre* attribute), 293
- params (*zfit.models.polynomials.Legendre* attribute), 301
- params (*zfit.models.polynomials.RecursivePolynomial* attribute), 307
- params (*zfit.models.special.SimpleFunctorPDF* attribute), 315
- params (*zfit.models.special.SimplePDF* attribute), 322
- params (*zfit.models.special.ZPDF* attribute), 328
- params (*zfit.param.ComplexParameter* attribute), 396
- params (*zfit.param.ComposedParameter* attribute), 394
- params (*zfit.param.ConstantParameter* attribute), 383
- params (*zfit.param.Parameter* attribute), 389
- params (*zfit.Parameter* attribute), 37
- params (*zfit.pdf.BaseFunctor* attribute), 407
- params (*zfit.pdf.BasePDF* attribute), 400
- params (*zfit.pdf.Chebyshev* attribute), 463
- params (*zfit.pdf.Chebyshev2* attribute), 478
- params (*zfit.pdf.CrystalBall* attribute), 421
- params (*zfit.pdf.DoubleCB* attribute), 429
- params (*zfit.pdf.Exponential* attribute), 414
- params (*zfit.pdf.Gauss* attribute), 436
- params (*zfit.pdf.Hermite* attribute), 485
- params (*zfit.pdf.Laguerre* attribute), 492
- params (*zfit.pdf.Legendre* attribute), 470
- params (*zfit.pdf.ProductPDF* attribute), 505
- params (*zfit.pdf.RecursivePolynomial* attribute), 499
- params (*zfit.pdf.SimpleFunctorPDF* attribute), 532
- params (*zfit.pdf.SimplePDF* attribute), 526
- params (*zfit.pdf.SumPDF* attribute), 512
- params (*zfit.pdf.TruncatedGauss* attribute), 449
- params (*zfit.pdf.Uniform* attribute), 442
- params (*zfit.pdf.WrapDistribution* attribute), 456
- params (*zfit.pdf.ZPDF* attribute), 519
- partial\_analytic\_integrate()
  - (*zfit.core.basefunc.BaseFunc* method), 52
- partial\_analytic\_integrate()
  - (*zfit.core.basemodel.BaseModel* method), 56
- partial\_analytic\_integrate()
  - (*zfit.core.basepdf.BasePDF* method), 65
- partial\_analytic\_integrate()
  - (*zfit.func.BaseFunc* method), 357
- partial\_analytic\_integrate()
  - (*zfit.func.ProdFunc* method), 362
- partial\_analytic\_integrate()
  - (*zfit.func.SimpleFunc* method), 372
- partial\_analytic\_integrate()
  - (*zfit.func.SumFunc* method), 367
- partial\_analytic\_integrate()
  - (*zfit.models.basefunc.FunctorMixin* method), 156
- partial\_analytic\_integrate()
  - (*zfit.models.basic.CustomGaussOLD* method), 162
- partial\_analytic\_integrate()
  - (*zfit.models.basic.Exponential* method), 169
- partial\_analytic\_integrate()
  - (*zfit.models.dist\_tfp.ExponentialTFP* method),

176  
`partial_analytic_integrate()`  
 (`zfit.models.dist_tfp.Gauss` method), 183  
`partial_analytic_integrate()`  
 (`zfit.models.dist_tfp.TruncatedGauss` method), 190  
`partial_analytic_integrate()`  
 (`zfit.models.dist_tfp.Uniform` method), 197  
`partial_analytic_integrate()`  
 (`zfit.models.dist_tfp.WrapDistribution` method), 204  
`partial_analytic_integrate()`  
 (`zfit.models.functions.BaseFunc` method), 209  
`partial_analytic_integrate()`  
 (`zfit.models.functions.ProdFunc` method), 214  
`partial_analytic_integrate()`  
 (`zfit.models.functions.SimpleFunc` method), 219  
`partial_analytic_integrate()`  
 (`zfit.models.functions.SumFunc` method), 224  
`partial_analytic_integrate()`  
 (`zfit.models.functions.ZFunc` method), 229  
`partial_analytic_integrate()`  
 (`zfit.models.functor.BaseFunc` method), 236  
`partial_analytic_integrate()`  
 (`zfit.models.functor.ProductPDF` method), 243  
`partial_analytic_integrate()`  
 (`zfit.models.functor.SumPDF` method), 250  
`partial_analytic_integrate()`  
 (`zfit.models.physics.CrystalBall` method), 257  
`partial_analytic_integrate()`  
 (`zfit.models.physics.DoubleCB` method), 264  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.Chebyshev` method), 272  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.Chebyshev2` method), 279  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.Hermite` method), 286  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.Laguerre` method), 293  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.Legendre` method), 301  
`partial_analytic_integrate()`  
 (`zfit.models.polynomials.RecursivePolynomial` method), 307  
`partial_analytic_integrate()`  
 (`zfit.models.special.SimpleFuncPDF` method), 315  
`partial_analytic_integrate()`  
 (`zfit.models.special.SimplePDF` method), 322  
`partial_analytic_integrate()`  
 (`zfit.models.special.ZPDF` method), 328  
`partial_analytic_integrate()`  
 (`zfit.pdf.BaseFunc` method), 407  
`partial_analytic_integrate()`  
 (`zfit.pdf.BasePDF` method), 400  
`partial_analytic_integrate()`  
 (`zfit.pdf.Chebyshev` method), 463  
`partial_analytic_integrate()`  
 (`zfit.pdf.Chebyshev2` method), 478  
`partial_analytic_integrate()`  
 (`zfit.pdf.CrystalBall` method), 421  
`partial_analytic_integrate()`  
 (`zfit.pdf.DoubleCB` method), 429  
`partial_analytic_integrate()`  
 (`zfit.pdf.Exponential` method), 414  
`partial_analytic_integrate()` (`zfit.pdf.Gauss` method), 436  
`partial_analytic_integrate()`  
 (`zfit.pdf.Hermite` method), 485  
`partial_analytic_integrate()`  
 (`zfit.pdf.Laguerre` method), 492  
`partial_analytic_integrate()`  
 (`zfit.pdf.Legendre` method), 470  
`partial_analytic_integrate()`  
 (`zfit.pdf.ProductPDF` method), 505  
`partial_analytic_integrate()`  
 (`zfit.pdf.RecursivePolynomial` method), 499  
`partial_analytic_integrate()`  
 (`zfit.pdf.SimpleFuncPDF` method), 532  
`partial_analytic_integrate()`  
 (`zfit.pdf.SimplePDF` method), 526  
`partial_analytic_integrate()`  
 (`zfit.pdf.SumPDF` method), 512  
`partial_analytic_integrate()`  
 (`zfit.pdf.TruncatedGauss` method), 449  
`partial_analytic_integrate()`  
 (`zfit.pdf.Uniform` method), 442  
`partial_analytic_integrate()`  
 (`zfit.pdf.WrapDistribution` method), 456  
`partial_analytic_integrate()` (`zfit.pdf.ZPDF` method), 519  
`partial_integrate()`  
 (`zfit.core.basefunc.BaseFunc` method), 52  
`partial_integrate()`

[\(zfit.core.basemodel.BaseModel method\), 56](#)  
[partial\\_integrate\(\) \(zfit.core.basepdf.BasePDF method\), 65](#)  
[partial\\_integrate\(\) \(zfit.core.interfaces.ZfitFunc method\), 89](#)  
[partial\\_integrate\(\) \(zfit.core.interfaces.ZfitModel method\), 92](#)  
[partial\\_integrate\(\) \(zfit.core.interfaces.ZfitPDF method\), 94](#)  
[partial\\_integrate\(\) \(zfit.func.BaseFunc method\), 357](#)  
[partial\\_integrate\(\) \(zfit.func.ProdFunc method\), 362](#)  
[partial\\_integrate\(\) \(zfit.func.SimpleFunc method\), 372](#)  
[partial\\_integrate\(\) \(zfit.func.SumFunc method\), 367](#)  
[partial\\_integrate\(\) \(zfit.models.basefunctor.FunctorMixin method\), 156](#)  
[partial\\_integrate\(\) \(zfit.models.basic.CustomGaussOLD method\), 163](#)  
[partial\\_integrate\(\) \(zfit.models.basic.Exponential method\), 169](#)  
[partial\\_integrate\(\) \(zfit.models.dist\\_tfp.ExponentialTFP method\), 176](#)  
[partial\\_integrate\(\) \(zfit.models.dist\\_tfp.Gauss method\), 183](#)  
[partial\\_integrate\(\) \(zfit.models.dist\\_tfp.TruncatedGauss method\), 190](#)  
[partial\\_integrate\(\) \(zfit.models.dist\\_tfp.Uniform method\), 197](#)  
[partial\\_integrate\(\) \(zfit.models.dist\\_tfp.WrapDistribution method\), 204](#)  
[partial\\_integrate\(\) \(zfit.models.functions.BaseFunc method\), 209](#)  
[partial\\_integrate\(\) \(zfit.models.functions.ProdFunc method\), 214](#)  
[partial\\_integrate\(\) \(zfit.models.functions.SimpleFunc method\), 219](#)  
[partial\\_integrate\(\) \(zfit.models.functions.SumFunc method\), 224](#)  
[partial\\_integrate\(\) \(zfit.models.functions.ZFunc method\), 229](#)  
[partial\\_integrate\(\) \(zfit.models.functor.BaseFunc method\), 236](#)  
[partial\\_integrate\(\) \(zfit.models.functor.ProductPDF method\), 243](#)  
[partial\\_integrate\(\) \(zfit.models.functor.SumPDF method\), 250](#)  
[partial\\_integrate\(\) \(zfit.models.physics.CrystalBall method\), 257](#)  
[partial\\_integrate\(\) \(zfit.models.physics.DoubleCB method\), 265](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.Chebyshev method\), 272](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.Chebyshev2 method\), 279](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.Hermite method\), 286](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.Laguerre method\), 293](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.Legendre method\), 301](#)  
[partial\\_integrate\(\) \(zfit.models.polynomials.RecursivePolynomial method\), 308](#)  
[partial\\_integrate\(\) \(zfit.models.special.SimpleFuncPDF method\), 315](#)  
[partial\\_integrate\(\) \(zfit.models.special.SimplePDF method\), 322](#)  
[partial\\_integrate\(\) \(zfit.models.special.ZPDF method\), 329](#)  
[partial\\_integrate\(\) \(zfit.pdf.BaseFunc method\), 407](#)  
[partial\\_integrate\(\) \(zfit.pdf.BasePDF method\), 400](#)  
[partial\\_integrate\(\) \(zfit.pdf.Chebyshev method\), 463](#)  
[partial\\_integrate\(\) \(zfit.pdf.Chebyshev2 method\), 478](#)  
[partial\\_integrate\(\) \(zfit.pdf.CrystalBall method\), 421](#)  
[partial\\_integrate\(\) \(zfit.pdf.DoubleCB method\), 429](#)  
[partial\\_integrate\(\) \(zfit.pdf.Exponential method\), 414](#)  
[partial\\_integrate\(\) \(zfit.pdf.Gauss method\), 436](#)

`partial_integrate()` (*zfit.pdf.Hermite method*), 485  
`partial_integrate()` (*zfit.pdf.Laguerre method*), 492  
`partial_integrate()` (*zfit.pdf.Legendre method*), 470  
`partial_integrate()` (*zfit.pdf.ProductPDF method*), 505  
`partial_integrate()` (*zfit.pdf.RecursivePolynomial method*), 499  
`partial_integrate()` (*zfit.pdf.SimpleFuncPDF method*), 532  
`partial_integrate()` (*zfit.pdf.SimplePDF method*), 526  
`partial_integrate()` (*zfit.pdf.SumPDF method*), 512  
`partial_integrate()` (*zfit.pdf.TruncatedGauss method*), 449  
`partial_integrate()` (*zfit.pdf.Uniform method*), 442  
`partial_integrate()` (*zfit.pdf.WrapDistribution method*), 456  
`partial_integrate()` (*zfit.pdf.ZPDF method*), 519  
`partial_numeric_integrate()` (*zfit.core.basefunc.BaseFunc method*), 52  
`partial_numeric_integrate()` (*zfit.core.basemodel.BaseModel method*), 57  
`partial_numeric_integrate()` (*zfit.core.basepdf.BasePDF method*), 65  
`partial_numeric_integrate()` (*zfit.func.BaseFunc method*), 357  
`partial_numeric_integrate()` (*zfit.func.ProdFunc method*), 362  
`partial_numeric_integrate()` (*zfit.func.SimpleFunc method*), 372  
`partial_numeric_integrate()` (*zfit.func.SumFunc method*), 367  
`partial_numeric_integrate()` (*zfit.models.basefunc.FunctorMixin method*), 156  
`partial_numeric_integrate()` (*zfit.models.basic.CustomGaussOLD method*), 163  
`partial_numeric_integrate()` (*zfit.models.basic.Exponential method*), 169  
`partial_numeric_integrate()` (*zfit.models.dist\_tfp.ExponentialTFP method*), 176  
`partial_numeric_integrate()` (*zfit.models.dist\_tfp.Gauss method*), 183  
`partial_numeric_integrate()` (*zfit.models.dist\_tfp.TruncatedGauss method*), 190  
`partial_numeric_integrate()` (*zfit.models.dist\_tfp.Uniform method*), 197  
`partial_numeric_integrate()` (*zfit.models.dist\_tfp.WrapDistribution method*), 204  
`partial_numeric_integrate()` (*zfit.models.functions.BaseFunc method*), 209  
`partial_numeric_integrate()` (*zfit.models.functions.ProdFunc method*), 214  
`partial_numeric_integrate()` (*zfit.models.functions.SimpleFunc method*), 219  
`partial_numeric_integrate()` (*zfit.models.functions.SumFunc method*), 224  
`partial_numeric_integrate()` (*zfit.models.functions.ZFunc method*), 229  
`partial_numeric_integrate()` (*zfit.models.functor.BaseFunc method*), 236  
`partial_numeric_integrate()` (*zfit.models.functor.ProductPDF method*), 243  
`partial_numeric_integrate()` (*zfit.models.functor.SumPDF method*), 250  
`partial_numeric_integrate()` (*zfit.models.physics.CrystalBall method*), 257  
`partial_numeric_integrate()` (*zfit.models.physics.DoubleCB method*), 265  
`partial_numeric_integrate()` (*zfit.models.polynomials.Chebyshev method*), 272  
`partial_numeric_integrate()` (*zfit.models.polynomials.Chebyshev2 method*), 279  
`partial_numeric_integrate()` (*zfit.models.polynomials.Hermite method*), 286  
`partial_numeric_integrate()` (*zfit.models.polynomials.Laguerre method*), 293  
`partial_numeric_integrate()` (*zfit.models.polynomials.Legendre method*), 301  
`partial_numeric_integrate()` (*zfit.models.polynomials.RecursivePolynomial method*), 308  
`partial_numeric_integrate()` (*zfit.models.special.SimpleFuncPDF method*), 315



