
zfit Documentation

Release 0.3.7

zfit

Dec 08, 2019

Contents

1	Getting started with zfit	3
1.1	What did just happen?	6
2	Downloading and Installation	7
2.1	Prerequisites	7
2.2	Downloads	7
2.3	Installation	8
2.4	Testing	8
2.5	Getting help	8
2.6	Acknowledgements	8
2.7	License	8
3	Contributing	11
3.1	Get Started!	11
3.2	Pull Request Guidelines	12
4	Space, Observable and Range	13
4.1	Definitions	13
4.2	Limits	14
5	Parameter	17
5.1	Independent Parameter	17
5.2	Dependent Parameter	18
6	Building a model	19
6.1	Predefined PDFs and basic properties	19
6.2	Composite PDF	20
6.3	Extended PDF	21
6.4	Custom PDF	21
7	Data	25
7.1	Import dataset from a ROOT file	25
7.2	Import dataset from a pandas DataFrame or Numpy ndarray	26
8	Loss	27
8.1	Adding constraints	27
8.2	Simultaneous fits	28

9	Minimization	29
9.1	Baseline minimizers	29
10	zfit API documentation	31
10.1	zfit package	31
	Python Module Index	585
	Index	587

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

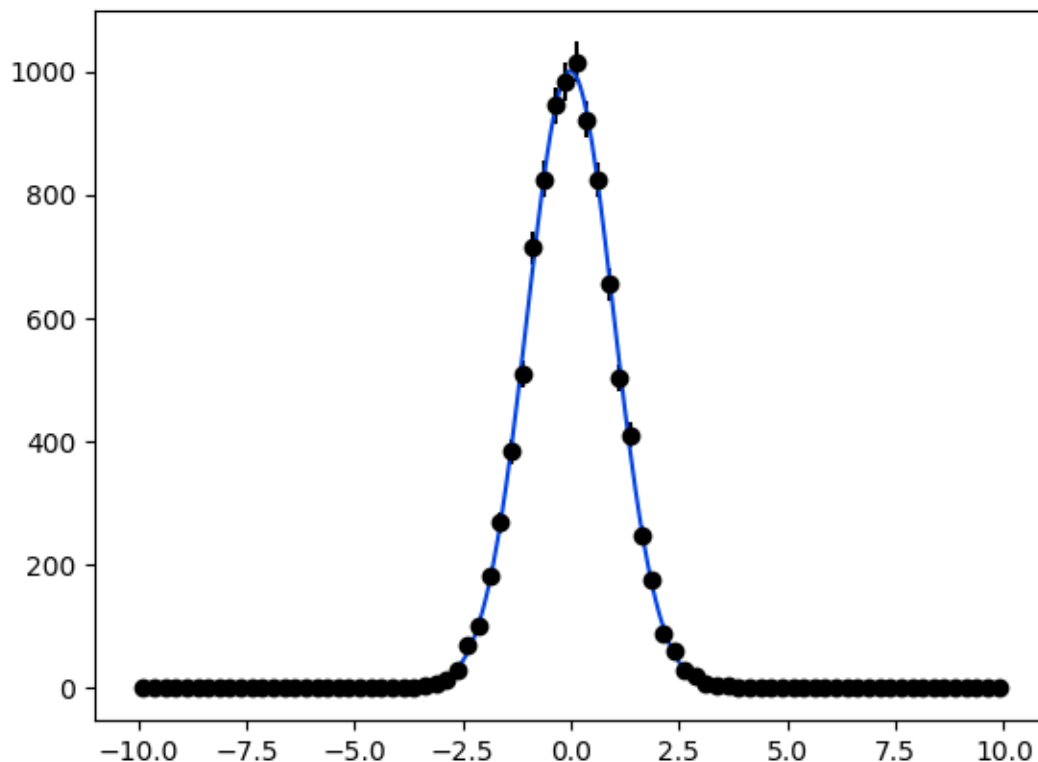
>>> # Printing information on specific parameters, e.g. mu
>>> print("mu={}".format(params[mu]['value']))
mu=0.012464509810750313
```

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

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

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

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

Downloading and Installation

2.1 Prerequisites

`zfit` works with Python versions 3.6 and 3.7. The following packages are required:

- `tensorflow >= 1.10.0`
- `tensorflow_probability >= 0.3.0`
- `scipy >=1.2`
- `numpy`
- `uproot`
- `iminuit`
- `typing`
- `colorlog`
- `texttable`

All of these are readily available on PyPI, and should be installed automatically if installing with `pip install zfit`.

In order to run the test suite, the `pytest` package is required

2.2 Downloads

The latest beta version is 0.3.0 and is available from *PyPi* <<https://pypi.org/project/zfit/>>.

2.3 Installation

The easiest way to install zfit is with

```
pip install zfit
```

To get the latest development version, use:

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

and install using:

```
python setup.py install
```

2.4 Testing

A battery of tests scripts that can be run with the `pytest` testing framework is distributed with zfit in the `tests` folder. These are automatically run as part of the development process. For any release or any master branch from the git repository, running `pytest` should run all of these tests to completion without errors or failures.

2.5 Getting help

If you have questions, comments, or suggestions for zfit, please drop a line in our [Gitter channel](#). If you find a bug in the code or documentation, open a [Github issue](#) and submit a report. If you have an idea for how to solve the problem and are familiar with Python and GitHub, submitting a GitHub Pull Request would be greatly appreciated. If you are unsure whether to use the Gitter channel or the Issue tracker, please start a conversation in the Gitter channel.

2.6 Acknowledgements

zfit has been developed with support from the University of Zürich and the Swiss National Science Foundation (SNSF) under contracts 168169 and 174182.

The idea of zfit is inspired by the [TensorFlowAnalysis](#) framework developed by Anton Poluektov using the TensorFlow open source library.

2.7 License

The zfit code is distributed under the BSD-3-Clause License:

Copyright (c) 2018, zfit All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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](#).

3.1 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 Pull Request Guidelines

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

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

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, ...)
```

4.1 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, ...)
```

(continues on next page)

(continued from previous page)

```
# 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.

4.2 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*.

4.2.1 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)

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.

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.

5.1 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
...     five_mu_val = zfit.run(five_mu) # is evaluated with mu = 5. -> five_mu_val is_
↪15
```

(continues on next page)

(continued from previous page)

```
>>> # 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
```

5.2 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_tensor = mu * 5 + mu2 # or any kind of computation
>>> dep_param = zfit.ComposedParameter("dependent_param", dependent_tensor)

>>> dependents = dep_param.get_dependents() # returns 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.

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`:

6.1 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', 'Exponential', 'CrystalBall', 'Gauss', 'Uniform', 'WrapDistribution',
↪ 'ProductPDF', 'SumPDF', 'ZPDF', 'SimplePDF', 'SimpleFuncPDF']
```

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)
```

(continues on next page)

(continued from previous page)

```
>>> # 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), norm_range=(5100, 5400))
>>> # 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.

6.2 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')]
```

6.3 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
```

6.4 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)
```

6.4.1 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.

6.4.2 Tensor sampling

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

6.4.3 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
```

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).

7.1 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.

7.2 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)
```

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 and the fit range (which internally sets `norm_range` in the PDF and makes sure data only within that range are used) **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,
>>>                                fit_range=(-10, 10))
```

8.1 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,
```

(continues on next page)

(continued from previous page)

```
>>> fit_range=(-10, 10),
>>> 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.

8.2 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, ...])
```

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.

9.1 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.train.AdamOptimizer())
```

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)
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
```

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!

10.1 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.util.execution.SessionHolderMixin, zfit.core.parameter.ZfitParameterMixin, zfit.core.parameter.TFBaseVariable, zfit.core.parameter.BaseParameter
```

Class for fit parameters, derived from TF Variable class.

Constructor. 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.

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 *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:

$$\text{num_prefix_dims} = \text{sparse_delta.indices.ndims} - 1 \quad \text{batch_dim} = \text{num_prefix_dims} + 1$$

```
*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** – *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 *ValueError* – 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.

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

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

Returns

Return type list(ZfitParameters)

get_shape ()

Alias of *Variable.shape*.

graph

The *Graph* of this variable.

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*: Union[int, float, complex, tensorflow.python.framework.ops.Tensor])

Parameter takes on the *value*. Is not part of the graph, does a session run.

Parameters *value* (*numerical*) –

lower_limit

name

The name of the object.

numpy ()

op

The op for this variable.

params

randomize (*minval=None, maxval=None, sampler=<built-in method uniform of 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 *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 subtraction has completed.

Raises *ValueError* – 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 $Q-I+P-K$ 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.

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

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 $Q-I+P-K$ 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.

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

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.

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

scatter_sub (*sparse_delta*, *use_locking=False*, *name=None*)

Subtracts *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 `ValueError` – if *sparse_delta* is not an *IndexedSlices*.

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

Parameters

- **sparse_delta** – *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 `ValueError` – if *sparse_delta* is not an *IndexedSlices*.

sess

set_sess (*sess*: *tensorflow.python.client.session.Session*)
Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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*, *tensor*, *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)

independent

load (*value*, *session*=None)

name

The name of the object.

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.ComplexParameter (name, value, 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\(ZfitParameters\)

    imag
    independent
    load (value, session=None)
    mod
    name
        The name of the object.
    params
    read_value ()

```

real

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.

value ()

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

Convert a *numerical* to a fixed 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[float]]]

Return the limits, either as *Space* objects or as pure limits-tuple.

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

Example

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

Returns

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

limit1d

return the tuple(lower, upper).

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

Return type tuple(float, float)

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

Type Simplified limits getter for 1 obs, 1 limit only

limit2d

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

Returns

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

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

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

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

limits

Return the limits.

Returns:

limits1d

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

Returns

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

Return type `tuple(float, float, ...)`

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

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

lower

Return the lower limits.

Returns:

n_limits

The number of different limits.

Returns `int >= 1`

n_obs

Return the number of observables/axes.

Returns `int >= 1`

name

The name of the object.

obs

The observables (“axes with str”)the space is defined in.

Returns:

obs_axes

reorder_by_indices (*indices: Tuple[int]*)

Return a *Space* reordered by the indices.

Parameters `()` (*indices*) –

upper

Return the upper limits.

Returns:

with_autofill_axes (*overwrite: bool = False*) → *zfit.Space*

Return a *Space* with filled axes corresponding to `range(len(n_obs))`.

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

Returns *Space*

with_axes (*axes*: Union[int, Iterable[int]]) → zfit.Space

Sort by *obs* and return the new instance.

Parameters () (*axes*) –

Returns *Space*

with_limits (*limits*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool], *name*: Optional[str] = None) → zfit.Space

Return a copy of the space with the new *limits* (and the new *name*).

Parameters

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

Returns *Space*

with_obs (*obs*: Union[str, Iterable[str], zfit.Space]) → zfit.Space

Sort by *obs* and return the new instance.

Parameters () (*obs*) –

Returns *Space*

with_obs_axes (*obs_axes*: Union[OrderedDict[str, int], Dict[str, int]], *ordered*: bool = False, *allow_subset*=False) → zfit.Space

Return a new *Space* with reordered observables and set the *axes*.

Parameters

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

Returns

Return type *Space*

zfit.convert_to_space (*obs*: Union[str, Iterable[str], zfit.Space, None] = None, *axes*: Union[int, Iterable[int], None] = None, *limits*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool, None] = None, *, *overwrite_limits*: bool = False, *one_dim_limits_only*: bool = True, *simple_limits_only*: bool = True) → Union[None, zfit.core.limits.Space, bool]

Convert *limits* to a *Space* object if not already None or False.

Parameters

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

Returns

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

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

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

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

10.1.1 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, 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'`) \rightarrow `Union[float, tensorflow.python.framework.ops.Tensor]`

The function evaluated at x .

Parameters

- \mathbf{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`)

integrate ($limits$: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, $norm_range$: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, $name$: `str = 'integrate'`) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class'tf.Tensor'`: the integral value as a scalar with shape $()$

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*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.

space

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

update_integration_options (*draws_per_dim=None, mc_sampler=None*)

Set the integration options.

Parameters

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

basemodel

Baseclass for a Model. Handle integration and sampling

```
class zfit.core.basemodel.BaseModel (obs: Union[str, Iterable[str], zfit.Space], params:
    Optional[Dict[str, zfit.core.interfaces.ZfitParameter]] =
    None, name: str = 'BaseModel', dtype=tf.float64,
    **kwargs)
```

Bases: *zfit.core.baseobject.BaseNumeric, zfit.util.cache.Cachable, zfit.core.dimension.BaseDimensional, zfit.core.interfaces.ZfitModel*

Base class for any generic model.

TODO instructions on how to use

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

Parameters

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

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

- **limits** (*tuple, Space*) – the limits to integrate over

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

Returns the integral value

Return type Tensor

Raises

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

axes

Return the axes.

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

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

Parameters

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

Returns:

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

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

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

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

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

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

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

Parameters

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

Returns

Return type `list(ZfitParameters)`

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

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class'tf.Tensor'`: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (`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`) \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.

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 = ztf.unstack_x(x)
        return ztf.exp(-ztf.square((x - mu) / sigma))
```

baseobject

Baseclass for most objects appearing in zfit.

class zfit.core.baseobject.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`

class zfit.core.baseobject.BaseNumeric (*name, dtype, params, **kwargs*)

Bases: `zfit.util.cache.Cachable`, `zfit.core.baseobject.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)

name

The name of the object.

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,
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_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

- (*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)

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) – name of the operation shown in the tf.Graph

Returns py:class'tf.Tensor': the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = `None`, *name*: *str* = `'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
 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 tf.Tensor of type *self.dtype*.

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 `()` (*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*) –

constraint

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

Bases: `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)`

name

The name of the object.

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.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)

name

The name of the object.

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.**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)`

name

The name of the object.

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()

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

The name of the object.

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** `()`**data**

```
class zfit.core.data.Data (dataset: Union[tensorflow.python.data.ops.dataset_ops.DatasetV1, 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.execution.SessionHolderMixin, 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,
                      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()
```

```
initialize()
```

```
iterator
```

```
n_obs
```

Return the number of observables.

```
name
```

The name of the object.

```
nevents
```


obs
Return the observables.

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.

sess

set_data_range (*data_range*)

set_sess (*sess*: *tensorflow.python.client.session.Session*)
Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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.DatasetV1,
                                             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: tensor-
    flow.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: tensor-
    flow.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** ()**initialize** ()**iterator****n_obs**

Return the number of observables.

name

The name of the object.

nevents**obs**

Return the observables.

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.

sess**set_data_range** (*data_range*)**set_sess** (*sess: tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –**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: Union[tensorflow.python.data.ops.dataset_ops.DatasetV1,
                                           LightDataset], sample_holder: tensorflow.python.ops.variables.VariableV1,
                                           n_holder: tensorflow.python.ops.variables.VariableV1,
                                           weights=None, fixed_params: Dict[zfit.Parameter, Union[int, float, complex,
                                           tensorflow.python.framework.ops.Tensor]] = None, obs: Union[str,
                                           Iterable[str], zfit.Space] = None, name: str = None, dtype:
                                           tensorflow.python.framework.dtypes.DType = tf.float64)
```

Bases: `zfit.core.data.Data`

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

axes

Return the axes.

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

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

Parameters

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

Returns:

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

data_range

dtype

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

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

Parameters

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

Returns

Return type *zfit.Data*

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

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

Parameters

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

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

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

Parameters

- **path** (*str*) –
- **treepath** (*str*) –
- **branches** (*List[str]*) –
- **branches_alias** (*dict*) – A mapping from the *branches* (as keys) to the actual *observables* (as values). This allows to have different *observable* names, independent of the branch name in the file.
- **weights** (*tf.Tensor*, *None*, *np.ndarray*, *str*) – Weights of the data. Has to be 1-D and match the shape of the data (nevents). Can be a column of the ROOT file by using a string corresponding to a column.
- **name** (*str*) –
- **()** (*root_dir_options*) –

Returns

Return type `zfit.Data`

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

classmethod `from_sample` (*sample: tensorflow.python.framework.ops.Tensor*, *n_holder: tensorflow.python.ops.variables.VariableV1*, *obs: Union[str, Iterable[str], zfit.Space]*, *fixed_params=None*, *name: str = None*, *weights=None*)

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

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

Parameters

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

Returns

Return type `zfit.core.Data`

classmethod `get_cache_counting` ()

`get_iteration` ()

`initialize` ()

`iterator`

n_obs

Return the number of observables.

n_samples

name

The name of the object.

nevents

obs

Return the observables.

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*) –

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.

sess

set_data_range (*data_range*)

set_sess (*sess: tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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)`

dimension

class `zfit.core.dimension.BaseDimensional`

Bases: `zfit.core.interfaces.ZfitDimensional`

axes

Return the axes.

copy (*deep: bool = False, **overwrite_params*) → `zfit.core.interfaces.ZfitObject`

n_obs

Return the number of observables.

name

Name prepended to all ops created by this *model*.

obs

Return the observables.

space

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

`zfit.core.dimension.add_spaces(spaces: Iterable[zfit.Space])`

Add two spaces and merge their limits if possible or return False.

Parameters *spaces* (Iterable[*Space*]) –

Returns

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

Raises *LimitsIncompatibleError* – if limits of the *spaces* cannot be merged because they overlap

`zfit.core.dimension.combine_spaces(spaces: Iterable[zfit.Space])`

Combine spaces with different *obs* and *limits* to one *space*.

Checks if the limits in each obs coincide *exactly*. If this is not the case, the combination is not unambiguous and *False* is returned

Parameters *spaces* (List[*Space*]) –

Returns

Returns False if the limits don't coincide in one or more obs. Otherwise return the *Space* with all obs from *spaces* sorted by the order of *spaces* and with the combined limits.

Return type *zfit.Space* or False

Raises

- *ValueError* – if only one space is given
- *LimitsIncompatibleError* – If the limits of one or more spaces (or within a space) overlap
- *LimitsNotSpecifiedError* – If the limits for one or more obs but not all are None.

`zfit.core.dimension.common_obs(spaces: Union[zfit.Space, Iterable[zfit.Space]]) → List[str]`

Extract the union of *obs* from *spaces* in the order of *spaces*.

For example:

```
space1.obs: ['obs1', 'obs3']
space2.obs: ['obs2', 'obs3', 'obs1']
space3.obs: ['obs2']
returns ['obs1', 'obs3', 'obs2']
```

Parameters *()* (*spaces*) – :py:class:`~zfit.Space`'s to extract the obs from

Returns The observables as *str*

Return type List[str]

```
zfit.core.dimension.get_same_obs
```

```
zfit.core.dimension.is_combinable(spaces)
```

```
zfit.core.dimension.limits_consistent(spaces: Iterable[zfit.Space])
```

Check if space limits are the *exact* same in each obs they are defined and therefore are compatible.

In this case, if a space has several limits, e.g. from -1 to 1 and from 2 to 3 (all in the same observable), to be consistent with this limits, other limits have to have (in this obs) also the limits from -1 to 1 and from 2 to 3. Only having the limit -1 to 1 *_or_* 2 to 3 is considered *_not_* consistent.

This function is useful to check if several spaces with *different* observables can be *_combined_*.

Parameters `spaces` (`List[zfit.Space]`) –

Returns

Return type `bool`

```
zfit.core.dimension.limits_overlap(spaces: Union[zfit.Space, Iterable[zfit.Space]], allow_exact_match: bool = False) → bool
```

Check if *_any_* of the limits of *spaces* overlaps with *_any_* other of *spaces*.

This also checks multiple limits within one space. If *allow_exact_match* is set to true, then an *exact* overlap of limits is allowed.

Parameters

- **spaces** (`Iterable[zfit.Space]`) –
- **allow_exact_match** (`bool`) – An exact overlap of two limits is counted as “not overlapping”. Example: limits from -1 to 3 and 4 to 5 to *NOT* overlap with the limits 4 to 5 iff *allow_exact_match* is True.

Returns if there are overlapping limits.

Return type `bool`

integration

This module contains functions for the numeric as well as the analytic (partial) integration.

```
class zfit.core.integration.AnalyticIntegral(*args, **kwargs)
```

Bases: `object`

Hold analytic integrals and manage their dimensions, limits etc.

```
get_max_axes(limits: Union[Tuple[Tuple[float, ...], Tuple[float, ...], bool], axes: Union[int, Iterable[int]] = None) → Tuple[int]
```

Return the maximal available axes to integrate over analytically for given limits

Parameters

- **limits** (`Space`) – The integral function will be able to integrate over this limits
- **axes** (`tuple`) – The axes over which (or over a subset) it will integrate

Returns

Return type `Tuple[int]`

```
get_max_integral(limits: Union[Tuple[Tuple[float, ...], Tuple[float, ...], bool], axes: Union[int, Iterable[int]] = None) → Union[None, zfit.core.integration.Integral]
```

Return the integral over the *limits* with *axes* (or a subset of them).

Parameters

- **limits** (*Space*) –
- **axes** (*Tuple[int]*) –

Returns Return a callable that integrated over the given limits.

Return type Union[None, *Integral*]

integrate (*x*: Union[float, tensorflow.python.framework.ops.Tensor, None], *limits*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], *axes*: Union[int, Iterable[int]] = None, *norm_range*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, *model*: zfit.core.interfaces.ZfitModel = None, *params*: dict = None) → Union[float, tensorflow.python.framework.ops.Tensor]
Integrate analytically over the axes if available.

Parameters

- **x** (*numeric*) – If a partial integration is made, x are the value to be evaluated for the partial integrated function. If a full integration is performed, this should be None.
- **limits** (*Space*) – The limits to integrate
- **axes** (*Tuple[int]*) – The dimensions to integrate over
- **norm_range** (*bool*) – **norm_range_arg_descr**
- **params** (*dict*) – The parameters of the function

Returns

Return type Union[tf.Tensor, float]

Raises `NotImplementedError` – If the requested integral is not available.

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

Parameters

- **func** (*callable*) – The integral function. Takes 1 argument.
- **axes** (*tuple*) – **dims_arg_descr**
- **limits** (*possible*) – **limits_arg_descr** Limits can be None if *func* works for any
- **limits** –
- **priority** (*int*) – If two or more integrals can integrate over certain limits, the one with the higher priority is taken (usually around 0-100).
- **supports_norm_range** (*bool*) – If True, norm_range will (if needed) be given to *func* as an argument.
- **supports_multiple_limits** (*bool*) – If True, multiple limits may be given as an argument to *func*.

class zfit.core.integration.**Integral** (*func*: Callable, *limits*: zfit.core.limits.Space, *priority*: Union[int, float])

Bases: `object`

A lightweight holder for the integral function.

class zfit.core.integration.**Integration** (*mc_sampler*, *draws_per_dim*)

Bases: `object`

```
class zfit.core.integration.PartialIntegralSampleData (sample:
                                                    List[tensorflow.python.framework.ops.Tensor],
                                                    space:
                                                    zfit.core.interfaces.ZfitSpace)
```

Bases: `zfit.core.dimension.BaseDimensional`, `zfit.core.interfaces.ZfitData`

Takes a list of tensors and “fakes” a dataset. Useful for tensors with non-matching shapes.

Parameters

- **sample** (`List[tf.Tensor]`) –
- **()** (`space`) –

axes

Return the axes.

copy (*deep*: `bool = False`, ***overwrite_params*) → `zfit.core.interfaces.ZfitObject`

n_obs

Return the number of observables.

name

Name prepended to all ops created by this *model*.

obs

Return the observables.

sort_by_axes (*axes*, *allow_superset*: `bool = False`)

sort_by_obs (*obs*, *allow_superset*: `bool = False`)

space

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

unstack_x (*always_list*=`False`)

value (*obs*: `List[str] = None`)

weights

`zfit.core.integration.auto_integrate` (*, *norm_range*: `bool = False`, *multiple_limits*: `bool = False`) → `Callable`

`zfit.core.integration.chunked_average` (*func*, *x*, *num_batches*, *batch_size*, *space*, *mc_sampler*)

`zfit.core.integration.mc_integrate` (*func*: `Callable`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *axes*: `Union[int, Iterable[int], None] = None`, *x*: `Union[float, tensorflow.python.framework.ops.Tensor, None] = None`, *n_axes*: `Optional[int] = None`, *draws_per_dim*: `int = 20000`, *method*: `str = None`, *dtype*: `Type[CT_co] = tf.float64`, *mc_sampler*: `Callable = <function sample_halton_sequence>`, *importance_sampling*: `Optional[Callable] = None`) → `tensorflow.python.framework.ops.Tensor`

Monte Carlo integration of *func* over *limits*.

Parameters

- **func** (*callable*) – The function to be integrated over
- **limits** (*Space*) – The limits of the integral

- **axes** (*tuple(int)*) – The row to integrate over. None means integration over all value
- **x** (*numeric*) – If a partial integration is performed, this are the value where x will be evaluated.
- **n_axes** (*int*) – the number of total dimensions (old?)
- **draws_per_dim** (*int*) – How many random points to draw per dimensions
- **method** (*str*) – Which integration method to use
- **dtype** (*dtype*) – **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.ZfitData`

Bases: `zfit.core.interfaces.ZfitDimensional`

axes

Return the axes.

copy (*deep: bool = False, **overwrite_params*) → `zfit.core.interfaces.ZfitObject`

n_obs

Return the number of observables.

name

Name prepended to all ops created by this *model*.

obs

Return the observables.

sort_by_axes (*axes, allow_superset: bool = False*)

sort_by_obs (*obs, allow_superset: bool = False*)

space

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

value (*obs: List[str] = None*) → `Union[float, tensorflow.python.framework.ops.Tensor]`

weights

class `zfit.core.interfaces.ZfitDependentsMixin`

Bases: `object`

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

class `zfit.core.interfaces.ZfitDimensional`
Bases: `zfit.core.interfaces.ZfitObject`

axes
Return the axes.

copy (*deep*: bool = False, ***overwrite_params*) → `zfit.core.interfaces.ZfitObject`

n_obs
Return the number of observables.

name
Name prepended to all ops created by this *model*.

obs
Return the observables.

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

class `zfit.core.interfaces.ZfitFunc`
Bases: `zfit.core.interfaces.ZfitModel`

as_pdf ()

axes
Return the axes.

copy (*deep*: bool = False, ***overwrite_params*) → `zfit.core.interfaces.ZfitObject`

dtype
The *DType* of *Tensor*’s handled by this ‘*model*’.

func (*x*: Union[float, tensorflow.python.framework.ops.Tensor], *name*: str = ‘value’) → Union[float, tensorflow.python.framework.ops.Tensor]

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

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

integrate (*limits*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], *norm_range*: Union[Tuple[Tuple[float, ...], Tuple[float, ...], bool] = None, *name*: str = ‘integrate’) → Union[float, tensorflow.python.framework.ops.Tensor]
Integrate the function over *limits* (normalized over *norm_range* if not False).

Parameters

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

Returns the integral value

Return type Tensor

n_obs
Return the number of observables.

name

Name prepended to all ops created by this *model*.

obs

Return the observables.

params

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

Partially integrate the function over the *limits* and evaluate it at *x*.

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

Parameters

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

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

Return type Tensor

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

Register an analytic integral with the class.

Parameters

- **()** (*limits*) –
- **()** – **limits_arg_descr!**
- **priority** (*int*) –
- **supports_multiple_limits** (*bool*) –
- **supports_norm_range** (*bool*) –

Returns:

classmethod register_inverse_analytic_integral (*func*: `Callable`)

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

Parameters **()** (*func*) –

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

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

Parameters

- **n** (*int*) – The number of samples to be generated

- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns Tensor(n_obs, n_samples)

space

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

update_integration_options (*args, **kwargs)

class zfit.core.interfaces.ZfitFunctorMixin

Bases: *object*

get_models () → List[zfit.core.interfaces.ZfitModel]

models

class zfit.core.interfaces.ZfitLoss

Bases: *zfit.core.interfaces.ZfitObject*, *zfit.core.interfaces.ZfitDependentsMixin*

add_constraints (constraints: List[*tensorflow.python.framework.ops.Tensor*])

copy (deep: bool = False, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

data

errordef

fit_range

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

gradients (params: Union[zfit.core.interfaces.ZfitParameter, *int*, *float*, *complex*, *tensorflow.python.framework.ops.Tensor*] = *None*) → List[*tensorflow.python.framework.ops.Tensor*]

model

name

Name prepended to all ops created by this *model*.

value () → Union[*tensorflow.python.framework.ops.Tensor*, *numpy.array*]

class zfit.core.interfaces.ZfitModel

Bases: *zfit.core.interfaces.ZfitNumeric*, *zfit.core.interfaces.ZfitDimensional*

axes

Return the axes.

copy (deep: bool = False, ***overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The *DType* of *Tensor*'s handled by this *model*.

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

get_params (only_floating: bool = False, names: Union[str, List[str], None] = None) → List[zfit.core.interfaces.ZfitParameter]

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

Integrate the function over *limits* (normalized over *norm_range* if not False).

Parameters

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

Returns the integral value

Return type Tensor

n_obs

Return the number of observables.

name

Name prepended to all ops created by this *model*.

obs

Return the observables.

params

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

Partially integrate the function over the *limits* and evaluate it at *x*.

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

Parameters

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

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

Return type Tensor

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

Register an analytic integral with the class.

Parameters

- **()** (*limits*) –
- **()** – **limits_arg_descr!**
- **priority** (*int*) –
- **supports_multiple_limits** (*bool*) –
- **supports_norm_range** (*bool*) –

Returns:

classmethod register_inverse_analytic_integral (*func: Callable*)

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

Parameters () (*func*) –

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

Parameters

- **n** (*int*) – The number of samples to be generated
- **limits** (tuple, *Space*) – In which region to sample in
- **name** (*str*) –

Returns Tensor(*n_obs, n_samples*)

space

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

update_integration_options (**args, **kwargs*)

class zfit.core.interfaces.ZfitNumeric

Bases: *zfit.core.interfaces.ZfitDependentsMixin, zfit.core.interfaces.ZfitObject*

copy (*deep: bool = False, **overwrite_params*) → zfit.core.interfaces.ZfitObject

dtype

The *DType* of *Tensor*’s handled by this *model*.

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

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

name

Name prepended to all ops created by this *model*.

params

class zfit.core.interfaces.ZfitObject

Bases: *object*

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[float]]]

Return the limits, either as *Space* objects or as pure limits-tuple.