`partial_numeric_integrate()`  
     (`zfit.models.special.SimplePDF` *method*), 322  
`partial_numeric_integrate()`  
     (`zfit.models.special.ZPDF` *method*), 329  
`partial_numeric_integrate()`  
     (`zfit.pdf.BaseFunctor` *method*), 407  
`partial_numeric_integrate()`  
     (`zfit.pdf.BasePDF` *method*), 400  
`partial_numeric_integrate()`  
     (`zfit.pdf.Chebyshev` *method*), 463  
`partial_numeric_integrate()`  
     (`zfit.pdf.Chebyshev2` *method*), 478  
`partial_numeric_integrate()`  
     (`zfit.pdf.CrystalBall` *method*), 421  
`partial_numeric_integrate()`  
     (`zfit.pdf.DoubleCB` *method*), 429  
`partial_numeric_integrate()`  
     (`zfit.pdf.Exponential` *method*), 414  
`partial_numeric_integrate()` (`zfit.pdf.Gauss` *method*), 436  
`partial_numeric_integrate()` (`zfit.pdf.Hermite` *method*), 485  
`partial_numeric_integrate()`  
     (`zfit.pdf.Laguerre` *method*), 492  
`partial_numeric_integrate()`  
     (`zfit.pdf.Legendre` *method*), 470  
`partial_numeric_integrate()`  
     (`zfit.pdf.ProductPDF` *method*), 505  
`partial_numeric_integrate()`  
     (`zfit.pdf.RecursivePolynomial` *method*), 499  
`partial_numeric_integrate()`  
     (`zfit.pdf.SimpleFunctorPDF` *method*), 532  
`partial_numeric_integrate()`  
     (`zfit.pdf.SimplePDF` *method*), 526  
`partial_numeric_integrate()`  
     (`zfit.pdf.SumPDF` *method*), 512  
`partial_numeric_integrate()`  
     (`zfit.pdf.TruncatedGauss` *method*), 449  
`partial_numeric_integrate()`  
     (`zfit.pdf.Uniform` *method*), 442  
`partial_numeric_integrate()`  
     (`zfit.pdf.WrapDistribution` *method*), 456  
`partial_numeric_integrate()` (`zfit.pdf.ZPDF` *method*), 519  
`PartialIntegralSampleData` (class in `zfit.core.integration`), 86  
`pdf()` (`zfit.core.basepdf.BasePDF` *method*), 65  
`pdf()` (`zfit.core.interfaces.ZfitPDF` *method*), 94  
`pdf()` (`zfit.models.basic.CustomGaussOLD` *method*), 163  
`pdf()` (`zfit.models.basic.Exponential` *method*), 170  
`pdf()` (`zfit.models.dist_tfp.ExponentialTFP` *method*), 176  
`pdf()` (`zfit.models.dist_tfp.Gauss` *method*), 183  
`pdf()` (`zfit.models.dist_tfp.TruncatedGauss` *method*), 190  
`pdf()` (`zfit.models.dist_tfp.Uniform` *method*), 197  
`pdf()` (`zfit.models.dist_tfp.WrapDistribution` *method*), 204  
`pdf()` (`zfit.models.functor.BaseFunctor` *method*), 236  
`pdf()` (`zfit.models.functor.ProductPDF` *method*), 243  
`pdf()` (`zfit.models.functor.SumPDF` *method*), 250  
`pdf()` (`zfit.models.physics.CrystalBall` *method*), 257  
`pdf()` (`zfit.models.physics.DoubleCB` *method*), 265  
`pdf()` (`zfit.models.polynomials.Chebyshev` *method*), 272  
`pdf()` (`zfit.models.polynomials.Chebyshev2` *method*), 279  
`pdf()` (`zfit.models.polynomials.Hermite` *method*), 287  
`pdf()` (`zfit.models.polynomials.Laguerre` *method*), 294  
`pdf()` (`zfit.models.polynomials.Legendre` *method*), 301  
`pdf()` (`zfit.models.polynomials.RecursivePolynomial` *method*), 308  
`pdf()` (`zfit.models.special.SimpleFunctorPDF` *method*), 316  
`pdf()` (`zfit.models.special.SimplePDF` *method*), 322  
`pdf()` (`zfit.models.special.ZPDF` *method*), 329  
`pdf()` (`zfit.pdf.BaseFunctor` *method*), 408  
`pdf()` (`zfit.pdf.BasePDF` *method*), 401  
`pdf()` (`zfit.pdf.Chebyshev` *method*), 464  
`pdf()` (`zfit.pdf.Chebyshev2` *method*), 478  
`pdf()` (`zfit.pdf.CrystalBall` *method*), 422  
`pdf()` (`zfit.pdf.DoubleCB` *method*), 429  
`pdf()` (`zfit.pdf.Exponential` *method*), 414  
`pdf()` (`zfit.pdf.Gauss` *method*), 436  
`pdf()` (`zfit.pdf.Hermite` *method*), 485  
`pdf()` (`zfit.pdf.Laguerre` *method*), 492  
`pdf()` (`zfit.pdf.Legendre` *method*), 471  
`pdf()` (`zfit.pdf.ProductPDF` *method*), 506  
`pdf()` (`zfit.pdf.RecursivePolynomial` *method*), 499  
`pdf()` (`zfit.pdf.SimpleFunctorPDF` *method*), 533  
`pdf()` (`zfit.pdf.SimplePDF` *method*), 526  
`pdf()` (`zfit.pdf.SumPDF` *method*), 513  
`pdf()` (`zfit.pdf.TruncatedGauss` *method*), 450  
`pdf()` (`zfit.pdf.Uniform` *method*), 443  
`pdf()` (`zfit.pdf.WrapDistribution` *method*), 457  
`pdf()` (`zfit.pdf.ZPDF` *method*), 519  
`PDFCompatibilityError`, 339  
`pdfs_extended` (`zfit.models.functor.BaseFunctor` *attribute*), 236  
`pdfs_extended` (`zfit.models.functor.ProductPDF` *attribute*), 243  
`pdfs_extended` (`zfit.models.functor.SumPDF` *attribute*), 250  
`pdfs_extended` (`zfit.models.special.SimpleFunctorPDF` *attribute*), 316  
`pdfs_extended` (`zfit.pdf.BaseFunctor` *attribute*), 408  
`pdfs_extended` (`zfit.pdf.ProductPDF` *attribute*), 506

pdfs\_extended (*zfit.pdf.SimpleFunctorPDF* attribute), 533

pdfs\_extended (*zfit.pdf.SumPDF* attribute), 513

poisson() (in module *zfit.sample*), 535

poly\_complex() (in module *zfit.z.math*), 345

pop() (*zfit.util.container.DotDict* method), 335

popitem() (*zfit.util.container.DotDict* method), 335

pow() (in module *zfit.z.wrapping\_tf*), 346

print\_gradients() (in module *zfit.minimizers.baseminimizer*), 144

print\_params() (in module *zfit.minimizers.baseminimizer*), 144

ProdFunc (class in *zfit.func*), 359

ProdFunc (class in *zfit.models.functions*), 211

ProductPDF (class in *zfit.models.functor*), 238

ProductPDF (class in *zfit.pdf*), 501

## R

raise\_error\_if\_norm\_range() (in module *zfit.models.special*), 331

random\_normal() (in module *zfit.z.wrapping\_tf*), 346

random\_poisson() (in module *zfit.z.wrapping\_tf*), 346

random\_uniform() (in module *zfit.z.wrapping\_tf*), 346

randomize() (*zfit.core.parameter.Parameter* method), 122

randomize() (*zfit.param.Parameter* method), 389

randomize() (*zfit.Parameter* method), 37

read\_value() (*zfit.ComplexParameter* method), 43

read\_value() (*zfit.ComposedParameter* method), 42

read\_value() (*zfit.core.parameter.BaseComposedParameter* method), 110

read\_value() (*zfit.core.parameter.BaseZParameter* method), 112

read\_value() (*zfit.core.parameter.ComplexParameter* method), 113

read\_value() (*zfit.core.parameter.ComposedParameter* method), 114

read\_value() (*zfit.core.parameter.ComposedVariable* method), 115

read\_value() (*zfit.core.parameter.ConstantParameter* method), 116

read\_value() (*zfit.core.parameter.Parameter* method), 122

read\_value() (*zfit.core.parameter.TFBaseVariable* method), 132

read\_value() (*zfit.param.ComplexParameter* method), 396

read\_value() (*zfit.param.ComposedParameter* method), 394

read\_value() (*zfit.param.ConstantParameter* method), 383

read\_value() (*zfit.param.Parameter* method), 389

read\_value() (*zfit.Parameter* method), 37

real (*zfit.ComplexParameter* attribute), 43

real (*zfit.core.parameter.ComplexParameter* attribute), 113

real (*zfit.param.ComplexParameter* attribute), 396

RecursivePolynomial (class in *zfit.models.polynomials*), 303

RecursivePolynomial (class in *zfit.pdf*), 494

register() (*zfit.core.integration.AnalyticIntegral* method), 86

register() (*zfit.core.parameter.MetaBaseParameter* method), 116

register\_additional\_repr() (*zfit.core.basefunc.BaseFunc* class method), 52

register\_additional\_repr() (*zfit.core.basemodel.BaseModel* class method), 57

register\_additional\_repr() (*zfit.core.basepdf.BasePDF* class method), 65

register\_additional\_repr() (*zfit.func.BaseFunc* class method), 357

register\_additional\_repr() (*zfit.func.ProdFunc* class method), 362

register\_additional\_repr() (*zfit.func.SimpleFunc* class method), 372

register\_additional\_repr() (*zfit.func.SumFunc* class method), 367

register\_additional\_repr() (*zfit.models.basefunctor.FunctorMixin* class method), 157

register\_additional\_repr() (*zfit.models.basic.CustomGaussOLD* class method), 163

register\_additional\_repr() (*zfit.models.basic.Exponential* class method), 170

register\_additional\_repr() (*zfit.models.dist\_tfp.ExponentialTFP* class method), 177

register\_additional\_repr() (*zfit.models.dist\_tfp.Gauss* class method), 184

register\_additional\_repr() (*zfit.models.dist\_tfp.TruncatedGauss* class method), 191

register\_additional\_repr() (*zfit.models.dist\_tfp.Uniform* class method), 197

register\_additional\_repr() (*zfit.models.dist\_tfp.WrapDistribution* class method), 204

register\_additional\_repr() (*zfit.models.functions.BaseFunctorFunc* class method), 204