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

Example

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

Returns

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

limit1d

return the tuple(lower, upper).

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

Return type tuple(float, float)

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

Type Simplified limits getter for 1 obs, 1 limit only

limit2d

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

Returns

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

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

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

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

limits

Return the limits.

Returns:

limits1d

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

Returns

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

Return type `tuple(float, float, ...)`

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

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

lower

Return the lower limits.

Returns:

n_limits

The number of different limits.

Returns `int >= 1`

n_obs

Return the number of observables/axes.

Returns `int >= 1`

name

The name of the object.

obs

The observables (“axes with str”)the space is defined in.

Returns:

obs_axes

reorder_by_indices (*indices: Tuple[int]*)

Return a *Space* reordered by the indices.

Parameters `() (indices)` –

upper

Return the upper limits.

Returns:

with_autofill_axes (*overwrite: bool = False*) → *zfit.Space*

Return a *Space* with filled axes corresponding to `range(len(n_obs))`.

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

Returns *Space*

with_axes (*axes: Union[int, Iterable[int]]*) → *zfit.Space*

Sort by *obs* and return the new instance.

Parameters `() (axes)` –

Returns *Space*

with_limits (*limits: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*, *name: Optional[str] = None*) → *zfit.Space*

Return a copy of the space with the new *limits* (and the new *name*).

Parameters

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

Returns *Space*

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

Parameters **()** (*obs*) –

Returns *Space*

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

Parameters

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

Returns

Return type *Space*

zfit.core.limits.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.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, data, fit_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, constraints: List[tensorflow.python.framework.ops.Tensor] = None)
Bases: zfit.core.baseobject.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*]

model

name

Name prepended to all ops created by this *model*.

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.loss.CachedLoss* (*model*, *data*, *fit_range*=*None*, *constraints*=*None*)

Bases: *zfit.core.loss.BaseLoss*

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

add_constraints (*constraints*)

constraints

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

data

errordef

fit_range

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

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

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

gradients (*params:* `Union[zfit.core.interfaces.ZfitParameter, tensorflow.python.framework.ops.Tensor]`, *int,* *float,* *com-*
plex, *tensorflow.python.framework.ops.Tensor] *=* *None*) *→*
`List[tensorflow.python.framework.ops.Tensor]`*

model

name
Name prepended to all ops created by this *model*.

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

class `zfit.core.loss.ExtendedUnbinnedNLL` (*model, data, fit_range=None, 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]]`, *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*.

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, tensorflow.python.framework.ops.Tensor]`, *int,* *float,* *com-*
plex, *tensorflow.python.framework.ops.Tensor] *=* *None*) *→*
`List[tensorflow.python.framework.ops.Tensor]`*

model

name

Name prepended to all ops created by this *model*.

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.loss.SimpleLoss (*func*: Callable, *dependents*: Optional[Iterable[zfit.Parameter]] = None, *errordef*: Optional[float] = None)

Bases: zfit.core.loss.CachedLoss

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]

model

name

Name prepended to all ops created by this *model*.

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.

value ()

class zfit.core.loss.UnbinnedNLL (*model*, *data*, *fit_range*=None, *constraints*=None)

Bases: zfit.core.loss.CachedLoss

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) *Iter-*

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]

model

name
Name prepended to all ops created by this *model*.

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 ()

math

`zfit.core.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.core.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*

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,
                                                    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)`

independent

```
load (value, session=None)
```

name

The name of the object.

params

```
read_value ()
```

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

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

Parameters **()** (`cacher`) –

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

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

value ()

class zfit.core.parameter.BaseParameter
    Bases: zfit.core.interfaces.ZfitParameter

    copy (deep: bool = False, **overwrite_params) → zfit.core.interfaces.ZfitObject

    dtype
        The DType of Tensor's handled by this 'model'.

    floating

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

    get_params (only_floating: bool = False, names: Union[str, List[str], None] = None) → List[zfit.core.interfaces.ZfitParameter]

    independent

    name
        Name prepended to all ops created by this model.

    params

    value () → tensorflow.python.framework.ops.Tensor

class zfit.core.parameter.BaseZParameter (name, initial_value, **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

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

Returns

Return type list(ZfitParameters)

independent

load (*value*, *session*=None)

name

The name of the object.

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*, *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)*

imag

independent

load (*value, session=None*)

mod

name

The name of the object.

params

read_value ()

real

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.ComposedParameter* (*name, tensor, 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*)

independent

load (*value*, *session=None*)

name

The name of the object.

params

read_value ()

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

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

Parameters **()** (*cacher*) –

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

reset_cache_self ()

Clear the cache of self and all dependent cachers.

value ()

class *zfit.core.parameter.ComposedResourceVariable* (*name*, *initial_value*, ***kwargs*)

Bases: *tensorflow.python.ops.resource_variable_ops.ResourceVariable*

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.

`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 *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.ndim - 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 subensors that result of slicing the first dimension. This is a valid option for *ndims* = 1, but less efficient than this implementation.

Parameters

- **sparse_delta** – *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 *ValueError* – 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.

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 *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 subtraction has completed.

Raises *ValueError* – 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.

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

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.

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

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.

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

scatter_sub (*sparse_delta*, *use_locking=False*, *name=None*)

Subtracts *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 `ValueError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update (*sparse_delta*, *use_locking=False*, *name=None*)

Assigns *IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *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 `ValueError` – 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.ComposedVariable` (*name: str, initial_value: tensorflow.python.framework.ops.Tensor, **kwargs*)

Bases: *object*

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

dtype

load (*value*, *session=None*)

name

read_value ()

value ()

class `zfit.core.parameter.ConstantParameter` (*name*, *value*)

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)

independent

load (`value, session=None`)

name

The name of the object.

params

read_value ()

register_cacher (`catcher: Union[zfit.core.interfaces.ZfitCachable, Iter-
able[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.**MetaBaseParameter**

Bases: tensorflow.python.ops.variables.VariableMetaclass, type

mro ()

Return a type's method resolution order.

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

Bases: `zfit.util.execution.SessionHolderMixin`, `zfit.core.parameter.ZfitParameterMixin`, `zfit.core.parameter.TFBaseVariable`, `zfit.core.parameter.BaseParameter`

Class for fit parameters, derived from TF Variable class.

Constructor. `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.

`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 *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** – *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 *ValueError* – 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.

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.

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: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)

Parameter takes on the *value*. Is not part of the graph, does a session run.

Parameters **value** (*numerical*) –

lower_limit

name

The name of the object.

numpy ()

op

The op for this variable.

params

randomize (*minval=None, maxval=None, sampler=<built-in method uniform of 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, Iter-
able[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 *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 subtraction has completed.

Raises `ValueError` – 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.

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

`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.

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

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.

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

scatter_sub (*sparse_delta*, *use_locking=False*, *name=None*)

Subtracts *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 `ValueError` – if *sparse_delta* is not an *IndexedSlices*.

scatter_update (*sparse_delta*, *use_locking=False*, *name=None*)

Assigns *IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *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 `ValueError` – if *sparse_delta* is not an *IndexedSlices*.

sess

set_sess (*sess: tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

set_shape (*shape*)

Unsupported.

set_value (*value: Union[int, float, complex, tensorflow.python.framework.ops.Tensor]*)

Set the *Parameter* to *value* (temporarily if used in a context manager).

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

shape

The shape of this variable.

sparse_read (*indices*, *name=None*)

Reads the value of this variable sparsely, using *gather*.

step_size

synchronization

to_proto (*export_scope=None*)

Converts a *ResourceVariable* to a *VariableDef* protocol buffer.

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

Raises `RuntimeError` – If run in EAGER mode.

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

trainable

upper_limit

value()

A cached operation which reads the value of this variable.

```
class zfit.core.parameter.TFBaseVariable(initial_value=None, trainable=None, collections=None, validate_shape=True,
                                         caching_device=None, name=None, dtype=None, variable_def=None, import_scope=None,
                                         constraint=None, distribute_strategy=None, synchronization=None, aggregation=None, shape=None)
```

Bases: tensorflow.python.ops.resource_variable_ops.ResourceVariable

Creates a variable.

Parameters

- **initial_value** – A *Tensor*, or Python object convertible to a *Tensor*, which is the initial value for the *Variable*. Can also be a callable with no argument that returns the initial value when called. (Note that initializer functions from `init_ops.py` must first be bound

to a shape before being used here.)

- **trainable** – If *True*, the default, also adds the variable to the graph collection *GraphKeys.TRAINABLE_VARIABLES*. This collection is used as the default list of variables to use by the *Optimizer* classes.

Defaults to *True* unless *synchronization* is set to *ON_READ*.

- **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. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- **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.

__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.

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 *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** – *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 *ValueError* – 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.

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 *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 subtraction has completed.

Raises `ValueError` – 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.

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

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.

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

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.

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

scatter_sub (*sparse_delta, use_locking=False, name=None*)

Subtracts *IndexedSlices* from this variable.

Parameters

- **sparse_delta** – *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 *ValueError* – if *sparse_delta* is not an *IndexedSlices*.

scatter_update (*sparse_delta*, *use_locking=False*, *name=None*)

Assigns *IndexedSlices* to this variable.

Parameters

- **sparse_delta** – *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 *ValueError* – 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.VariableV1,
                                             **kwargs)
```

Bases: *object*

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

dtype

value ()

```
class zfit.core.parameter.ZfitParameterMixin (name, initial_value, **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)

name

The name of the object.

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) → zfit.core.interfaces.ZfitParameter

Convert a *numerical* to a fixed 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.feed_function(feed, feed_val)
zfit.core.parameter.feed_function_for_partial_run(feed)
zfit.core.parameter.fetch_function(variable)
zfit.core.parameter.get_auto_number()
```

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.accept_reject_sample (*prob*: *Callable*, *n*: *int*, *limits*: *zfit.core.limits.Space*, *sample_and_weights_factory*: *Callable* = *<class 'zfit.core.sample.UniformSampleAndWeights'>*, *dtype*=*tf.float64*, *prob_max*: *Union[None, int]* = *None*, *efficiency_estimation*: *float* = *1.0*) → *tensorflow.python.framework.ops.Tensor*

Accept reject sample from a probability distribution.

Parameters

- **prob** (*function*) – A function taking *x* a *Tensor* as an argument and returning the probability (or anything that is proportional to the probability).
- **n** (*int*) – Number of samples to produce
- **limits** (*Space*) – The limits to sample from
- **sample_and_weights_factory** (*Callable*) – A factory function that returns the following function: A function that returns the sample to insert into *prob* and the weights (probability density) of each sample together with the random thresholds. The API looks as follows:
 - Parameters:
 - * *n_to_produce* (*int*, *tf.Tensor*): The number of events to produce (not exactly).
 - * *limits* (*Space*): the limits in which the samples will be.
 - * *dtype* (*dtype*): *DType* of the output.
 - **Return:** A tuple of length 5:
 - proposed sample (*tf.Tensor* with shape=(*n_to_produce*, *n_obs*)): The new (proposed) sample whose values are inside *limits*.

- * **thresholds_unscaled** (`tf.Tensor` with `shape=(n_to_produce,)`): Uniformly distributed random values **between 0 and 1**.
- * **weights** (`tf.Tensor` with `shape=(n_to_produce,)`): (Proportional to the) probability for each sample of the distribution it was drawn from.
- * **weights_max** (`int`, `tf.Tensor`, `None`): The maximum of the weights (if known). This is what the probability maximum will be scaled with, so it should be rather lower than the maximum if the peaks do not exactly coincide. Otherwise return `None` (which will **assume** that the peaks coincide).
- * **n_produced**: the number of events produced. Can deviate from the requested number.

- `() (dtype)` –
- **prob_max** (`Union[None, int]`) – The maximum of the model function for the given limits. If `None` is given, it will be automatically, safely estimated (by a 10% increase in computation time (constant weak scaling)).
- **efficiency_estimation** (`float`) – estimation of the initial sampling efficiency.

Returns

Return type `tf.Tensor`

`zfit.core.sample.extended_sampling` (`pdfs`: `Union[Iterable[zfit.core.interfaces.ZfitPDF], zfit.core.interfaces.ZfitPDF]`, `limits`: `zfit.core.limits.Space`)
→ `tensorflow.python.framework.ops.Tensor`

Create a sample from extended pdfs by sampling poissonian using the yield.

Parameters

- **pdfs** (`Iterable[ZfitPDF]`) –
- **limits** (`Space`) –

Returns

Return type `Union[tf.Tensor]`

`zfit.core.sample.extract_extended_pdfs` (`pdfs`: `Union[Iterable[zfit.core.interfaces.ZfitPDF], zfit.core.interfaces.ZfitPDF]`)
→ `List[zfit.core.interfaces.ZfitPDF]`

Return all extended pdfs that are daughters.

Parameters **pdfs** (`Iterable[pdfs]`) –

Returns

Return type `List[pdfs]`

testing

Module for testing of the zfit components. Contains a singleton instance to register new PDFs and let them be tested.

`zfit.core.testing.setup_function()`

`zfit.core.testing.teardown_function()`

minimizers

Submodules

base_tf

```
class zfit.minimizers.base_tf.WrapOptimizer (optimizer, tolerance=None, verbosity=None,
                                             name=None, **kwargs)
    Bases: zfit.minimizers.baseminimizer.BaseMinimizer

    copy ()

    minimize (loss: zfit.core.interfaces.ZfitLoss, params: Optional[Iterable[zfit.core.interfaces.ZfitParameter]]
              = None) → zfit.minimizers.fitresult.FitResult
        Fully minimize the loss with respect to params.

        Parameters
        • loss (ZfitLoss) – Loss to be minimized.
        • params (list(zfit.Parameter)) – The parameters with respect to which to minimize
          the loss. If None, the parameters will be taken from the loss.

        Returns The fit result.

        Return type FitResult

    sess

    set_sess (sess: tensorflow.python.client.session.Session)
        Set the session (temporarily) for this instance. If None, the auto-created default is taken.

        Parameters sess (tf.compat.v1.Session) –

    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 mini-
          mizer.

    tolerance
```

baseminimizer

Definition of minimizers, wrappers etc.

```
class zfit.minimizers.baseminimizer.BaseMinimizer (name, tolerance, verbosity, min-
                                                    imizer_options, strategy=None,
                                                    **kwargs)
    Bases: zfit.util.execution.SessionHolderMixin, zfit.minimizers.interface.
            ZfitMinimizer

    Minimizer for loss functions.

    Additional minimizer_options (given as **kwargs) can be accessed and changed via the attribute (dict) mini-
    mizer.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*

sess

set_sess (*sess*: *tensorflow.python.client.session.Session*)
Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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)
```

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.util.execution.SessionHolderMixin, 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) → collec-
tions.OrderedDict
```

Calculate and set for *params* the asymmetric error using the set error method.

Parameters

- **params** (list(*Parameter* or str)) – The parameters or their names to calculate the errors. If *params* is *None*, use all *floating* parameters.

- **method** (*str* or *Callable*) – The method to use to calculate the errors. Valid choices are {‘minuit_minos’} or a Callable.
- **sigma** (*float*) – Errors are calculated with respect to *sigma* std deviations. The definition of 1 sigma depends on the loss function and is defined there.

For example, the negative log-likelihood (without the factor of 2) has a correspondents of Δ NLL of 1 corresponds to 1 std deviation.
- **error_name** (*str*) – The name for the error in the dictionary.

Returns

A *OrderedDict* containing as keys the parameter names and as value a *dict* which contains (next to probably more things) two keys ‘lower’ and ‘upper’, holding the calculated errors. Example: result[‘par1’][‘upper’] -> the asymmetric upper error of ‘par1’

Return type *OrderedDict*

fmin

Function value at the minimum.

Returns numeric

hesse (*params*: *Optional[Iterable[zfit.core.interfaces.ZfitParameter]]* = *None*, *method*: *Union[str, Callable]* = ‘minuit_hesse’, *error_name*: *Optional[str]* = *None*, *sigma*=1.0) → *collections.OrderedDict*
Calculate for *params* the symmetric error using the Hessian matrix.

Parameters

- **params** (*list(Parameter)*) – The parameters to calculate the Hessian symmetric error. If *None*, use all parameters.
- **method** (*str*) – the method to calculate the hessian. Can be {‘minuit’} or a callable.
- **error_name** (*str*) – The name for the error in the dictionary.

Returns

Result of the hessian (symmetric) error as dict with each parameter holding the error dict {‘error’: sym_error}.

So given *param_a* (from *zfit.Parameter(.)*) *error_a* = *result.hesse(params=param_a)[param_a][‘error’]* *error_a* is the hessian error.

Return type *OrderedDict*

info**loss****minimizer****params****sess**

set_sess (*sess*: *tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters *sess* (*tf.compat.v1.Session*) –

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 ()

```
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`

```
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.

sess

set_sess (*sess: tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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.**BFGSMinimizer** (*args, **kwargs)

Bases: `zfit.minimizers.baseminimizer.BaseMinimizer`

copy ()

minimize (*sess=None, params=None*)

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*

sess

set_sess (*sess: tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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

sess

set_sess (sess: tensorflow.python.client.session.Session)
    Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters sess (tf.compat.v1.Session) –

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*

sess

set_sess (*sess*: *tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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 nonnegative.

- **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 $\pm np.infty$). 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 (*session=None*, *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 component is = 1.
equalities = [vector[1] - 1.] # Ensure the vector's x component is >= 1.
inequalities = [vector[0] - 1.]

# Our default SciPy optimization algorithm, L-BFGS-B, does not support # general constraints. Thus we use SLSQP instead.
optimizer = ScipyOptimizerInterface(
    loss, equalities=equalities, inequalities=inequalities, method='SLSQP')

with tf.compat.v1.Session() as session: optimizer.minimize(session)

# The value of vector should now be [1., 1.]. """
```

Initialize a new interface instance.

Parameters

- **loss** – A scalar *Tensor* to be minimized.
- **var_list** – Optional *list* of *Variable* objects to update to minimize *loss*. Defaults to the list of variables collected in the graph under the key *GraphKeys.TRAINABLE_VARIABLES*.
- **equalities** – Optional *list* of equality constraint scalar *Tensor*'s to be held equal to zero.
- **inequalities** – Optional *list* of inequality constraint scalar *Tensor*'s to be held nonnegative.
- **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 (*session=None, 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape *()*

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*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.

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`) \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`)

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`) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`**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_rangeReturn 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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: `str = 'model'`) → `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type `self.dtype`.

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.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'*) \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`) → `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`)

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`) – name of the operation shown in the `tf.Graph`

Returns `py:class'tf.Tensor'`: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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]]*) *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', 'r', '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`)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class'tf.Tensor':` the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, `norm_range`: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, `name`: `str = 'model'`) \rightarrow `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type `self.dtype`.

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]]*)

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.dist_tfp.Gauss (mu: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], sigma: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str], zfit.Space], name: str = 'Gauss')
```

Bases: `zfit.models.dist_tfp.WrapDistribution`

Gaussian or Normal distribution with a mean (`mu`) and a standartdevation (`sigma`).

The gaussian shape is defined as

$$f(x \mid \mu, \sigma^2) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

with the normalization over `[-inf, inf]` of

$$\frac{1}{\sqrt{2\pi\sigma^2}}$$

The normalization changes for different normalization ranges

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type Tensor

Raises

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

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

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

Parameters

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

Returns numerical

as_func (*norm_range*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

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

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

Returns

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

Return type `model`

create_extended (*yield_*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, name_addition='_extended']`) → `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `yield_`. 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape *()*

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*, bool) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

params

```
partial_analytic_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_analytic_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

```
partial_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

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

```
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 (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
```

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 tf.Tensor of type *self.dtype*.

```
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.dist_tfp.TruncatedGauss` (*mu*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *sigma*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *low*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *high*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *obs*: `Union[str, Iterable[str], zfit.Space]`, *name*: `str = 'TruncatedGauss'`)

Bases: `zfit.models.dist_tfp.WrapDistribution`

Gaussian distribution that is 0 outside of *low*, *high*. Equivalent to the product of `Gauss` and `Uniform`.

Parameters

- **mu** (`Parameter`) – Mean of the gaussian dist
- **sigma** (`Parameter`) – Standard deviation or spread of the gaussian
- **low** (`Parameter`) – Below this value, the pdf is zero.

- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type *Tensor*

Raises

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

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

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

Parameters

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

Returns *numerical*

as_func (*norm_range*: *Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]* = *False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

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

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

Returns

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

Return type model

create_extended (*yield_*: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], *name_addition*='_extended') → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

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

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

distribution**dtype**

The dtype of the object

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

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

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

get_params (*only_floating: bool = False*, *names: Union[str, List[str], None] = None*) -> *List[zfit.core.interfaces.ZfitParameter]*

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

Parameters

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

Returns

Return type *list(ZfitParameters)*

get_yield () -> *Optional[zfit.core.parameter.Parameter]*

Return the yield (only for extended models).

Returns the yield of the current model or *None*

Return type *Parameter*

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

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range:
Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') →
Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.


```
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') → 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*) -> *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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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*) –

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.BaseFunc (funcs, name='BaseFunc',
                                     **kwargs)
```

Bases: *zfit.models.basefunc.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, 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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class*‘tf.Tensor’: the integral value as a scalar with shape **()**

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (`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`) \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.

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.BaseFuncorFunc*

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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape **()**

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (`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`) \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.

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` (*func: Callable, obs: Union[str, Iterable[str], zfit.Space], 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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class'tf.Tensor'*: the integral value as a scalar with shape **()**

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (`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`) \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.

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, 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape **()**

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (`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`) \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.

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, 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)

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*) – name of the operation shown in the tf.Graph

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*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.

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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range:
Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') →
Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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, 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.SampleDataSample *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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range:
Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') →
Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.SampleDataSample *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],      frags:
                                Union[zfit.core.interfaces.ZfitParameter, int, float, com-
                                plex, 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 *frags* as coefficients.

Parameters

- **pdfs** (*pdf*) – The pdfs to add.
- **frags** (*iterable*) – coefficients for the linear combination of the pdfs. If pdfs are extended, this throws an error.
 - `len(frag) == len(basic) - 1` results in the interpretation of a non-extended pdf. The last coefficient will equal to `1 - sum(frag)`
 - `len(frag) == 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, 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.

dtype

The dtype of the object

fracs

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

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

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

get_models (*names*=*None*) → *List*[*zfit.core.interfaces.ZfitModel*]

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

Parameters

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

Returns

Return type list(ZfitParameters)

get_yield () → Optional[zfit.core.parameter.Parameter]
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type Parameter

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

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) – name of the operation shown in the tf.Graph

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, `norm_range`: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, `name`: `str = 'model'`) \rightarrow `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type `self.dtype`.

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]]*)

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*) –

physics

```
class zfit.models.physics.CrystalBall(mu: Union[zfit.core.interfaces.ZfitParameter,
int, float, complex, tensorflow.python.framework.ops.Tensor],
sigma: Union[zfit.core.interfaces.ZfitParameter,
int, float, complex, tensorflow.python.framework.ops.Tensor],
alpha: Union[zfit.core.interfaces.ZfitParameter, int, float,
complex, tensorflow.python.framework.ops.Tensor],
n: Union[zfit.core.interfaces.ZfitParameter, int, float,
complex, tensorflow.python.framework.ops.Tensor],
obs: Union[str, Iterable[str], zfit.Space], name: str =
'CrystalBall', dtype: Type[CT_co] = tf.float64)
```

Bases: `zfit.core.basepdf.BasePDF`

‘Crystal Ball shaped PDF’ __. A combination of a Gaussian with an powerlaw tail.

The function is defined as follows:

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

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, 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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],
                                   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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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*) –

`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*, *alpha*, *nl*, *alpha*, *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'`) \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`) → `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)

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) – name of the operation shown in the tf.Graph

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = `'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: `str = 'model'`) → `Union[float, tensorflow.python.framework.ops.Tensor]`
 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 `tf.Tensor` of type *self.dtype*.

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, Iter-able[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.

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

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

space

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

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

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

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

- **composition** (*certain*) – pdfs.
- **name** (*str*) –

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

Return type `tf.Tensor`

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

Parameters

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

```
class zfit.models.polynomials.Chebyshev2(obs, coeffs: list, apply_scaling: bool = True,  
                                         coeff0: Union[zfit.core.interfaces.ZfitParameter,  
                                         int, float, complex, tensorflow.python.framework.ops.Tensor, None] =  
                                         None, name: str = 'Chebyshev2')
```

Bases: `zfit.models.polynomials.RecursivePolynomial`

Linear combination of Chebyshev (second kind) polynomials of order `len(coeffs)`, `coeffs` are scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with `coeff0`. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the `coeffs` are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

with $T_0 = 1, T_1 = 2x$

Notice that T_1 is $2x$ as opposed to x in Chebyshev polynomials of the first kind.

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent $(-1, 1)$.
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type `Tensor`

Raises

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

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

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

Parameters

- **value** (`numerical`) –
- **()** (`norm_range`) –
- **log** (`bool`) –

Returns `numerical`

as_func (*norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = `False`)

Return a `Function` with the function `model(x, norm_range=norm_range)`.

Parameters **()** (`norm_range`) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

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

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

Returns

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

Return type model

create_extended (*yield_*: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], *name_addition*='_extended') → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

Parameters *limits_to_integrate* (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

degree

degree of the polynomial, starting from 0.

Type `int`

dtype

The dtype of the object

get_dependents (*only_floating*: `bool = True`) \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`)

integrate (*limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'integrate'`) \rightarrow `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`) – name of the operation shown in the `tf.Graph`

Returns py:class`tf.Tensor`: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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.models.polynomials.Hermite` (*obs*, *coeffs*: *list*, *apply_scaling*: *bool* = True, *coeff0*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor, None] = None`, *name*: `str = 'Hermite'`)

Bases: `zfit.models.polynomials.RecursivePolynomial`

Linear combination of Hermite polynomials (for physics) of order `len(coeffs)`, with coeffs as scaling factors.

The 0th coefficient is set to 1 by default but can be explicitly set with *coeff0*. Since the PDF normalization removes a degree of freedom, the 0th coefficient is redundant and leads to an arbitrary overall scaling of all parameters.

Notice that this is already a sum of polynomials and the coeffs are simply scaling the individual orders of the polynomials.

The recursive definition of _a single **order**_ of the polynomial is

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

with $P_0 = 1$ $P_1 = 2x$

Parameters

- **obs** – The default space the PDF is defined in.
- **coeffs** (*list[params]*) – A list of the coefficients for the polynomials of order 1+ in the sum.
- **apply_scaling** (*bool*) – Rescale the data so that the actual limits represent (-1, 1).
- **coeff0** (*param*) – The scaling factor of the 0th order polynomial. If not given, it is set to 1.
- **name** (*str*) – Name of the polynomial

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type *Tensor*

Raises

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

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

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

Parameters

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

Returns numerical

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

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

Parameters

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

Returns:

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

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

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

Returns

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

Return type model

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

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

```
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.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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape **()**

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.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`) \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`)

integrate (*limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'integrate'`) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.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_*. 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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*) –

zfit.models.polynomials.chebyshev2_shape (*x*, *coeffs*)

zfit.models.polynomials.chebyshev_recurrence (*p1*, *p2*, *_*, *x*)

Recurrence relation for Chebyshev polynomials.

$$T_{\{n+1\}}(x) = 2 \times T_{\{n\}}(x) - T_{\{n-1\}}(x)$$

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 polynomial.

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_recurrence(p1, p2, n, x)
    Recurrence relation for Hermite polynomials (physics).

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

zfit.models.polynomials.hermite_shape(x, coeffs)
zfit.models.polynomials.laguerre_recurrence(p1, p2, n, x)
    Recurrence relation for Laguerre polynomials.

$$(n+1)L_{n+1}(x) = (2n+1+lpha-x)L_n(x) - (n+lpha)L_{n-1}(x)$$

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_recurrence(p1, p2, n, x)
    Recurrence relation for Legendre polynomials.

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

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.SimpleFuncPDF(obs, pdfs, func, name='SimpleFuncPDF',  
**params)
```

Bases: `zfit.models.functor.BaseFuncPDF`, `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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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 (*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.

set_norm_range (*norm_range*: *Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]*)

Set the normalization range (temporarily if used with contextmanager).

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

space

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

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

PDF "unnormalized". Use *functions* for unnormalized pdfs. this is only for performance in special cases.

Parameters

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

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

Return type `tf.Tensor`

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

Parameters

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

class `zfit.models.special.SimplePDF` (*obs, func, name='SimplePDF', **params*)

Bases: `zfit.core.basepdf.BasePDF`

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type `Tensor`

Raises

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

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

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`) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`**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_rangeReturn 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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = *'partial_integrate'*) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: *str* = *'model'*) → `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type `self.dtype`.

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.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, 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_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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, $limits$: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, $norm_range$: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, $name$: `str = 'partial_integrate'`) \rightarrow `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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

pdf (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, $norm_range$: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, $name$: `str = 'model'`) \rightarrow `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type *self.dtype*.

```
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*) –

```
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.

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.

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

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.ConversionError
    Bases: Exception

    args

    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception zfit.util.exception.DueToLazynessNotImplementedError
    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.
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.
```

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]

    auto_initialize (variable: tensorflow.python.ops.variables.VariableV1)

    chunksize

    create_session (*args, close_current=True, reset_graph=False, **kwargs)
        Create a new session (or replace the current one). Arguments will overwrite the already set arguments.
```

Parameters

- **close_current** (*bool*) – Closes the current open session before replacement. Has no effect if no session was created before.
- **reset_graph** (*bool*) – Resets the current (default) graph before creating a new `tf.compat.v1.Session`.
- **()** (***kwargs*) –
- **()** –

Returns `tf.compat.v1.Session`

```
n_cpu
reset()
sess
set_n_cpu (n_cpu='auto')
```

```
class zfit.util.execution.SessionHolderMixin (*args, **kwargs)
    Bases: object
```

Creates a *self.sess* attribute, a setter *set_sess* (with a fallback to the zfit default session).

```
sess
```

set_sess (*sess*: `tensorflow.python.client.session.Session`)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters *sess* (`tf.compat.v1.Session`) –

graph

`zfit.util.graph.all_parents` (*op*, *current_obs*=None)

`zfit.util.graph.get_dependents_auto` (*tensor*: `tensorflow.python.framework.ops.Tensor`, *candidates*: `List[tensorflow.python.framework.ops.Tensor]`)
→ `List[tensorflow.python.framework.ops.Tensor]`

Return the nodes in *candidates* that *tensor* depends on.

Parameters

- *()* (*candidates*) –
- *()* –

logging

This module controls the zfit logging.

The base logger for zfit is called *zfit*, and all loggers created by this module have the form *zfit.XX*, where *XX* is their name.

By default, time, name of the logger and message with the default colorlog color scheme are printed.

`zfit.util.logging.get_logger` (*name*, *stdout_level*=None, *file_level*=None, *file_name*=None)

Get and configure logger.

This logger has two handlers:

- A stdout handler is always configured with *colorlog*.
- A file handler is configured if *file_name* is given. Once it has been configure, it is not

necessary to give it to modify its properties.

Once the logger has been created, *get_logger* can be called again to modify its log levels, independently for the stream and file handlers.

Note: If the logger name doesn't start with "zfit", it is automatically added.

Note: Default logging level at first instantiation is WARNING.

Parameters

- **name** (*str*) – Name of the logger.
- **stdout_level** (*int*, *optional*) – Logging level for the stream handler. Defaults to *logging.WARNING*.
- **file_level** (*int*, *optional*) – Logging level for the file handler. Defaults to *logging.WARNING*.
- **file_name** (*str*, *optional*) – File to log to. If not given, no file logging is performed.

Returns The requested logger.

Return type *logging.Logger*

Raise: `ValueError` if *file_level* has been specified without having configured the output file.

temporary

class `zfit.util.temporary.TemporarilySet` (*value: Any, setter: Callable, getter: Callable, setter_args=None, setter_kwargs=None, getter_args=None, getter_kwargs=None*)

Bases: `object`

Temporarily set *value* with *setter* and reset to the old value after leaving the context.

This class can be used to have a setter that can permanently set a value *as well as* just for the time inside a context manager. The usage is as follows:

```
>>> class SimpleX:
>>>     def __init__(self):
>>>         self.x = None
>>>     def _set_x(self, x):
>>>         self.x = x
>>>     def get_x(self):
>>>         return self.x
>>>     def set_x(self, x):
>>>         return TemporarilySet(value=x, setter=self._set_x,
↪getter=self.get_x)
```

```
>>> simple_x = SimpleX()
Now we can either set x permanently
>>> simple_x.set_x(42)
>>> print(simple.x)
42
```

or temporarily >>> with simple_x.set_x(13) as value: >>> print("Value from contextmanager:", value) >>> print("simple_x.get_x():", simple_x.get_x()) 13 13

and is afterwards unset again >>> print(simple.x) 42

Parameters

- **value** (*Any*) – The value to be (temporarily) set (and returned if a context manager is applied).
- **setter** (*Callable*) – The setter function with a signature that is compatible to the call: `setter(value, *setter_args, **setter_kwargs)`
- **getter** (*Callable*) – The getter function with a signature that is compatible to the call: `getter(*getter_args, **getter_kwargs)`
- **setter_args** (*List*) – A list of arguments given to the setter
- **setter_kwargs** (*Dict*) – A dict of keyword-arguments given to the setter
- **getter_args** (*List*) – A list of arguments given to the getter
- **getter_kwargs** (*Dict*) – A dict of keyword-arguments given to the getter

ztyping

ztf

ztf 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 *ztf* whenever possible and *tf* for the rest.

Submodules

random

```
zfit.ztf.random.counts_multinomial (total_count: Union[int, tensor-
                                     flow.python.framework.ops.Tensor],
                                     probs: Iterable[Union[float, tensor-
                                     flow.python.framework.ops.Tensor]] =
                                     None, logits: Iterable[Union[float, ten-
                                     sorflow.python.framework.ops.Tensor]]
                                     = None, dtype=tf.int32) → tensor-
                                     flow.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.ztf.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.ztf.wrapping_tf.complex (real, imag, name=None)
```

```
zfit.ztf.wrapping_tf.exp (x, name=None)
```

```
zfit.ztf.wrapping_tf.log (x, name=None)
```



```

zfit.ztf.wrapping_tf.pow(x, y, name=None)
zfit.ztf.wrapping_tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float64,
                                   seed=None, name=None)
zfit.ztf.wrapping_tf.random_poisson(lam: Any, shape: Any, dtype: tensorflow.python.framework.dtypes.DType = tf.float64,
                                   seed: Any = None, name: Any = None)
zfit.ztf.wrapping_tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float64,
                                   seed=None, name=None)
zfit.ztf.wrapping_tf.sqrt(x, name=None)
zfit.ztf.wrapping_tf.square(x, name=None)

```

zextension

```

zfit.ztf.zextension.abs_square(x)
zfit.ztf.zextension.constant(value, dtype=tf.float64, shape=None, name='Const', verify_shape=None)
zfit.ztf.zextension.convert_to_tensor(value, dtype=None, name=None, preferred_dtype=None)
zfit.ztf.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.ztf.zextension.run_no_nan(func, x)
```

```

zfit.ztf.zextension.safe_where(condition: tensorflow.python.framework.ops.Tensor,
                              func: Callable, safe_func: Callable, values: tensorflow.python.framework.ops.Tensor,
                              value_safer: Callable = <function add_dispatch_support.<locals>.wrapper>) → 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* 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* 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.ztf.zextension.stack_x(values, axis: int = -1, name: str = 'stack_x')`

`zfit.ztf.zextension.to_complex(number, dtype=tf.complex128)`

`zfit.ztf.zextension.to_real(x, dtype=tf.float64)`

`zfit.ztf.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-

10.1.2 Submodules

constraint

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

Return negative log likelihood graph for gaussian constraints on a list of parameters.

Parameters

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

Returns the constraint object

Return type `GaussianConstraint`

Raises `ShapeIncompatibleError` – if params, mu and sigma don't have the same size

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

name

The name of the object.

params

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

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

Parameters **()** (*cacher*) –

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

reset_cache_self ()

Clear the cache of self and all dependent cachers.

sample (*n*)

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

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

Returns `n_samples`)

Return type `Dict(Parameter)`

`value()`

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

Bases: `zfit.core.constraint.DistributionConstraint`

Gaussian constraints on a list of parameters.

Parameters

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

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

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

distribution

dtype

The dtype of the object

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

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

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

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

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

Parameters

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

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

Returns**Return type** `list(ZfitParameters)`**name**

The name of the object.

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.data.Data` (*dataset*: `Union[tensorflow.python.data.ops.dataset_ops.DatasetV1, 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.execution.SessionHolderMixin`, `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,
                           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, 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()
```

```
initialize()
```

```
iterator
```

```
n_obs
```

Return the number of observables.

```
name
```

The name of the object.

```
nevents
```

obs
Return the observables.

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.

sess

set_data_range (*data_range*)

set_sess (*sess*: *tensorflow.python.client.session.Session*)
Set the session (temporarily) for this instance. If *None*, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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

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

sort_by_axes (*axes*: *Union[int, Iterable[int]]*, *allow_superset*: *bool = False*)

sort_by_obs (*obs*: *Union[str, Iterable[str], zfit.Space]*, *allow_superset*: *bool = False*)

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

to_pandas (*obs*: *Union[str, Iterable[str], zfit.Space] = None*)
Create a *pd.DataFrame* from *obs* as columns and return it.

Parameters () (*obs*) – The observables to use as columns. If *None*, all observables are used.

Returns:

unstack_x (*obs*: *Union[str, Iterable[str], zfit.Space] = None*, *always_list*: *bool = False*)
Return the unstacked data: a list of tensors or a single Tensor.

Parameters

- () (*obs*) – which observables to return
- **always_list** (*bool*) – If True, always return a list (also if length 1)

Returns List(*tf.Tensor*)

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

weights

func

class *zfit.func.BaseFunc* (*obs=None*, *dtype: Type[CT_co] = tf.float64*, *name: str = 'BaseFunc'*,
params: Any = None)
Bases: *zfit.core.basemodel.BaseModel, zfit.core.interfaces.ZfitFunc*
TODO(docs): explain subclassing

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type Tensor

Raises

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

as_pdf () → zfit.core.interfaces.ZfitPDF

Create a PDF out of the function

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

Return type ZfitPDF

axes

Return the axes.

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

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

Parameters

- () (*limits*) –
- () –
- () –

Returns:

copy (***override_params*)

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

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

Parameters

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

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

Return type tf.Tensor

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

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

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

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

Parameters

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

Returns

Return type list(ZfitParameters)

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

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*) – name of the operation shown in the `tf.Graph`

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

```
partial_analytic_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_analytic_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

```
partial_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

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

```
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 (*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.

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'`) \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.

as_pdf () \rightarrow `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`) \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_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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape ()

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*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.

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.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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class*‘*tf.Tensor*’: the integral value as a scalar with shape **()**

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*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.

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` (*func*: Callable, *obs*: Union[str, Iterable[str], zfit.Space], *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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class*‘tf.Tensor’: the integral value as a scalar with shape **()**

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*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.

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*=None, *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*) \rightarrow `zfit.core.interfaces.ZfitObject`

data

errordef

fit_range

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`

gradients (*params*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]` = `None`) \rightarrow `List[tensorflow.python.framework.ops.Tensor]`

model

name

Name prepended to all ops created by this *model*.

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

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

Parameters `() (cacher) -`

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

reset_cache_self (`()`)

Clear the cache of self and all dependent cachers.

value (`()`)

```
class zfit.loss.UnbinnedNLL(model, data, fit_range=None, constraints=None)
```

```
    Bases: zfit.core.loss.CachedLoss
```

The Unbinned Negative Log Likelihood.

```
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*.

```
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]
```

```
model
```

```
name
```

Name prepended to all ops created by this *model*.

```
register_cacher (cache: Union[zfit.core.interfaces.ZfitCachable, Iter-  
                able[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.

```
value ()
```

```
class zfit.loss.BaseLoss(model, data, fit_range: Union[Tuple[Tuple[Tuple[float,  
                    ...]]], Tuple[float, float], bool] = None, constraints:  
                        List[tensorflow.python.framework.ops.Tensor] = None)  
    Bases: zfit.core.baseobject.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*]

model

name

Name prepended to all ops created by this *model*.

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

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

Parameters *()* (*cacher*) –

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

reset_cache_self()

Clear the cache of self and all dependent cachers.

value()

class `zfit.loss.SimpleLoss` (*func: Callable, dependents: Optional[Iterable[zfit.Parameter]] = None, errordef: Optional[float] = None*)

Bases: `zfit.core.loss.CachedLoss`

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]`

model

name

Name prepended to all ops created by this *model*.

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 `()` (*cache*) –

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

reset_cache_self ()

Clear the cache of self and all dependent cachers.

value ()

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`

sess

set_sess (*sess*: `tensorflow.python.client.session.Session`)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (`tf.compat.v1.Session`) –

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`

sess

set_sess (*sess*: `tensorflow.python.client.session.Session`)

Set the session (temporarily) for this instance. If `None`, the auto-created default is taken.

Parameters **sess** (`tf.compat.v1.Session`) –

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_` `low_non_cachable` if `False`.

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`

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.

sess

set_sess (*sess*: `tensorflow.python.client.session.Session`)
Set the session (temporarily) for this instance. If `None`, the auto-created default is taken.

Parameters **sess** (`tf.compat.v1.Session`) –

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*

sess

set_sess (*sess*: *tensorflow.python.client.session.Session*)

Set the session (temporarily) for this instance. If None, the auto-created default is taken.

Parameters **sess** (*tf.compat.v1.Session*) –

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*)

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)`**independent****load** (*value*, *session=None*)**name**

The name of the object.

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 ()**pdf**

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

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

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

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

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

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

Parameters

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

Returns numerical

as_func (*norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False`)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → `zfit.core.basepdf.BasePDF`

Creates a copy of the model.

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

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

Returns

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

Return type model

create_extended (*yield_*: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], *name_addition*='_extended') → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

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

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

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

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

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

get_params (*only_floating*: *bool* = *False*, *names*: *Union*[*str*, *List*[*str*], *None*] = *None*) -> *List*[*zfit.core.interfaces.ZfitParameter*]

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

Parameters

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

Returns

Return type *list*(*ZfitParameters*)

get_yield () -> *Optional*[*zfit.core.parameter.Parameter*]

Return the yield (only for extended models).

Returns the yield of the current model or *None*

Return type *Parameter*

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

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape *()*

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.BaseFuncтор` (*pdfs, name='BaseFuncтор', **kwargs*)

Bases: `zfit.models.basefuncтор.FuncторMixin`, `zfit.core.basepdf.BasePDF`

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type `Tensor`

Raises

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

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

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

Parameters

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

Returns numerical

as_func (*norm_range*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

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

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

Returns

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

Return type model

create_extended (*yield_*: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], *name_addition*='_extended') → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

Parameters *limits_to_integrate* (*Space*) –

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type ZfitPDF

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

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

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

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

get_models (*names*=None) → List[zfit.core.interfaces.ZfitModel]

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

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

Parameters

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

Returns

Return type `list(ZfitParameters)`

get_yield `()` → `Optional[zfit.core.parameter.Parameter]`
Return the yield (only for extended models).

Returns the yield of the current model or None

Return type `Parameter`

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

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`) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = `'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
 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 tf.Tensor of type *self.dtype*.

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 () (*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.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'*) \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`) \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_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)

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) – name of the operation shown in the tf.Graph

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = `'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: `str = 'model'`) → `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type *self.dtype*.

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, Iter-
able[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.CrystalBall(mu: Union[zfit.core.interfaces.ZfitParameter, int, float,
complex, tensorflow.python.framework.ops.Tensor], sigma:
Union[zfit.core.interfaces.ZfitParameter, int, float, complex,
tensorflow.python.framework.ops.Tensor], alpha:
Union[zfit.core.interfaces.ZfitParameter, int, float, complex,
tensorflow.python.framework.ops.Tensor], n:
Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], obs: Union[str, Iterable[str],
zfit.Space], name: str = 'CrystalBall', dtype: Type[CT_co] =
tf.float64)
```

Bases: `zfit.core.basepdf.BasePDF`

‘Crystal Ball shaped PDF’_. A combination of a Gaussian with an powerlaw tail.

The function is defined as follows:

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

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, 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`)

integrate (*limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'integrate'`) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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*) – 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'`) \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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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

pdf (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: `str = 'model'`) \rightarrow `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type *self.dtype*.

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 (*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.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], alpha: 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], alphan: 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
- **alpha_l** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the left
- **side** –
- **n_l** (*zfit.Parameter*) – Exponent of the powertail on the left side
- **alpha_r** (*zfit.Parameter*) – parameter where to switch from a gaussian to the powertail on the right
- **side** –
- **n_r** (*zfit.Parameter*) – Exponent of the powertail on the right side
- **obs** (*Space*) –
- **name** (*str*) –
- **dtype** (*tf.DType*) –

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

- **name** (*str*) –

Returns the integral value

Return type Tensor

Raises

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

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

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

Parameters

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

Returns numerical

as_func (*norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False`)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → `zfit.core.basepdf.BasePDF`

Creates a copy of the model.

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

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

Returns

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

Return type model

```
create_extended (yield_: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], name_addition='_extended') → zfit.core.interfaces.ZfitPDF
```

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

```
create_projection_pdf (limits_to_integrate: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF
```

Create a PDF projection by integrating out some of the dimensions.

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

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

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

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

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

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

get_params (*only_floating*: *bool* = *False*, *names*: *Union*[*str*, *List*[*str*], *None*] = *None*) -> *List*[*zfit.core.interfaces.ZfitParameter*]

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

Parameters

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

Returns

Return type *list*(*ZfitParameters*)

get_yield () -> *Optional*[*zfit.core.parameter.Parameter*]

Return the yield (only for extended models).

Returns the yield of the current model or *None*

Return type *Parameter*

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

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape **()**

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.

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

$$f(x | \mu, \sigma^2) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

with the normalization over `[-inf, inf]` of

$$\frac{1}{\sqrt{2\pi\sigma^2}}$$

The normalization changes for different normalization ranges

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type Tensor

Raises

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

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

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

Parameters

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

Returns numerical

as_func (*norm_range*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = False)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → *zfit.core.basepdf.BasePDF*

Creates a copy of the model.

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

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

Returns

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

Return type `model`

create_extended (*yield_*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *name_addition*=`'_extended'`) → `zfit.core.interfaces.ZfitPDF`

Return an extended version of this pdf with `yield_`. 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class‘tf.Tensor‘`: the integral value as a scalar with shape *()*

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*, bool) – the limits to normalize over
- **name** (*str*) –

Returns the integral value

Return type Tensor

obs

Return the observables.

params

```
partial_analytic_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_analytic_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

```
partial_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

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

```
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 (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
```

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 tf.Tensor of type *self.dtype*.

```
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.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_`. 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`)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.TruncatedGauss` (*mu*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *sigma*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *low*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *high*: `Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor]`, *obs*: `Union[str, Iterable[str], zfit.Space]`, *name*: `str = 'TruncatedGauss'`)

Bases: `zfit.models.dist_tfp.WrapDistribution`

Gaussian distribution that is 0 outside of *low*, *high*. Equivalent to the product of Gauss and Uniform.

Parameters

- **mu** (*Parameter*) – Mean of the gaussian dist
- **sigma** (*Parameter*) – Standard deviation or spread of the gaussian
- **low** (*Parameter*) – Below this value, the pdf is zero.

- **high** (*Parameter*) – Above this value, the pdf is zero.
- **obs** (*Space*) – Observables and normalization range the pdf is defined in
- **name** (*str*) – Name of the pdf

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

Add dependents that render the cache invalid if they change.

Parameters

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

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

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

Analytical integration over function and raise Error if not possible.

Parameters

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

Returns the integral value

Return type *Tensor*

Raises

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

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

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

Parameters

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

Returns *numerical*

as_func (*norm_range*: *Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]* = *False*)

Return a *Function* with the function *model(x, norm_range=norm_range)*.

Parameters **()** (*norm_range*) –

axes

Return the axes.

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

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

Parameters

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

Returns:

copy (***override_parameters*) → zfit.core.basepdf.BasePDF

Creates a copy of the model.

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

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

Returns

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

Return type model

create_extended (*yield_*: Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensorflow.python.framework.ops.Tensor], *name_addition*='_extended') → zfit.core.interfaces.ZfitPDF

Return an extended version of this pdf with yield *yield_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

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

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

distribution**dtype**

The dtype of the object

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

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

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

get_params (*only_floating: bool = False*, *names: Union[str, List[str], None] = None*) -> *List[zfit.core.interfaces.ZfitParameter]*

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

Parameters

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

Returns

Return type *list(ZfitParameters)*

get_yield () -> *Optional[zfit.core.parameter.Parameter]*

Return the yield (only for extended models).

Returns the yield of the current model or *None*

Return type *Parameter*

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

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.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_*. 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*)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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_`. 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)

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) – name of the operation shown in the tf.Graph

Returns py:class‘tf.Tensor’: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = `'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *norm_range*: `Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None`, *name*: `str = 'model'`) → `Union[float, tensorflow.python.framework.ops.Tensor]`
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 `tf.Tensor` of type *self.dtype*.

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') → 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*)

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`) – name of the operation shown in the `tf.Graph`

Returns py:class`tf.Tensor`: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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.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 *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)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape ()

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

```
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.Hermite(obs, coeffs: list, apply_scaling: bool = True, coeff0:
    Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensor-
    flow.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, 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.

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*)

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*) – name of the operation shown in the *tf.Graph*

Returns *py:class* 'tf.Tensor': the integral value as a scalar with shape **()**

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.Laguerre(obs, coeffs: list, apply_scaling: bool = True, coeff0:  
    Union[zfit.core.interfaces.ZfitParameter, int, float, complex, tensor-  
    flow.python.framework.ops.Tensor, None] = None, name: str = 'La-  
    guerre')
```

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, 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.

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`)

integrate (*limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'integrate'`) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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.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`)

integrate (*limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'integrate'`) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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') → 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_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*)

integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'integrate'*) \rightarrow `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*) – name of the operation shown in the `tf.Graph`

Returns `py:class'tf.Tensor'`: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_analytic_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]` = None, *name*: *str* = 'partial_integrate') → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

```
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 (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
```

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 tf.Tensor of type *self.dtype*.

pdfs_extended

```
classmethod register_additional_repr (**kwargs)
```

Register an additional attribute to add to the repr.

Parameters

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

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

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:

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

register_cacher (*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.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'`) \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_*. The parameters are shared.

Parameters

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

Returns *ZfitPDF*

create_projection_pdf (*limits_to_integrate*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool]) → zfit.core.interfaces.ZfitPDF

Create a PDF projection by integrating out some of the dimensions.

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

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

Returns a pdf without the dimensions from *limits_to_integrate*.

Return type *ZfitPDF*

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

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

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

Parameters

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

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

Raises

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

dtype

The dtype of the object

fracs

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

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

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

get_models (*names=None*) → `List[zfit.core.interfaces.ZfitModel]`

get_params (*only_floating: bool = False, names: Union[str, List[str], None] = None*) → `List[zfit.core.interfaces.ZfitParameter]`

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

Parameters

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

Returns

Return type `list(ZfitParameters)`

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

Return the yield (only for extended models).

Returns the yield of the current model or `None`

Return type *Parameter*

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

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *limits*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], *norm_range*: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, *name*: str = 'partial_analytic_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (*x*: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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.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`)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

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

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

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

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]

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 tf.Tensor of type *self.dtype*.

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*, *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`)

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*) – name of the operation shown in the `tf.Graph`

Returns `py:class`tf.Tensor``: the integral value as a scalar with shape `()`

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.

params

partial_analytic_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_analytic_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

partial_integrate (*x*: `Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data]`, *limits*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool]`, *norm_range*: `Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None`, *name*: `str = 'partial_integrate'`) → `Union[tensorflow.python.framework.ops.Tensor, zfit.Data]`

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

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 (x : Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], *norm_range*: Union[Tuple[Tuple[Tuple[float, ...]]], Tuple[float, float], bool] = None, *name*: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
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 tf.Tensor of type *self.dtype*.

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.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_`. 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_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*)

integrate (*limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'integrate'*) \rightarrow `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`) – name of the operation shown in the `tf.Graph`

Returns `py:class‘tf.Tensor‘`: the integral value as a scalar with shape `()`

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.

params


```
partial_analytic_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_analytic_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

Do analytical 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

Raises

- `NotImplementedError` – if the *analytic* integral (over this limits) is not implemented
- `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.

```
partial_integrate (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], limits: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool], norm_range: Union[Tuple[Tuple[float, ...]], Tuple[float, ...], bool] = None, name: str = 'partial_integrate') → Union[tensorflow.python.framework.ops.Tensor, zfit.Data]
```

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

```
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 (x: Union[numpy.ndarray, tensorflow.python.framework.ops.Tensor, zfit.Data], norm_range: Union[Tuple[Tuple[Tuple[float, ...]], Tuple[float, float], bool] = None, name: str = 'model') → Union[float, tensorflow.python.framework.ops.Tensor]
```

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 tf.Tensor of type *self.dtype*.

pdfs_extended

```
classmethod register_additional_repr (**kwargs)
```

Register an additional attribute to add to the repr.

Parameters

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

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

Register an analytic integral with the class.

Parameters

- **func** (*callable*) – A function that calculates the (partial) integral over the axes *limits*. The signature has to be the following:

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

register_cacher (*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*) –

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`, 382
- `zfit.core`, 47
 - `basefunc`, 47
 - `basemodel`, 53
 - `baseobject`, 59
 - `basepdf`, 60
 - `constraint`, 69
 - `data`, 74
 - `dimension`, 84
 - `integration`, 86
 - `interfaces`, 89
 - `limits`, 98
 - `loss`, 104
 - `math`, 109
 - `operations`, 109
 - `parameter`, 110
 - `sample`, 143
 - `testing`, 148
- `zfit.data`, 385
- `zfit.func`, 388
- `zfit.loss`, 413
- `zfit.minimize`, 417
- `zfit.minimizers`, 149
 - `base_tf`, 149
 - `baseminimizer`, 149
 - `fitresult`, 151
 - `interface`, 153
 - `minimizer_minuit`, 154
 - `minimizer_tfp`, 155
 - `minimizers_scipy`, 156
 - `optimizers_tf`, 156
 - `tf_external_optimizer`, 157
- `zfit.models`, 160
 - `basefunctor`, 160
 - `basic`, 166
 - `dist_tfp`, 182
 - `functions`, 223
 - `functor`, 253
 - `physics`, 277
 - `polynomials`, 294
 - `special`, 346
- `zfit.param`, 420
- `zfit.pdf`, 421
- `zfit.sample`, 584
- `zfit.settings`, 584
- `zfit.util`, 370
 - `cache`, 370
 - `checks`, 371
 - `container`, 372
 - `diverse`, 373
 - `exception`, 373
 - `execution`, 377
 - `graph`, 378
 - `logging`, 378
 - `temporary`, 379
 - `ztyping`, 380
- `zfit.ztf`, 380
 - `random`, 380
 - `tools`, 380
 - `wrapping_tf`, 380
 - `zextension`, 381

Symbols

`__iter__()` (*zfit.Parameter* method), 32
`__iter__()` (*zfit.core.parameter.ComposedResourceVariable* method), 116
`__iter__()` (*zfit.core.parameter.Parameter* method), 125
`__iter__()` (*zfit.core.parameter.TFBaseVariable* method), 135

A

`abs_square()` (in module *zfit.ztf.zextension*), 381
`accept_reject_sample()` (in module *zfit.core.sample*), 147
Adam (class in *zfit.minimize*), 417
Adam (class in *zfit.minimizers.optimizers_tf*), 156
AdamMinimizer (in module *zfit.minimize*), 417
`add()` (in module *zfit.core.operations*), 109
`add()` (*zfit.core.limits.Space* method), 99
`add()` (*zfit.core.sample.EventSpace* method), 143
`add()` (*zfit.Space* method), 42
`add_cache_dependents()` (*zfit.ComplexParameter* method), 41
`add_cache_dependents()` (*zfit.ComposedParameter* method), 39
`add_cache_dependents()` (*zfit.constraint.GaussianConstraint* method), 384
`add_cache_dependents()` (*zfit.constraint.SimpleConstraint* method), 383
`add_cache_dependents()` (*zfit.core.basefunc.BaseFunc* method), 47
`add_cache_dependents()` (*zfit.core.basemodel.BaseModel* method), 53
`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), 69
`add_cache_dependents()` (*zfit.core.constraint.DistributionConstraint* method), 70
`add_cache_dependents()` (*zfit.core.constraint.GaussianConstraint* method), 72
`add_cache_dependents()` (*zfit.constraint.SimpleConstraint* method), 73
`add_cache_dependents()` (*zfit.core.data.Data* method), 74
`add_cache_dependents()` (*zfit.core.data.SampleData* method), 78
`add_cache_dependents()` (*zfit.core.data.Sampler* method), 81
`add_cache_dependents()` (*zfit.core.loss.BaseLoss* method), 104
`add_cache_dependents()` (*zfit.core.loss.CachedLoss* method), 105
`add_cache_dependents()` (*zfit.core.loss.ExtendedUnbinnedNLL* method), 106
`add_cache_dependents()` (*zfit.core.loss.SimpleLoss* method), 107
`add_cache_dependents()` (*zfit.core.loss.UnbinnedNLL* method), 108
`add_cache_dependents()` (*zfit.core.parameter.BaseComposedParameter* method), 111
`add_cache_dependents()` (*zfit.core.parameter.BaseZParameter* method), 112
`add_cache_dependents()` (*zfit.core.parameter.ComplexParameter* method), 113
`add_cache_dependents()` (*zfit.core.parameter.ComposedParameter*

[method](#)), 114
[add_cache_dependents\(\)](#) ([zfit.core.parameter.ConstantParameter](#) [method](#)), 123
[add_cache_dependents\(\)](#) ([zfit.core.parameter.Parameter](#) [method](#)), 125
[add_cache_dependents\(\)](#) ([zfit.core.parameter.ZfitParameterMixin](#) [method](#)), 141
[add_cache_dependents\(\)](#) ([zfit.data.Data](#) [method](#)), 385
[add_cache_dependents\(\)](#) ([zfit.func.BaseFunc](#) [method](#)), 388
[add_cache_dependents\(\)](#) ([zfit.func.ProdFunc](#) [method](#)), 395
[add_cache_dependents\(\)](#) ([zfit.func.SimpleFunc](#) [method](#)), 407
[add_cache_dependents\(\)](#) ([zfit.func.SumFunc](#) [method](#)), 401
[add_cache_dependents\(\)](#) ([zfit.loss.BaseLoss](#) [method](#)), 415
[add_cache_dependents\(\)](#) ([zfit.loss.ExtendedUnbinnedNLL](#) [method](#)), 413
[add_cache_dependents\(\)](#) ([zfit.loss.SimpleLoss](#) [method](#)), 416
[add_cache_dependents\(\)](#) ([zfit.loss.UnbinnedNLL](#) [method](#)), 414
[add_cache_dependents\(\)](#) ([zfit.minimize.Minuit](#) [method](#)), 418
[add_cache_dependents\(\)](#) ([zfit.minimizers.minimizer_minuit.Minuit](#) [method](#)), 154
[add_cache_dependents\(\)](#) ([zfit.models.basefunctor.FunctorMixin](#) [method](#)), 160
[add_cache_dependents\(\)](#) ([zfit.models.basic.CustomGaussOLD](#) [method](#)), 166
[add_cache_dependents\(\)](#) ([zfit.models.basic.Exponential](#) [method](#)), 174
[add_cache_dependents\(\)](#) ([zfit.models.dist_tfp.ExponentialTFP](#) [method](#)), 182
[add_cache_dependents\(\)](#) ([zfit.models.dist_tfp.Gauss](#) [method](#)), 190
[add_cache_dependents\(\)](#) ([zfit.models.dist_tfp.TruncatedGauss](#) [method](#)), 199
[add_cache_dependents\(\)](#) ([zfit.models.dist_tfp.Uniform](#) [method](#)), 207
[add_cache_dependents\(\)](#) ([zfit.models.dist_tfp.WrapDistribution](#) [method](#)), 215
[add_cache_dependents\(\)](#) ([zfit.models.functions.BaseFunctorFunc](#) [method](#)), 223
[add_cache_dependents\(\)](#) ([zfit.models.functions.ProdFunc](#) [method](#)), 229
[add_cache_dependents\(\)](#) ([zfit.models.functions.SimpleFunc](#) [method](#)), 235
[add_cache_dependents\(\)](#) ([zfit.models.functions.SumFunc](#) [method](#)), 241
[add_cache_dependents\(\)](#) ([zfit.models.functions.ZFunc](#) [method](#)), 247
[add_cache_dependents\(\)](#) ([zfit.models.functor.BaseFunctor](#) [method](#)), 253
[add_cache_dependents\(\)](#) ([zfit.models.functor.ProductPDF](#) [method](#)), 261
[add_cache_dependents\(\)](#) ([zfit.models.functor.SumPDF](#) [method](#)), 269
[add_cache_dependents\(\)](#) ([zfit.models.physics.CrystalBall](#) [method](#)), 278
[add_cache_dependents\(\)](#) ([zfit.models.physics.DoubleCB](#) [method](#)), 286
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.Chebyshev](#) [method](#)), 295
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.Chebyshev2](#) [method](#)), 303
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.Hermite](#) [method](#)), 312
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.Laguerre](#) [method](#)), 320
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.Legendre](#) [method](#)), 328
[add_cache_dependents\(\)](#) ([zfit.models.polynomials.RecursivePolynomial](#) [method](#)), 337
[add_cache_dependents\(\)](#) ([zfit.models.special.SimpleFunctorPDF](#) [method](#)), 346
[add_cache_dependents\(\)](#) ([zfit.models.special.SimplePDF](#) [method](#)), 354
[add_cache_dependents\(\)](#)