`method`), 210  
`register_additional_repr()`  
     (`zfit.models.functions.ProdFunc` class method), 215  
`register_additional_repr()`  
     (`zfit.models.functions.SimpleFunc` class method), 220  
`register_additional_repr()`  
     (`zfit.models.functions.SumFunc` class method), 225  
`register_additional_repr()`  
     (`zfit.models.functions.ZFunc` class method), 230  
`register_additional_repr()`  
     (`zfit.models.functor.BaseFunc` class method), 236  
`register_additional_repr()`  
     (`zfit.models.functor.ProductPDF` class method), 243  
`register_additional_repr()`  
     (`zfit.models.functor.SumPDF` class method), 250  
`register_additional_repr()`  
     (`zfit.models.physics.CrystalBall` class method), 257  
`register_additional_repr()`  
     (`zfit.models.physics.DoubleCB` class method), 265  
`register_additional_repr()`  
     (`zfit.models.polynomials.Chebyshev` class method), 272  
`register_additional_repr()`  
     (`zfit.models.polynomials.Chebyshev2` class method), 280  
`register_additional_repr()`  
     (`zfit.models.polynomials.Hermite` class method), 287  
`register_additional_repr()`  
     (`zfit.models.polynomials.Laguerre` class method), 294  
`register_additional_repr()`  
     (`zfit.models.polynomials.Legendre` class method), 301  
`register_additional_repr()`  
     (`zfit.models.polynomials.RecursivePolynomial` class method), 308  
`register_additional_repr()`  
     (`zfit.models.special.SimpleFunc` class method), 316  
`register_additional_repr()`  
     (`zfit.models.special.SimplePDF` class method), 322  
`register_additional_repr()`  
     (`zfit.models.special.ZPDF` class method), 329  
`register_additional_repr()`  
     (`zfit.pdf.BaseFunc` class method), 408  
`register_additional_repr()` (`zfit.pdf.BasePDF` class method), 401  
`register_additional_repr()`  
     (`zfit.pdf.Chebyshev` class method), 464  
`register_additional_repr()`  
     (`zfit.pdf.Chebyshev2` class method), 478  
`register_additional_repr()`  
     (`zfit.pdf.CrystalBall` class method), 422  
`register_additional_repr()`  
     (`zfit.pdf.DoubleCB` class method), 429  
`register_additional_repr()`  
     (`zfit.pdf.Exponential` class method), 414  
`register_additional_repr()` (`zfit.pdf.Gauss` class method), 436  
`register_additional_repr()` (`zfit.pdf.Hermite` class method), 485  
`register_additional_repr()` (`zfit.pdf.Laguerre` class method), 492  
`register_additional_repr()` (`zfit.pdf.Legendre` class method), 471  
`register_additional_repr()`  
     (`zfit.pdf.ProductPDF` class method), 506  
`register_additional_repr()`  
     (`zfit.pdf.RecursivePolynomial` class method), 499  
`register_additional_repr()`  
     (`zfit.pdf.SimpleFunc` class method), 533  
`register_additional_repr()`  
     (`zfit.pdf.SimplePDF` class method), 526  
`register_additional_repr()` (`zfit.pdf.SumPDF` class method), 513  
`register_additional_repr()`  
     (`zfit.pdf.TruncatedGauss` class method), 450  
`register_additional_repr()` (`zfit.pdf.Uniform` class method), 443  
`register_additional_repr()`  
     (`zfit.pdf.WrapDistribution` class method), 457  
`register_additional_repr()` (`zfit.pdf.ZPDF` class method), 520  
`register_analytic_integral()`  
     (`zfit.core.basefunc.BaseFunc` class method), 52  
`register_analytic_integral()`  
     (`zfit.core.basemodel.BaseModel` class method), 57  
`register_analytic_integral()`  
     (`zfit.core.basepdf.BasePDF` class method), 65  
`register_analytic_integral()`

[\(zfit.core.interfaces.ZfitFunc class method\), 90](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.core.interfaces.ZfitModel class method\), 92](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.core.interfaces.ZfitPDF class method\), 94](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.func.BaseFunc class method\), 357](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.func.ProdFunc class method\), 362](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.func.SimpleFunc class method\), 373](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.func.SumFunc class method\), 367](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.basefunctor.FunctorMixin class method\), 157](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.basic.CustomGaussOLD class method\), 163](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.basic.Exponential class method\), 170](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.dist\\_tfp.ExponentialTFP class method\), 177](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.dist\\_tfp.Gauss class method\), 184](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.dist\\_tfp.TruncatedGauss class method\), 191](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.dist\\_tfp.Uniform class method\), 197](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.dist\\_tfp.WrapDistribution class method\), 204](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functions.BaseFunctorFunc class method\), 210](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functions.ProdFunc class method\), 215](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functions.SimpleFunc class method\), 220](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functions.SumFunc class method\), 225](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functions.ZFunc class method\), 230](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functor.BaseFunctor class method\), 236](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functor.ProductPDF class method\), 243](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.functor.SumPDF class method\), 250](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.physics.CrystalBall class method\), 258](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.physics.DoubleCB class method\), 265](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.Chebyshev class method\), 272](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.Chebyshev2 class method\), 280](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.Hermite class method\), 287](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.Laguerre class method\), 294](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.Legendre class method\), 301](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.polynomials.RecursivePolynomial class method\), 308](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.special.SimpleFunctorPDF class method\), 316](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.special.SimplePDF class method\), 322](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.models.special.ZPDF class method\), 329](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.BaseFunctor class method\), 408](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.BasePDF class method\), 401](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.Chebyshev class method\), 464](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.Chebyshev2 class method\), 478](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.CrystalBall class method\), 422](#)  
[register\\_analytic\\_integral\(\)](#)  
[\(zfit.pdf.DoubleCB class method\), 429](#)

```

register_analytic_integral()
    (zfit.pdf.Exponential class method), 415
register_analytic_integral() (zfit.pdf.Gauss
    class method), 436
register_analytic_integral()
    (zfit.pdf.Hermite class method), 485
register_analytic_integral()
    (zfit.pdf.Laguerre class method), 492
register_analytic_integral()
    (zfit.pdf.Legendre class method), 471
register_analytic_integral()
    (zfit.pdf.ProductPDF class method), 506
register_analytic_integral()
    (zfit.pdf.RecursivePolynomial class method),
    499
register_analytic_integral()
    (zfit.pdf.SimpleFunctorPDF class method),
    533
register_analytic_integral()
    (zfit.pdf.SimplePDF class method), 526
register_analytic_integral()
    (zfit.pdf.SumPDF class method), 513
register_analytic_integral()
    (zfit.pdf.TruncatedGauss class method),
    450
register_analytic_integral()
    (zfit.pdf.Uniform class method), 443
register_analytic_integral()
    (zfit.pdf.WrapDistribution class method),
    457
register_analytic_integral() (zfit.pdf.ZPDF
    class method), 520
register_cacher() (zfit.ComplexParameter
    method), 43
register_cacher() (zfit.ComposedParameter
    method), 42
register_cacher()
    (zfit.constraint.GaussianConstraint method),
    350
register_cacher()
    (zfit.constraint.SimpleConstraint method),
    349
register_cacher() (zfit.core.basefunc.BaseFunc
    method), 53
register_cacher()
    (zfit.core.basemodel.BaseModel method),
    58
register_cacher()
    (zfit.core.baseobject.BaseNumeric method), 60
register_cacher() (zfit.core.basepdf.BasePDF
    method), 66
register_cacher()
    (zfit.core.constraint.BaseConstraint method),
    68
register_cacher()
    (zfit.core.constraint.DistributionConstraint
    method), 70
register_cacher()
    (zfit.core.constraint.GaussianConstraint
    method), 71
register_cacher()
    (zfit.core.constraint.SimpleConstraint method),
    72
register_cacher() (zfit.core.data.Data method),
    75
register_cacher() (zfit.core.data.SampleData
    method), 79
register_cacher() (zfit.core.data.Sampler
    method), 82
register_cacher() (zfit.core.loss.BaseLoss
    method), 104
register_cacher() (zfit.core.loss.CachedLoss
    method), 105
register_cacher()
    (zfit.core.loss.ExtendedUnbinnedNLL method),
    106
register_cacher() (zfit.core.loss.SimpleLoss
    method), 107
register_cacher() (zfit.core.loss.UnbinnedNLL
    method), 108
register_cacher()
    (zfit.core.parameter.BaseComposedParameter
    method), 110
register_cacher()
    (zfit.core.parameter.BaseZParameter method),
    112
register_cacher()
    (zfit.core.parameter.ComplexParameter
    method), 113
register_cacher()
    (zfit.core.parameter.ComposedParameter
    method), 114
register_cacher()
    (zfit.core.parameter.ConstantParameter
    method), 116
register_cacher() (zfit.core.parameter.Parameter
    method), 122
register_cacher()
    (zfit.core.parameter.ZfitParameterMixin
    method), 137
register_cacher() (zfit.data.Data method), 353
register_cacher() (zfit.func.BaseFunc method),
    358
register_cacher() (zfit.func.ProdFunc method),
    363
register_cacher() (zfit.func.SimpleFunc method),
    373
register_cacher() (zfit.func.SumFunc method),

```



```

368
register_cacher() (zfit.loss.BaseLoss method),
377
register_cacher()
(zfit.loss.ExtendedUnbinnedNLL method),
375
register_cacher() (zfit.loss.SimpleLoss method),
378
register_cacher() (zfit.loss.UnbinnedNLL
method), 376
register_cacher() (zfit.minimize.Minuit method),
381
register_cacher()
(zfit.minimizers.minimizer_minuit.Minuit
method), 148
register_cacher()
(zfit.models.basefunctor.FunctorMixin
method), 157
register_cacher()
(zfit.models.basic.CustomGaussOLD method),
164
register_cacher() (zfit.models.basic.Exponential
method), 170
register_cacher()
(zfit.models.dist_tfp.ExponentialTFP method),
177
register_cacher() (zfit.models.dist_tfp.Gauss
method), 184
register_cacher()
(zfit.models.dist_tfp.TruncatedGauss method),
191
register_cacher() (zfit.models.dist_tfp.Uniform
method), 198
register_cacher()
(zfit.models.dist_tfp.WrapDistribution method),
205
register_cacher()
(zfit.models.functions.BaseFunctorFunc
method), 210
register_cacher()
(zfit.models.functions.ProdFunc method),
215
register_cacher()
(zfit.models.functions.SimpleFunc method),
221
register_cacher() (zfit.models.functions.SumFunc
method), 226
register_cacher() (zfit.models.functions.ZFunc
method), 231
register_cacher()
(zfit.models.functor.BaseFunctor method),
237
register_cacher()
(zfit.models.functor.ProductPDF method),
244
register_cacher() (zfit.models.functor.SumPDF
method), 251
register_cacher()
(zfit.models.physics.CrystalBall method),
258
register_cacher() (zfit.models.physics.DoubleCB
method), 266
register_cacher()
(zfit.models.polynomials.Chebyshev method),
273
register_cacher()
(zfit.models.polynomials.Chebyshev2 method),
280
register_cacher()
(zfit.models.polynomials.Hermite method),
287
register_cacher()
(zfit.models.polynomials.Laguerre method),
295
register_cacher()
(zfit.models.polynomials.Legendre method),
302
register_cacher()
(zfit.models.polynomials.RecursivePolynomial
method), 309
register_cacher()
(zfit.models.special.SimpleFunctorPDF
method), 316
register_cacher() (zfit.models.special.SimplePDF
method), 323
register_cacher() (zfit.models.special.ZPDF
method), 330
register_cacher() (zfit.param.ComplexParameter
method), 396
register_cacher()
(zfit.param.ComposedParameter method),
394
register_cacher() (zfit.param.ConstantParameter
method), 383
register_cacher() (zfit.param.Parameter method),
389
register_cacher() (zfit.Parameter method), 37
register_cacher() (zfit.pdf.BaseFunctor method),
408
register_cacher() (zfit.pdf.BasePDF method), 402
register_cacher() (zfit.pdf.Chebyshev method),
464
register_cacher() (zfit.pdf.Chebyshev2 method),
479
register_cacher() (zfit.pdf.CrystalBall method),
422
register_cacher() (zfit.pdf.DoubleCB method),
430

```

`register_cacher()` (*zfit.pdf.Exponential method*), 415  
`register_cacher()` (*zfit.pdf.Gauss method*), 437  
`register_cacher()` (*zfit.pdf.Hermite method*), 486  
`register_cacher()` (*zfit.pdf.Laguerre method*), 493  
`register_cacher()` (*zfit.pdf.Legendre method*), 472  
`register_cacher()` (*zfit.pdf.ProductPDF method*), 507  
`register_cacher()` (*zfit.pdf.RecursivePolynomial method*), 500  
`register_cacher()` (*zfit.pdf.SimpleFunctorPDF method*), 533  
`register_cacher()` (*zfit.pdf.SimplePDF method*), 527  
`register_cacher()` (*zfit.pdf.SumPDF method*), 514  
`register_cacher()` (*zfit.pdf.TruncatedGauss method*), 451  
`register_cacher()` (*zfit.pdf.Uniform method*), 444  
`register_cacher()` (*zfit.pdf.WrapDistribution method*), 457  
`register_cacher()` (*zfit.pdf.ZPDF method*), 520  
`register_cacher()` (*zfit.util.cache.Cachable method*), 332  
`register_cacher()` (*zfit.util.cache.FunctionCacheHolder method*), 333  
`register_cacher()` (*zfit.util.cache.ZfitCachable method*), 334  
`register_inverse_analytic_integral()` (*zfit.core.basefunc.BaseFunc class method*), 53  
`register_inverse_analytic_integral()` (*zfit.core.basemodel.BaseModel class method*), 58  
`register_inverse_analytic_integral()` (*zfit.core.basepdf.BasePDF class method*), 66  
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitFunc class method*), 90  
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitModel class method*), 92  
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitPDF class method*), 95  
`register_inverse_analytic_integral()` (*zfit.func.BaseFunc class method*), 358  
`register_inverse_analytic_integral()` (*zfit.func.ProdFunc class method*), 363  
`register_inverse_analytic_integral()` (*zfit.func.SimpleFunc class method*), 373  
`register_inverse_analytic_integral()` (*zfit.func.SumFunc class method*), 368  
`register_inverse_analytic_integral()` (*zfit.models.basefunctor.FunctorMixin class method*), 157  
`register_inverse_analytic_integral()` (*zfit.models.basic.CustomGaussOLD class method*), 164  
`register_inverse_analytic_integral()` (*zfit.models.basic.Exponential class method*), 171  
`register_inverse_analytic_integral()` (*zfit.models.dist\_tfp.ExponentialTFP class method*), 177  
`register_inverse_analytic_integral()` (*zfit.models.dist\_tfp.Gauss class method*), 184  
`register_inverse_analytic_integral()` (*zfit.models.dist\_tfp.TruncatedGauss class method*), 191  
`register_inverse_analytic_integral()` (*zfit.models.dist\_tfp.Uniform class method*), 198  
`register_inverse_analytic_integral()` (*zfit.models.dist\_tfp.WrapDistribution class method*), 205  
`register_inverse_analytic_integral()` (*zfit.models.functions.BaseFunctorFunc class method*), 211  
`register_inverse_analytic_integral()` (*zfit.models.functions.ProdFunc class method*), 216  
`register_inverse_analytic_integral()` (*zfit.models.functions.SimpleFunc class method*), 221  
`register_inverse_analytic_integral()` (*zfit.models.functions.SumFunc class method*), 226  
`register_inverse_analytic_integral()` (*zfit.models.functions.ZFunc class method*), 231  
`register_inverse_analytic_integral()` (*zfit.models.functor.BaseFunctor class method*), 237  
`register_inverse_analytic_integral()` (*zfit.models.functor.ProductPDF class method*), 244  
`register_inverse_analytic_integral()` (*zfit.models.functor.SumPDF class method*), 251  
`register_inverse_analytic_integral()` (*zfit.models.physics.CrystalBall class method*), 258  
`register_inverse_analytic_integral()` (*zfit.models.physics.DoubleCB class method*), 266  
`register_inverse_analytic_integral()` (*zfit.models.polynomials.Chebyshev class method*), 273  
`register_inverse_analytic_integral()` (*zfit.models.polynomials.Chebyshev2 class method*), 273