- [\(zfit.models.special.ZPDF method\), 362](#)
- [add_cache_dependents\(\) \(zfit.param.ConstantParameter method\), 420](#)
- [add_cache_dependents\(\) \(zfit.Parameter method\), 32](#)
- [add_cache_dependents\(\) \(zfit.pdf.BaseFuncтор method\), 429](#)
- [add_cache_dependents\(\) \(zfit.pdf.BasePDF method\), 421](#)
- [add_cache_dependents\(\) \(zfit.pdf.Chebyshev method\), 495](#)
- [add_cache_dependents\(\) \(zfit.pdf.Chebyshev2 method\), 512](#)
- [add_cache_dependents\(\) \(zfit.pdf.CrystalBall method\), 445](#)
- [add_cache_dependents\(\) \(zfit.pdf.DoubleCB method\), 454](#)
- [add_cache_dependents\(\) \(zfit.pdf.Exponential method\), 437](#)
- [add_cache_dependents\(\) \(zfit.pdf.Gauss method\), 462](#)
- [add_cache_dependents\(\) \(zfit.pdf.Hermite method\), 520](#)
- [add_cache_dependents\(\) \(zfit.pdf.Laguerre method\), 528](#)
- [add_cache_dependents\(\) \(zfit.pdf.Legendre method\), 503](#)
- [add_cache_dependents\(\) \(zfit.pdf.ProductPDF method\), 544](#)
- [add_cache_dependents\(\) \(zfit.pdf.RecursivePolynomial method\), 536](#)
- [add_cache_dependents\(\) \(zfit.pdf.SimpleFuncторPDF method\), 576](#)
- [add_cache_dependents\(\) \(zfit.pdf.SimplePDF method\), 568](#)
- [add_cache_dependents\(\) \(zfit.pdf.SumPDF method\), 553](#)
- [add_cache_dependents\(\) \(zfit.pdf.TruncatedGauss method\), 479](#)
- [add_cache_dependents\(\) \(zfit.pdf.Uniform method\), 470](#)
- [add_cache_dependents\(\) \(zfit.pdf.WrapDistribution method\), 486](#)
- [add_cache_dependents\(\) \(zfit.pdf.ZPDF method\), 561](#)
- [add_cache_dependents\(\) \(zfit.util.cache.Cachable method\), 370](#)
- [add_cache_dependents\(\) \(zfit.util.cache.ZfitCachable method\), 371](#)
- [add_constraints\(\) \(zfit.core.interfaces.ZfitLoss method\), 92](#)
- [add_constraints\(\) \(zfit.core.loss.BaseLoss method\), 104](#)
- [add_constraints\(\) \(zfit.core.loss.CachedLoss method\), 105](#)
- [add_constraints\(\) \(zfit.core.loss.ExtendedUnbinnedNLL method\), 106](#)
- [add_constraints\(\) \(zfit.core.loss.SimpleLoss method\), 107](#)
- [add_constraints\(\) \(zfit.core.loss.UnbinnedNLL method\), 108](#)
- [add_constraints\(\) \(zfit.loss.BaseLoss method\), 415](#)
- [add_constraints\(\) \(zfit.loss.ExtendedUnbinnedNLL method\), 413](#)
- [add_constraints\(\) \(zfit.loss.SimpleLoss method\), 416](#)
- [add_constraints\(\) \(zfit.loss.UnbinnedNLL method\), 414](#)
- [add_func_func\(\) \(in module zfit.core.operations\), 109](#)
- [add_param_func\(\) \(in module zfit.core.operations\), 109](#)
- [add_param_param\(\) \(in module zfit.core.operations\), 110](#)
- [add_pdf_pdf\(\) \(in module zfit.core.operations\), 110](#)
- [add_spaces\(\) \(in module zfit.core.dimension\), 85](#)
- [aggregation \(zfit.core.parameter.ComposedResourceVariable attribute\), 116](#)
- [aggregation \(zfit.core.parameter.Parameter attribute\), 125](#)
- [aggregation \(zfit.core.parameter.TFBaseVariable attribute\), 135](#)
- [aggregation \(zfit.Parameter attribute\), 32](#)
- [all_parents\(\) \(in module zfit.util.graph\), 378](#)
- [AlreadyExtendedPDFError, 373](#)
- [analytic_integrate\(\) \(zfit.core.basefunc.BaseFunc method\), 47](#)
- [analytic_integrate\(\) \(zfit.core.basemodel.BaseModel method\), 53](#)
- [analytic_integrate\(\) \(zfit.core.basepdf.BasePDF method\), 61](#)
- [analytic_integrate\(\) \(zfit.func.BaseFunc method\), 389](#)
- [analytic_integrate\(\) \(zfit.func.ProdFunc method\), 395](#)
- [analytic_integrate\(\) \(zfit.func.SimpleFunc method\), 407](#)
- [analytic_integrate\(\) \(zfit.func.SumFunc method\), 401](#)
- [analytic_integrate\(\) \(zfit.models.basefuncтор.FuncторMixin method\), 160](#)
- [analytic_integrate\(\)](#)

[\(zfit.models.basic.CustomGaussOLD method\), 166](#)
[analytic_integrate\(\)](#)
[\(zfit.models.basic.Exponential method\), 174](#)
[analytic_integrate\(\)](#)
[\(zfit.models.dist_tfp.ExponentialTFP method\), 182](#)
[analytic_integrate\(\)](#) [\(zfit.models.dist_tfp.Gauss method\), 190](#)
[analytic_integrate\(\)](#)
[\(zfit.models.dist_tfp.TruncatedGauss method\), 199](#)
[analytic_integrate\(\)](#)
[\(zfit.models.dist_tfp.Uniform method\), 207](#)
[analytic_integrate\(\)](#)
[\(zfit.models.dist_tfp.WrapDistribution method\), 215](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functions.BaseFunc method\), 223](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functions.ProdFunc method\), 229](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functions.SimpleFunc method\), 235](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functions.SumFunc method\), 241](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functions.ZFunc method\), 247](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functor.BaseFunc method\), 253](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functor.ProductPDF method\), 261](#)
[analytic_integrate\(\)](#)
[\(zfit.models.functor.SumPDF method\), 269](#)
[analytic_integrate\(\)](#)
[\(zfit.models.physics.CrystalBall method\), 278](#)
[analytic_integrate\(\)](#)
[\(zfit.models.physics.DoubleCB method\), 287](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.Chebyshev method\), 295](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.Chebyshev2 method\), 304](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.Hermite method\), 312](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.Laguerre method\), 320](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.Legendre method\), 329](#)
[analytic_integrate\(\)](#)
[\(zfit.models.polynomials.RecursivePolynomial method\), 337](#)
[analytic_integrate\(\)](#)
[\(zfit.models.special.SimpleFuncPDF method\), 346](#)
[analytic_integrate\(\)](#)
[\(zfit.models.special.SimplePDF method\), 354](#)
[analytic_integrate\(\)](#) [\(zfit.models.special.ZPDF method\), 362](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.BaseFunc method\), 429](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.BasePDF method\), 421](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Chebyshev method\), 495](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Chebyshev2 method\), 512](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.CrystalBall method\), 446](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.DoubleCB method\), 454](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Exponential method\), 437](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Gauss method\), 462](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Hermite method\), 520](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Laguerre method\), 529](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Legendre method\), 504](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.ProductPDF method\), 545](#)
[analytic_integrate\(\)](#)
[\(zfit.pdf.RecursivePolynomial method\), 537](#)
[analytic_integrate\(\)](#)
[\(zfit.pdf.SimpleFuncPDF method\), 576](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.SimplePDF method\), 569](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.SumPDF method\), 553](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.TruncatedGauss method\), 479](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.Uniform method\), 471](#)
[analytic_integrate\(\)](#) [\(zfit.pdf.WrapDistribution](#)

- method*), 487
- `analytic_integrate()` (*zfit.pdf.ZPDF method*), 561
- `AnalyticIntegral` (*class in zfit.core.integration*), 86
- `Any` (*class in zfit.core.limits*), 98
- `ANY` (*zfit.core.limits.Space attribute*), 99
- `ANY` (*zfit.core.sample.EventSpace attribute*), 143
- `ANY` (*zfit.Space attribute*), 42
- `ANY_LOWER` (*zfit.core.limits.Space attribute*), 99
- `ANY_LOWER` (*zfit.core.sample.EventSpace attribute*), 143
- `ANY_LOWER` (*zfit.Space attribute*), 42
- `ANY_UPPER` (*zfit.core.limits.Space attribute*), 99
- `ANY_UPPER` (*zfit.core.sample.EventSpace attribute*), 143
- `ANY_UPPER` (*zfit.Space attribute*), 42
- `AnyLower` (*class in zfit.core.limits*), 99
- `AnyUpper` (*class in zfit.core.limits*), 99
- `apply_yield()` (*zfit.core.basepdf.BasePDF method*), 62
- `apply_yield()` (*zfit.models.basic.CustomGaussOLD method*), 166
- `apply_yield()` (*zfit.models.basic.Exponential method*), 175
- `apply_yield()` (*zfit.models.dist_tfp.ExponentialTFP method*), 183
- `apply_yield()` (*zfit.models.dist_tfp.Gauss method*), 191
- `apply_yield()` (*zfit.models.dist_tfp.TruncatedGauss method*), 199
- `apply_yield()` (*zfit.models.dist_tfp.Uniform method*), 207
- `apply_yield()` (*zfit.models.dist_tfp.WrapDistribution method*), 215
- `apply_yield()` (*zfit.models.functor.BaseFunctor method*), 253
- `apply_yield()` (*zfit.models.functor.ProductPDF method*), 261
- `apply_yield()` (*zfit.models.functor.SumPDF method*), 270
- `apply_yield()` (*zfit.models.physics.CrystalBall method*), 278
- `apply_yield()` (*zfit.models.physics.DoubleCB method*), 287
- `apply_yield()` (*zfit.models.polynomials.Chebyshev method*), 296
- `apply_yield()` (*zfit.models.polynomials.Chebyshev2 method*), 304
- `apply_yield()` (*zfit.models.polynomials.Hermite method*), 312
- `apply_yield()` (*zfit.models.polynomials.Laguerre method*), 321
- `apply_yield()` (*zfit.models.polynomials.Legendre method*), 329
- `apply_yield()` (*zfit.models.polynomials.RecursivePolynomial method*), 337
- `apply_yield()` (*zfit.models.special.SimpleFunctorPDF method*), 347
- `apply_yield()` (*zfit.models.special.SimplePDF method*), 354
- `apply_yield()` (*zfit.models.special.ZPDF method*), 362
- `apply_yield()` (*zfit.pdf.BaseFunctor method*), 429
- `apply_yield()` (*zfit.pdf.BasePDF method*), 422
- `apply_yield()` (*zfit.pdf.Chebyshev method*), 496
- `apply_yield()` (*zfit.pdf.Chebyshev2 method*), 512
- `apply_yield()` (*zfit.pdf.CrystalBall method*), 446
- `apply_yield()` (*zfit.pdf.DoubleCB method*), 455
- `apply_yield()` (*zfit.pdf.Exponential method*), 438
- `apply_yield()` (*zfit.pdf.Gauss method*), 463
- `apply_yield()` (*zfit.pdf.Hermite method*), 521
- `apply_yield()` (*zfit.pdf.Laguerre method*), 529
- `apply_yield()` (*zfit.pdf.Legendre method*), 504
- `apply_yield()` (*zfit.pdf.ProductPDF method*), 545
- `apply_yield()` (*zfit.pdf.RecursivePolynomial method*), 537
- `apply_yield()` (*zfit.pdf.SimpleFunctorPDF method*), 577
- `apply_yield()` (*zfit.pdf.SimplePDF method*), 569
- `apply_yield()` (*zfit.pdf.SumPDF method*), 553
- `apply_yield()` (*zfit.pdf.TruncatedGauss method*), 479
- `apply_yield()` (*zfit.pdf.Uniform method*), 471
- `apply_yield()` (*zfit.pdf.WrapDistribution method*), 487
- `apply_yield()` (*zfit.pdf.ZPDF method*), 561
- `acquire_cpu()` (*zfit.util.execution.RunManager method*), 377
- `area()` (*zfit.core.interfaces.ZfitSpace method*), 97
- `area()` (*zfit.core.limits.Space method*), 99
- `area()` (*zfit.core.sample.EventSpace method*), 143
- `area()` (*zfit.Space method*), 42
- `arg` (*zfit.ComplexParameter attribute*), 41
- `arg` (*zfit.core.parameter.ComplexParameter attribute*), 113
- `args` (*zfit.minimizers.baseminimizer.FailMinimizeNaN attribute*), 150
- `args` (*zfit.util.exception.AlreadyExtendedPDFError attribute*), 373
- `args` (*zfit.util.exception.AxesNotSpecifiedError attribute*), 373
- `args` (*zfit.util.exception.AxesNotUnambiguousError attribute*), 373
- `args` (*zfit.util.exception.BasePDFSubclassingError attribute*), 373
- `args` (*zfit.util.exception.ConversionError attribute*), 373
- `args` (*zfit.util.exception.DueToLazynessNotImplementedError attribute*), 373
- `args` (*zfit.util.exception.ExtendedPDFError attribute*), 374

```

args (zfit.util.exception.IncompatibleError attribute), 374
args (zfit.util.exception.IntentionNotUnambiguousError attribute), 374
args (zfit.util.exception.LimitsIncompatibleError attribute), 374
args (zfit.util.exception.LimitsNotSpecifiedError attribute), 374
args (zfit.util.exception.LimitsOverdefinedError attribute), 374
args (zfit.util.exception.LimitsUnderdefinedError attribute), 374
args (zfit.util.exception.LogicalUndefinedOperationError attribute), 374
args (zfit.util.exception.ModelIncompatibleError attribute), 375
args (zfit.util.exception.MultipleLimitsNotImplementedError attribute), 375
args (zfit.util.exception.NameAlreadyTakenError attribute), 375
args (zfit.util.exception.NormRangeNotImplementedError attribute), 375
args (zfit.util.exception.NormRangeNotSpecifiedError attribute), 375
args (zfit.util.exception.NoSessionSpecifiedError attribute), 375
args (zfit.util.exception.NotExtendedPDFError attribute), 375
args (zfit.util.exception.NotMinimizedError attribute), 375
args (zfit.util.exception.NotSpecifiedError attribute), 376
args (zfit.util.exception.ObsIncompatibleError attribute), 376
args (zfit.util.exception.ObsNotSpecifiedError attribute), 376
args (zfit.util.exception.OverdefinedError attribute), 376
args (zfit.util.exception.PDFCompatibilityError attribute), 376
args (zfit.util.exception.ShapeIncompatibleError attribute), 376
args (zfit.util.exception.SpaceIncompatibleError attribute), 376
args (zfit.util.exception.SubclassingError attribute), 376
args (zfit.util.exception.UnderdefinedError attribute), 377
args (zfit.util.exception.WeightsNotImplementedError attribute), 377
as_func () (zfit.core.basepdf.BasePDF method), 62
as_func () (zfit.core.interfaces.ZfitPDF method), 94
as_func () (zfit.models.basic.CustomGaussOLD method), 167
as_func () (zfit.models.basic.Exponential method), 175
as_func () (zfit.models.dist_tfp.ExponentialTFP method), 183
as_func () (zfit.models.dist_tfp.Gauss method), 191
as_func () (zfit.models.dist_tfp.TruncatedGauss method), 199
as_func () (zfit.models.dist_tfp.Uniform method), 207
as_func () (zfit.models.dist_tfp.WrapDistribution method), 215
as_func () (zfit.models.functor.BaseFunctor method), 254
as_func () (zfit.models.functor.ProductPDF method), 261
as_func () (zfit.models.functor.SumPDF method), 270
as_func () (zfit.models.physics.CrystalBall method), 278
as_func () (zfit.models.physics.DoubleCB method), 287
as_func () (zfit.models.polynomials.Chebyshev method), 296
as_func () (zfit.models.polynomials.Chebyshev2 method), 304
as_func () (zfit.models.polynomials.Hermite method), 313
as_func () (zfit.models.polynomials.Laguerre method), 321
as_func () (zfit.models.polynomials.Legendre method), 329
as_func () (zfit.models.polynomials.RecursivePolynomial method), 337
as_func () (zfit.models.special.SimpleFunctorPDF method), 347
as_func () (zfit.models.special.SimplePDF method), 355
as_func () (zfit.models.special.ZPDF method), 363
as_func () (zfit.pdf.BaseFunctor method), 430
as_func () (zfit.pdf.BasePDF method), 422
as_func () (zfit.pdf.Chebyshev method), 496
as_func () (zfit.pdf.Chebyshev2 method), 513
as_func () (zfit.pdf.CrystalBall method), 446
as_func () (zfit.pdf.DoubleCB method), 455
as_func () (zfit.pdf.Exponential method), 438
as_func () (zfit.pdf.Gauss method), 463
as_func () (zfit.pdf.Hermite method), 521
as_func () (zfit.pdf.Laguerre method), 529
as_func () (zfit.pdf.Legendre method), 504
as_func () (zfit.pdf.ProductPDF method), 545
as_func () (zfit.pdf.RecursivePolynomial method), 537
as_func () (zfit.pdf.SimpleFunctorPDF method), 577
as_func () (zfit.pdf.SimplePDF method), 569
as_func () (zfit.pdf.SumPDF method), 553
as_func () (zfit.pdf.TruncatedGauss method), 479
as_func () (zfit.pdf.Uniform method), 471
as_func () (zfit.pdf.WrapDistribution method), 487
as_func () (zfit.pdf.ZPDF method), 561

```


- `as_pdf()` (`zfit.core.basefunc.BaseFunc` method), 48
- `as_pdf()` (`zfit.core.interfaces.ZfitFunc` method), 90
- `as_pdf()` (`zfit.func.BaseFunc` method), 389
- `as_pdf()` (`zfit.func.ProdFunc` method), 395
- `as_pdf()` (`zfit.func.SimpleFunc` method), 407
- `as_pdf()` (`zfit.func.SumFunc` method), 401
- `as_pdf()` (`zfit.models.functions.BaseFuncorFunc` method), 223
- `as_pdf()` (`zfit.models.functions.ProdFunc` method), 229
- `as_pdf()` (`zfit.models.functions.SimpleFunc` method), 235
- `as_pdf()` (`zfit.models.functions.SumFunc` method), 241
- `as_pdf()` (`zfit.models.functions.ZFunc` method), 247
- `assign()` (`zfit.ComplexParameter` method), 41
- `assign()` (`zfit.ComposedParameter` method), 40
- `assign()` (`zfit.core.parameter.BaseComposedParameter` method), 111
- `assign()` (`zfit.core.parameter.BaseZParameter` method), 112
- `assign()` (`zfit.core.parameter.ComplexParameter` method), 113
- `assign()` (`zfit.core.parameter.ComposedParameter` method), 115
- `assign()` (`zfit.core.parameter.ComposedResourceVariable` method), 116
- `assign()` (`zfit.core.parameter.ComposedVariable` method), 123
- `assign()` (`zfit.core.parameter.ConstantParameter` method), 123
- `assign()` (`zfit.core.parameter.Parameter` method), 125
- `assign()` (`zfit.core.parameter.TFBaseVariable` method), 135
- `assign()` (`zfit.core.parameter.ZfitBaseVariable` method), 141
- `assign()` (`zfit.param.ConstantParameter` method), 420
- `assign()` (`zfit.Parameter` method), 32
- `assign_add()` (`zfit.core.parameter.ComposedResourceVariable` method), 117
- `assign_add()` (`zfit.core.parameter.Parameter` method), 126
- `assign_add()` (`zfit.core.parameter.TFBaseVariable` method), 135
- `assign_add()` (`zfit.Parameter` method), 33
- `assign_sub()` (`zfit.core.parameter.ComposedResourceVariable` method), 117
- `assign_sub()` (`zfit.core.parameter.Parameter` method), 126
- `assign_sub()` (`zfit.core.parameter.TFBaseVariable` method), 135
- `assign_sub()` (`zfit.Parameter` method), 33
- `AUTO_FILL` (`zfit.core.limits.Space` attribute), 99
- `AUTO_FILL` (`zfit.core.sample.EventSpace` attribute), 143
- `AUTO_FILL` (`zfit.Space` attribute), 42
- `auto_initialize()` (`zfit.util.execution.RunManager` method), 377
- `auto_integrate()` (in module `zfit.core.integration`), 88
- `axes` (`zfit.core.basefunc.BaseFunc` attribute), 48
- `axes` (`zfit.core.basemodel.BaseModel` attribute), 54
- `axes` (`zfit.core.basepdf.BasePDF` attribute), 62
- `axes` (`zfit.core.data.Data` attribute), 75
- `axes` (`zfit.core.data.SampleData` attribute), 78
- `axes` (`zfit.core.data.Sampler` attribute), 81
- `axes` (`zfit.core.dimension.BaseDimensional` attribute), 84
- `axes` (`zfit.core.integration.PartialIntegralSampleData` attribute), 88
- `axes` (`zfit.core.interfaces.ZfitData` attribute), 89
- `axes` (`zfit.core.interfaces.ZfitDimensional` attribute), 90
- `axes` (`zfit.core.interfaces.ZfitFunc` attribute), 90
- `axes` (`zfit.core.interfaces.ZfitModel` attribute), 92
- `axes` (`zfit.core.interfaces.ZfitPDF` attribute), 94
- `axes` (`zfit.core.interfaces.ZfitSpace` attribute), 97
- `axes` (`zfit.core.limits.Space` attribute), 99
- `axes` (`zfit.core.sample.EventSpace` attribute), 143
- `axes` (`zfit.data.Data` attribute), 386
- `axes` (`zfit.func.BaseFunc` attribute), 389
- `axes` (`zfit.func.ProdFunc` attribute), 395
- `axes` (`zfit.func.SimpleFunc` attribute), 407
- `axes` (`zfit.func.SumFunc` attribute), 401
- `axes` (`zfit.models.basefuncor.FunctorMixin` attribute), 161
- `axes` (`zfit.models.basic.CustomGaussOLD` attribute), 167
- `axes` (`zfit.models.basic.Exponential` attribute), 175
- `axes` (`zfit.models.dist_tfp.ExponentialTFP` attribute), 183
- `axes` (`zfit.models.dist_tfp.Gauss` attribute), 191
- `axes` (`zfit.models.dist_tfp.TruncatedGauss` attribute), 199
- `axes` (`zfit.models.dist_tfp.Uniform` attribute), 208
- `axes` (`zfit.models.dist_tfp.WrapDistribution` attribute), 216
- `axes` (`zfit.models.functions.BaseFuncorFunc` attribute), 223
- `axes` (`zfit.models.functions.ProdFunc` attribute), 229
- `axes` (`zfit.models.functions.SimpleFunc` attribute), 236
- `axes` (`zfit.models.functions.SumFunc` attribute), 241
- `axes` (`zfit.models.functions.ZFunc` attribute), 247
- `axes` (`zfit.models.functor.BaseFuncor` attribute), 254
- `axes` (`zfit.models.functor.ProductPDF` attribute), 262
- `axes` (`zfit.models.functor.SumPDF` attribute), 270
- `axes` (`zfit.models.physics.CrystalBall` attribute), 279
- `axes` (`zfit.models.physics.DoubleCB` attribute), 287
- `axes` (`zfit.models.polynomials.Chebyshev` attribute), 296

axes (*zfit.models.polynomials.Chebyshev2* attribute), 304

axes (*zfit.models.polynomials.Hermite* attribute), 313

axes (*zfit.models.polynomials.Laguerre* attribute), 321

axes (*zfit.models.polynomials.Legendre* attribute), 329

axes (*zfit.models.polynomials.RecursivePolynomial* attribute), 337

axes (*zfit.models.special.SimpleFunctorPDF* attribute), 347

axes (*zfit.models.special.SimplePDF* attribute), 355

axes (*zfit.models.special.ZPDF* attribute), 363

axes (*zfit.pdf.BaseFunctor* attribute), 430

axes (*zfit.pdf.BasePDF* attribute), 422

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

axes (*zfit.pdf.Chebyshev2* attribute), 513

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

axes (*zfit.pdf.DoubleCB* attribute), 455

axes (*zfit.pdf.Exponential* attribute), 438

axes (*zfit.pdf.Gauss* attribute), 463

axes (*zfit.pdf.Hermite* attribute), 521

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

axes (*zfit.pdf.Legendre* attribute), 504

axes (*zfit.pdf.ProductPDF* attribute), 545

axes (*zfit.pdf.RecursivePolynomial* attribute), 537

axes (*zfit.pdf.SimpleFunctorPDF* attribute), 577

axes (*zfit.pdf.SimplePDF* attribute), 569

axes (*zfit.pdf.SumPDF* attribute), 553

axes (*zfit.pdf.TruncatedGauss* attribute), 479

axes (*zfit.pdf.Uniform* attribute), 471

axes (*zfit.pdf.WrapDistribution* attribute), 487

axes (*zfit.pdf.ZPDF* attribute), 561

axes (*zfit.Space* attribute), 42

AxesNotSpecifiedError, 373

AxesNotUnambiguousError, 373

B

BaseComposedParameter (class in *zfit.core.parameter*), 110

BaseConstraint (class in *zfit.core.constraint*), 69

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

BaseDimensional (class in *zfit.core.dimension*), 84

BaseFunc (class in *zfit.core.basefunc*), 47

BaseFunc (class in *zfit.func*), 388

BaseFunctor (class in *zfit.models.functor*), 253

BaseFunctor (class in *zfit.pdf*), 429

BaseFunctorFunc (class in *zfit.models.functions*), 223

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

BaseLoss (class in *zfit.loss*), 414

BaseMinimizer (class in *zfit.minimizers.baseminimizer*), 149

BaseModel (class in *zfit.core.basemodel*), 53

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

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

BaseParameter (class in *zfit.core.parameter*), 112

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

BasePDF (class in *zfit.pdf*), 421

BasePDFSubclassingError, 373

BaseStrategy (class in *zfit.minimizers.baseminimizer*), 150

BaseZParameter (class in *zfit.core.parameter*), 112

batch_scatter_update () (*zfit.core.parameter.ComposedResourceVariable* method), 117

batch_scatter_update () (*zfit.core.parameter.Parameter* method), 126

batch_scatter_update () (*zfit.core.parameter.TFBaseVariable* method), 136

batch_scatter_update () (*zfit.Parameter* method), 33

BFGSMinimizer (class in *zfit.minimizers.minimizer_tfp*), 155

C

Cachable (class in *zfit.util.cache*), 370

CachedLoss (class in *zfit.core.loss*), 105

Chebyshev (class in *zfit.models.polynomials*), 294

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

Chebyshev2 (class in *zfit.models.polynomials*), 303

Chebyshev2 (class in *zfit.pdf*), 511

chebyshev2_shape () (in module *zfit.models.polynomials*), 344

chebyshev_recurrence () (in module *zfit.models.polynomials*), 344

chebyshev_shape () (in module *zfit.models.polynomials*), 344

check_numerics () (in module *zfit.ztf.wrapping_tf*), 380

chunked_average () (in module *zfit.core.integration*), 88

chunksize (*zfit.util.execution.RunManager* attribute), 377

clear () (*zfit.util.container.DotDict* method), 372

combine () (*zfit.core.limits.Space* method), 99

combine () (*zfit.core.sample.EventSpace* method), 143

combine () (*zfit.Space* method), 42

combine_spaces () (in module *zfit.core.dimension*), 85

common_obs () (in module *zfit.core.dimension*), 85

complex () (in module *zfit.ztf.wrapping_tf*), 380

ComplexParameter (class in *zfit*), 40

ComplexParameter (class in *zfit.core.parameter*), 113

ComposedParameter (class in *zfit*), 39

ComposedParameter (class in *zfit.core.parameter*), 114
 ComposedResourceVariable (class in *zfit.core.parameter*), 115
 ComposedResourceVariable.SaveSliceInfo (class in *zfit.core.parameter*), 115
 ComposedVariable (class in *zfit.core.parameter*), 123
 conj (*zfit.ComplexParameter* attribute), 41
 conj (*zfit.core.parameter.ComplexParameter* attribute), 113
 constant () (in module *zfit.ztf.zextension*), 381
 ConstantParameter (class in *zfit.core.parameter*), 123
 ConstantParameter (class in *zfit.param*), 420
 constraint (*zfit.core.parameter.ComposedResourceVariable* attribute), 118
 constraint (*zfit.core.parameter.Parameter* attribute), 127
 constraint (*zfit.core.parameter.TFBaseVariable* attribute), 136
 constraint (*zfit.Parameter* attribute), 34
 constraints (*zfit.core.loss.BaseLoss* attribute), 104
 constraints (*zfit.core.loss.CachedLoss* attribute), 105
 constraints (*zfit.core.loss.ExtendedUnbinnedNLL* attribute), 106
 constraints (*zfit.core.loss.SimpleLoss* attribute), 107
 constraints (*zfit.core.loss.UnbinnedNLL* attribute), 108
 constraints (*zfit.loss.BaseLoss* attribute), 415
 constraints (*zfit.loss.ExtendedUnbinnedNLL* attribute), 413
 constraints (*zfit.loss.SimpleLoss* attribute), 416
 constraints (*zfit.loss.UnbinnedNLL* attribute), 414
 converged (*zfit.minimizers.fitresult.FitResult* attribute), 151
 ConversionError, 373
 convert_coeffs_dict_to_list () (in module *zfit.models.polynomials*), 344
 convert_func_to_pdf () (in module *zfit.core.operations*), 110
 convert_pdf_to_func () (in module *zfit.core.operations*), 110
 convert_sort_space () (*zfit.core.basefunc.BaseFunc* method), 48
 convert_sort_space () (*zfit.core.basemodel.BaseModel* method), 54
 convert_sort_space () (*zfit.core.basepdf.BasePDF* method), 62
 convert_sort_space () (*zfit.core.data.Data* method), 75
 convert_sort_space () (*zfit.core.data.SampleData* method), 78
 convert_sort_space () (*zfit.core.data.Sampler* method), 81
 convert_sort_space () (*zfit.data.Data* method), 386
 convert_sort_space () (*zfit.func.BaseFunc* method), 389
 convert_sort_space () (*zfit.func.ProdFunc* method), 395
 convert_sort_space () (*zfit.func.SimpleFunc* method), 408
 convert_sort_space () (*zfit.func.SumFunc* method), 401
 convert_sort_space () (*zfit.models.basefuncor.FunctorMixin* method), 161
 convert_sort_space () (*zfit.models.basic.CustomGaussOLD* method), 167
 convert_sort_space () (*zfit.models.basic.Exponential* method), 175
 convert_sort_space () (*zfit.models.dist_tfp.ExponentialTFP* method), 183
 convert_sort_space () (*zfit.models.dist_tfp.Gauss* method), 191
 convert_sort_space () (*zfit.models.dist_tfp.TruncatedGauss* method), 200
 convert_sort_space () (*zfit.models.dist_tfp.Uniform* method), 208
 convert_sort_space () (*zfit.models.dist_tfp.WrapDistribution* method), 216
 convert_sort_space () (*zfit.models.functions.BaseFuncorFunc* method), 223
 convert_sort_space () (*zfit.models.functions.ProdFunc* method), 229
 convert_sort_space () (*zfit.models.functions.SimpleFunc* method), 236
 convert_sort_space () (*zfit.models.functions.SumFunc* method), 241
 convert_sort_space () (*zfit.models.functions.ZFunc* method), 247
 convert_sort_space () (*zfit.models.functor.BaseFunctor* method), 254
 convert_sort_space () (*zfit.models.functor.ProductPDF* method), 262

`convert_sort_space()` (`zfit.models.functor.SumPDF` method), 270
`convert_sort_space()` (`zfit.models.physics.CrystalBall` method), 279
`convert_sort_space()` (`zfit.models.physics.DoubleCB` method), 287
`convert_sort_space()` (`zfit.models.polynomials.Chebyshev` method), 296
`convert_sort_space()` (`zfit.models.polynomials.Chebyshev2` method), 304
`convert_sort_space()` (`zfit.models.polynomials.Hermite` method), 313
`convert_sort_space()` (`zfit.models.polynomials.Laguerre` method), 321
`convert_sort_space()` (`zfit.models.polynomials.Legendre` method), 329
`convert_sort_space()` (`zfit.models.polynomials.RecursivePolynomial` method), 337
`convert_sort_space()` (`zfit.models.special.SimpleFunctorPDF` method), 347
`convert_sort_space()` (`zfit.models.special.SimplePDF` method), 355
`convert_sort_space()` (`zfit.models.special.ZPDF` method), 363
`convert_sort_space()` (`zfit.pdf.BaseFunctor` method), 430
`convert_sort_space()` (`zfit.pdf.BasePDF` method), 422
`convert_sort_space()` (`zfit.pdf.Chebyshev` method), 496
`convert_sort_space()` (`zfit.pdf.Chebyshev2` method), 513
`convert_sort_space()` (`zfit.pdf.CrystalBall` method), 446
`convert_sort_space()` (`zfit.pdf.DoubleCB` method), 455
`convert_sort_space()` (`zfit.pdf.Exponential` method), 438
`convert_sort_space()` (`zfit.pdf.Gauss` method), 463
`convert_sort_space()` (`zfit.pdf.Hermite` method), 521
`convert_sort_space()` (`zfit.pdf.Laguerre` method), 529
`convert_sort_space()` (`zfit.pdf.Legendre` method), 504
`convert_sort_space()` (`zfit.pdf.ProductPDF` method), 545
`convert_sort_space()` (`zfit.pdf.RecursivePolynomial` method), 537
`convert_sort_space()` (`zfit.pdf.SimpleFunctorPDF` method), 577
`convert_sort_space()` (`zfit.pdf.SimplePDF` method), 569
`convert_sort_space()` (`zfit.pdf.SumPDF` method), 553
`convert_sort_space()` (`zfit.pdf.TruncatedGauss` method), 480
`convert_sort_space()` (`zfit.pdf.Uniform` method), 471
`convert_sort_space()` (`zfit.pdf.WrapDistribution` method), 487
`convert_sort_space()` (`zfit.pdf.ZPDF` method), 562
`convert_to_container()` (in module `zfit.util.container`), 372
`convert_to_obs_str()` (in module `zfit.core.limits`), 103
`convert_to_parameter()` (in module `zfit`), 42
`convert_to_parameter()` (in module `zfit.core.parameter`), 142
`convert_to_space()` (in module `zfit`), 46
`convert_to_space()` (in module `zfit.core.limits`), 103
`convert_to_tensor()` (in module `zfit.ztf.zextension`), 381
`copy()` (`zfit.ComplexParameter` method), 41
`copy()` (`zfit.ComposedParameter` method), 40
`copy()` (`zfit.constraint.GaussianConstraint` method), 384
`copy()` (`zfit.constraint.SimpleConstraint` method), 383
`copy()` (`zfit.core.basefunc.BaseFunc` method), 48
`copy()` (`zfit.core.basemodel.BaseModel` method), 54
`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), 69
`copy()` (`zfit.core.constraint.DistributionConstraint` method), 70
`copy()` (`zfit.core.constraint.GaussianConstraint` method), 72
`copy()` (`zfit.core.constraint.SimpleConstraint` method), 73
`copy()` (`zfit.core.data.Data` method), 75
`copy()` (`zfit.core.data.SampleData` method), 78
`copy()` (`zfit.core.data.Sampler` method), 81
`copy()` (`zfit.core.dimension.BaseDimensional` method), 84

- `copy ()` (`zfit.core.integration.PartialIntegralSampleData method`), 88
- `copy ()` (`zfit.core.interfaces.ZfitData method`), 89
- `copy ()` (`zfit.core.interfaces.ZfitDimensional method`), 90
- `copy ()` (`zfit.core.interfaces.ZfitFunc method`), 90
- `copy ()` (`zfit.core.interfaces.ZfitLoss method`), 92
- `copy ()` (`zfit.core.interfaces.ZfitModel method`), 92
- `copy ()` (`zfit.core.interfaces.ZfitNumeric method`), 94
- `copy ()` (`zfit.core.interfaces.ZfitObject method`), 94
- `copy ()` (`zfit.core.interfaces.ZfitParameter method`), 96
- `copy ()` (`zfit.core.interfaces.ZfitPDF method`), 94
- `copy ()` (`zfit.core.interfaces.ZfitSpace method`), 97
- `copy ()` (`zfit.core.limits.Space method`), 99
- `copy ()` (`zfit.core.loss.BaseLoss method`), 104
- `copy ()` (`zfit.core.loss.CachedLoss method`), 105
- `copy ()` (`zfit.core.loss.ExtendedUnbinnedNLL method`), 106
- `copy ()` (`zfit.core.loss.SimpleLoss method`), 107
- `copy ()` (`zfit.core.loss.UnbinnedNLL method`), 108
- `copy ()` (`zfit.core.parameter.BaseComposedParameter method`), 111
- `copy ()` (`zfit.core.parameter.BaseParameter method`), 112
- `copy ()` (`zfit.core.parameter.BaseZParameter method`), 112
- `copy ()` (`zfit.core.parameter.ComplexParameter method`), 113
- `copy ()` (`zfit.core.parameter.ComposedParameter method`), 115
- `copy ()` (`zfit.core.parameter.ConstantParameter method`), 123
- `copy ()` (`zfit.core.parameter.Parameter method`), 127
- `copy ()` (`zfit.core.parameter.ZfitParameterMixin method`), 142
- `copy ()` (`zfit.core.sample.EventSpace method`), 143
- `copy ()` (`zfit.data.Data method`), 386
- `copy ()` (`zfit.func.BaseFunc method`), 389
- `copy ()` (`zfit.func.ProdFunc method`), 396
- `copy ()` (`zfit.func.SimpleFunc method`), 408
- `copy ()` (`zfit.func.SumFunc method`), 402
- `copy ()` (`zfit.loss.BaseLoss method`), 415
- `copy ()` (`zfit.loss.ExtendedUnbinnedNLL method`), 413
- `copy ()` (`zfit.loss.SimpleLoss method`), 416
- `copy ()` (`zfit.loss.UnbinnedNLL method`), 414
- `copy ()` (`zfit.minimize.Adam method`), 417
- `copy ()` (`zfit.minimize.Minuit method`), 419
- `copy ()` (`zfit.minimize.Scipy method`), 419
- `copy ()` (`zfit.minimize.WrapOptimizer method`), 417
- `copy ()` (`zfit.minimizers.base_tf.WrapOptimizer method`), 149
- `copy ()` (`zfit.minimizers.baseminimizer.BaseMinimizer method`), 149
- `copy ()` (`zfit.minimizers.minimizer_minuit.Minuit method`), 154
- `copy ()` (`zfit.minimizers.minimizer_tfp.BFGSMinimizer method`), 155
- `copy ()` (`zfit.minimizers.minimizers_scipy.Scipy method`), 156
- `copy ()` (`zfit.minimizers.optimizers_tf.Adam method`), 156
- `copy ()` (`zfit.models.basefunctor.FunctorMixin method`), 161
- `copy ()` (`zfit.models.basic.CustomGaussOLD method`), 167
- `copy ()` (`zfit.models.basic.Exponential method`), 175
- `copy ()` (`zfit.models.dist_tfp.ExponentialTFP method`), 183
- `copy ()` (`zfit.models.dist_tfp.Gauss method`), 191
- `copy ()` (`zfit.models.dist_tfp.TruncatedGauss method`), 200
- `copy ()` (`zfit.models.dist_tfp.Uniform method`), 208
- `copy ()` (`zfit.models.dist_tfp.WrapDistribution method`), 216
- `copy ()` (`zfit.models.functions.BaseFunctorFunc method`), 224
- `copy ()` (`zfit.models.functions.ProdFunc method`), 230
- `copy ()` (`zfit.models.functions.SimpleFunc method`), 236
- `copy ()` (`zfit.models.functions.SumFunc method`), 242
- `copy ()` (`zfit.models.functions.ZFunc method`), 248
- `copy ()` (`zfit.models.functor.BaseFunctor method`), 254
- `copy ()` (`zfit.models.functor.ProductPDF method`), 262
- `copy ()` (`zfit.models.functor.SumPDF method`), 270
- `copy ()` (`zfit.models.physics.CrystalBall method`), 279
- `copy ()` (`zfit.models.physics.DoubleCB method`), 288
- `copy ()` (`zfit.models.polynomials.Chebyshev method`), 296
- `copy ()` (`zfit.models.polynomials.Chebyshev2 method`), 304
- `copy ()` (`zfit.models.polynomials.Hermite method`), 313
- `copy ()` (`zfit.models.polynomials.Laguerre method`), 321
- `copy ()` (`zfit.models.polynomials.Legendre method`), 330
- `copy ()` (`zfit.models.polynomials.RecursivePolynomial method`), 338
- `copy ()` (`zfit.models.special.SimpleFunctorPDF method`), 347
- `copy ()` (`zfit.models.special.SimplePDF method`), 355
- `copy ()` (`zfit.models.special.ZPDF method`), 363
- `copy ()` (`zfit.param.ConstantParameter method`), 420
- `copy ()` (`zfit.Parameter method`), 34
- `copy ()` (`zfit.pdf.BaseFunctor method`), 430
- `copy ()` (`zfit.pdf.BasePDF method`), 422
- `copy ()` (`zfit.pdf.Chebyshev method`), 496
- `copy ()` (`zfit.pdf.Chebyshev2 method`), 513
- `copy ()` (`zfit.pdf.CrystalBall method`), 447
- `copy ()` (`zfit.pdf.DoubleCB method`), 455
- `copy ()` (`zfit.pdf.Exponential method`), 438

`copy()` (*zfit.pdf.Gauss method*), 463
`copy()` (*zfit.pdf.Hermite method*), 521
`copy()` (*zfit.pdf.Laguerre method*), 530
`copy()` (*zfit.pdf.Legendre method*), 504
`copy()` (*zfit.pdf.ProductPDF method*), 545
`copy()` (*zfit.pdf.RecursivePolynomial method*), 538
`copy()` (*zfit.pdf.SimpleFunctorPDF method*), 577
`copy()` (*zfit.pdf.SimplePDF method*), 570
`copy()` (*zfit.pdf.SumPDF method*), 554
`copy()` (*zfit.pdf.TruncatedGauss method*), 480
`copy()` (*zfit.pdf.Uniform method*), 472
`copy()` (*zfit.pdf.WrapDistribution method*), 488
`copy()` (*zfit.pdf.ZPDF method*), 562
`copy()` (*zfit.Space method*), 43
`copy()` (*zfit.util.container.DotDict method*), 372
`count_up_to()` (*zfit.core.parameter.ComposedResourceVariable method*), 118
`count_up_to()` (*zfit.core.parameter.Parameter method*), 127
`count_up_to()` (*zfit.core.parameter.TFBaseVariable method*), 136
`count_up_to()` (*zfit.Parameter method*), 34
`counts_multinomial()` (*in module zfit.ztf.random*), 380
`covariance()` (*zfit.minimizers.fitresult.FitResult method*), 151
`create` (*zfit.core.parameter.ComposedResourceVariable attribute*), 118
`create` (*zfit.core.parameter.Parameter attribute*), 127
`create` (*zfit.core.parameter.TFBaseVariable attribute*), 137
`create` (*zfit.Parameter attribute*), 34
`create_extended()` (*zfit.core.basepdf.BasePDF method*), 62
`create_extended()` (*zfit.core.interfaces.ZfitPDF method*), 94
`create_extended()` (*zfit.models.basic.CustomGaussOLD method*), 167
`create_extended()` (*zfit.models.basic.Exponential method*), 175
`create_extended()` (*zfit.models.dist_tfp.ExponentialTFP method*), 183
`create_extended()` (*zfit.models.dist_tfp.Gauss method*), 192
`create_extended()` (*zfit.models.dist_tfp.TruncatedGauss method*), 200
`create_extended()` (*zfit.models.dist_tfp.Uniform method*), 208
`create_extended()` (*zfit.models.dist_tfp.WrapDistribution method*), 216
`create_extended()` (*zfit.models.functor.BaseFunctor method*), 254
`create_extended()` (*zfit.models.functor.ProductPDF method*), 262
`create_extended()` (*zfit.models.functor.SumPDF method*), 270
`create_extended()` (*zfit.models.physics.CrystalBall method*), 279
`create_extended()` (*zfit.models.physics.DoubleCB method*), 288
`create_extended()` (*zfit.models.polynomials.Chebyshev method*), 296
`create_extended()` (*zfit.models.polynomials.Chebyshev2 method*), 305
`create_extended()` (*zfit.models.polynomials.Hermite method*), 313
`create_extended()` (*zfit.models.polynomials.Laguerre method*), 321
`create_extended()` (*zfit.models.polynomials.Legendre method*), 330
`create_extended()` (*zfit.models.polynomials.RecursivePolynomial method*), 338
`create_extended()` (*zfit.models.special.SimpleFunctorPDF method*), 347
`create_extended()` (*zfit.models.special.SimplePDF method*), 355
`create_extended()` (*zfit.models.special.ZPDF method*), 363
`create_extended()` (*zfit.pdf.BaseFunctor method*), 430
`create_extended()` (*zfit.pdf.BasePDF method*), 422
`create_extended()` (*zfit.pdf.Chebyshev method*), 496
`create_extended()` (*zfit.pdf.Chebyshev2 method*), 513
`create_extended()` (*zfit.pdf.CrystalBall method*), 447
`create_extended()` (*zfit.pdf.DoubleCB method*), 455
`create_extended()` (*zfit.pdf.Exponential method*), 438
`create_extended()` (*zfit.pdf.Gauss method*), 464
`create_extended()` (*zfit.pdf.Hermite method*), 521
`create_extended()` (*zfit.pdf.Laguerre method*), 530

[create_extended\(\) \(zfit.pdf.Legendre method\), 505](#)
[create_extended\(\) \(zfit.pdf.ProductPDF method\), 546](#)
[create_extended\(\) \(zfit.pdf.RecursivePolynomial method\), 538](#)
[create_extended\(\) \(zfit.pdf.SimpleFunctorPDF method\), 578](#)
[create_extended\(\) \(zfit.pdf.SimplePDF method\), 570](#)
[create_extended\(\) \(zfit.pdf.SumPDF method\), 554](#)
[create_extended\(\) \(zfit.pdf.TruncatedGauss method\), 480](#)
[create_extended\(\) \(zfit.pdf.Uniform method\), 472](#)
[create_extended\(\) \(zfit.pdf.WrapDistribution method\), 488](#)
[create_extended\(\) \(zfit.pdf.ZPDF method\), 562](#)
[create_limits\(\) \(zfit.core.sample.EventSpace method\), 144](#)
[create_poly\(\) \(in module zfit.models.polynomials\), 345](#)
[create_projection_pdf\(\) \(zfit.core.basepdf.BasePDF method\), 63](#)
[create_projection_pdf\(\) \(zfit.models.basic.CustomGaussOLD method\), 167](#)
[create_projection_pdf\(\) \(zfit.models.basic.Exponential method\), 176](#)
[create_projection_pdf\(\) \(zfit.models.dist_tfp.ExponentialTFP method\), 184](#)
[create_projection_pdf\(\) \(zfit.models.dist_tfp.Gauss method\), 192](#)
[create_projection_pdf\(\) \(zfit.models.dist_tfp.TruncatedGauss method\), 200](#)
[create_projection_pdf\(\) \(zfit.models.dist_tfp.Uniform method\), 208](#)
[create_projection_pdf\(\) \(zfit.models.dist_tfp.WrapDistribution method\), 216](#)
[create_projection_pdf\(\) \(zfit.models.functor.BaseFunctor method\), 254](#)
[create_projection_pdf\(\) \(zfit.models.functor.ProductPDF method\), 262](#)
[create_projection_pdf\(\) \(zfit.models.functor.SumPDF method\), 271](#)
[create_projection_pdf\(\) \(zfit.models.physics.CrystalBall method\), 279](#)
[create_projection_pdf\(\) \(zfit.models.physics.DoubleCB method\), 288](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.Chebyshev method\), 297](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.Chebyshev2 method\), 305](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.Hermite method\), 313](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.Laguerre method\), 322](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.Legendre method\), 330](#)
[create_projection_pdf\(\) \(zfit.models.polynomials.RecursivePolynomial method\), 338](#)
[create_projection_pdf\(\) \(zfit.models.special.SimpleFunctorPDF method\), 348](#)
[create_projection_pdf\(\) \(zfit.models.special.SimplePDF method\), 355](#)
[create_projection_pdf\(\) \(zfit.models.special.ZPDF method\), 363](#)
[create_projection_pdf\(\) \(zfit.pdf.BaseFunctor method\), 430](#)
[create_projection_pdf\(\) \(zfit.pdf.BasePDF method\), 423](#)
[create_projection_pdf\(\) \(zfit.pdf.Chebyshev method\), 497](#)
[create_projection_pdf\(\) \(zfit.pdf.Chebyshev2 method\), 513](#)
[create_projection_pdf\(\) \(zfit.pdf.CrystalBall method\), 447](#)
[create_projection_pdf\(\) \(zfit.pdf.DoubleCB method\), 456](#)
[create_projection_pdf\(\) \(zfit.pdf.Exponential method\), 439](#)
[create_projection_pdf\(\) \(zfit.pdf.Gauss method\), 464](#)
[create_projection_pdf\(\) \(zfit.pdf.Hermite method\), 522](#)
[create_projection_pdf\(\) \(zfit.pdf.Laguerre method\), 530](#)
[create_projection_pdf\(\) \(zfit.pdf.Legendre method\), 505](#)
[create_projection_pdf\(\) \(zfit.pdf.ProductPDF method\), 546](#)
[create_projection_pdf\(\) \(zfit.pdf.RecursivePolynomial method\), 538](#)
[create_projection_pdf\(\) \(zfit.pdf.SimpleFunctorPDF method\), 578](#)

`create_projection_pdf()` (*zfit.pdf.SimplePDF method*), 570
`create_projection_pdf()` (*zfit.pdf.SumPDF method*), 554
`create_projection_pdf()` (*zfit.pdf.TruncatedGauss method*), 480
`create_projection_pdf()` (*zfit.pdf.Uniform method*), 472
`create_projection_pdf()` (*zfit.pdf.WrapDistribution method*), 488
`create_projection_pdf()` (*zfit.pdf.ZPDF method*), 562
`create_sampler()` (*zfit.core.basefunc.BaseFunc method*), 48
`create_sampler()` (*zfit.core.basemodel.BaseModel method*), 54
`create_sampler()` (*zfit.core.basepdf.BasePDF method*), 63
`create_sampler()` (*zfit.func.BaseFunc method*), 390
`create_sampler()` (*zfit.func.ProdFunc method*), 396
`create_sampler()` (*zfit.func.SimpleFunc method*), 408
`create_sampler()` (*zfit.func.SumFunc method*), 402
`create_sampler()` (*zfit.models.basefunc.FunctorMixin method*), 161
`create_sampler()` (*zfit.models.basic.CustomGaussOLD method*), 168
`create_sampler()` (*zfit.models.basic.Exponential method*), 176
`create_sampler()` (*zfit.models.dist_tfp.ExponentialTFP method*), 184
`create_sampler()` (*zfit.models.dist_tfp.Gauss method*), 192
`create_sampler()` (*zfit.models.dist_tfp.TruncatedGauss method*), 200
`create_sampler()` (*zfit.models.dist_tfp.Uniform method*), 208
`create_sampler()` (*zfit.models.dist_tfp.WrapDistribution method*), 216
`create_sampler()` (*zfit.models.functions.BaseFunc method*), 224
`create_sampler()` (*zfit.models.functions.ProdFunc method*), 230
`create_sampler()` (*zfit.models.functions.SimpleFunc method*), 236
`create_sampler()` (*zfit.models.functions.SumFunc method*), 242
`create_sampler()` (*zfit.models.functions.ZFunc method*), 248
`create_sampler()` (*zfit.models.functor.BaseFunc method*), 254
`create_sampler()` (*zfit.models.functor.ProductPDF method*), 262
`create_sampler()` (*zfit.models.functor.SumPDF method*), 271
`create_sampler()` (*zfit.models.physics.CrystalBall method*), 279
`create_sampler()` (*zfit.models.physics.DoubleCB method*), 288
`create_sampler()` (*zfit.models.polynomials.Chebyshev method*), 297
`create_sampler()` (*zfit.models.polynomials.Chebyshev2 method*), 305
`create_sampler()` (*zfit.models.polynomials.Hermite method*), 313
`create_sampler()` (*zfit.models.polynomials.Laguerre method*), 322
`create_sampler()` (*zfit.models.polynomials.Legendre method*), 330
`create_sampler()` (*zfit.models.polynomials.RecursivePolynomial method*), 338
`create_sampler()` (*zfit.models.special.SimpleFuncPDF method*), 348
`create_sampler()` (*zfit.models.special.SimplePDF method*), 356
`create_sampler()` (*zfit.models.special.ZPDF method*), 363
`create_sampler()` (*zfit.pdf.BaseFunc method*), 431
`create_sampler()` (*zfit.pdf.BasePDF method*), 423
`create_sampler()` (*zfit.pdf.Chebyshev method*), 497
`create_sampler()` (*zfit.pdf.Chebyshev2 method*), 513
`create_sampler()` (*zfit.pdf.CrystalBall method*), 447
`create_sampler()` (*zfit.pdf.DoubleCB method*), 456
`create_sampler()` (*zfit.pdf.Exponential method*), 439
`create_sampler()` (*zfit.pdf.Gauss method*), 464
`create_sampler()` (*zfit.pdf.Hermite method*), 522
`create_sampler()` (*zfit.pdf.Laguerre method*), 530
`create_sampler()` (*zfit.pdf.Legendre method*), 505
`create_sampler()` (*zfit.pdf.ProductPDF method*), 546
`create_sampler()` (*zfit.pdf.RecursivePolynomial method*), 538
`create_sampler()` (*zfit.pdf.SimpleFuncPDF method*), 578
`create_sampler()` (*zfit.pdf.SimplePDF method*), 570
`create_sampler()` (*zfit.pdf.SumPDF method*), 554
`create_sampler()` (*zfit.pdf.TruncatedGauss method*), 480
`create_sampler()` (*zfit.pdf.Uniform method*), 472
`create_sampler()` (*zfit.pdf.WrapDistribution method*), 488
`create_sampler()` (*zfit.pdf.ZPDF method*), 562
`create_session()` (*zfit.util.execution.RunManager*

- method), 377
- CrystalBall (class in `zfit.models.physics`), 277
- CrystalBall (class in `zfit.pdf`), 445
- `crystalball_func()` (in module `zfit.models.physics`), 294
- `crystalball_integral()` (in module `zfit.models.physics`), 294
- CustomGaussOLD (class in `zfit.models.basic`), 166
- ## D
- Data (class in `zfit.core.data`), 74
- Data (class in `zfit.data`), 385
- data (`zfit.core.interfaces.ZfitLoss` attribute), 92
- data (`zfit.core.loss.BaseLoss` attribute), 104
- data (`zfit.core.loss.CachedLoss` attribute), 105
- data (`zfit.core.loss.ExtendedUnbinnedNLL` attribute), 106
- data (`zfit.core.loss.SimpleLoss` attribute), 107
- data (`zfit.core.loss.UnbinnedNLL` attribute), 108
- data (`zfit.loss.BaseLoss` attribute), 415
- data (`zfit.loss.ExtendedUnbinnedNLL` attribute), 413
- data (`zfit.loss.SimpleLoss` attribute), 416
- data (`zfit.loss.UnbinnedNLL` attribute), 414
- data_range (`zfit.core.data.Data` attribute), 75
- data_range (`zfit.core.data.SampleData` attribute), 78
- data_range (`zfit.core.data.Sampler` attribute), 81
- data_range (`zfit.data.Data` attribute), 386
- DefaultStrategy (class in `zfit.minimizers.baseminimizer`), 150
- degree (`zfit.models.polynomials.Chebyshev` attribute), 297
- degree (`zfit.models.polynomials.Chebyshev2` attribute), 306
- degree (`zfit.models.polynomials.Hermite` attribute), 314
- degree (`zfit.models.polynomials.Laguerre` attribute), 322
- degree (`zfit.models.polynomials.Legendre` attribute), 331
- degree (`zfit.models.polynomials.RecursivePolynomial` attribute), 339
- degree (`zfit.pdf.Chebyshev` attribute), 497
- degree (`zfit.pdf.Chebyshev2` attribute), 514
- degree (`zfit.pdf.Hermite` attribute), 522
- degree (`zfit.pdf.Laguerre` attribute), 531
- degree (`zfit.pdf.Legendre` attribute), 506
- degree (`zfit.pdf.RecursivePolynomial` attribute), 539
- device (`zfit.core.parameter.ComposedResourceVariable` attribute), 118
- device (`zfit.core.parameter.Parameter` attribute), 127
- device (`zfit.core.parameter.TFBaseVariable` attribute), 137
- device (`zfit.Parameter` attribute), 34
- `dict_to_matrix()` (in module `zfit.minimizers.fitresult`), 152
- distribution (`zfit.constraint.GaussianConstraint` attribute), 384
- distribution (`zfit.core.constraint.DistributionConstraint` attribute), 71
- distribution (`zfit.core.constraint.GaussianConstraint` attribute), 72
- distribution (`zfit.models.dist_tfp.ExponentialTFP` attribute), 184
- distribution (`zfit.models.dist_tfp.Gauss` attribute), 193
- distribution (`zfit.models.dist_tfp.TruncatedGauss` attribute), 201
- distribution (`zfit.models.dist_tfp.Uniform` attribute), 209
- distribution (`zfit.models.dist_tfp.WrapDistribution` attribute), 217
- distribution (`zfit.pdf.Gauss` attribute), 465
- distribution (`zfit.pdf.TruncatedGauss` attribute), 481
- distribution (`zfit.pdf.Uniform` attribute), 473
- distribution (`zfit.pdf.WrapDistribution` attribute), 489
- DistributionConstraint (class in `zfit.core.constraint`), 70
- `do_recurrence()` (in module `zfit.models.polynomials`), 345
- DotDict (class in `zfit.util.container`), 372
- `double_crystalball_func()` (in module `zfit.models.physics`), 294
- `double_crystalball_integral()` (in module `zfit.models.physics`), 294
- DoubleCB (class in `zfit.models.physics`), 285
- DoubleCB (class in `zfit.pdf`), 453
- dtype (`zfit.ComplexParameter` attribute), 41
- dtype (`zfit.ComposedParameter` attribute), 40
- dtype (`zfit.constraint.GaussianConstraint` attribute), 384
- dtype (`zfit.constraint.SimpleConstraint` attribute), 383
- dtype (`zfit.core.basefunc.BaseFunc` attribute), 49
- dtype (`zfit.core.basemodel.BaseModel` attribute), 55
- dtype (`zfit.core.baseobject.BaseNumeric` attribute), 60
- dtype (`zfit.core.basepdf.BasePDF` attribute), 63
- dtype (`zfit.core.constraint.BaseConstraint` attribute), 69
- dtype (`zfit.core.constraint.DistributionConstraint` attribute), 71
- dtype (`zfit.core.constraint.GaussianConstraint` attribute), 72
- dtype (`zfit.core.constraint.SimpleConstraint` attribute), 73
- dtype (`zfit.core.data.Data` attribute), 75
- dtype (`zfit.core.data.SampleData` attribute), 78
- dtype (`zfit.core.data.Sampler` attribute), 81
- dtype (`zfit.core.interfaces.ZfitFunc` attribute), 90
- dtype (`zfit.core.interfaces.ZfitModel` attribute), 92