[method](#)), 280  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.polynomials.Hermite class method](#)), 287  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.polynomials.Laguerre class method](#)), 295  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.polynomials.Legendre class method](#)), 302  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.polynomials.RecursivePolynomial class method](#)), 309  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.special.SimpleFunctorPDF class method](#)), 317  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.special.SimplePDF class method](#)), 323  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.models.special.ZPDF class method](#)), 330  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.BaseFunctor class method](#)), 408  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.BasePDF class method](#)), 402  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Chebyshev class method](#)), 464  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Chebyshev2 class method](#)), 479  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.CrystalBall class method](#)), 423  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.DoubleCB class method](#)), 430  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Exponential class method](#)), 415  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Gauss class method](#)), 437  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Hermite class method](#)), 486  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Laguerre class method](#)), 493  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Legendre class method](#)), 472  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.ProductPDF class method](#)), 507  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.RecursivePolynomial class method](#)), 500  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.SimpleFunctorPDF class method](#)), 534  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.SimplePDF class method](#)), 527  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.SumPDF class method](#)), 514  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.TruncatedGauss class method](#)), 451  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.Uniform class method](#)), 444  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.WrapDistribution class method](#)), 457  
[register\\_inverse\\_analytic\\_integral\(\)](#)  
     ([zfit.pdf.ZPDF class method](#)), 520  
[register\\_tensor\\_conversion\(\)](#) (in module [zfit.core.parameter](#)), 138  
[registries](#) ([zfit.z.zextension.FunctionWrapperRegistry attribute](#)), 346  
[reorder\\_by\\_indices\(\)](#) ([zfit.core.limits.Space method](#)), 101  
[reorder\\_by\\_indices\(\)](#)  
     ([zfit.core.sample.EventSpace method](#)), 141  
[reorder\\_by\\_indices\(\)](#) ([zfit.Space method](#)), 47  
[resample\(\)](#) ([zfit.core.data.Sampler method](#)), 82  
[rescale\\_minus\\_plus\\_one\(\)](#) (in module [zfit.models.polynomials](#)), 311  
[reset\(\)](#) ([zfit.z.zextension.FunctionWrapperRegistry method](#)), 346  
[reset\\_cache\(\)](#) ([zfit.ComplexParameter method](#)), 43  
[reset\\_cache\(\)](#) ([zfit.ComposedParameter method](#)), 42  
[reset\\_cache\(\)](#) ([zfit.constraint.GaussianConstraint method](#)), 350  
[reset\\_cache\(\)](#) ([zfit.constraint.SimpleConstraint method](#)), 349  
[reset\\_cache\(\)](#) ([zfit.core.basefunc.BaseFunc method](#)), 53  
[reset\\_cache\(\)](#) ([zfit.core.basemodel.BaseModel method](#)), 58  
[reset\\_cache\(\)](#) ([zfit.core.baseobject.BaseNumeric method](#)), 60  
[reset\\_cache\(\)](#) ([zfit.core.basepdf.BasePDF method](#)), 66  
[reset\\_cache\(\)](#) ([zfit.core.constraint.BaseConstraint method](#)), 68  
[reset\\_cache\(\)](#) ([zfit.core.constraint.DistributionConstraint method](#)), 70  
[reset\\_cache\(\)](#) ([zfit.core.constraint.GaussianConstraint method](#)), 71  
[reset\\_cache\(\)](#) ([zfit.core.constraint.SimpleConstraint method](#)), 72  
[reset\\_cache\(\)](#) ([zfit.core.data.Data method](#)), 75  
[reset\\_cache\(\)](#) ([zfit.core.data.SampleData method](#)), 79  
[reset\\_cache\(\)](#) ([zfit.core.data.Sampler method](#)), 82  
[reset\\_cache\(\)](#) ([zfit.core.loss.BaseLoss method](#)), 104  
[reset\\_cache\(\)](#) ([zfit.core.loss.CachedLoss method](#)), 105  
[reset\\_cache\(\)](#) ([zfit.core.loss.ExtendedUnbinnedNLL method](#)), 106



`reset_cache()` (`zfit.core.loss.SimpleLoss` method), 107  
`reset_cache()` (`zfit.core.loss.UnbinnedNLL` method), 108  
`reset_cache()` (`zfit.core.parameter.BaseComposedParameter` method), 110  
`reset_cache()` (`zfit.core.parameter.BaseZParameter` method), 112  
`reset_cache()` (`zfit.core.parameter.ComplexParameter` method), 113  
`reset_cache()` (`zfit.core.parameter.ComposedParameter` method), 114  
`reset_cache()` (`zfit.core.parameter.ConstantParameter` method), 116  
`reset_cache()` (`zfit.core.parameter.Parameter` method), 122  
`reset_cache()` (`zfit.core.parameter.ZfitParameterMixin` method), 137  
`reset_cache()` (`zfit.data.Data` method), 353  
`reset_cache()` (`zfit.func.BaseFunc` method), 358  
`reset_cache()` (`zfit.func.ProdFunc` method), 363  
`reset_cache()` (`zfit.func.SimpleFunc` method), 373  
`reset_cache()` (`zfit.func.SumFunc` method), 368  
`reset_cache()` (`zfit.loss.BaseLoss` method), 377  
`reset_cache()` (`zfit.loss.ExtendedUnbinnedNLL` method), 375  
`reset_cache()` (`zfit.loss.SimpleLoss` method), 379  
`reset_cache()` (`zfit.loss.UnbinnedNLL` method), 376  
`reset_cache()` (`zfit.minimize.Minuit` method), 381  
`reset_cache()` (`zfit.minimizers.minimizer_minuit.Minuit` method), 148  
`reset_cache()` (`zfit.models.basefunctor.FunctorMixin` method), 158  
`reset_cache()` (`zfit.models.basic.CustomGaussOLD` method), 164  
`reset_cache()` (`zfit.models.basic.Exponential` method), 171  
`reset_cache()` (`zfit.models.dist_tfp.ExponentialTFP` method), 177  
`reset_cache()` (`zfit.models.dist_tfp.Gauss` method), 184  
`reset_cache()` (`zfit.models.dist_tfp.TruncatedGauss` method), 191  
`reset_cache()` (`zfit.models.dist_tfp.Uniform` method), 198  
`reset_cache()` (`zfit.models.dist_tfp.WrapDistribution` method), 205  
`reset_cache()` (`zfit.models.functions.BaseFunctorFunc` method), 211  
`reset_cache()` (`zfit.models.functions.ProdFunc` method), 216  
`reset_cache()` (`zfit.models.functions.SimpleFunc` method), 221  
`reset_cache()` (`zfit.models.functions.SumFunc` method), 226  
`reset_cache()` (`zfit.models.functions.ZFunc` method), 231  
`reset_cache()` (`zfit.models.functor.BaseFunctor` method), 244  
`reset_cache()` (`zfit.models.functor.ProductPDF` method), 251  
`reset_cache()` (`zfit.models.functor.SumPDF` method), 258  
`reset_cache()` (`zfit.models.physics.CrystalBall` method), 266  
`reset_cache()` (`zfit.models.physics.DoubleCB` method), 273  
`reset_cache()` (`zfit.models.polynomials.Chebyshev` method), 280  
`reset_cache()` (`zfit.models.polynomials.Chebyshev2` method), 288  
`reset_cache()` (`zfit.models.polynomials.Hermite` method), 295  
`reset_cache()` (`zfit.models.polynomials.Laguerre` method), 302  
`reset_cache()` (`zfit.models.polynomials.Legendre` method), 309  
`reset_cache()` (`zfit.models.polynomials.RecursivePolynomial` method), 317  
`reset_cache()` (`zfit.models.special.SimpleFunctorPDF` method), 323  
`reset_cache()` (`zfit.models.special.SimplePDF` method), 330  
`reset_cache()` (`zfit.models.special.ZPDF` method), 396  
`reset_cache()` (`zfit.param.ComplexParameter` method), 395  
`reset_cache()` (`zfit.param.ComposedParameter` method), 383  
`reset_cache()` (`zfit.param.ConstantParameter` method), 389  
`reset_cache()` (`zfit.Parameter` method), 37  
`reset_cache()` (`zfit.pdf.BaseFunctor` method), 408  
`reset_cache()` (`zfit.pdf.BasePDF` method), 402  
`reset_cache()` (`zfit.pdf.Chebyshev` method), 465  
`reset_cache()` (`zfit.pdf.Chebyshev2` method), 479  
`reset_cache()` (`zfit.pdf.CrystalBall` method), 423  
`reset_cache()` (`zfit.pdf.DoubleCB` method), 430  
`reset_cache()` (`zfit.pdf.Exponential` method), 415  
`reset_cache()` (`zfit.pdf.Gauss` method), 437  
`reset_cache()` (`zfit.pdf.Hermite` method), 486  
`reset_cache()` (`zfit.pdf.Laguerre` method), 493  
`reset_cache()` (`zfit.pdf.Legendre` method), 472  
`reset_cache()` (`zfit.pdf.ProductPDF` method), 507  
`reset_cache()` (`zfit.pdf.RecursivePolynomial` method), 500  
`reset_cache()` (`zfit.pdf.SimpleFunctorPDF` method),

534

`reset_cache()` (*zfit.pdf.SimplePDF method*), 527

`reset_cache()` (*zfit.pdf.SumPDF method*), 514

`reset_cache()` (*zfit.pdf.TruncatedGauss method*), 451

`reset_cache()` (*zfit.pdf.Uniform method*), 444

`reset_cache()` (*zfit.pdf.WrapDistribution method*), 457

`reset_cache()` (*zfit.pdf.ZPDF method*), 520

`reset_cache()` (*zfit.util.cache.Cachable method*), 332

`reset_cache()` (*zfit.util.cache.FunctionCacheHolder method*), 334

`reset_cache()` (*zfit.util.cache.ZfitCachable method*), 334

`reset_cache_self()` (*zfit.ComplexParameter method*), 43

`reset_cache_self()` (*zfit.ComposedParameter method*), 42

`reset_cache_self()` (*zfit.constraint.GaussianConstraint method*), 350

`reset_cache_self()` (*zfit.constraint.SimpleConstraint method*), 349

`reset_cache_self()` (*zfit.core.basefunc.BaseFunc method*), 53

`reset_cache_self()` (*zfit.core.basemodel.BaseModel method*), 58

`reset_cache_self()` (*zfit.core.baseobject.BaseNumeric method*), 60

`reset_cache_self()` (*zfit.core.basepdf.BasePDF method*), 66

`reset_cache_self()` (*zfit.core.constraint.BaseConstraint method*), 68

`reset_cache_self()` (*zfit.core.constraint.DistributionConstraint method*), 70

`reset_cache_self()` (*zfit.core.constraint.GaussianConstraint method*), 71

`reset_cache_self()` (*zfit.core.constraint.SimpleConstraint method*), 72

`reset_cache_self()` (*zfit.core.data.Data method*), 75

`reset_cache_self()` (*zfit.core.data.SampleData method*), 79

`reset_cache_self()` (*zfit.core.data.Sampler method*), 82

`reset_cache_self()` (*zfit.core.loss.BaseLoss method*), 104

`reset_cache_self()` (*zfit.core.loss.CachedLoss method*), 105

`reset_cache_self()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 106

`reset_cache_self()` (*zfit.core.loss.SimpleLoss method*), 107

`reset_cache_self()` (*zfit.core.loss.UnbinnedNLL method*), 108

`reset_cache_self()` (*zfit.core.parameter.BaseComposedParameter method*), 110

`reset_cache_self()` (*zfit.core.parameter.BaseZParameter method*), 112

`reset_cache_self()` (*zfit.core.parameter.ComplexParameter method*), 113

`reset_cache_self()` (*zfit.core.parameter.ComposedParameter method*), 114

`reset_cache_self()` (*zfit.core.parameter.ConstantParameter method*), 116

`reset_cache_self()` (*zfit.core.parameter.Parameter method*), 122

`reset_cache_self()` (*zfit.core.parameter.ZfitParameterMixin method*), 137

`reset_cache_self()` (*zfit.data.Data method*), 353

`reset_cache_self()` (*zfit.func.BaseFunc method*), 358

`reset_cache_self()` (*zfit.func.ProdFunc method*), 363

`reset_cache_self()` (*zfit.func.SimpleFunc method*), 373

`reset_cache_self()` (*zfit.func.SumFunc method*), 368

`reset_cache_self()` (*zfit.loss.BaseLoss method*), 377

`reset_cache_self()` (*zfit.loss.ExtendedUnbinnedNLL method*), 375

`reset_cache_self()` (*zfit.loss.SimpleLoss method*), 379

`reset_cache_self()` (*zfit.loss.UnbinnedNLL method*), 376

`reset_cache_self()` (*zfit.minimize.Minuit method*), 381

`reset_cache_self()` (*zfit.minimizers.minimizer\_minuit.Minuit method*), 148

`reset_cache_self()`

```

        (zfit.models.basefunctor.FunctorMixin
         method), 158
reset_cache_self()
    (zfit.models.basic.CustomGaussOLD method),
    164
reset_cache_self()
    (zfit.models.basic.Exponential method), 171
reset_cache_self()
    (zfit.models.dist_tfp.ExponentialTFP method),
    177
reset_cache_self() (zfit.models.dist_tfp.Gauss
    method), 184
reset_cache_self()
    (zfit.models.dist_tfp.TruncatedGauss method),
    191
reset_cache_self() (zfit.models.dist_tfp.Uniform
    method), 198
reset_cache_self()
    (zfit.models.dist_tfp.WrapDistribution method),
    205
reset_cache_self()
    (zfit.models.functions.BaseFunctorFunc
     method), 211
reset_cache_self()
    (zfit.models.functions.ProdFunc method),
    216
reset_cache_self()
    (zfit.models.functions.SimpleFunc method),
    221
reset_cache_self()
    (zfit.models.functions.SumFunc method),
    226
reset_cache_self() (zfit.models.functions.ZFunc
    method), 231
reset_cache_self()
    (zfit.models.functor.BaseFunctor method),
    237
reset_cache_self()
    (zfit.models.functor.ProductPDF method),
    244
reset_cache_self() (zfit.models.functor.SumPDF
    method), 251
reset_cache_self()
    (zfit.models.physics.CrystalBall method),
    258
reset_cache_self()
    (zfit.models.physics.DoubleCB method),
    266
reset_cache_self()
    (zfit.models.polynomials.Chebyshev method),
    273
reset_cache_self()
    (zfit.models.polynomials.Chebyshev2 method),
    280
reset_cache_self()
    (zfit.models.polynomials.Hermite method),
    288
reset_cache_self()
    (zfit.models.polynomials.Laguerre method),
    295
reset_cache_self()
    (zfit.models.polynomials.Legendre method),
    302
reset_cache_self()
    (zfit.models.polynomials.RecursivePolynomial
     method), 309
reset_cache_self()
    (zfit.models.special.SimpleFunctorPDF
     method), 317
reset_cache_self()
    (zfit.models.special.SimplePDF method),
    323
reset_cache_self() (zfit.models.special.ZPDF
    method), 330
reset_cache_self()
    (zfit.param.ComplexParameter method),
    396
reset_cache_self()
    (zfit.param.ComposedParameter method),
    395
reset_cache_self()
    (zfit.param.ConstantParameter method),
    383
reset_cache_self() (zfit.param.Parameter
    method), 389
reset_cache_self() (zfit.Parameter method), 37
reset_cache_self() (zfit.pdf.BaseFunctor
    method), 408
reset_cache_self() (zfit.pdf.BasePDF method),
    402
reset_cache_self() (zfit.pdf.Chebyshev method),
    465
reset_cache_self() (zfit.pdf.Chebyshev2 method),
    479
reset_cache_self() (zfit.pdf.CrystalBall method),
    423
reset_cache_self() (zfit.pdf.DoubleCB method),
    430
reset_cache_self() (zfit.pdf.Exponential method),
    415
reset_cache_self() (zfit.pdf.Gauss method), 437
reset_cache_self() (zfit.pdf.Hermite method), 486
reset_cache_self() (zfit.pdf.Laguerre method),
    493
reset_cache_self() (zfit.pdf.Legendre method),
    472
reset_cache_self() (zfit.pdf.ProductPDF
    method), 507

```

[reset\\_cache\\_self\(\)](#) ([zfit.pdf.RecursivePolynomial method](#)), 500  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.SimpleFunctorPDF method](#)), 534  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.SimplePDF method](#)), 527  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.SumPDF method](#)), 514  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.TruncatedGauss method](#)), 451  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.Uniform method](#)), 444  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.WrapDistribution method](#)), 457  
[reset\\_cache\\_self\(\)](#) ([zfit.pdf.ZPDF method](#)), 520  
[reset\\_cache\\_self\(\)](#) ([zfit.util.cache.Cachable method](#)), 332  
[reset\\_cache\\_self\(\)](#) ([zfit.util.cache.FunctionCacheHolder method](#)), 334  
[reset\\_cache\\_self\(\)](#) ([zfit.util.cache.ZfitCachable method](#)), 334  
[run\\_no\\_nan\(\)](#) (in module [zfit.z.zextension](#)), 347  
[RunManager](#) (class in [zfit.util.execution](#)), 340

## S

[safe\\_where\(\)](#) (in module [zfit.z.zextension](#)), 347  
[sample\(\)](#) ([zfit.constraint.GaussianConstraint method](#)), 351  
[sample\(\)](#) ([zfit.constraint.SimpleConstraint method](#)), 349  
[sample\(\)](#) ([zfit.core.basefunc.BaseFunc method](#)), 53  
[sample\(\)](#) ([zfit.core.basemodel.BaseModel method](#)), 58  
[sample\(\)](#) ([zfit.core.basepdf.BasePDF method](#)), 66  
[sample\(\)](#) ([zfit.core.constraint.BaseConstraint method](#)), 68  
[sample\(\)](#) ([zfit.core.constraint.DistributionConstraint method](#)), 70  
[sample\(\)](#) ([zfit.core.constraint.GaussianConstraint method](#)), 71  
[sample\(\)](#) ([zfit.core.constraint.SimpleConstraint method](#)), 72  
[sample\(\)](#) ([zfit.core.interfaces.ZfitFunc method](#)), 90  
[sample\(\)](#) ([zfit.core.interfaces.ZfitModel method](#)), 92  
[sample\(\)](#) ([zfit.core.interfaces.ZfitPDF method](#)), 95  
[sample\(\)](#) ([zfit.func.BaseFunc method](#)), 358  
[sample\(\)](#) ([zfit.func.ProdFunc method](#)), 363  
[sample\(\)](#) ([zfit.func.SimpleFunc method](#)), 373  
[sample\(\)](#) ([zfit.func.SumFunc method](#)), 368  
[sample\(\)](#) ([zfit.models.basefunctor.FunctorMixin method](#)), 158  
[sample\(\)](#) ([zfit.models.basic.CustomGaussOLD method](#)), 164  
[sample\(\)](#) ([zfit.models.basic.Exponential method](#)), 171  
[sample\(\)](#) ([zfit.models.dist\\_tfp.ExponentialTFP method](#)), 177  
[sample\(\)](#) ([zfit.models.dist\\_tfp.Gauss method](#)), 184  
[sample\(\)](#) ([zfit.models.dist\\_tfp.TruncatedGauss method](#)), 191  
[sample\(\)](#) ([zfit.models.dist\\_tfp.Uniform method](#)), 198  
[sample\(\)](#) ([zfit.models.dist\\_tfp.WrapDistribution method](#)), 205  
[sample\(\)](#) ([zfit.models.functions.BaseFunctorFunc method](#)), 211  
[sample\(\)](#) ([zfit.models.functions.ProdFunc method](#)), 216  
[sample\(\)](#) ([zfit.models.functions.SimpleFunc method](#)), 221  
[sample\(\)](#) ([zfit.models.functions.SumFunc method](#)), 226  
[sample\(\)](#) ([zfit.models.functions.ZFunc method](#)), 231  
[sample\(\)](#) ([zfit.models.functor.BaseFunctor method](#)), 237  
[sample\(\)](#) ([zfit.models.functor.ProductPDF method](#)), 244  
[sample\(\)](#) ([zfit.models.functor.SumPDF method](#)), 251  
[sample\(\)](#) ([zfit.models.physics.CrystalBall method](#)), 258  
[sample\(\)](#) ([zfit.models.physics.DoubleCB method](#)), 266  
[sample\(\)](#) ([zfit.models.polynomials.Chebyshev method](#)), 273  
[sample\(\)](#) ([zfit.models.polynomials.Chebyshev2 method](#)), 280  
[sample\(\)](#) ([zfit.models.polynomials.Hermite method](#)), 288  
[sample\(\)](#) ([zfit.models.polynomials.Laguerre method](#)), 295  
[sample\(\)](#) ([zfit.models.polynomials.Legendre method](#)), 302  
[sample\(\)](#) ([zfit.models.polynomials.RecursivePolynomial method](#)), 309  
[sample\(\)](#) ([zfit.models.special.SimpleFunctorPDF method](#)), 317  
[sample\(\)](#) ([zfit.models.special.SimplePDF method](#)), 323  
[sample\(\)](#) ([zfit.models.special.ZPDF method](#)), 330  
[sample\(\)](#) ([zfit.pdf.BaseFunctor method](#)), 409  
[sample\(\)](#) ([zfit.pdf.BasePDF method](#)), 402  
[sample\(\)](#) ([zfit.pdf.Chebyshev method](#)), 465  
[sample\(\)](#) ([zfit.pdf.Chebyshev2 method](#)), 479  
[sample\(\)](#) ([zfit.pdf.CrystalBall method](#)), 423  
[sample\(\)](#) ([zfit.pdf.DoubleCB method](#)), 430  
[sample\(\)](#) ([zfit.pdf.Exponential method](#)), 415  
[sample\(\)](#) ([zfit.pdf.Gauss method](#)), 437  
[sample\(\)](#) ([zfit.pdf.Hermite method](#)), 486  
[sample\(\)](#) ([zfit.pdf.Laguerre method](#)), 493  
[sample\(\)](#) ([zfit.pdf.Legendre method](#)), 472  
[sample\(\)](#) ([zfit.pdf.ProductPDF method](#)), 507  
[sample\(\)](#) ([zfit.pdf.RecursivePolynomial method](#)), 500



`sample()` (*zfit.pdf.SimpleFunctorPDF* method), 534  
`sample()` (*zfit.pdf.SimplePDF* method), 527  
`sample()` (*zfit.pdf.SumPDF* method), 514  
`sample()` (*zfit.pdf.TruncatedGauss* method), 451  
`sample()` (*zfit.pdf.Uniform* method), 444  
`sample()` (*zfit.pdf.WrapDistribution* method), 458  
`sample()` (*zfit.pdf.ZPDF* method), 520  
`SampleData` (class in *zfit.core.data*), 76  
`Sampler` (class in *zfit.core.data*), 79  
`scatter_add()` (*zfit.core.parameter.Parameter* method), 122  
`scatter_add()` (*zfit.core.parameter.TFBaseVariable* method), 132  
`scatter_add()` (*zfit.param.Parameter* method), 389  
`scatter_add()` (*zfit.Parameter* method), 37  
`scatter_div()` (*zfit.core.parameter.Parameter* method), 122  
`scatter_div()` (*zfit.core.parameter.TFBaseVariable* method), 133  
`scatter_div()` (*zfit.param.Parameter* method), 390  
`scatter_div()` (*zfit.Parameter* method), 37  
`scatter_max()` (*zfit.core.parameter.Parameter* method), 123  
`scatter_max()` (*zfit.core.parameter.TFBaseVariable* method), 133  
`scatter_max()` (*zfit.param.Parameter* method), 390  
`scatter_max()` (*zfit.Parameter* method), 38  
`scatter_min()` (*zfit.core.parameter.Parameter* method), 123  
`scatter_min()` (*zfit.core.parameter.TFBaseVariable* method), 133  
`scatter_min()` (*zfit.param.Parameter* method), 390  
`scatter_min()` (*zfit.Parameter* method), 38  
`scatter_mul()` (*zfit.core.parameter.Parameter* method), 123  
`scatter_mul()` (*zfit.core.parameter.TFBaseVariable* method), 133  
`scatter_mul()` (*zfit.param.Parameter* method), 390  
`scatter_mul()` (*zfit.Parameter* method), 38  
`scatter_nd_add()` (*zfit.core.parameter.Parameter* method), 123  
`scatter_nd_add()` (*zfit.core.parameter.TFBaseVariable* method), 134  
`scatter_nd_add()` (*zfit.param.Parameter* method), 391  
`scatter_nd_add()` (*zfit.Parameter* method), 38  
`scatter_nd_sub()` (*zfit.core.parameter.Parameter* method), 124  
`scatter_nd_sub()` (*zfit.core.parameter.TFBaseVariable* method), 134  
`scatter_nd_sub()` (*zfit.param.Parameter* method), 391  
`scatter_nd_sub()` (*zfit.Parameter* method), 39  
`scatter_nd_update()` (*zfit.core.parameter.Parameter* method), 125  
`scatter_nd_update()` (*zfit.core.parameter.TFBaseVariable* method), 135  
`scatter_nd_update()` (*zfit.param.Parameter* method), 392  
`scatter_nd_update()` (*zfit.Parameter* method), 39  
`scatter_sub()` (*zfit.core.parameter.Parameter* method), 125  
`scatter_sub()` (*zfit.core.parameter.TFBaseVariable* method), 135  
`scatter_sub()` (*zfit.param.Parameter* method), 393  
`scatter_sub()` (*zfit.Parameter* method), 40  
`scatter_update()` (*zfit.core.parameter.Parameter* method), 125  
`scatter_update()` (*zfit.core.parameter.TFBaseVariable* method), 136  
`scatter_update()` (*zfit.param.Parameter* method), 393  
`scatter_update()` (*zfit.Parameter* method), 40  
`Scipy` (class in *zfit.minimize*), 381  
`Scipy` (class in *zfit.minimizers.minimizers\_scipy*), 149  
`ScipyMinimizer` (in module *zfit.minimize*), 379  
`ScipyOptimizerInterface` (class in *zfit.minimizers.tf\_external\_optimizer*), 152  
`set_cpus_explicit()` (*zfit.util.execution.RunManager* method), 340  
`set_data_range()` (*zfit.core.data.Data* method), 75  
`set_data_range()` (*zfit.core.data.SampleData* method), 79  
`set_data_range()` (*zfit.core.data.Sampler* method), 82  
`set_data_range()` (*zfit.data.Data* method), 353  
`set_n_cpu()` (*zfit.util.execution.RunManager* method), 340  
`set_norm_range()` (*zfit.core.basepdf.BasePDF* method), 67  
`set_norm_range()` (*zfit.core.interfaces.ZfitPDF* method), 95  
`set_norm_range()` (*zfit.models.basic.CustomGaussOLD* method), 164  
`set_norm_range()` (*zfit.models.basic.Exponential* method), 171  
`set_norm_range()` (*zfit.models.dist\_tfp.ExponentialTFP* method), 178  
`set_norm_range()` (*zfit.models.dist\_tfp.Gauss* method), 185  
`set_norm_range()` (*zfit.models.dist\_tfp.TruncatedGauss* method), 192  
`set_norm_range()` (*zfit.models.dist\_tfp.Uniform* method), 199  
`set_norm_range()` (*zfit.models.dist\_tfp.WrapDistribution*