- `dtype (zfit.core.interfaces.ZfitNumeric attribute)`, 94
 - `dtype (zfit.core.interfaces.ZfitParameter attribute)`, 96
 - `dtype (zfit.core.interfaces.ZfitPDF attribute)`, 94
 - `dtype (zfit.core.parameter.BaseComposedParameter attribute)`, 111
 - `dtype (zfit.core.parameter.BaseParameter attribute)`, 112
 - `dtype (zfit.core.parameter.BaseZParameter attribute)`, 112
 - `dtype (zfit.core.parameter.ComplexParameter attribute)`, 113
 - `dtype (zfit.core.parameter.ComposedParameter attribute)`, 115
 - `dtype (zfit.core.parameter.ComposedResourceVariable attribute)`, 118
 - `dtype (zfit.core.parameter.ComposedVariable attribute)`, 123
 - `dtype (zfit.core.parameter.ConstantParameter attribute)`, 123
 - `dtype (zfit.core.parameter.Parameter attribute)`, 127
 - `dtype (zfit.core.parameter.TFBaseVariable attribute)`, 137
 - `dtype (zfit.core.parameter.ZfitBaseVariable attribute)`, 141
 - `dtype (zfit.core.parameter.ZfitParameterMixin attribute)`, 142
 - `dtype (zfit.data.Data attribute)`, 386
 - `dtype (zfit.func.BaseFunc attribute)`, 390
 - `dtype (zfit.func.ProdFunc attribute)`, 396
 - `dtype (zfit.func.SimpleFunc attribute)`, 408
 - `dtype (zfit.func.SumFunc attribute)`, 402
 - `dtype (zfit.models.basefunctor.FunctorMixin attribute)`, 162
 - `dtype (zfit.models.basic.CustomGaussOLD attribute)`, 168
 - `dtype (zfit.models.basic.Exponential attribute)`, 176
 - `dtype (zfit.models.dist_tfp.ExponentialTFP attribute)`, 184
 - `dtype (zfit.models.dist_tfp.Gauss attribute)`, 193
 - `dtype (zfit.models.dist_tfp.TruncatedGauss attribute)`, 201
 - `dtype (zfit.models.dist_tfp.Uniform attribute)`, 209
 - `dtype (zfit.models.dist_tfp.WrapDistribution attribute)`, 217
 - `dtype (zfit.models.functions.BaseFunctorFunc attribute)`, 224
 - `dtype (zfit.models.functions.ProdFunc attribute)`, 230
 - `dtype (zfit.models.functions.SimpleFunc attribute)`, 236
 - `dtype (zfit.models.functions.SumFunc attribute)`, 242
 - `dtype (zfit.models.functions.ZFunc attribute)`, 248
 - `dtype (zfit.models.functor.BaseFunctor attribute)`, 255
 - `dtype (zfit.models.functor.ProductPDF attribute)`, 263
 - `dtype (zfit.models.functor.SumPDF attribute)`, 271
 - `dtype (zfit.models.physics.CrystalBall attribute)`, 280
 - `dtype (zfit.models.physics.DoubleCB attribute)`, 289
 - `dtype (zfit.models.polynomials.Chebyshev attribute)`, 297
 - `dtype (zfit.models.polynomials.Chebyshev2 attribute)`, 306
 - `dtype (zfit.models.polynomials.Hermite attribute)`, 314
 - `dtype (zfit.models.polynomials.Laguerre attribute)`, 322
 - `dtype (zfit.models.polynomials.Legendre attribute)`, 331
 - `dtype (zfit.models.polynomials.RecursivePolynomial attribute)`, 339
 - `dtype (zfit.models.special.SimpleFunctorPDF attribute)`, 348
 - `dtype (zfit.models.special.SimplePDF attribute)`, 356
 - `dtype (zfit.models.special.ZPDF attribute)`, 364
 - `dtype (zfit.param.ConstantParameter attribute)`, 420
 - `dtype (zfit.Parameter attribute)`, 34
 - `dtype (zfit.pdf.BaseFunctor attribute)`, 431
 - `dtype (zfit.pdf.BasePDF attribute)`, 423
 - `dtype (zfit.pdf.Chebyshev attribute)`, 497
 - `dtype (zfit.pdf.Chebyshev2 attribute)`, 514
 - `dtype (zfit.pdf.CrystalBall attribute)`, 448
 - `dtype (zfit.pdf.DoubleCB attribute)`, 456
 - `dtype (zfit.pdf.Exponential attribute)`, 439
 - `dtype (zfit.pdf.Gauss attribute)`, 465
 - `dtype (zfit.pdf.Hermite attribute)`, 522
 - `dtype (zfit.pdf.Laguerre attribute)`, 531
 - `dtype (zfit.pdf.Legendre attribute)`, 506
 - `dtype (zfit.pdf.ProductPDF attribute)`, 547
 - `dtype (zfit.pdf.RecursivePolynomial attribute)`, 539
 - `dtype (zfit.pdf.SimpleFunctorPDF attribute)`, 579
 - `dtype (zfit.pdf.SimplePDF attribute)`, 571
 - `dtype (zfit.pdf.SumPDF attribute)`, 555
 - `dtype (zfit.pdf.TruncatedGauss attribute)`, 481
 - `dtype (zfit.pdf.Uniform attribute)`, 473
 - `dtype (zfit.pdf.WrapDistribution attribute)`, 489
 - `dtype (zfit.pdf.ZPDF attribute)`, 563
 - `DueToLazynessNotImplementedError`, 373
- ## E
- `edm (zfit.minimizers.fitresult.FitResult attribute)`, 151
 - `error()` (`zfit.minimizers.fitresult.FitResult` method), 151
 - `error()` (`zfit.minimizers.interface.ZfitResult` method), 153
 - `errordef (zfit.core.interfaces.ZfitLoss attribute)`, 92
 - `errordef (zfit.core.loss.BaseLoss attribute)`, 104
 - `errordef (zfit.core.loss.CachedLoss attribute)`, 105
 - `errordef (zfit.core.loss.ExtendedUnbinnedNLL attribute)`, 106
 - `errordef (zfit.core.loss.SimpleLoss attribute)`, 107
 - `errordef (zfit.core.loss.UnbinnedNLL attribute)`, 108
 - `errordef (zfit.loss.BaseLoss attribute)`, 415
 - `errordef (zfit.loss.ExtendedUnbinnedNLL attribute)`, 413

errordef (*zfit.loss.SimpleLoss* attribute), 416
 errordef (*zfit.loss.UnbinnedNLL* attribute), 414
 eval() (*zfit.core.parameter.ComposedResourceVariable* method), 118
 eval() (*zfit.core.parameter.Parameter* method), 127
 eval() (*zfit.core.parameter.TFBaseVariable* method), 137
 eval() (*zfit.Parameter* method), 34
 EventSpace (class in *zfit.core.sample*), 143
 exp() (in module *zfit.ztf.wrapping_tf*), 380
 Exponential (class in *zfit.models.basic*), 174
 Exponential (class in *zfit.pdf*), 437
 ExponentialTFP (class in *zfit.models.dist_tfp*), 182
 extended_sampling() (in module *zfit.core.sample*), 148
 ExtendedPDFError, 374
 ExtendedUnbinnedNLL (class in *zfit.core.loss*), 106
 ExtendedUnbinnedNLL (class in *zfit.loss*), 413
 ExternalOptimizerInterface (class in *zfit.minimizers.tf_external_optimizer*), 157
 extract_extended_pdfs() (in module *zfit.core.sample*), 148

F

factory (*zfit.core.sample.EventSpace* attribute), 144
 FailMinimizeNaN, 150
 feed_function() (in module *zfit.core.data*), 84
 feed_function() (in module *zfit.core.parameter*), 143
 feed_function_for_partial_run() (in module *zfit.core.data*), 84
 feed_function_for_partial_run() (in module *zfit.core.parameter*), 143
 fetch_function() (in module *zfit.core.data*), 84
 fetch_function() (in module *zfit.core.parameter*), 143
 fit_range (*zfit.core.interfaces.ZfitLoss* attribute), 92
 fit_range (*zfit.core.loss.BaseLoss* attribute), 104
 fit_range (*zfit.core.loss.CachedLoss* attribute), 105
 fit_range (*zfit.core.loss.ExtendedUnbinnedNLL* attribute), 106
 fit_range (*zfit.core.loss.SimpleLoss* attribute), 107
 fit_range (*zfit.core.loss.UnbinnedNLL* attribute), 108
 fit_range (*zfit.loss.BaseLoss* attribute), 415
 fit_range (*zfit.loss.ExtendedUnbinnedNLL* attribute), 413
 fit_range (*zfit.loss.SimpleLoss* attribute), 416
 fit_range (*zfit.loss.UnbinnedNLL* attribute), 414
 FitResult (class in *zfit.minimizers.fitresult*), 151
 floating (*zfit.ComplexParameter* attribute), 41
 floating (*zfit.ComposedParameter* attribute), 40
 floating (*zfit.core.interfaces.ZfitParameter* attribute), 96

floating (*zfit.core.parameter.BaseComposedParameter* attribute), 111
 floating (*zfit.core.parameter.BaseParameter* attribute), 112
 floating (*zfit.core.parameter.BaseZParameter* attribute), 112
 floating (*zfit.core.parameter.ComplexParameter* attribute), 113
 floating (*zfit.core.parameter.ComposedParameter* attribute), 115
 floating (*zfit.core.parameter.ConstantParameter* attribute), 123
 floating (*zfit.core.parameter.Parameter* attribute), 127
 floating (*zfit.core.parameter.ZfitParameterMixin* attribute), 142
 floating (*zfit.param.ConstantParameter* attribute), 420
 floating (*zfit.Parameter* attribute), 34
 fmin (*zfit.minimizers.fitresult.FitResult* attribute), 152
 fmin (*zfit.minimizers.interface.ZfitResult* attribute), 153
 fracs (*zfit.models.functor.SumPDF* attribute), 271
 fracs (*zfit.pdf.SumPDF* attribute), 555
 from_axes() (*zfit.core.limits.Space* class method), 100
 from_axes() (*zfit.core.sample.EventSpace* class method), 144
 from_axes() (*zfit.Space* class method), 43
 from_cartesian() (*zfit.ComplexParameter* static method), 41
 from_cartesian() (*zfit.core.parameter.ComplexParameter* static method), 114
 from_numpy() (*zfit.core.data.Data* class method), 75
 from_numpy() (*zfit.core.data.SampleData* class method), 78
 from_numpy() (*zfit.core.data.Sampler* class method), 81
 from_numpy() (*zfit.data.Data* class method), 386
 from_pandas() (*zfit.core.data.Data* class method), 75
 from_pandas() (*zfit.core.data.SampleData* class method), 78
 from_pandas() (*zfit.core.data.Sampler* class method), 82
 from_pandas() (*zfit.data.Data* class method), 386
 from_polar() (*zfit.ComplexParameter* static method), 41
 from_polar() (*zfit.core.parameter.ComplexParameter* static method), 114
 from_proto() (*zfit.core.parameter.ComposedResourceVariable* static method), 118
 from_proto() (*zfit.core.parameter.Parameter* static method), 128
 from_proto() (*zfit.core.parameter.TFBaseVariable* static method), 137

- `from_proto()` (*zfit.Parameter static method*), 34
- `from_root()` (*zfit.core.data.Data class method*), 75
- `from_root()` (*zfit.core.data.SampleData class method*), 79
- `from_root()` (*zfit.core.data.Sampler class method*), 82
- `from_root()` (*zfit.data.Data class method*), 386
- `from_root_iter()` (*zfit.core.data.Data class method*), 76
- `from_root_iter()` (*zfit.core.data.SampleData class method*), 79
- `from_root_iter()` (*zfit.core.data.Sampler class method*), 83
- `from_root_iter()` (*zfit.data.Data class method*), 387
- `from_sample()` (*zfit.core.data.SampleData class method*), 79
- `from_sample()` (*zfit.core.data.Sampler class method*), 83
- `from_tensor()` (*zfit.core.data.Data class method*), 76
- `from_tensor()` (*zfit.core.data.LightDataset class method*), 77
- `from_tensor()` (*zfit.core.data.SampleData class method*), 79
- `from_tensor()` (*zfit.core.data.Sampler class method*), 83
- `from_tensor()` (*zfit.data.Data class method*), 387
- `fromkeys()` (*zfit.util.container.DotDict method*), 372
- `func()` (*zfit.core.basefunc.BaseFunc method*), 49
- `func()` (*zfit.core.interfaces.ZfitFunc method*), 90
- `func()` (*zfit.func.BaseFunc method*), 390
- `func()` (*zfit.func.ProdFunc method*), 396
- `func()` (*zfit.func.SimpleFunc method*), 408
- `func()` (*zfit.func.SumFunc method*), 402
- `func()` (*zfit.models.functions.BaseFuncorFunc method*), 224
- `func()` (*zfit.models.functions.ProdFunc method*), 230
- `func()` (*zfit.models.functions.SimpleFunc method*), 236
- `func()` (*zfit.models.functions.SumFunc method*), 242
- `func()` (*zfit.models.functions.ZFunc method*), 248
- `func_integral_chebyshev1()` (*in module zfit.models.polynomials*), 345
- `func_integral_chebyshev2()` (*in module zfit.models.polynomials*), 345
- `func_integral_hermite()` (*in module zfit.models.polynomials*), 345
- `func_integral_laguerre()` (*in module zfit.models.polynomials*), 345
- `FuncorMixin` (*class in zfit.models.basefuncor*), 160
- G**
- `gather_nd()` (*zfit.core.parameter.ComposedResourceVariable method*), 118
- `gather_nd()` (*zfit.core.parameter.Parameter method*), 128
- `gather_nd()` (*zfit.core.parameter.TFBaseVariable method*), 137
- `gather_nd()` (*zfit.Parameter method*), 34
- `Gauss` (*class in zfit.models.dist_tfp*), 190
- `Gauss` (*class in zfit.pdf*), 462
- `gauss_2d()` (*in module zfit.util.diverse*), 373
- `gauss_4d()` (*in module zfit.util.diverse*), 373
- `GaussianConstraint` (*class in zfit.constraint*), 384
- `GaussianConstraint` (*class in zfit.core.constraint*), 71
- `GaussianMixture2D` (*class in zfit.util.diverse*), 373
- `GaussianMixture4D` (*class in zfit.util.diverse*), 373
- `generalized_laguerre_polys_factory()` (*in module zfit.models.polynomials*), 345
- `generalized_laguerre_recurrence_factory()` (*in module zfit.models.polynomials*), 345
- `generalized_laguerre_shape_factory()` (*in module zfit.models.polynomials*), 345
- `get()` (*zfit.util.container.DotDict method*), 372
- `get_auto_number()` (*in module zfit.core.parameter*), 143
- `get_axes()` (*zfit.core.interfaces.ZfitSpace method*), 97
- `get_axes()` (*zfit.core.limits.Space method*), 100
- `get_axes()` (*zfit.core.sample.EventSpace method*), 144
- `get_axes()` (*zfit.Space method*), 43
- `get_cache_counting()` (*zfit.core.data.SampleData class method*), 80
- `get_cache_counting()` (*zfit.core.data.Sampler class method*), 83
- `get_dependents()` (*zfit.ComplexParameter method*), 41
- `get_dependents()` (*zfit.ComposedParameter method*), 40
- `get_dependents()` (*zfit.constraint.GaussianConstraint method*), 384
- `get_dependents()` (*zfit.constraint.SimpleConstraint method*), 383
- `get_dependents()` (*zfit.core.basefunc.BaseFunc method*), 49
- `get_dependents()` (*zfit.core.basemodel.BaseModel method*), 55
- `get_dependents()` (*zfit.core.baseobject.BaseDependentsMixin method*), 59
- `get_dependents()` (*zfit.core.baseobject.BaseNumeric method*), 60
- `get_dependents()` (*zfit.core.basepdf.BasePDF method*), 63
- `get_dependents()` (*zfit.core.constraint.BaseConstraint method*), 69
- `get_dependents()` (*zfit.core.constraint.DistributionConstraint method*), 71

`get_dependents()` (`zfit.core.constraint.GaussianConstraint` method), 72
`get_dependents()` (`zfit.core.constraint.SimpleConstraint` method), 73
`get_dependents()` (`zfit.core.interfaces.ZfitDependentsMixin` method), 89
`get_dependents()` (`zfit.core.interfaces.ZfitFunc` method), 90
`get_dependents()` (`zfit.core.interfaces.ZfitLoss` method), 92
`get_dependents()` (`zfit.core.interfaces.ZfitModel` method), 92
`get_dependents()` (`zfit.core.interfaces.ZfitNumeric` method), 94
`get_dependents()` (`zfit.core.interfaces.ZfitParameter` method), 96
`get_dependents()` (`zfit.core.interfaces.ZfitPDF` method), 95
`get_dependents()` (`zfit.core.loss.BaseLoss` method), 104
`get_dependents()` (`zfit.core.loss.CachedLoss` method), 105
`get_dependents()` (`zfit.core.loss.ExtendedUnbinnedNLL` method), 106
`get_dependents()` (`zfit.core.loss.SimpleLoss` method), 107
`get_dependents()` (`zfit.core.loss.UnbinnedNLL` method), 108
`get_dependents()` (`zfit.core.parameter.BaseComposedParameter` method), 111
`get_dependents()` (`zfit.core.parameter.BaseParameter` method), 112
`get_dependents()` (`zfit.core.parameter.BaseZParameter` method), 112
`get_dependents()` (`zfit.core.parameter.ComplexParameter` method), 114
`get_dependents()` (`zfit.core.parameter.ComposedParameter` method), 115
`get_dependents()` (`zfit.core.parameter.ConstantParameter` method), 123
`get_dependents()` (`zfit.core.parameter.Parameter` method), 128
`get_dependents()` (`zfit.core.parameter.ZfitParameterMixin` method), 142
`get_dependents()` (`zfit.func.BaseFunc` method), 390
`get_dependents()` (`zfit.func.ProdFunc` method), 396
`get_dependents()` (`zfit.func.SimpleFunc` method), 409
`get_dependents()` (`zfit.func.SumFunc` method), 402
`get_dependents()` (`zfit.loss.BaseLoss` method), 415
`get_dependents()` (`zfit.loss.ExtendedUnbinnedNLL` method), 413
`get_dependents()` (`zfit.loss.SimpleLoss` method), 416
`get_dependents()` (`zfit.loss.UnbinnedNLL` method), 414
`get_dependents()` (`zfit.models.basefunctor.FunctorMixin` method), 162
`get_dependents()` (`zfit.models.basic.CustomGaussOLD` method), 168
`get_dependents()` (`zfit.models.basic.Exponential` method), 176
`get_dependents()` (`zfit.models.dist_tfp.ExponentialTFP` method), 184
`get_dependents()` (`zfit.models.dist_tfp.Gauss` method), 193
`get_dependents()` (`zfit.models.dist_tfp.TruncatedGauss` method), 201
`get_dependents()` (`zfit.models.dist_tfp.Uniform` method), 209
`get_dependents()` (`zfit.models.dist_tfp.WrapDistribution` method), 217
`get_dependents()` (`zfit.models.functions.BaseFunctorFunc` method), 225
`get_dependents()` (`zfit.models.functions.ProdFunc` method), 231
`get_dependents()` (`zfit.models.functions.SimpleFunc` method), 237
`get_dependents()` (`zfit.models.functions.SumFunc` method), 243
`get_dependents()` (`zfit.models.functions.ZFunc` method), 249
`get_dependents()` (`zfit.models.functor.BaseFunctor` method), 255
`get_dependents()` (`zfit.models.functor.ProductPDF` method), 263
`get_dependents()` (`zfit.models.functor.SumPDF` method), 271
`get_dependents()` (`zfit.models.physics.CrystalBall` method), 280
`get_dependents()` (`zfit.models.physics.DoubleCB` method), 289
`get_dependents()` (`zfit.models.polynomials.Chebyshev` method), 297
`get_dependents()` (`zfit.models.polynomials.Chebyshev2` method), 306
`get_dependents()` (`zfit.models.polynomials.Hermite` method), 314
`get_dependents()` (`zfit.models.polynomials.Laguerre` method), 322
`get_dependents()` (`zfit.models.polynomials.Legendre` method), 331
`get_dependents()` (`zfit.models.polynomials.RecursivePolynomial` method), 339
`get_dependents()` (`zfit.models.special.SimpleFunctorPDF` method), 348
`get_dependents()` (`zfit.models.special.SimplePDF` method), 356

[get_dependents\(\)](#) ([zfit.models.special.ZPDF method](#)), 364
[get_dependents\(\)](#) ([zfit.param.ConstantParameter method](#)), 420
[get_dependents\(\)](#) ([zfit.Parameter method](#)), 34
[get_dependents\(\)](#) ([zfit.pdf.BaseFuncor method](#)), 431
[get_dependents\(\)](#) ([zfit.pdf.BasePDF method](#)), 423
[get_dependents\(\)](#) ([zfit.pdf.Chebyshev method](#)), 497
[get_dependents\(\)](#) ([zfit.pdf.Chebyshev2 method](#)), 514
[get_dependents\(\)](#) ([zfit.pdf.CrystalBall method](#)), 448
[get_dependents\(\)](#) ([zfit.pdf.DoubleCB method](#)), 456
[get_dependents\(\)](#) ([zfit.pdf.Exponential method](#)), 439
[get_dependents\(\)](#) ([zfit.pdf.Gauss method](#)), 465
[get_dependents\(\)](#) ([zfit.pdf.Hermite method](#)), 522
[get_dependents\(\)](#) ([zfit.pdf.Laguerre method](#)), 531
[get_dependents\(\)](#) ([zfit.pdf.Legendre method](#)), 506
[get_dependents\(\)](#) ([zfit.pdf.ProductPDF method](#)), 547
[get_dependents\(\)](#) ([zfit.pdf.RecursivePolynomial method](#)), 539
[get_dependents\(\)](#) ([zfit.pdf.SimpleFuncorPDF method](#)), 579
[get_dependents\(\)](#) ([zfit.pdf.SimplePDF method](#)), 571
[get_dependents\(\)](#) ([zfit.pdf.SumPDF method](#)), 555
[get_dependents\(\)](#) ([zfit.pdf.TruncatedGauss method](#)), 481
[get_dependents\(\)](#) ([zfit.pdf.Uniform method](#)), 473
[get_dependents\(\)](#) ([zfit.pdf.WrapDistribution method](#)), 489
[get_dependents\(\)](#) ([zfit.pdf.ZPDF method](#)), 563
[get_dependents_auto\(\)](#) (in [module zfit.util.graph](#)), 378
[get_iteration\(\)](#) ([zfit.core.data.Data method](#)), 76
[get_iteration\(\)](#) ([zfit.core.data.SampleData method](#)), 80
[get_iteration\(\)](#) ([zfit.core.data.Sampler method](#)), 83
[get_iteration\(\)](#) ([zfit.data.Data method](#)), 387
[get_logger\(\)](#) (in [module zfit.util.logging](#)), 378
[get_max_axes\(\)](#) ([zfit.core.integration.AnalyticIntegral method](#)), 86
[get_max_integral\(\)](#) ([zfit.core.integration.AnalyticIntegral method](#)), 86
[get_models\(\)](#) ([zfit.core.interfaces.ZfitFuncorMixin method](#)), 92
[get_models\(\)](#) ([zfit.func.ProdFunc method](#)), 397
[get_models\(\)](#) ([zfit.func.SumFunc method](#)), 403
[get_models\(\)](#) ([zfit.models.basefuncor.FuncorMixin method](#)), 162
[get_models\(\)](#) ([zfit.models.functions.BaseFuncorFunc method](#)), 225
[get_models\(\)](#) ([zfit.models.functions.ProdFunc method](#)), 231
[get_models\(\)](#) ([zfit.models.functions.SumFunc method](#)), 243
[get_models\(\)](#) ([zfit.models.funcor.BaseFuncor method](#)), 255
[get_models\(\)](#) ([zfit.models.funcor.ProductPDF method](#)), 263
[get_models\(\)](#) ([zfit.models.funcor.SumPDF method](#)), 271
[get_models\(\)](#) ([zfit.models.special.SimpleFuncorPDF method](#)), 348
[get_models\(\)](#) ([zfit.pdf.BaseFuncor method](#)), 431
[get_models\(\)](#) ([zfit.pdf.ProductPDF method](#)), 547
[get_models\(\)](#) ([zfit.pdf.SimpleFuncorPDF method](#)), 579
[get_models\(\)](#) ([zfit.pdf.SumPDF method](#)), 555
[get_obs_axes\(\)](#) ([zfit.core.limits.Space method](#)), 100
[get_obs_axes\(\)](#) ([zfit.core.sample.EventSpace method](#)), 144
[get_obs_axes\(\)](#) ([zfit.Space method](#)), 43
[get_params\(\)](#) ([zfit.ComplexParameter method](#)), 41
[get_params\(\)](#) ([zfit.ComposedParameter method](#)), 40
[get_params\(\)](#) ([zfit.constraint.GaussianConstraint method](#)), 384
[get_params\(\)](#) ([zfit.constraint.SimpleConstraint method](#)), 383
[get_params\(\)](#) ([zfit.core.basefunc.BaseFunc method](#)), 49
[get_params\(\)](#) ([zfit.core.basemodel.BaseModel method](#)), 55
[get_params\(\)](#) ([zfit.core.baseobject.BaseNumeric method](#)), 60
[get_params\(\)](#) ([zfit.core.basepdf.BasePDF method](#)), 63
[get_params\(\)](#) ([zfit.core.constraint.BaseConstraint method](#)), 69
[get_params\(\)](#) ([zfit.core.constraint.DistributionConstraint method](#)), 71
[get_params\(\)](#) ([zfit.core.constraint.GaussianConstraint method](#)), 72
[get_params\(\)](#) ([zfit.core.constraint.SimpleConstraint method](#)), 73
[get_params\(\)](#) ([zfit.core.interfaces.ZfitFunc method](#)), 90
[get_params\(\)](#) ([zfit.core.interfaces.ZfitModel method](#)), 92
[get_params\(\)](#) ([zfit.core.interfaces.ZfitNumeric method](#)), 94
[get_params\(\)](#) ([zfit.core.interfaces.ZfitParameter method](#)), 96

`get_params()` (*zfit.core.interfaces.ZfitPDF method*), 95
`get_params()` (*zfit.core.parameter.BaseComposedParameter method*), 111
`get_params()` (*zfit.core.parameter.BaseParameter method*), 112
`get_params()` (*zfit.core.parameter.BaseZParameter method*), 112
`get_params()` (*zfit.core.parameter.ComplexParameter method*), 114
`get_params()` (*zfit.core.parameter.ComposedParameter method*), 115
`get_params()` (*zfit.core.parameter.ConstantParameter method*), 124
`get_params()` (*zfit.core.parameter.Parameter method*), 128
`get_params()` (*zfit.core.parameter.ZfitParameterMixin method*), 142
`get_params()` (*zfit.func.BaseFunc method*), 390
`get_params()` (*zfit.func.ProdFunc method*), 397
`get_params()` (*zfit.func.SimpleFunc method*), 409
`get_params()` (*zfit.func.SumFunc method*), 403
`get_params()` (*zfit.models.basefunctor.FunctorMixin method*), 162
`get_params()` (*zfit.models.basic.CustomGaussOLD method*), 168
`get_params()` (*zfit.models.basic.Exponential method*), 176
`get_params()` (*zfit.models.dist_tfp.ExponentialTFP method*), 185
`get_params()` (*zfit.models.dist_tfp.Gauss method*), 193
`get_params()` (*zfit.models.dist_tfp.TruncatedGauss method*), 201
`get_params()` (*zfit.models.dist_tfp.Uniform method*), 209
`get_params()` (*zfit.models.dist_tfp.WrapDistribution method*), 217
`get_params()` (*zfit.models.functions.BaseFunctorFunc method*), 225
`get_params()` (*zfit.models.functions.ProdFunc method*), 231
`get_params()` (*zfit.models.functions.SimpleFunc method*), 237
`get_params()` (*zfit.models.functions.SumFunc method*), 243
`get_params()` (*zfit.models.functions.ZFunc method*), 249
`get_params()` (*zfit.models.functor.BaseFunctor method*), 255
`get_params()` (*zfit.models.functor.ProductPDF method*), 263
`get_params()` (*zfit.models.functor.SumPDF method*), 271
`get_params()` (*zfit.models.physics.CrystalBall method*), 280
`get_params()` (*zfit.models.physics.DoubleCB method*), 289
`get_params()` (*zfit.models.polynomials.Chebyshev method*), 297
`get_params()` (*zfit.models.polynomials.Chebyshev2 method*), 306
`get_params()` (*zfit.models.polynomials.Hermite method*), 314
`get_params()` (*zfit.models.polynomials.Laguerre method*), 323
`get_params()` (*zfit.models.polynomials.Legendre method*), 331
`get_params()` (*zfit.models.polynomials.RecursivePolynomial method*), 339
`get_params()` (*zfit.models.special.SimpleFunctorPDF method*), 348
`get_params()` (*zfit.models.special.SimplePDF method*), 356
`get_params()` (*zfit.models.special.ZPDF method*), 364
`get_params()` (*zfit.param.ConstantParameter method*), 420
`get_params()` (*zfit.Parameter method*), 34
`get_params()` (*zfit.pdf.BaseFunctor method*), 431
`get_params()` (*zfit.pdf.BasePDF method*), 424
`get_params()` (*zfit.pdf.Chebyshev method*), 497
`get_params()` (*zfit.pdf.Chebyshev2 method*), 514
`get_params()` (*zfit.pdf.CrystalBall method*), 448
`get_params()` (*zfit.pdf.DoubleCB method*), 457
`get_params()` (*zfit.pdf.Exponential method*), 439
`get_params()` (*zfit.pdf.Gauss method*), 465
`get_params()` (*zfit.pdf.Hermite method*), 523
`get_params()` (*zfit.pdf.Laguerre method*), 531
`get_params()` (*zfit.pdf.Legendre method*), 506
`get_params()` (*zfit.pdf.ProductPDF method*), 547
`get_params()` (*zfit.pdf.RecursivePolynomial method*), 539
`get_params()` (*zfit.pdf.SimpleFunctorPDF method*), 579
`get_params()` (*zfit.pdf.SimplePDF method*), 571
`get_params()` (*zfit.pdf.SumPDF method*), 555
`get_params()` (*zfit.pdf.TruncatedGauss method*), 481
`get_params()` (*zfit.pdf.Uniform method*), 473
`get_params()` (*zfit.pdf.WrapDistribution method*), 489
`get_params()` (*zfit.pdf.ZPDF method*), 563
`get_reorder_indices()` (*zfit.core.limits.Space method*), 100
`get_reorder_indices()` (*zfit.core.sample.EventSpace method*), 144
`get_reorder_indices()` (*zfit.Space method*), 43
`get_same_obs` (in module *zfit.core.dimension*), 85

[get_shape\(\) \(zfit.core.parameter.ComposedResourceVariable method\), 357](#)
[get_shape\(\) \(zfit.core.parameter.Parameter method\), 118](#)
[get_shape\(\) \(zfit.core.parameter.TFBaseVariable method\), 137](#)
[get_shape\(\) \(zfit.Parameter method\), 35](#)
[get_subspace\(\) \(zfit.core.interfaces.ZfitSpace method\), 97](#)
[get_subspace\(\) \(zfit.core.limits.Space method\), 100](#)
[get_subspace\(\) \(zfit.core.sample.EventSpace method\), 144](#)
[get_subspace\(\) \(zfit.Space method\), 44](#)
[get_verbosity\(\) \(in module zfit.settings\), 584](#)
[get_yield\(\) \(zfit.core.basepdf.BasePDF method\), 64](#)
[get_yield\(\) \(zfit.core.interfaces.ZfitPDF method\), 95](#)
[get_yield\(\) \(zfit.models.basic.CustomGaussOLD method\), 169](#)
[get_yield\(\) \(zfit.models.basic.Exponential method\), 177](#)
[get_yield\(\) \(zfit.models.dist_tfp.ExponentialTFP method\), 185](#)
[get_yield\(\) \(zfit.models.dist_tfp.Gauss method\), 193](#)
[get_yield\(\) \(zfit.models.dist_tfp.TruncatedGauss method\), 201](#)
[get_yield\(\) \(zfit.models.dist_tfp.Uniform method\), 209](#)
[get_yield\(\) \(zfit.models.dist_tfp.WrapDistribution method\), 217](#)
[get_yield\(\) \(zfit.models.functor.BaseFunctor method\), 255](#)
[get_yield\(\) \(zfit.models.functor.ProductPDF method\), 263](#)
[get_yield\(\) \(zfit.models.functor.SumPDF method\), 272](#)
[get_yield\(\) \(zfit.models.physics.CrystalBall method\), 280](#)
[get_yield\(\) \(zfit.models.physics.DoubleCB method\), 289](#)
[get_yield\(\) \(zfit.models.polynomials.Chebyshev method\), 298](#)
[get_yield\(\) \(zfit.models.polynomials.Chebyshev2 method\), 306](#)
[get_yield\(\) \(zfit.models.polynomials.Hermite method\), 314](#)
[get_yield\(\) \(zfit.models.polynomials.Laguerre method\), 323](#)
[get_yield\(\) \(zfit.models.polynomials.Legendre method\), 331](#)
[get_yield\(\) \(zfit.models.polynomials.RecursivePolynomial method\), 339](#)
[get_yield\(\) \(zfit.models.special.SimpleFunctorPDF method\), 349](#)
[get_yield\(\) \(zfit.models.special.SimplePDF method\), 357](#)
[get_yield\(\) \(zfit.models.special.ZPDF method\), 364](#)
[get_yield\(\) \(zfit.pdf.BaseFunctor method\), 432](#)
[get_yield\(\) \(zfit.pdf.BasePDF method\), 424](#)
[get_yield\(\) \(zfit.pdf.Chebyshev method\), 498](#)
[get_yield\(\) \(zfit.pdf.Chebyshev2 method\), 514](#)
[get_yield\(\) \(zfit.pdf.CrystalBall method\), 448](#)
[get_yield\(\) \(zfit.pdf.DoubleCB method\), 457](#)
[get_yield\(\) \(zfit.pdf.Exponential method\), 440](#)
[get_yield\(\) \(zfit.pdf.Gauss method\), 465](#)
[get_yield\(\) \(zfit.pdf.Hermite method\), 523](#)
[get_yield\(\) \(zfit.pdf.Laguerre method\), 531](#)
[get_yield\(\) \(zfit.pdf.Legendre method\), 506](#)
[get_yield\(\) \(zfit.pdf.ProductPDF method\), 547](#)
[get_yield\(\) \(zfit.pdf.RecursivePolynomial method\), 539](#)
[get_yield\(\) \(zfit.pdf.SimpleFunctorPDF method\), 579](#)
[get_yield\(\) \(zfit.pdf.SimplePDF method\), 571](#)
[get_yield\(\) \(zfit.pdf.SumPDF method\), 555](#)
[get_yield\(\) \(zfit.pdf.TruncatedGauss method\), 481](#)
[get_yield\(\) \(zfit.pdf.Uniform method\), 473](#)
[get_yield\(\) \(zfit.pdf.WrapDistribution method\), 489](#)
[get_yield\(\) \(zfit.pdf.ZPDF method\), 563](#)
[gradients\(\) \(zfit.core.basefunc.BaseFunc method\), 49](#)
[gradients\(\) \(zfit.core.basemodel.BaseModel method\), 55](#)
[gradients\(\) \(zfit.core.basepdf.BasePDF method\), 64](#)
[gradients\(\) \(zfit.core.interfaces.ZfitLoss method\), 92](#)
[gradients\(\) \(zfit.core.loss.BaseLoss method\), 105](#)
[gradients\(\) \(zfit.core.loss.CachedLoss method\), 105](#)
[gradients\(\) \(zfit.core.loss.ExtendedUnbinnedNLL method\), 106](#)
[gradients\(\) \(zfit.core.loss.SimpleLoss method\), 107](#)
[gradients\(\) \(zfit.core.loss.UnbinnedNLL method\), 108](#)
[gradients\(\) \(zfit.func.BaseFunc method\), 391](#)
[gradients\(\) \(zfit.func.ProdFunc method\), 397](#)
[gradients\(\) \(zfit.func.SimpleFunc method\), 409](#)
[gradients\(\) \(zfit.func.SumFunc method\), 403](#)
[gradients\(\) \(zfit.loss.BaseLoss method\), 415](#)
[gradients\(\) \(zfit.loss.ExtendedUnbinnedNLL method\), 413](#)
[gradients\(\) \(zfit.loss.SimpleLoss method\), 416](#)
[gradients\(\) \(zfit.loss.UnbinnedNLL method\), 414](#)
[gradients\(\) \(zfit.models.basefunctor.FunctorMixin method\), 162](#)
[gradients\(\) \(zfit.models.basic.CustomGaussOLD method\), 169](#)
[gradients\(\) \(zfit.models.basic.Exponential method\), 177](#)
[gradients\(\) \(zfit.models.dist_tfp.ExponentialTFP method\), 185](#)

- `gradients()` (*zfit.models.dist_tfp.Gauss method*), 193
 - `gradients()` (*zfit.models.dist_tfp.TruncatedGauss method*), 201
 - `gradients()` (*zfit.models.dist_tfp.Uniform method*), 210
 - `gradients()` (*zfit.models.dist_tfp.WrapDistribution method*), 217
 - `gradients()` (*zfit.models.functions.BaseFuncorFunc method*), 225
 - `gradients()` (*zfit.models.functions.ProdFunc method*), 231
 - `gradients()` (*zfit.models.functions.SimpleFunc method*), 237
 - `gradients()` (*zfit.models.functions.SumFunc method*), 243
 - `gradients()` (*zfit.models.functions.ZFunc method*), 249
 - `gradients()` (*zfit.models.funcor.BaseFuncor method*), 256
 - `gradients()` (*zfit.models.funcor.ProductPDF method*), 264
 - `gradients()` (*zfit.models.funcor.SumPDF method*), 272
 - `gradients()` (*zfit.models.physics.CrystalBall method*), 280
 - `gradients()` (*zfit.models.physics.DoubleCB method*), 289
 - `gradients()` (*zfit.models.polynomials.Chebyshev method*), 298
 - `gradients()` (*zfit.models.polynomials.Chebyshev2 method*), 306
 - `gradients()` (*zfit.models.polynomials.Hermite method*), 315
 - `gradients()` (*zfit.models.polynomials.Laguerre method*), 323
 - `gradients()` (*zfit.models.polynomials.Legendre method*), 331
 - `gradients()` (*zfit.models.polynomials.RecursivePolynomial method*), 339
 - `gradients()` (*zfit.models.special.SimpleFuncorPDF method*), 349
 - `gradients()` (*zfit.models.special.SimplePDF method*), 357
 - `gradients()` (*zfit.models.special.ZPDF method*), 365
 - `gradients()` (*zfit.pdf.BaseFuncor method*), 432
 - `gradients()` (*zfit.pdf.BasePDF method*), 424
 - `gradients()` (*zfit.pdf.Chebyshev method*), 498
 - `gradients()` (*zfit.pdf.Chebyshev2 method*), 515
 - `gradients()` (*zfit.pdf.CrystalBall method*), 448
 - `gradients()` (*zfit.pdf.DoubleCB method*), 457
 - `gradients()` (*zfit.pdf.Exponential method*), 440
 - `gradients()` (*zfit.pdf.Gauss method*), 465
 - `gradients()` (*zfit.pdf.Hermite method*), 523
 - `gradients()` (*zfit.pdf.Laguerre method*), 531
 - `gradients()` (*zfit.pdf.Legendre method*), 506
 - `gradients()` (*zfit.pdf.ProductPDF method*), 547
 - `gradients()` (*zfit.pdf.RecursivePolynomial method*), 539
 - `gradients()` (*zfit.pdf.SimpleFuncorPDF method*), 579
 - `gradients()` (*zfit.pdf.SimplePDF method*), 571
 - `gradients()` (*zfit.pdf.SumPDF method*), 555
 - `gradients()` (*zfit.pdf.TruncatedGauss method*), 481
 - `gradients()` (*zfit.pdf.Uniform method*), 473
 - `gradients()` (*zfit.pdf.WrapDistribution method*), 489
 - `gradients()` (*zfit.pdf.ZPDF method*), 563
 - `graph` (*zfit.core.parameter.ComposedResourceVariable attribute*), 118
 - `graph` (*zfit.core.parameter.Parameter attribute*), 128
 - `graph` (*zfit.core.parameter.TFBaseVariable attribute*), 137
 - `graph` (*zfit.Parameter attribute*), 35
- ## H
- `handle` (*zfit.core.parameter.ComposedResourceVariable attribute*), 118
 - `handle` (*zfit.core.parameter.Parameter attribute*), 128
 - `handle` (*zfit.core.parameter.TFBaseVariable attribute*), 137
 - `handle` (*zfit.Parameter attribute*), 35
 - `has_limits` (*zfit.core.parameter.Parameter attribute*), 128
 - `has_limits` (*zfit.Parameter attribute*), 35
 - `Hermite` (*class in zfit.models.polynomials*), 311
 - `Hermite` (*class in zfit.pdf*), 519
 - `hermite_recurrence()` (*in module zfit.models.polynomials*), 345
 - `hermite_shape()` (*in module zfit.models.polynomials*), 345
 - `hesse()` (*zfit.minimizers.fitresult.FitResult method*), 152
 - `hesse()` (*zfit.minimizers.interface.ZfitResult method*), 153
- ## I
- `imag` (*zfit.ComplexParameter attribute*), 41
 - `imag` (*zfit.core.parameter.ComplexParameter attribute*), 114
 - `IncompatibleError`, 374
 - `independent` (*zfit.ComplexParameter attribute*), 41
 - `independent` (*zfit.ComposedParameter attribute*), 40
 - `independent` (*zfit.core.interfaces.ZfitParameter attribute*), 97
 - `independent` (*zfit.core.parameter.BaseComposedParameter attribute*), 111
 - `independent` (*zfit.core.parameter.BaseParameter attribute*), 112

- `independent` (`zfit.core.parameter.BaseZParameter attribute`), 113
- `independent` (`zfit.core.parameter.ComplexParameter attribute`), 114
- `independent` (`zfit.core.parameter.ComposedParameter attribute`), 115
- `independent` (`zfit.core.parameter.ConstantParameter attribute`), 124
- `independent` (`zfit.core.parameter.Parameter attribute`), 128
- `independent` (`zfit.param.ConstantParameter attribute`), 421
- `independent` (`zfit.Parameter attribute`), 35
- `info` (`zfit.minimizers.fitresult.FitResult attribute`), 152
- `initial_value` (`zfit.core.parameter.ComposedResourceVariable attribute`), 119
- `initial_value` (`zfit.core.parameter.Parameter attribute`), 128
- `initial_value` (`zfit.core.parameter.TFBaseVariable attribute`), 137
- `initial_value` (`zfit.Parameter attribute`), 35
- `initialize()` (`zfit.core.data.Data method`), 76
- `initialize()` (`zfit.core.data.SampleData method`), 80
- `initialize()` (`zfit.core.data.Sampler method`), 83
- `initialize()` (`zfit.data.Data method`), 387
- `initialized_value()` (`zfit.core.parameter.ComposedResourceVariable method`), 119
- `initialized_value()` (`zfit.core.parameter.Parameter method`), 128
- `initialized_value()` (`zfit.core.parameter.TFBaseVariable method`), 137
- `initialized_value()` (`zfit.Parameter method`), 35
- `initializer` (`zfit.core.parameter.ComposedResourceVariable attribute`), 119
- `initializer` (`zfit.core.parameter.Parameter attribute`), 128
- `initializer` (`zfit.core.parameter.TFBaseVariable attribute`), 137
- `initializer` (`zfit.Parameter attribute`), 35
- `Integral` (class in `zfit.core.integration`), 87
- `integrate()` (`zfit.core.basefunc.BaseFunc method`), 49
- `integrate()` (`zfit.core.basemodel.BaseModel method`), 55
- `integrate()` (`zfit.core.basepdf.BasePDF method`), 64
- `integrate()` (`zfit.core.integration.AnalyticIntegral method`), 87
- `integrate()` (`zfit.core.interfaces.ZfitFunc method`), 90
- `integrate()` (`zfit.core.interfaces.ZfitModel method`), 92
- `integrate()` (`zfit.core.interfaces.ZfitPDF method`), 95
- `integrate()` (`zfit.func.BaseFunc method`), 391
- `integrate()` (`zfit.func.ProdFunc method`), 397
- `integrate()` (`zfit.func.SimpleFunc method`), 409
- `integrate()` (`zfit.func.SumFunc method`), 403
- `integrate()` (`zfit.models.basefunc.FunctorMixin method`), 162
- `integrate()` (`zfit.models.basic.CustomGaussOLD method`), 169
- `integrate()` (`zfit.models.basic.Exponential method`), 177
- `integrate()` (`zfit.models.dist_tfp.ExponentialTFP method`), 185
- `integrate()` (`zfit.models.dist_tfp.Gauss method`), 193
- `integrate()` (`zfit.models.dist_tfp.TruncatedGauss method`), 201
- `integrate()` (`zfit.models.dist_tfp.Uniform method`), 210
- `integrate()` (`zfit.models.dist_tfp.WrapDistribution method`), 217
- `integrate()` (`zfit.models.functions.BaseFunctorFunc method`), 225
- `integrate()` (`zfit.models.functions.ProdFunc method`), 231
- `integrate()` (`zfit.models.functions.SimpleFunc method`), 237
- `integrate()` (`zfit.models.functions.SumFunc method`), 243
- `integrate()` (`zfit.models.functions.ZFunc method`), 249
- `integrate()` (`zfit.models.functor.BaseFunctor method`), 256
- `integrate()` (`zfit.models.functor.ProductPDF method`), 264
- `integrate()` (`zfit.models.functor.SumPDF method`), 272
- `integrate()` (`zfit.models.physics.CrystalBall method`), 280
- `integrate()` (`zfit.models.physics.DoubleCB method`), 289
- `integrate()` (`zfit.models.polynomials.Chebyshev method`), 298
- `integrate()` (`zfit.models.polynomials.Chebyshev2 method`), 306
- `integrate()` (`zfit.models.polynomials.Hermite method`), 315
- `integrate()` (`zfit.models.polynomials.Laguerre method`), 323
- `integrate()` (`zfit.models.polynomials.Legendre method`), 331
- `integrate()` (`zfit.models.polynomials.RecursivePolynomial method`), 339
- `integrate()` (`zfit.models.special.SimpleFunctorPDF method`), 347

- [method\), 349](#)
- [integrate\(\) \(zfit.models.special.SimplePDF method\), 357](#)
- [integrate\(\) \(zfit.models.special.ZPDF method\), 365](#)
- [integrate\(\) \(zfit.pdf.BaseFunctor method\), 432](#)
- [integrate\(\) \(zfit.pdf.BasePDF method\), 424](#)
- [integrate\(\) \(zfit.pdf.Chebyshev method\), 498](#)
- [integrate\(\) \(zfit.pdf.Chebyshev2 method\), 515](#)
- [integrate\(\) \(zfit.pdf.CrystalBall method\), 448](#)
- [integrate\(\) \(zfit.pdf.DoubleCB method\), 457](#)
- [integrate\(\) \(zfit.pdf.Exponential method\), 440](#)
- [integrate\(\) \(zfit.pdf.Gauss method\), 465](#)
- [integrate\(\) \(zfit.pdf.Hermite method\), 523](#)
- [integrate\(\) \(zfit.pdf.Laguerre method\), 531](#)
- [integrate\(\) \(zfit.pdf.Legendre method\), 506](#)
- [integrate\(\) \(zfit.pdf.ProductPDF method\), 547](#)
- [integrate\(\) \(zfit.pdf.RecursivePolynomial method\), 539](#)
- [integrate\(\) \(zfit.pdf.SimpleFunctorPDF method\), 579](#)
- [integrate\(\) \(zfit.pdf.SimplePDF method\), 571](#)
- [integrate\(\) \(zfit.pdf.SumPDF method\), 555](#)
- [integrate\(\) \(zfit.pdf.TruncatedGauss method\), 481](#)
- [integrate\(\) \(zfit.pdf.Uniform method\), 473](#)
- [integrate\(\) \(zfit.pdf.WrapDistribution method\), 489](#)
- [integrate\(\) \(zfit.pdf.ZPDF method\), 563](#)
- [Integration \(class in zfit.core.integration\), 87](#)
- [IntentionNotUnambiguousError, 374](#)
- [interpolate\(\) \(in module zfit.core.math\), 109](#)
- [invalidates_cache\(\) \(in module zfit.util.cache\), 371](#)
- [is_combinable\(\) \(in module zfit.core.dimension\), 86](#)
- [is_container\(\) \(in module zfit.util.container\), 372](#)
- [is_extended \(zfit.core.basepdf.BasePDF attribute\), 64](#)
- [is_extended \(zfit.core.interfaces.ZfitPDF attribute\), 95](#)
- [is_extended \(zfit.models.basic.CustomGaussOLD attribute\), 169](#)
- [is_extended \(zfit.models.basic.Exponential attribute\), 177](#)
- [is_extended \(zfit.models.dist_tfp.ExponentialTFP attribute\), 185](#)
- [is_extended \(zfit.models.dist_tfp.Gauss attribute\), 193](#)
- [is_extended \(zfit.models.dist_tfp.TruncatedGauss attribute\), 202](#)
- [is_extended \(zfit.models.dist_tfp.Uniform attribute\), 210](#)
- [is_extended \(zfit.models.dist_tfp.WrapDistribution attribute\), 218](#)
- [is_extended \(zfit.models.functor.BaseFunctor attribute\), 256](#)
- [is_extended \(zfit.models.functor.ProductPDF attribute\), 264](#)
- [is_extended \(zfit.models.functor.SumPDF attribute\), 272](#)
- [is_extended \(zfit.models.physics.CrystalBall attribute\), 281](#)
- [is_extended \(zfit.models.physics.DoubleCB attribute\), 290](#)
- [is_extended \(zfit.models.polynomials.Chebyshev attribute\), 298](#)
- [is_extended \(zfit.models.polynomials.Chebyshev2 attribute\), 306](#)
- [is_extended \(zfit.models.polynomials.Hermite attribute\), 315](#)
- [is_extended \(zfit.models.polynomials.Laguerre attribute\), 323](#)
- [is_extended \(zfit.models.polynomials.Legendre attribute\), 332](#)
- [is_extended \(zfit.models.polynomials.RecursivePolynomial attribute\), 340](#)
- [is_extended \(zfit.models.special.SimpleFunctorPDF attribute\), 349](#)
- [is_extended \(zfit.models.special.SimplePDF attribute\), 357](#)
- [is_extended \(zfit.models.special.ZPDF attribute\), 365](#)
- [is_extended \(zfit.pdf.BaseFunctor attribute\), 432](#)
- [is_extended \(zfit.pdf.BasePDF attribute\), 424](#)
- [is_extended \(zfit.pdf.Chebyshev attribute\), 498](#)
- [is_extended \(zfit.pdf.Chebyshev2 attribute\), 515](#)
- [is_extended \(zfit.pdf.CrystalBall attribute\), 449](#)
- [is_extended \(zfit.pdf.DoubleCB attribute\), 457](#)
- [is_extended \(zfit.pdf.Exponential attribute\), 440](#)
- [is_extended \(zfit.pdf.Gauss attribute\), 465](#)
- [is_extended \(zfit.pdf.Hermite attribute\), 523](#)
- [is_extended \(zfit.pdf.Laguerre attribute\), 532](#)
- [is_extended \(zfit.pdf.Legendre attribute\), 506](#)
- [is_extended \(zfit.pdf.ProductPDF attribute\), 547](#)
- [is_extended \(zfit.pdf.RecursivePolynomial attribute\), 540](#)
- [is_extended \(zfit.pdf.SimpleFunctorPDF attribute\), 579](#)
- [is_extended \(zfit.pdf.SimplePDF attribute\), 572](#)
- [is_extended \(zfit.pdf.SumPDF attribute\), 556](#)
- [is_extended \(zfit.pdf.TruncatedGauss attribute\), 482](#)
- [is_extended \(zfit.pdf.Uniform attribute\), 474](#)
- [is_extended \(zfit.pdf.WrapDistribution attribute\), 490](#)
- [is_extended \(zfit.pdf.ZPDF attribute\), 564](#)
- [is_generator \(zfit.core.sample.EventSpace attribute\), 145](#)
- [is_initialized\(\) \(zfit.core.parameter.ComposedResourceVariable method\), 119](#)
- [is_initialized\(\) \(zfit.core.parameter.Parameter](#)

- method*), 128
 - `is_initialized()` (*zfit.core.parameter.TFBaseVariable* *method*), 137
 - `is_initialized()` (*zfit.Parameter* *method*), 35
 - `items()` (*zfit.util.container.DotDict* *method*), 372
 - `iter_areas()` (*zfit.core.interfaces.ZfitSpace* *method*), 97
 - `iter_areas()` (*zfit.core.limits.Space* *method*), 100
 - `iter_areas()` (*zfit.core.sample.EventSpace* *method*), 145
 - `iter_areas()` (*zfit.Space* *method*), 44
 - `iter_limits()` (*zfit.core.interfaces.ZfitSpace* *method*), 97
 - `iter_limits()` (*zfit.core.limits.Space* *method*), 101
 - `iter_limits()` (*zfit.core.sample.EventSpace* *method*), 145
 - `iter_limits()` (*zfit.Space* *method*), 44
 - `iterator` (*zfit.core.data.Data* *attribute*), 76
 - `iterator` (*zfit.core.data.SampleData* *attribute*), 80
 - `iterator` (*zfit.core.data.Sampler* *attribute*), 83
 - `iterator` (*zfit.data.Data* *attribute*), 387
- ## K
- `keys()` (*zfit.util.container.DotDict* *method*), 372
- ## L
- Laguerre* (*class in zfit.models.polynomials*), 319
 - Laguerre* (*class in zfit.pdf*), 528
 - `laguerre_recurrence()` (*in module zfit.models.polynomials*), 345
 - `laguerre_shape()` (*in module zfit.models.polynomials*), 345
 - `laguerre_shape_alpha_minusone()` (*in module zfit.models.polynomials*), 345
 - Legendre* (*class in zfit.models.polynomials*), 328
 - Legendre* (*class in zfit.pdf*), 503
 - `legendre_integral()` (*in module zfit.models.polynomials*), 345
 - `legendre_recurrence()` (*in module zfit.models.polynomials*), 345
 - `legendre_shape()` (*in module zfit.models.polynomials*), 345
 - LightDataset* (*class in zfit.core.data*), 77
 - `limit1d` (*zfit.core.limits.Space* *attribute*), 101
 - `limit1d` (*zfit.core.sample.EventSpace* *attribute*), 145
 - `limit1d` (*zfit.Space* *attribute*), 44
 - `limit2d` (*zfit.core.limits.Space* *attribute*), 101
 - `limit2d` (*zfit.core.sample.EventSpace* *attribute*), 145
 - `limit2d` (*zfit.Space* *attribute*), 44
 - `limits` (*zfit.core.interfaces.ZfitSpace* *attribute*), 97
 - `limits` (*zfit.core.limits.Space* *attribute*), 101
 - `limits` (*zfit.core.sample.EventSpace* *attribute*), 145
 - `limits` (*zfit.Space* *attribute*), 45
 - `limits1d` (*zfit.core.limits.Space* *attribute*), 101
 - `limits1d` (*zfit.core.sample.EventSpace* *attribute*), 145
 - `limits1d` (*zfit.Space* *attribute*), 45
 - `limits_consistent()` (*in module zfit.core.dimension*), 86
 - `limits_overlap()` (*in module zfit.core.dimension*), 86
 - LimitsIncompatibleError*, 374
 - LimitsNotSpecifiedError*, 374
 - LimitsOverdefinedError*, 374
 - LimitsUnderdefinedError*, 374
 - `load()` (*zfit.ComplexParameter* *method*), 41
 - `load()` (*zfit.ComposedParameter* *method*), 40
 - `load()` (*zfit.core.parameter.BaseComposedParameter* *method*), 111
 - `load()` (*zfit.core.parameter.BaseZParameter* *method*), 113
 - `load()` (*zfit.core.parameter.ComplexParameter* *method*), 114
 - `load()` (*zfit.core.parameter.ComposedParameter* *method*), 115
 - `load()` (*zfit.core.parameter.ComposedResourceVariable* *method*), 119
 - `load()` (*zfit.core.parameter.ComposedVariable* *method*), 123
 - `load()` (*zfit.core.parameter.ConstantParameter* *method*), 124
 - `load()` (*zfit.core.parameter.Parameter* *method*), 129
 - `load()` (*zfit.core.parameter.TFBaseVariable* *method*), 137
 - `load()` (*zfit.param.ConstantParameter* *method*), 421
 - `load()` (*zfit.Parameter* *method*), 35
 - `log()` (*in module zfit.ztf.wrapping_tf*), 380
 - `log_pdf()` (*zfit.core.basepdf.BasePDF* *method*), 64
 - `log_pdf()` (*zfit.models.basic.CustomGaussOLD* *method*), 169
 - `log_pdf()` (*zfit.models.basic.Exponential* *method*), 177
 - `log_pdf()` (*zfit.models.dist_tfp.ExponentialTFP* *method*), 185
 - `log_pdf()` (*zfit.models.dist_tfp.Gauss* *method*), 193
 - `log_pdf()` (*zfit.models.dist_tfp.TruncatedGauss* *method*), 202
 - `log_pdf()` (*zfit.models.dist_tfp.Uniform* *method*), 210
 - `log_pdf()` (*zfit.models.dist_tfp.WrapDistribution* *method*), 218
 - `log_pdf()` (*zfit.models.functor.BaseFunctor* *method*), 256
 - `log_pdf()` (*zfit.models.functor.ProductPDF* *method*), 264
 - `log_pdf()` (*zfit.models.functor.SumPDF* *method*), 272
 - `log_pdf()` (*zfit.models.physics.CrystalBall* *method*), 281
 - `log_pdf()` (*zfit.models.physics.DoubleCB* *method*), 290