- method*), 205
- `set_norm_range()` (*zfit.models.functor.BaseFunctor method*), 238
- `set_norm_range()` (*zfit.models.functor.ProductPDF method*), 244
- `set_norm_range()` (*zfit.models.functor.SumPDF method*), 251
- `set_norm_range()` (*zfit.models.physics.CrystalBall method*), 259
- `set_norm_range()` (*zfit.models.physics.DoubleCB method*), 266
- `set_norm_range()` (*zfit.models.polynomials.Chebyshev method*), 274
- `set_norm_range()` (*zfit.models.polynomials.Chebyshev2 method*), 281
- `set_norm_range()` (*zfit.models.polynomials.Hermite method*), 288
- `set_norm_range()` (*zfit.models.polynomials.Laguerre method*), 295
- `set_norm_range()` (*zfit.models.polynomials.Legendre method*), 302
- `set_norm_range()` (*zfit.models.polynomials.RecursivePolynomial method*), 309
- `set_norm_range()` (*zfit.models.special.SimpleFunctorPDF method*), 317
- `set_norm_range()` (*zfit.models.special.SimplePDF method*), 324
- `set_norm_range()` (*zfit.models.special.ZPDF method*), 330
- `set_norm_range()` (*zfit.pdf.BaseFunctor method*), 409
- `set_norm_range()` (*zfit.pdf.BasePDF method*), 402
- `set_norm_range()` (*zfit.pdf.Chebyshev method*), 465
- `set_norm_range()` (*zfit.pdf.Chebyshev2 method*), 479
- `set_norm_range()` (*zfit.pdf.CrystalBall method*), 423
- `set_norm_range()` (*zfit.pdf.DoubleCB method*), 430
- `set_norm_range()` (*zfit.pdf.Exponential method*), 416
- `set_norm_range()` (*zfit.pdf.Gauss method*), 437
- `set_norm_range()` (*zfit.pdf.Hermite method*), 487
- `set_norm_range()` (*zfit.pdf.Laguerre method*), 494
- `set_norm_range()` (*zfit.pdf.Legendre method*), 472
- `set_norm_range()` (*zfit.pdf.ProductPDF method*), 507
- `set_norm_range()` (*zfit.pdf.RecursivePolynomial method*), 501
- `set_norm_range()` (*zfit.pdf.SimpleFunctorPDF method*), 534
- `set_norm_range()` (*zfit.pdf.SimplePDF method*), 527
- `set_norm_range()` (*zfit.pdf.SumPDF method*), 514
- `set_norm_range()` (*zfit.pdf.TruncatedGauss method*), 451
- `set_norm_range()` (*zfit.pdf.Uniform method*), 444
- `set_norm_range()` (*zfit.pdf.WrapDistribution method*), 458
- `set_norm_range()` (*zfit.pdf.ZPDF method*), 521
- `set_seed()` (*in module zfit.settings*), 535
- `set_shape()` (*zfit.core.parameter.Parameter method*), 126
- `set_shape()` (*zfit.core.parameter.TFBaseVariable method*), 136
- `set_shape()` (*zfit.param.Parameter method*), 393
- `set_shape()` (*zfit.Parameter method*), 41
- `set_value()` (*zfit.core.parameter.Parameter method*), 126
- `set_value()` (*zfit.param.Parameter method*), 393
- `set_value()` (*zfit.Parameter method*), 41
- `set_verbosity()` (*in module zfit.settings*), 535
- `set_weights()` (*zfit.core.data.Data method*), 76
- `set_weights()` (*zfit.core.data.SampleData method*), 79
- `set_weights()` (*zfit.core.data.Sampler method*), 82
- `set_weights()` (*zfit.data.Data method*), 354
- `setdefault()` (*zfit.util.container.DotDict method*), 335
- `setup_function()` (*in module zfit.core.testing*), 142
- `shape` (*zfit.core.parameter.Parameter attribute*), 126
- `shape` (*zfit.core.parameter.TFBaseVariable attribute*), 136
- `shape` (*zfit.param.Parameter attribute*), 393
- `shape` (*zfit.Parameter attribute*), 41
- `shape_np_tf()` (*in module zfit.core.limits*), 102
- `ShapeIncompatibleError`, 339
- `SimpleConstraint` (*class in zfit.constraint*), 348
- `SimpleConstraint` (*class in zfit.core.constraint*), 71
- `SimpleFunc` (*class in zfit.func*), 369
- `SimpleFunc` (*class in zfit.models.functions*), 216
- `SimpleFunctorPDF` (*class in zfit.models.special*), 311
- `SimpleFunctorPDF` (*class in zfit.pdf*), 528
- `SimpleLoss` (*class in zfit.core.loss*), 106
- `SimpleLoss` (*class in zfit.loss*), 377
- `SimpleModelSubclassMixin` (*class in zfit.core.basemodel*), 58
- `SimplePDF` (*class in zfit.models.special*), 318
- `SimplePDF` (*class in zfit.pdf*), 521
- `sort_by_axes()` (*zfit.core.data.Data method*), 76
- `sort_by_axes()` (*zfit.core.data.SampleData method*), 79
- `sort_by_axes()` (*zfit.core.data.Sampler method*), 82
- `sort_by_axes()` (*zfit.core.integration.PartialIntegralSampleData method*), 87
- `sort_by_axes()` (*zfit.core.interfaces.ZfitData method*), 88
- `sort_by_axes()` (*zfit.data.Data method*), 354
- `sort_by_obs()` (*zfit.core.data.Data method*), 76

- `sort_by_obs()` (`zfit.core.data.SampleData` method), 79
- `sort_by_obs()` (`zfit.core.data.Sampler` method), 82
- `sort_by_obs()` (`zfit.core.integration.PartialIntegralSampleData` method), 87
- `sort_by_obs()` (`zfit.core.interfaces.ZfitData` method), 88
- `sort_by_obs()` (`zfit.data.Data` method), 354
- `Space` (class in `zfit`), 44
- `Space` (class in `zfit.core.limits`), 97
- `space` (`zfit.core.basefunc.BaseFunc` attribute), 53
- `space` (`zfit.core.basemodel.BaseModel` attribute), 58
- `space` (`zfit.core.basepdf.BasePDF` attribute), 67
- `space` (`zfit.core.data.Data` attribute), 76
- `space` (`zfit.core.data.SampleData` attribute), 79
- `space` (`zfit.core.data.Sampler` attribute), 82
- `space` (`zfit.core.dimension.BaseDimensional` attribute), 83
- `space` (`zfit.core.integration.PartialIntegralSampleData` attribute), 87
- `space` (`zfit.core.interfaces.ZfitData` attribute), 88
- `space` (`zfit.core.interfaces.ZfitDimensional` attribute), 89
- `space` (`zfit.core.interfaces.ZfitFunc` attribute), 90
- `space` (`zfit.core.interfaces.ZfitModel` attribute), 93
- `space` (`zfit.core.interfaces.ZfitPDF` attribute), 95
- `space` (`zfit.data.Data` attribute), 354
- `space` (`zfit.func.BaseFunc` attribute), 359
- `space` (`zfit.func.ProdFunc` attribute), 364
- `space` (`zfit.func.SimpleFunc` attribute), 374
- `space` (`zfit.func.SumFunc` attribute), 369
- `space` (`zfit.models.basefunctor.FunctorMixin` attribute), 158
- `space` (`zfit.models.basic.CustomGaussOLD` attribute), 164
- `space` (`zfit.models.basic.Exponential` attribute), 171
- `space` (`zfit.models.dist_tfp.ExponentialTFP` attribute), 178
- `space` (`zfit.models.dist_tfp.Gauss` attribute), 185
- `space` (`zfit.models.dist_tfp.TruncatedGauss` attribute), 192
- `space` (`zfit.models.dist_tfp.Uniform` attribute), 199
- `space` (`zfit.models.dist_tfp.WrapDistribution` attribute), 206
- `space` (`zfit.models.functions.BaseFunctorFunc` attribute), 211
- `space` (`zfit.models.functions.ProdFunc` attribute), 216
- `space` (`zfit.models.functions.SimpleFunc` attribute), 221
- `space` (`zfit.models.functions.SumFunc` attribute), 226
- `space` (`zfit.models.functions.ZFunc` attribute), 231
- `space` (`zfit.models.functor.BaseFunctor` attribute), 238
- `space` (`zfit.models.functor.ProductPDF` attribute), 244
- `space` (`zfit.models.functor.SumPDF` attribute), 251
- `space` (`zfit.models.physics.CrystalBall` attribute), 259
- `space` (`zfit.models.physics.DoubleCB` attribute), 266
- `space` (`zfit.models.polynomials.Chebyshev` attribute), 274
- `space` (`zfit.models.polynomials.Chebyshev2` attribute), 281
- `space` (`zfit.models.polynomials.Hermite` attribute), 288
- `space` (`zfit.models.polynomials.Laguerre` attribute), 295
- `space` (`zfit.models.polynomials.Legendre` attribute), 303
- `space` (`zfit.models.polynomials.RecursivePolynomial` attribute), 309
- `space` (`zfit.models.special.SimpleFunctorPDF` attribute), 317
- `space` (`zfit.models.special.SimplePDF` attribute), 324
- `space` (`zfit.models.special.ZPDF` attribute), 330
- `space` (`zfit.pdf.BaseFunctor` attribute), 409
- `space` (`zfit.pdf.BasePDF` attribute), 402
- `space` (`zfit.pdf.Chebyshev` attribute), 465
- `space` (`zfit.pdf.Chebyshev2` attribute), 480
- `space` (`zfit.pdf.CrystalBall` attribute), 423
- `space` (`zfit.pdf.DoubleCB` attribute), 431
- `space` (`zfit.pdf.Exponential` attribute), 416
- `space` (`zfit.pdf.Gauss` attribute), 438
- `space` (`zfit.pdf.Hermite` attribute), 487
- `space` (`zfit.pdf.Laguerre` attribute), 494
- `space` (`zfit.pdf.Legendre` attribute), 472
- `space` (`zfit.pdf.ProductPDF` attribute), 507
- `space` (`zfit.pdf.RecursivePolynomial` attribute), 501
- `space` (`zfit.pdf.SimpleFunctorPDF` attribute), 534
- `space` (`zfit.pdf.SimplePDF` attribute), 528
- `space` (`zfit.pdf.SumPDF` attribute), 514
- `space` (`zfit.pdf.TruncatedGauss` attribute), 451
- `space` (`zfit.pdf.Uniform` attribute), 444
- `space` (`zfit.pdf.WrapDistribution` attribute), 458
- `space` (`zfit.pdf.ZPDF` attribute), 521
- `SpaceIncompatibleError`, 339
- `sparse_read()` (`zfit.core.parameter.Parameter` method), 126
- `sparse_read()` (`zfit.core.parameter.TFBaseVariable` method), 136
- `sparse_read()` (`zfit.param.Parameter` method), 393
- `sparse_read()` (`zfit.Parameter` method), 41
- `spec` (`zfit.core.parameter.Parameter.SaveSliceInfo` attribute), 117
- `spec` (`zfit.core.parameter.TFBaseVariable.SaveSliceInfo` attribute), 128
- `spec` (`zfit.param.Parameter.SaveSliceInfo` attribute), 384
- `spec` (`zfit.Parameter.SaveSliceInfo` attribute), 32
- `sqrt()` (in module `zfit.z.wrapping_tf`), 346
- `square()` (in module `zfit.z.wrapping_tf`), 346
- `stack_x()` (in module `zfit.z.zextension`), 347
- `status` (`zfit.minimizers.fitresult.FitResult` attribute), 146
- `step()` (`zfit.minimize.Adam` method), 380
- `step()` (`zfit.minimize.BFGS` method), 382
- `step()` (`zfit.minimize.Minuit` method), 381
- `step()` (`zfit.minimize.Scipy` method), 381

- `step()` (*zfit.minimize.WrapOptimizer* method), 379
- `step()` (*zfit.minimizers.base\_tf.WrapOptimizer* method), 143
- `step()` (*zfit.minimizers.baseminimizer.BaseMinimizer* method), 144
- `step()` (*zfit.minimizers.interface.ZfitMinimizer* method), 146
- `step()` (*zfit.minimizers.minimizer\_minuit.Minuit* method), 148
- `step()` (*zfit.minimizers.minimizer\_tfp.BFGS* method), 149
- `step()` (*zfit.minimizers.minimizers\_scipy.Scipy* method), 150
- `step()` (*zfit.minimizers.optimizers\_tf.Adam* method), 150
- `step_size` (*zfit.core.parameter.Parameter* attribute), 126
- `step_size` (*zfit.param.Parameter* attribute), 393
- `step_size` (*zfit.Parameter* attribute), 41
- `SubclassingError`, 339
- `SumFunc` (class in *zfit.func*), 364
- `SumFunc` (class in *zfit.models.functions*), 221
- `SumPDF` (class in *zfit.models.functor*), 245
- `SumPDF` (class in *zfit.pdf*), 508
- `supports()` (in module *zfit*), 48
- `supports()` (in module *zfit.core.limits*), 102
- `synchronization` (*zfit.core.parameter.Parameter* attribute), 126
- `synchronization` (*zfit.core.parameter.TFBaseVariable* attribute), 136
- `synchronization` (*zfit.param.Parameter* attribute), 393
- `synchronization` (*zfit.Parameter* attribute), 41
- T**
- `teardown_function()` (in module *zfit.core.testing*), 142
- `TemporarilySet` (class in *zfit.util.temporary*), 341
- `TFBaseVariable` (class in *zfit.core.parameter*), 126
- `TFBaseVariable.SaveSliceInfo` (class in *zfit.core.parameter*), 127
- `tfd_analytic_sample()` (in module *zfit.models.dist\_tfp*), 206
- `to_complex()` (in module *zfit.z.zextension*), 347
- `to_pandas()` (*zfit.core.data.Data* method), 76
- `to_pandas()` (*zfit.core.data.SampleData* method), 79
- `to_pandas()` (*zfit.core.data.Sampler* method), 82
- `to_pandas()` (*zfit.data.Data* method), 354
- `to_proto()` (*zfit.core.parameter.Parameter* method), 126
- `to_proto()` (*zfit.core.parameter.Parameter.SaveSliceInfo* method), 117
- `to_proto()` (*zfit.core.parameter.TFBaseVariable* method), 136
- `to_proto()` (*zfit.core.parameter.TFBaseVariable.SaveSliceInfo* method), 128
- `to_proto()` (*zfit.param.Parameter* method), 393
- `to_proto()` (*zfit.param.Parameter.SaveSliceInfo* method), 384
- `to_proto()` (*zfit.Parameter* method), 41
- `to_proto()` (*zfit.Parameter.SaveSliceInfo* method), 32
- `to_real()` (in module *zfit.z.zextension*), 347
- `tolerance` (*zfit.minimize.Adam* attribute), 380
- `tolerance` (*zfit.minimize.BFGS* attribute), 382
- `tolerance` (*zfit.minimize.Minuit* attribute), 381
- `tolerance` (*zfit.minimize.Scipy* attribute), 381
- `tolerance` (*zfit.minimize.WrapOptimizer* attribute), 379
- `tolerance` (*zfit.minimizers.base\_tf.WrapOptimizer* attribute), 143
- `tolerance` (*zfit.minimizers.baseminimizer.BaseMinimizer* attribute), 144
- `tolerance` (*zfit.minimizers.interface.ZfitMinimizer* attribute), 146
- `tolerance` (*zfit.minimizers.minimizer\_minuit.Minuit* attribute), 148
- `tolerance` (*zfit.minimizers.minimizer\_tfp.BFGS* attribute), 149
- `tolerance` (*zfit.minimizers.minimizers\_scipy.Scipy* attribute), 150
- `tolerance` (*zfit.minimizers.optimizers\_tf.Adam* attribute), 150
- `ToyStrategyFail` (class in *zfit.minimizers.baseminimizer*), 144
- `trainable` (*zfit.core.parameter.Parameter* attribute), 126
- `trainable` (*zfit.core.parameter.TFBaseVariable* attribute), 136
- `trainable` (*zfit.param.Parameter* attribute), 393
- `trainable` (*zfit.Parameter* attribute), 41
- `TruncatedGauss` (class in *zfit.models.dist\_tfp*), 185
- `TruncatedGauss` (class in *zfit.pdf*), 445
- U**
- `UnbinnedNLL` (class in *zfit.core.loss*), 107
- `UnbinnedNLL` (class in *zfit.loss*), 375
- `UnderdefinedError`, 339
- `Uniform` (class in *zfit.models.dist\_tfp*), 192
- `Uniform` (class in *zfit.pdf*), 438
- `UniformSampleAndWeights` (class in *zfit.core.sample*), 142
- `unnormalized_pdf()` (*zfit.core.basepdf.BasePDF* method), 67
- `unnormalized_pdf()` (*zfit.models.basic.CustomGaussOLD* method), 164
- `unnormalized_pdf()` (*zfit.models.basic.Exponential* method), 171



`unnormalized_pdf()`  
     (`zfit.models.dist_tfp.ExponentialTFP` method), 178  
`unnormalized_pdf()` (`zfit.models.dist_tfp.Gauss` method), 185  
`unnormalized_pdf()`  
     (`zfit.models.dist_tfp.TruncatedGauss` method), 192  
`unnormalized_pdf()` (`zfit.models.dist_tfp.Uniform` method), 199  
`unnormalized_pdf()`  
     (`zfit.models.dist_tfp.WrapDistribution` method), 206  
`unnormalized_pdf()`  
     (`zfit.models.functor.BaseFunctor` method), 238  
`unnormalized_pdf()`  
     (`zfit.models.functor.ProductPDF` method), 245  
`unnormalized_pdf()` (`zfit.models.functor.SumPDF` method), 252  
`unnormalized_pdf()`  
     (`zfit.models.physics.CrystalBall` method), 259  
`unnormalized_pdf()`  
     (`zfit.models.physics.DoubleCB` method), 266  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.Chebyshev` method), 274  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.Chebyshev2` method), 281  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.Hermite` method), 288  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.Laguerre` method), 295  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.Legendre` method), 303  
`unnormalized_pdf()`  
     (`zfit.models.polynomials.RecursivePolynomial` method), 309  
`unnormalized_pdf()`  
     (`zfit.models.special.SimpleFunctorPDF` method), 317  
`unnormalized_pdf()`  
     (`zfit.models.special.SimplePDF` method), 324  
`unnormalized_pdf()` (`zfit.models.special.ZPDF` method), 330  
`unnormalized_pdf()` (`zfit.pdf.BaseFunctor` method), 409  
`unnormalized_pdf()` (`zfit.pdf.BasePDF` method), 402  
`unnormalized_pdf()` (`zfit.pdf.Chebyshev` method), 465  
`unnormalized_pdf()` (`zfit.pdf.Chebyshev2` method), 480  
`unnormalized_pdf()` (`zfit.pdf.CrystalBall` method), 423  
`unnormalized_pdf()` (`zfit.pdf.DoubleCB` method), 431  
`unnormalized_pdf()` (`zfit.pdf.Exponential` method), 416  
`unnormalized_pdf()` (`zfit.pdf.Gauss` method), 438  
`unnormalized_pdf()` (`zfit.pdf.Hermite` method), 487  
`unnormalized_pdf()` (`zfit.pdf.Laguerre` method), 494  
`unnormalized_pdf()` (`zfit.pdf.Legendre` method), 472  
`unnormalized_pdf()` (`zfit.pdf.ProductPDF` method), 507  
`unnormalized_pdf()` (`zfit.pdf.RecursivePolynomial` method), 501  
`unnormalized_pdf()` (`zfit.pdf.SimpleFunctorPDF` method), 534  
`unnormalized_pdf()` (`zfit.pdf.SimplePDF` method), 528  
`unnormalized_pdf()` (`zfit.pdf.SumPDF` method), 514  
`unnormalized_pdf()` (`zfit.pdf.TruncatedGauss` method), 451  
`unnormalized_pdf()` (`zfit.pdf.Uniform` method), 444  
`unnormalized_pdf()` (`zfit.pdf.WrapDistribution` method), 458  
`unnormalized_pdf()` (`zfit.pdf.ZPDF` method), 521  
`unstack_x()` (in module `zfit.z.zextension`), 347  
`unstack_x()` (`zfit.core.data.Data` method), 76  
`unstack_x()` (`zfit.core.data.SampleData` method), 79  
`unstack_x()` (`zfit.core.data.Sampler` method), 83  
`unstack_x()` (`zfit.core.integration.PartialIntegralSampleData` method), 87  
`unstack_x()` (`zfit.data.Data` method), 354  
`update()` (`zfit.util.container.DotDict` method), 335  
`update_integration_options()`  
     (`zfit.core.basefunc.BaseFunc` method), 53  
`update_integration_options()`  
     (`zfit.core.basemodel.BaseModel` method), 58  
`update_integration_options()`  
     (`zfit.core.basepdf.BasePDF` method), 67  
`update_integration_options()`  
     (`zfit.core.interfaces.ZfitFunc` method), 90  
`update_integration_options()`

(*zfit.core.interfaces.ZfitModel method*), 93  
 update\_integration\_options()  
     (*zfit.core.interfaces.ZfitPDF method*), 95  
 update\_integration\_options()  
     (*zfit.func.BaseFunc method*), 359  
 update\_integration\_options()  
     (*zfit.func.ProdFunc method*), 364  
 update\_integration\_options()  
     (*zfit.func.SimpleFunc method*), 374  
 update\_integration\_options()  
     (*zfit.func.SumFunc method*), 369  
 update\_integration\_options()  
     (*zfit.models.basefunctor.FunctorMixin method*), 158  
 update\_integration\_options()  
     (*zfit.models.basic.CustomGaussOLD method*), 165  
 update\_integration\_options()  
     (*zfit.models.basic.Exponential method*), 172  
 update\_integration\_options()  
     (*zfit.models.dist\_tfp.ExponentialTFP method*), 178  
 update\_integration\_options()  
     (*zfit.models.dist\_tfp.Gauss method*), 185  
 update\_integration\_options()  
     (*zfit.models.dist\_tfp.TruncatedGauss method*), 192  
 update\_integration\_options()  
     (*zfit.models.dist\_tfp.Uniform method*), 199  
 update\_integration\_options()  
     (*zfit.models.dist\_tfp.WrapDistribution method*), 206  
 update\_integration\_options()  
     (*zfit.models.functions.BaseFunctorFunc method*), 211  
 update\_integration\_options()  
     (*zfit.models.functions.ProdFunc method*), 216  
 update\_integration\_options()  
     (*zfit.models.functions.SimpleFunc method*), 221  
 update\_integration\_options()  
     (*zfit.models.functions.SumFunc method*), 226  
 update\_integration\_options()  
     (*zfit.models.functions.ZFunc method*), 231  
 update\_integration\_options()  
     (*zfit.models.functor.BaseFunctor method*), 238  
 update\_integration\_options()  
     (*zfit.models.functor.ProductPDF method*), 245  
 update\_integration\_options()  
     (*zfit.models.functor.SumPDF method*), 252  
 update\_integration\_options()  
     (*zfit.models.physics.CrystalBall method*), 259  
 update\_integration\_options()  
     (*zfit.models.physics.DoubleCB method*), 267  
 update\_integration\_options()  
     (*zfit.models.polynomials.Chebyshev method*), 274  
 update\_integration\_options()  
     (*zfit.models.polynomials.Chebyshev2 method*), 281  
 update\_integration\_options()  
     (*zfit.models.polynomials.Hermite method*), 289  
 update\_integration\_options()  
     (*zfit.models.polynomials.Laguerre method*), 296  
 update\_integration\_options()  
     (*zfit.models.polynomials.Legendre method*), 303  
 update\_integration\_options()  
     (*zfit.models.polynomials.RecursivePolynomial method*), 310  
 update\_integration\_options()  
     (*zfit.models.special.SimpleFunctorPDF method*), 318  
 update\_integration\_options()  
     (*zfit.models.special.SimplePDF method*), 324  
 update\_integration\_options()  
     (*zfit.models.special.ZPDF method*), 331  
 update\_integration\_options()  
     (*zfit.pdf.BaseFunctor method*), 409  
 update\_integration\_options()  
     (*zfit.pdf.BasePDF method*), 403  
 update\_integration\_options()  
     (*zfit.pdf.Chebyshev method*), 466  
 update\_integration\_options()  
     (*zfit.pdf.Chebyshev2 method*), 480  
 update\_integration\_options()  
     (*zfit.pdf.CrystalBall method*), 424  
 update\_integration\_options()  
     (*zfit.pdf.DoubleCB method*), 431  
 update\_integration\_options()  
     (*zfit.pdf.Exponential method*), 416  
 update\_integration\_options()  
     (*zfit.pdf.Gauss method*), 438  
 update\_integration\_options()  
     (*zfit.pdf.Hermite method*), 487  
 update\_integration\_options()  
     (*zfit.pdf.Laguerre method*), 494  
 update\_integration\_options()  
     (*zfit.pdf.Legendre method*), 473

`update_integration_options()`  
     (*zfit.pdf.ProductPDF method*), 508  
`update_integration_options()`  
     (*zfit.pdf.RecursivePolynomial method*), 501  
`update_integration_options()`  
     (*zfit.pdf.SimpleFunctorPDF method*), 535  
`update_integration_options()`  
     (*zfit.pdf.SimplePDF method*), 528  
`update_integration_options()`  
     (*zfit.pdf.SumPDF method*), 515  
`update_integration_options()`  
     (*zfit.pdf.TruncatedGauss method*), 452  
`update_integration_options()`  
     (*zfit.pdf.Uniform method*), 445  
`update_integration_options()`  
     (*zfit.pdf.WrapDistribution method*), 458  
`update_integration_options()` (*zfit.pdf.ZPDF method*), 521  
`upper` (*zfit.core.interfaces.ZfitSpace attribute*), 96  
`upper` (*zfit.core.limits.Space attribute*), 101  
`upper` (*zfit.core.sample.EventSpace attribute*), 141  
`upper` (*zfit.Space attribute*), 47  
`upper_limit` (*zfit.core.parameter.Parameter attribute*), 126  
`upper_limit` (*zfit.param.Parameter attribute*), 393  
`upper_limit` (*zfit.Parameter attribute*), 41

## V

`value()` (*zfit.ComplexParameter method*), 43  
`value()` (*zfit.ComposedParameter method*), 42  
`value()` (*zfit.constraint.GaussianConstraint method*), 351  
`value()` (*zfit.constraint.SimpleConstraint method*), 349  
`value()` (*zfit.core.constraint.BaseConstraint method*), 69  
`value()` (*zfit.core.constraint.DistributionConstraint method*), 70  
`value()` (*zfit.core.constraint.GaussianConstraint method*), 71  
`value()` (*zfit.core.constraint.SimpleConstraint method*), 73  
`value()` (*zfit.core.data.Data method*), 76  
`value()` (*zfit.core.data.LightDataset method*), 76  
`value()` (*zfit.core.data.SampleData method*), 79  
`value()` (*zfit.core.data.Sampler method*), 83  
`value()` (*zfit.core.integration.PartialIntegralSampleData method*), 87  
`value()` (*zfit.core.interfaces.ZfitConstraint method*), 88  
`value()` (*zfit.core.interfaces.ZfitData method*), 88  
`value()` (*zfit.core.interfaces.ZfitLoss method*), 91  
`value()` (*zfit.core.interfaces.ZfitParameter method*), 95  
`value()` (*zfit.core.loss.BaseLoss method*), 104  
`value()` (*zfit.core.loss.CachedLoss method*), 105  
`value()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 106  
`value()` (*zfit.core.loss.SimpleLoss method*), 107  
`value()` (*zfit.core.loss.UnbinnedNLL method*), 108  
`value()` (*zfit.core.parameter.BaseComposedParameter method*), 111  
`value()` (*zfit.core.parameter.BaseParameter method*), 111  
`value()` (*zfit.core.parameter.BaseZParameter method*), 112  
`value()` (*zfit.core.parameter.ComplexParameter method*), 113  
`value()` (*zfit.core.parameter.ComposedParameter method*), 115  
`value()` (*zfit.core.parameter.ComposedVariable method*), 115  
`value()` (*zfit.core.parameter.ConstantParameter method*), 116  
`value()` (*zfit.core.parameter.Parameter method*), 126  
`value()` (*zfit.core.parameter.TFBaseVariable method*), 136  
`value()` (*zfit.core.parameter.ZfitBaseVariable method*), 136  
`value()` (*zfit.data.Data method*), 354  
`value()` (*zfit.loss.BaseLoss method*), 377  
`value()` (*zfit.loss.ExtendedUnbinnedNLL method*), 375  
`value()` (*zfit.loss.SimpleLoss method*), 379  
`value()` (*zfit.loss.UnbinnedNLL method*), 376  
`value()` (*zfit.param.ComplexParameter method*), 396  
`value()` (*zfit.param.ComposedParameter method*), 395  
`value()` (*zfit.param.ConstantParameter method*), 383  
`value()` (*zfit.param.Parameter method*), 393  
`value()` (*zfit.Parameter method*), 41  
`value_gradients()` (*zfit.core.loss.BaseLoss method*), 104  
`value_gradients()` (*zfit.core.loss.CachedLoss method*), 105  
`value_gradients()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 106  
`value_gradients()` (*zfit.core.loss.SimpleLoss method*), 107  
`value_gradients()` (*zfit.core.loss.UnbinnedNLL method*), 108  
`value_gradients()` (*zfit.loss.BaseLoss method*), 377  
`value_gradients()` (*zfit.loss.ExtendedUnbinnedNLL method*), 375  
`value_gradients()` (*zfit.loss.SimpleLoss method*), 379  
`value_gradients()` (*zfit.loss.UnbinnedNLL method*), 376  
`value_gradients_hessian()`

[\(zfit.core.loss.BaseLoss method\), 104](#)  
[value\\_gradients\\_hessian\(\) \(zfit.core.loss.CachedLoss method\), 105](#)  
[value\\_gradients\\_hessian\(\) \(zfit.core.loss.ExtendedUnbinnedNLL method\), 106](#)  
[value\\_gradients\\_hessian\(\) \(zfit.core.loss.SimpleLoss method\), 107](#)  
[value\\_gradients\\_hessian\(\) \(zfit.core.loss.UnbinnedNLL method\), 108](#)  
[value\\_gradients\\_hessian\(\) \(zfit.loss.BaseLoss method\), 377](#)  
[value\\_gradients\\_hessian\(\) \(zfit.loss.ExtendedUnbinnedNLL method\), 375](#)  
[value\\_gradients\\_hessian\(\) \(zfit.loss.SimpleLoss method\), 379](#)  
[value\\_gradients\\_hessian\(\) \(zfit.loss.UnbinnedNLL method\), 376](#)  
[values\(\) \(zfit.util.container.DotDict method\), 335](#)

## W

[weights \(zfit.core.data.Data attribute\), 76](#)  
[weights \(zfit.core.data.SampleData attribute\), 79](#)  
[weights \(zfit.core.data.Sampler attribute\), 83](#)  
[weights \(zfit.core.integration.PartialIntegralSampleData attribute\), 87](#)  
[weights \(zfit.core.interfaces.ZfitData attribute\), 88](#)  
[weights \(zfit.data.Data attribute\), 354](#)  
[WeightsNotImplementedError, 339](#)  
[with\\_autofill\\_axes\(\) \(zfit.core.interfaces.ZfitSpace method\), 96](#)  
[with\\_autofill\\_axes\(\) \(zfit.core.limits.Space method\), 101](#)  
[with\\_autofill\\_axes\(\) \(zfit.core.sample.EventSpace method\), 141](#)  
[with\\_autofill\\_axes\(\) \(zfit.Space method\), 47](#)  
[with\\_axes\(\) \(zfit.core.interfaces.ZfitSpace method\), 96](#)  
[with\\_axes\(\) \(zfit.core.limits.Space method\), 101](#)  
[with\\_axes\(\) \(zfit.core.sample.EventSpace method\), 141](#)  
[with\\_axes\(\) \(zfit.Space method\), 47](#)  
[with\\_limits\(\) \(zfit.core.interfaces.ZfitSpace method\), 96](#)  
[with\\_limits\(\) \(zfit.core.limits.Space method\), 101](#)  
[with\\_limits\(\) \(zfit.core.sample.EventSpace method\), 141](#)  
[with\\_limits\(\) \(zfit.Space method\), 47](#)  
[with\\_obs\(\) \(zfit.core.interfaces.ZfitSpace method\), 96](#)  
[with\\_obs\(\) \(zfit.core.limits.Space method\), 101](#)  
[with\\_obs\(\) \(zfit.core.sample.EventSpace method\), 142](#)  
[with\\_obs\(\) \(zfit.Space method\), 48](#)

[with\\_obs\\_axes\(\) \(zfit.core.limits.Space method\), 101](#)  
[with\\_obs\\_axes\(\) \(zfit.core.sample.EventSpace method\), 142](#)  
[with\\_obs\\_axes\(\) \(zfit.Space method\), 48](#)  
[with\\_traceback\(\) \(zfit.minimizers.baseminimizer.FailMinimizeNaN method\), 144](#)  
[with\\_traceback\(\) \(zfit.util.exception.AlreadyExtendedPDFError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.AxesNotSpecifiedError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.AxesNotUnambiguousError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.BasePDFSubclassingError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.BreakingAPIChangeError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.ConversionError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.ExtendedPDFError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.IncompatibleError method\), 336](#)  
[with\\_traceback\(\) \(zfit.util.exception.IntentionNotUnambiguousError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.LimitsIncompatibleError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.LimitsNotSpecifiedError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.LimitsOverdefinedError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.LimitsUnderdefinedError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.LogicalUndefinedOperationError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.ModelIncompatibleError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.MultipleLimitsNotImplementedError method\), 337](#)  
[with\\_traceback\(\) \(zfit.util.exception.NameAlreadyTakenError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NormRangeNotImplementedError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NormRangeNotSpecifiedError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NoSessionSpecifiedError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NotExtendedPDFError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NotMinimizedError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.NotSpecifiedError method\), 338](#)  
[with\\_traceback\(\) \(zfit.util.exception.ObsIncompatibleError](#)



*method*), 338  
 with\_traceback() (*zfit.util.exception.ObsNotSpecifiedError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.OverdefinedError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.PDFCompatibilityError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.ShapeIncompatibleError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.SpaceIncompatibleError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.SubclassingError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.UnderdefinedError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.WeightsNotImplementedError*, *method*), 339  
 with\_traceback() (*zfit.util.exception.WorkInProgressError*, *method*), 340  
 WorkInProgressError, 339  
 WrapDistribution (class in *zfit.models.dist\_tfp*), 199  
 WrapDistribution (class in *zfit.pdf*), 452  
 WrapOptimizer (class in *zfit.minimize*), 379  
 WrapOptimizer (class in *zfit.minimizers.base\_tf*), 143  
 wrapped\_functions  
     (*zfit.z.zextension.FunctionWrapperRegistry* attribute), 346

**Z**

zfit (module), 31  
 zfit.constraint (module), 348  
 zfit.core (module), 49  
 zfit.core.basefunc (module), 49  
 zfit.core.basemodel (module), 54  
 zfit.core.baseobject (module), 59  
 zfit.core.basepdf (module), 60  
 zfit.core.constraint (module), 67  
 zfit.core.data (module), 73  
 zfit.core.dependents (module), 83  
 zfit.core.dimension (module), 83  
 zfit.core.integration (module), 85  
 zfit.core.interfaces (module), 88  
 zfit.core.limits (module), 97  
 zfit.core.loss (module), 103  
 zfit.core.operations (module), 108  
 zfit.core.parameter (module), 109  
 zfit.core.sample (module), 138  
 zfit.core.testing (module), 142  
 zfit.data (module), 351  
 zfit.func (module), 354  
 zfit.loss (module), 374  
 zfit.minimize (module), 379  
 zfit.minimizers (module), 143  
 zfit.minimizers.base\_tf (module), 143  
 zfit.minimizers.baseminimizer (module), 143  
 zfit.minimizers.fitresult (module), 144  
 zfit.minimizers.interface (module), 146  
 zfit.minimizers.minimizer\_minuit (module), 147  
 zfit.minimizers.minimizer\_tfp (module), 149  
 zfit.minimizers.minimizers\_scipy (module), 149  
 zfit.minimizers.optimizers\_tf (module), 150  
 zfit.minimizers.tf\_external\_optimizer (module), 150  
 zfit.models (module), 154  
 zfit.models.basefunc (module), 154  
 zfit.models.basic (module), 158  
 zfit.models.dist\_tfp (module), 172  
 zfit.models.functions (module), 206  
 zfit.models.functor (module), 232  
 zfit.models.physics (module), 252  
 zfit.models.polynomials (module), 267  
 zfit.models.special (module), 311  
 zfit.param (module), 382  
 zfit.pdf (module), 396  
 zfit.sample (module), 535  
 zfit.settings (module), 535  
 zfit.util (module), 331  
 zfit.util.cache (module), 331  
 zfit.util.checks (module), 334  
 zfit.util.container (module), 334  
 zfit.util.diverse (module), 335  
 zfit.util.exception (module), 336  
 zfit.util.execution (module), 340  
 zfit.util.graph (module), 340  
 zfit.util.logging (module), 341  
 zfit.util.temporary (module), 341  
 zfit.util.ztyping (module), 342  
 zfit.z (module), 342  
 zfit.z.const (module), 342  
 zfit.z.math (module), 343  
 zfit.z.random (module), 345  
 zfit.z.tools (module), 345  
 zfit.z.wrapping\_tf (module), 345  
 zfit.z.zextension (module), 346  
 ZfitBaseVariable (class in *zfit.core.parameter*), 136  
 ZfitCachable (class in *zfit.util.cache*), 334  
 ZfitConstraint (class in *zfit.core.interfaces*), 88  
 ZfitData (class in *zfit.core.interfaces*), 88  
 ZfitDependentsMixin (class in *zfit.core.interfaces*), 88  
 ZfitDimensional (class in *zfit.core.interfaces*), 88

[ZfitFunc \(class in `zfit.core.interfaces`\), 89](#)  
[ZfitFunctorMixin \(class in `zfit.core.interfaces`\), 90](#)  
[ZfitLoss \(class in `zfit.core.interfaces`\), 91](#)  
[ZfitMinimizer \(class in `zfit.minimizers.interface`\),  
\[146\]\(#\)](#)  
[ZfitModel \(class in `zfit.core.interfaces`\), 91](#)  
[ZfitNotImplemented \(class in `zfit.util.checks`\), 334](#)  
[ZfitNumeric \(class in `zfit.core.interfaces`\), 93](#)  
[ZfitObject \(class in `zfit.core.interfaces`\), 93](#)  
[ZfitParameter \(class in `zfit.core.interfaces`\), 95](#)  
[ZfitParameterMixin \(class in `zfit.core.parameter`\),  
\[136\]\(#\)](#)  
[ZfitPDF \(class in `zfit.core.interfaces`\), 93](#)  
[ZfitResult \(class in `zfit.minimizers.interface`\), 146](#)  
[ZfitSpace \(class in `zfit.core.interfaces`\), 95](#)  
[ZfitStrategy \(class in `zfit.minimizers.baseminimizer`\), 144](#)  
[ZFunc \(class in `zfit.models.functions`\), 226](#)  
[ZPDF \(class in `zfit.models.special`\), 324](#)  
[ZPDF \(class in `zfit.pdf`\), 515](#)