`log_pdf()` (*zfit.models.polynomials.Chebyshev method*), 298
`log_pdf()` (*zfit.models.polynomials.Chebyshev2 method*), 307
`log_pdf()` (*zfit.models.polynomials.Hermite method*), 315
`log_pdf()` (*zfit.models.polynomials.Laguerre method*), 323
`log_pdf()` (*zfit.models.polynomials.Legendre method*), 332
`log_pdf()` (*zfit.models.polynomials.RecursivePolynomial method*), 340
`log_pdf()` (*zfit.models.special.SimpleFunctorPDF method*), 349
`log_pdf()` (*zfit.models.special.SimplePDF method*), 357
`log_pdf()` (*zfit.models.special.ZPDF method*), 365
`log_pdf()` (*zfit.pdf.BaseFunctor method*), 432
`log_pdf()` (*zfit.pdf.BasePDF method*), 424
`log_pdf()` (*zfit.pdf.Chebyshev method*), 498
`log_pdf()` (*zfit.pdf.Chebyshev2 method*), 515
`log_pdf()` (*zfit.pdf.CrystalBall method*), 449
`log_pdf()` (*zfit.pdf.DoubleCB method*), 457
`log_pdf()` (*zfit.pdf.Exponential method*), 440
`log_pdf()` (*zfit.pdf.Gauss method*), 465
`log_pdf()` (*zfit.pdf.Hermite method*), 523
`log_pdf()` (*zfit.pdf.Laguerre method*), 532
`log_pdf()` (*zfit.pdf.Legendre method*), 507
`log_pdf()` (*zfit.pdf.ProductPDF method*), 548
`log_pdf()` (*zfit.pdf.RecursivePolynomial method*), 540
`log_pdf()` (*zfit.pdf.SimpleFunctorPDF method*), 579
`log_pdf()` (*zfit.pdf.SimplePDF method*), 572
`log_pdf()` (*zfit.pdf.SumPDF method*), 556
`log_pdf()` (*zfit.pdf.TruncatedGauss method*), 482
`log_pdf()` (*zfit.pdf.Uniform method*), 474
`log_pdf()` (*zfit.pdf.WrapDistribution method*), 490
`log_pdf()` (*zfit.pdf.ZPDF method*), 564
`LogicalUndefinedOperationError`, 374
`loss` (*zfit.minimizers.fitresult.FitResult attribute*), 152
`loss` (*zfit.minimizers.interface.ZfitResult attribute*), 153
`lower` (*zfit.core.interfaces.ZfitSpace attribute*), 97
`lower` (*zfit.core.limits.Space attribute*), 102
`lower` (*zfit.core.sample.EventSpace attribute*), 146
`lower` (*zfit.Space attribute*), 45
`lower_limit` (*zfit.core.parameter.Parameter attribute*), 129
`lower_limit` (*zfit.Parameter attribute*), 35
M
`mc_integrate()` (*in module zfit.core.integration*), 88
`MetaBaseParameter` (*class in zfit.core.parameter*), 124
`minimize()` (*zfit.minimize.Adam method*), 417
`minimize()` (*zfit.minimize.Minuit method*), 419
`minimize()` (*zfit.minimize.Scipy method*), 419
`minimize()` (*zfit.minimize.WrapOptimizer method*), 417
`minimize()` (*zfit.minimizers.base_tf.WrapOptimizer method*), 149
`minimize()` (*zfit.minimizers.baseminimizer.BaseMinimizer method*), 149
`minimize()` (*zfit.minimizers.interface.ZfitMinimizer method*), 153
`minimize()` (*zfit.minimizers.minimizer_minuit.Minuit method*), 154
`minimize()` (*zfit.minimizers.minimizer_tfp.BFGSMinimizer method*), 155
`minimize()` (*zfit.minimizers.minimizers_scipy.Scipy method*), 156
`minimize()` (*zfit.minimizers.optimizers_tf.Adam method*), 156
`minimize()` (*zfit.minimizers.tf_external_optimizer.ExternalOptimizerInterface method*), 158
`minimize()` (*zfit.minimizers.tf_external_optimizer.ScipyOptimizerInterface method*), 160
`minimize_nan()` (*zfit.minimizers.baseminimizer.BaseStrategy method*), 150
`minimize_nan()` (*zfit.minimizers.baseminimizer.DefaultStrategy method*), 150
`minimize_nan()` (*zfit.minimizers.baseminimizer.ToyStrategyFail method*), 150
`minimize_nan()` (*zfit.minimizers.baseminimizer.ZfitStrategy method*), 150
`minimizer` (*zfit.minimizers.fitresult.FitResult attribute*), 152
`minimizer` (*zfit.minimizers.interface.ZfitResult attribute*), 153
`Minuit` (*class in zfit.minimize*), 418
`Minuit` (*class in zfit.minimizers.minimizer_minuit*), 154
`MinuitMinimizer` (*in module zfit.minimize*), 417
`mod` (*zfit.ComplexParameter attribute*), 41
`mod` (*zfit.core.parameter.ComplexParameter attribute*), 114
`model` (*zfit.core.interfaces.ZfitLoss attribute*), 92
`model` (*zfit.core.loss.BaseLoss attribute*), 105
`model` (*zfit.core.loss.CachedLoss attribute*), 106
`model` (*zfit.core.loss.ExtendedUnbinnedNLL attribute*), 106
`model` (*zfit.core.loss.SimpleLoss attribute*), 108
`model` (*zfit.core.loss.UnbinnedNLL attribute*), 108
`model` (*zfit.loss.BaseLoss attribute*), 415
`model` (*zfit.loss.ExtendedUnbinnedNLL attribute*), 413
`model` (*zfit.loss.SimpleLoss attribute*), 416
`model` (*zfit.loss.UnbinnedNLL attribute*), 414
`model()` (*zfit.util.diverse.GaussianMixture2D method*), 373
`model()` (*zfit.util.diverse.GaussianMixture4D method*), 373

[ModelIncompatibleError](#), 375
[models \(zfit.core.interfaces.ZfitFuncMixin attribute\)](#), 92
[models \(zfit.func.ProdFunc attribute\)](#), 397
[models \(zfit.func.SumFunc attribute\)](#), 403
[models \(zfit.models.basefunc.FunctorMixin attribute\)](#), 162
[models \(zfit.models.functions.BaseFuncFunc attribute\)](#), 225
[models \(zfit.models.functions.ProdFunc attribute\)](#), 231
[models \(zfit.models.functions.SumFunc attribute\)](#), 243
[models \(zfit.models.functor.BaseFunc attribute\)](#), 256
[models \(zfit.models.functor.ProductPDF attribute\)](#), 264
[models \(zfit.models.functor.SumPDF attribute\)](#), 272
[models \(zfit.models.special.SimpleFuncPDF attribute\)](#), 349
[models \(zfit.pdf.BaseFunc attribute\)](#), 432
[models \(zfit.pdf.ProductPDF attribute\)](#), 548
[models \(zfit.pdf.SimpleFuncPDF attribute\)](#), 580
[models \(zfit.pdf.SumPDF attribute\)](#), 556
[mro\(\)](#) ([zfit.core.parameter.MetaBaseParameter method](#)), 124
[MultipleLimitsNotImplementedError](#), 375
[multiply\(\)](#) (in module [zfit.core.operations](#)), 110
[multiply_func_func\(\)](#) (in module [zfit.core.operations](#)), 110
[multiply_param_func\(\)](#) (in module [zfit.core.operations](#)), 110
[multiply_param_param\(\)](#) (in module [zfit.core.operations](#)), 110
[multiply_param_pdf\(\)](#) (in module [zfit.core.operations](#)), 110
[multiply_pdf_pdf\(\)](#) (in module [zfit.core.operations](#)), 110
[multivariate_gauss\(\)](#) (in module [zfit.util.diverse](#)), 373

N

[n_cpu \(zfit.util.execution.RunManager attribute\)](#), 377
[n_limits \(zfit.core.interfaces.ZfitSpace attribute\)](#), 97
[n_limits \(zfit.core.limits.Space attribute\)](#), 102
[n_limits \(zfit.core.sample.EventSpace attribute\)](#), 146
[n_limits \(zfit.Space attribute\)](#), 45
[n_obs \(zfit.core.basefunc.BaseFunc attribute\)](#), 49
[n_obs \(zfit.core.basemodel.BaseModel attribute\)](#), 55
[n_obs \(zfit.core.basepdf.BasePDF attribute\)](#), 64
[n_obs \(zfit.core.data.Data attribute\)](#), 76
[n_obs \(zfit.core.data.SampleData attribute\)](#), 80
[n_obs \(zfit.core.data.Sampler attribute\)](#), 83
[n_obs \(zfit.core.dimension.BaseDimensional attribute\)](#), 84
[n_obs \(zfit.core.integration.PartialIntegralSampleData attribute\)](#), 88
[n_obs \(zfit.core.interfaces.ZfitData attribute\)](#), 89

[n_obs \(zfit.core.interfaces.ZfitDimensional attribute\)](#), 90
[n_obs \(zfit.core.interfaces.ZfitFunc attribute\)](#), 90
[n_obs \(zfit.core.interfaces.ZfitModel attribute\)](#), 93
[n_obs \(zfit.core.interfaces.ZfitPDF attribute\)](#), 95
[n_obs \(zfit.core.interfaces.ZfitSpace attribute\)](#), 97
[n_obs \(zfit.core.limits.Space attribute\)](#), 102
[n_obs \(zfit.core.sample.EventSpace attribute\)](#), 146
[n_obs \(zfit.data.Data attribute\)](#), 387
[n_obs \(zfit.func.BaseFunc attribute\)](#), 391
[n_obs \(zfit.func.ProdFunc attribute\)](#), 397
[n_obs \(zfit.func.SimpleFunc attribute\)](#), 409
[n_obs \(zfit.func.SumFunc attribute\)](#), 403
[n_obs \(zfit.models.basefunc.FunctorMixin attribute\)](#), 162
[n_obs \(zfit.models.basic.CustomGaussOLD attribute\)](#), 169
[n_obs \(zfit.models.basic.Exponential attribute\)](#), 177
[n_obs \(zfit.models.dist_tfp.ExponentialTFP attribute\)](#), 185
[n_obs \(zfit.models.dist_tfp.Gauss attribute\)](#), 194
[n_obs \(zfit.models.dist_tfp.TruncatedGauss attribute\)](#), 202
[n_obs \(zfit.models.dist_tfp.Uniform attribute\)](#), 210
[n_obs \(zfit.models.dist_tfp.WrapDistribution attribute\)](#), 218
[n_obs \(zfit.models.functions.BaseFuncFunc attribute\)](#), 225
[n_obs \(zfit.models.functions.ProdFunc attribute\)](#), 231
[n_obs \(zfit.models.functions.SimpleFunc attribute\)](#), 237
[n_obs \(zfit.models.functions.SumFunc attribute\)](#), 243
[n_obs \(zfit.models.functions.ZFunc attribute\)](#), 249
[n_obs \(zfit.models.functor.BaseFunc attribute\)](#), 256
[n_obs \(zfit.models.functor.ProductPDF attribute\)](#), 264
[n_obs \(zfit.models.functor.SumPDF attribute\)](#), 272
[n_obs \(zfit.models.physics.CrystalBall attribute\)](#), 281
[n_obs \(zfit.models.physics.DoubleCB attribute\)](#), 290
[n_obs \(zfit.models.polynomials.Chebyshev attribute\)](#), 298
[n_obs \(zfit.models.polynomials.Chebyshev2 attribute\)](#), 307
[n_obs \(zfit.models.polynomials.Hermite attribute\)](#), 315
[n_obs \(zfit.models.polynomials.Laguerre attribute\)](#), 324
[n_obs \(zfit.models.polynomials.Legendre attribute\)](#), 332
[n_obs \(zfit.models.polynomials.RecursivePolynomial attribute\)](#), 340
[n_obs \(zfit.models.special.SimpleFuncPDF attribute\)](#), 349
[n_obs \(zfit.models.special.SimplePDF attribute\)](#), 357
[n_obs \(zfit.models.special.ZPDF attribute\)](#), 365
[n_obs \(zfit.pdf.BaseFunc attribute\)](#), 432
[n_obs \(zfit.pdf.BasePDF attribute\)](#), 425
[n_obs \(zfit.pdf.Chebyshev attribute\)](#), 498
[n_obs \(zfit.pdf.Chebyshev2 attribute\)](#), 515
[n_obs \(zfit.pdf.CrystalBall attribute\)](#), 449

- `n_obs` (`zfit.pdf.DoubleCB` attribute), 458
- `n_obs` (`zfit.pdf.Exponential` attribute), 440
- `n_obs` (`zfit.pdf.Gauss` attribute), 466
- `n_obs` (`zfit.pdf.Hermite` attribute), 524
- `n_obs` (`zfit.pdf.Laguerre` attribute), 532
- `n_obs` (`zfit.pdf.Legendre` attribute), 507
- `n_obs` (`zfit.pdf.ProductPDF` attribute), 548
- `n_obs` (`zfit.pdf.RecursivePolynomial` attribute), 540
- `n_obs` (`zfit.pdf.SimpleFuncPDF` attribute), 580
- `n_obs` (`zfit.pdf.SimplePDF` attribute), 572
- `n_obs` (`zfit.pdf.SumPDF` attribute), 556
- `n_obs` (`zfit.pdf.TruncatedGauss` attribute), 482
- `n_obs` (`zfit.pdf.Uniform` attribute), 474
- `n_obs` (`zfit.pdf.WrapDistribution` attribute), 490
- `n_obs` (`zfit.pdf.ZPDF` attribute), 564
- `n_obs` (`zfit.Space` attribute), 45
- `n_samples` (`zfit.core.data.Sampler` attribute), 83
- `name` (`zfit.ComplexParameter` attribute), 41
- `name` (`zfit.ComposedParameter` attribute), 40
- `name` (`zfit.constraint.GaussianConstraint` attribute), 385
- `name` (`zfit.constraint.SimpleConstraint` attribute), 383
- `name` (`zfit.core.basefunc.BaseFunc` attribute), 50
- `name` (`zfit.core.basemodel.BaseModel` attribute), 55
- `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), 70
- `name` (`zfit.core.constraint.DistributionConstraint` attribute), 71
- `name` (`zfit.core.constraint.GaussianConstraint` attribute), 72
- `name` (`zfit.core.constraint.SimpleConstraint` attribute), 74
- `name` (`zfit.core.data.Data` attribute), 76
- `name` (`zfit.core.data.SampleData` attribute), 80
- `name` (`zfit.core.data.Sampler` attribute), 83
- `name` (`zfit.core.dimension.BaseDimensional` attribute), 85
- `name` (`zfit.core.integration.PartialIntegralSampleData` attribute), 88
- `name` (`zfit.core.interfaces.ZfitData` attribute), 89
- `name` (`zfit.core.interfaces.ZfitDimensional` attribute), 90
- `name` (`zfit.core.interfaces.ZfitFunc` attribute), 90
- `name` (`zfit.core.interfaces.ZfitLoss` attribute), 92
- `name` (`zfit.core.interfaces.ZfitModel` attribute), 93
- `name` (`zfit.core.interfaces.ZfitNumeric` attribute), 94
- `name` (`zfit.core.interfaces.ZfitObject` attribute), 94
- `name` (`zfit.core.interfaces.ZfitParameter` attribute), 97
- `name` (`zfit.core.interfaces.ZfitPDF` attribute), 95
- `name` (`zfit.core.interfaces.ZfitSpace` attribute), 97
- `name` (`zfit.core.limits.Space` attribute), 102
- `name` (`zfit.core.loss.BaseLoss` attribute), 105
- `name` (`zfit.core.loss.CachedLoss` attribute), 106
- `name` (`zfit.core.loss.ExtendedUnbinnedNLL` attribute), 106
- `name` (`zfit.core.loss.SimpleLoss` attribute), 108
- `name` (`zfit.core.loss.UnbinnedNLL` attribute), 108
- `name` (`zfit.core.parameter.BaseComposedParameter` attribute), 111
- `name` (`zfit.core.parameter.BaseParameter` attribute), 112
- `name` (`zfit.core.parameter.BaseZParameter` attribute), 113
- `name` (`zfit.core.parameter.ComplexParameter` attribute), 114
- `name` (`zfit.core.parameter.ComposedParameter` attribute), 115
- `name` (`zfit.core.parameter.ComposedResourceVariable` attribute), 119
- `name` (`zfit.core.parameter.ComposedVariable` attribute), 123
- `name` (`zfit.core.parameter.ConstantParameter` attribute), 124
- `name` (`zfit.core.parameter.Parameter` attribute), 129
- `name` (`zfit.core.parameter.TFBaseVariable` attribute), 138
- `name` (`zfit.core.parameter.ZfitParameterMixin` attribute), 142
- `name` (`zfit.core.sample.EventSpace` attribute), 146
- `name` (`zfit.data.Data` attribute), 387
- `name` (`zfit.func.BaseFunc` attribute), 391
- `name` (`zfit.func.ProdFunc` attribute), 397
- `name` (`zfit.func.SimpleFunc` attribute), 409
- `name` (`zfit.func.SumFunc` attribute), 403
- `name` (`zfit.loss.BaseLoss` attribute), 415
- `name` (`zfit.loss.ExtendedUnbinnedNLL` attribute), 413
- `name` (`zfit.loss.SimpleLoss` attribute), 416
- `name` (`zfit.loss.UnbinnedNLL` attribute), 414
- `name` (`zfit.models.basefunctor.FunctorMixin` attribute), 162
- `name` (`zfit.models.basic.CustomGaussOLD` attribute), 169
- `name` (`zfit.models.basic.Exponential` attribute), 177
- `name` (`zfit.models.dist_tfp.ExponentialTFP` attribute), 186
- `name` (`zfit.models.dist_tfp.Gauss` attribute), 194
- `name` (`zfit.models.dist_tfp.TruncatedGauss` attribute), 202
- `name` (`zfit.models.dist_tfp.Uniform` attribute), 210
- `name` (`zfit.models.dist_tfp.WrapDistribution` attribute), 218
- `name` (`zfit.models.functions.BaseFunctorFunc` attribute), 225
- `name` (`zfit.models.functions.ProdFunc` attribute), 231
- `name` (`zfit.models.functions.SimpleFunc` attribute), 237
- `name` (`zfit.models.functions.SumFunc` attribute), 243
- `name` (`zfit.models.functions.ZFunc` attribute), 249
- `name` (`zfit.models.functor.BaseFunctor` attribute), 256

- name (*zfit.models.functor.ProductPDF* attribute), 264
- name (*zfit.models.functor.SumPDF* attribute), 273
- name (*zfit.models.physics.CrystalBall* attribute), 281
- name (*zfit.models.physics.DoubleCB* attribute), 290
- name (*zfit.models.polynomials.Chebyshev* attribute), 298
- name (*zfit.models.polynomials.Chebyshev2* attribute), 307
- name (*zfit.models.polynomials.Hermite* attribute), 315
- name (*zfit.models.polynomials.Laguerre* attribute), 324
- name (*zfit.models.polynomials.Legendre* attribute), 332
- name (*zfit.models.polynomials.RecursivePolynomial* attribute), 340
- name (*zfit.models.special.SimpleFunctorPDF* attribute), 350
- name (*zfit.models.special.SimplePDF* attribute), 357
- name (*zfit.models.special.ZPDF* attribute), 365
- name (*zfit.param.ConstantParameter* attribute), 421
- name (*zfit.Parameter* attribute), 35
- name (*zfit.pdf.BaseFunctor* attribute), 432
- name (*zfit.pdf.BasePDF* attribute), 425
- name (*zfit.pdf.Chebyshev* attribute), 498
- name (*zfit.pdf.Chebyshev2* attribute), 515
- name (*zfit.pdf.CrystalBall* attribute), 449
- name (*zfit.pdf.DoubleCB* attribute), 458
- name (*zfit.pdf.Exponential* attribute), 440
- name (*zfit.pdf.Gauss* attribute), 466
- name (*zfit.pdf.Hermite* attribute), 524
- name (*zfit.pdf.Laguerre* attribute), 532
- name (*zfit.pdf.Legendre* attribute), 507
- name (*zfit.pdf.ProductPDF* attribute), 548
- name (*zfit.pdf.RecursivePolynomial* attribute), 540
- name (*zfit.pdf.SimpleFunctorPDF* attribute), 580
- name (*zfit.pdf.SimplePDF* attribute), 572
- name (*zfit.pdf.SumPDF* attribute), 556
- name (*zfit.pdf.TruncatedGauss* attribute), 482
- name (*zfit.pdf.Uniform* attribute), 474
- name (*zfit.pdf.WrapDistribution* attribute), 490
- name (*zfit.pdf.ZPDF* attribute), 564
- name (*zfit.Space* attribute), 45
- NameAlreadyTakenError, 375
- nevents (*zfit.core.data.Data* attribute), 76
- nevents (*zfit.core.data.SampleData* attribute), 80
- nevents (*zfit.core.data.Sampler* attribute), 83
- nevents (*zfit.data.Data* attribute), 387
- nll_gaussian() (in module *zfit.constraint*), 382
- no_multiple_limits() (in module *zfit.core.limits*), 103
- no_norm_range() (in module *zfit.core.limits*), 103
- norm_range (*zfit.core.basepdf.BasePDF* attribute), 65
- norm_range (*zfit.models.basic.CustomGaussOLD* attribute), 169
- norm_range (*zfit.models.basic.Exponential* attribute), 177
- norm_range (*zfit.models.dist_tfp.ExponentialTFP* attribute), 186
- norm_range (*zfit.models.dist_tfp.Gauss* attribute), 194
- norm_range (*zfit.models.dist_tfp.TruncatedGauss* attribute), 202
- norm_range (*zfit.models.dist_tfp.Uniform* attribute), 210
- norm_range (*zfit.models.dist_tfp.WrapDistribution* attribute), 218
- norm_range (*zfit.models.functor.BaseFunctor* attribute), 256
- norm_range (*zfit.models.functor.ProductPDF* attribute), 264
- norm_range (*zfit.models.functor.SumPDF* attribute), 273
- norm_range (*zfit.models.physics.CrystalBall* attribute), 281
- norm_range (*zfit.models.physics.DoubleCB* attribute), 290
- norm_range (*zfit.models.polynomials.Chebyshev* attribute), 299
- norm_range (*zfit.models.polynomials.Chebyshev2* attribute), 307
- norm_range (*zfit.models.polynomials.Hermite* attribute), 315
- norm_range (*zfit.models.polynomials.Laguerre* attribute), 324
- norm_range (*zfit.models.polynomials.Legendre* attribute), 332
- norm_range (*zfit.models.polynomials.RecursivePolynomial* attribute), 340
- norm_range (*zfit.models.special.SimpleFunctorPDF* attribute), 350
- norm_range (*zfit.models.special.SimplePDF* attribute), 357
- norm_range (*zfit.models.special.ZPDF* attribute), 365
- norm_range (*zfit.pdf.BaseFunctor* attribute), 433
- norm_range (*zfit.pdf.BasePDF* attribute), 425
- norm_range (*zfit.pdf.Chebyshev* attribute), 499
- norm_range (*zfit.pdf.Chebyshev2* attribute), 515
- norm_range (*zfit.pdf.CrystalBall* attribute), 449
- norm_range (*zfit.pdf.DoubleCB* attribute), 458
- norm_range (*zfit.pdf.Exponential* attribute), 441
- norm_range (*zfit.pdf.Gauss* attribute), 466
- norm_range (*zfit.pdf.Hermite* attribute), 524
- norm_range (*zfit.pdf.Laguerre* attribute), 532
- norm_range (*zfit.pdf.Legendre* attribute), 507
- norm_range (*zfit.pdf.ProductPDF* attribute), 548
- norm_range (*zfit.pdf.RecursivePolynomial* attribute), 540
- norm_range (*zfit.pdf.SimpleFunctorPDF* attribute), 580
- norm_range (*zfit.pdf.SimplePDF* attribute), 572
- norm_range (*zfit.pdf.SumPDF* attribute), 556

[norm_range \(zfit.pdf.TruncatedGauss attribute\), 482](#)
[norm_range \(zfit.pdf.Uniform attribute\), 474](#)
[norm_range \(zfit.pdf.WrapDistribution attribute\), 490](#)
[norm_range \(zfit.pdf.ZPDF attribute\), 564](#)
[normalization\(\) \(zfit.core.basepdf.BasePDF method\), 65](#)
[normalization\(\) \(zfit.core.interfaces.ZfitPDF method\), 95](#)
[normalization\(\) \(zfit.models.basic.CustomGaussOLD method\), 170](#)
[normalization\(\) \(zfit.models.basic.Exponential method\), 178](#)
[normalization\(\) \(zfit.models.dist_tfp.ExponentialTFP method\), 186](#)
[normalization\(\) \(zfit.models.dist_tfp.Gauss method\), 194](#)
[normalization\(\) \(zfit.models.dist_tfp.TruncatedGauss method\), 202](#)
[normalization\(\) \(zfit.models.dist_tfp.Uniform method\), 210](#)
[normalization\(\) \(zfit.models.dist_tfp.WrapDistribution method\), 218](#)
[normalization\(\) \(zfit.models.functor.BaseFunctor method\), 256](#)
[normalization\(\) \(zfit.models.functor.ProductPDF method\), 264](#)
[normalization\(\) \(zfit.models.functor.SumPDF method\), 273](#)
[normalization\(\) \(zfit.models.physics.CrystalBall method\), 281](#)
[normalization\(\) \(zfit.models.physics.DoubleCB method\), 290](#)
[normalization\(\) \(zfit.models.polynomials.Chebyshev method\), 299](#)
[normalization\(\) \(zfit.models.polynomials.Chebyshev2 method\), 307](#)
[normalization\(\) \(zfit.models.polynomials.Hermite method\), 315](#)
[normalization\(\) \(zfit.models.polynomials.Laguerre method\), 324](#)
[normalization\(\) \(zfit.models.polynomials.Legendre method\), 332](#)
[normalization\(\) \(zfit.models.polynomials.RecursivePolynomial method\), 340](#)
[normalization\(\) \(zfit.models.special.SimpleFunctorPDF method\), 350](#)
[normalization\(\) \(zfit.models.special.SimplePDF method\), 358](#)
[normalization\(\) \(zfit.models.special.ZPDF method\), 365](#)
[normalization\(\) \(zfit.pdf.BaseFunctor method\), 433](#)
[normalization\(\) \(zfit.pdf.BasePDF method\), 425](#)
[normalization\(\) \(zfit.pdf.Chebyshev method\), 499](#)
[normalization\(\) \(zfit.pdf.Chebyshev2 method\), 515](#)
[normalization\(\) \(zfit.pdf.CrystalBall method\), 449](#)
[normalization\(\) \(zfit.pdf.DoubleCB method\), 458](#)
[normalization\(\) \(zfit.pdf.Exponential method\), 441](#)
[normalization\(\) \(zfit.pdf.Gauss method\), 466](#)
[normalization\(\) \(zfit.pdf.Hermite method\), 524](#)
[normalization\(\) \(zfit.pdf.Laguerre method\), 532](#)
[normalization\(\) \(zfit.pdf.Legendre method\), 507](#)
[normalization\(\) \(zfit.pdf.ProductPDF method\), 548](#)
[normalization\(\) \(zfit.pdf.RecursivePolynomial method\), 540](#)
[normalization\(\) \(zfit.pdf.SimpleFunctorPDF method\), 580](#)
[normalization\(\) \(zfit.pdf.SimplePDF method\), 572](#)
[normalization\(\) \(zfit.pdf.SumPDF method\), 556](#)
[normalization\(\) \(zfit.pdf.TruncatedGauss method\), 482](#)
[normalization\(\) \(zfit.pdf.Uniform method\), 474](#)
[normalization\(\) \(zfit.pdf.WrapDistribution method\), 490](#)
[normalization\(\) \(zfit.pdf.ZPDF method\), 564](#)
[normalization_chunked\(\) \(in module zfit.core.integration\), 89](#)
[normalization_nograd\(\) \(in module zfit.core.integration\), 89](#)
[NormRangeNotImplementedError, 375](#)
[NormRangeNotSpecifiedError, 375](#)
[NoSessionSpecifiedError, 375](#)
[NotExtendedPDFError, 375](#)
[NotMinimizedError, 375](#)
[NotSpecified \(class in zfit.util.checks\), 371](#)
[NotSpecifiedError, 376](#)
[nth_pow\(\) \(in module zfit.ztf.zextension\), 381](#)
[numeric_integrate\(\) \(in module zfit.core.integration\), 89](#)
[numeric_integrate\(\) \(zfit.core.basefunc.BaseFunc method\), 50](#)
[numeric_integrate\(\) \(zfit.core.basemodel.BaseModel method\), 55](#)
[numeric_integrate\(\) \(zfit.core.basepdf.BasePDF method\), 65](#)
[numeric_integrate\(\) \(zfit.func.BaseFunc method\), 391](#)
[numeric_integrate\(\) \(zfit.func.ProdFunc method\), 397](#)
[numeric_integrate\(\) \(zfit.func.SimpleFunc method\), 409](#)
[numeric_integrate\(\) \(zfit.func.SumFunc method\), 403](#)
[numeric_integrate\(\) \(zfit.models.basefunctor.FunctorMixin method\), 162](#)

<code>numeric_integrate()</code> (<code>zfit.models.basic.CustomGaussOLD</code> method), 170	<code>numeric_integrate()</code> (<code>zfit.models.polynomials.Laguerre</code> method), 324
<code>numeric_integrate()</code> (<code>zfit.models.basic.Exponential</code> method), 178	<code>numeric_integrate()</code> (<code>zfit.models.polynomials.Legendre</code> method), 332
<code>numeric_integrate()</code> (<code>zfit.models.dist_tfp.ExponentialTFP</code> method), 186	<code>numeric_integrate()</code> (<code>zfit.models.polynomials.RecursivePolynomial</code> method), 340
<code>numeric_integrate()</code> (<code>zfit.models.dist_tfp.Gauss</code> method), 194	<code>numeric_integrate()</code> (<code>zfit.models.special.SimpleFunctorPDF</code> method), 350
<code>numeric_integrate()</code> (<code>zfit.models.dist_tfp.TruncatedGauss</code> method), 202	<code>numeric_integrate()</code> (<code>zfit.models.special.SimplePDF</code> method), 358
<code>numeric_integrate()</code> (<code>zfit.models.dist_tfp.Uniform</code> method), 211	<code>numeric_integrate()</code> (<code>zfit.models.special.ZPDF</code> method), 366
<code>numeric_integrate()</code> (<code>zfit.models.dist_tfp.WrapDistribution</code> method), 218	<code>numeric_integrate()</code> (<code>zfit.pdf.BaseFunctor</code> method), 433
<code>numeric_integrate()</code> (<code>zfit.models.functions.BaseFunctorFunc</code> method), 225	<code>numeric_integrate()</code> (<code>zfit.pdf.BasePDF</code> method), 425
<code>numeric_integrate()</code> (<code>zfit.models.functions.ProdFunc</code> method), 231	<code>numeric_integrate()</code> (<code>zfit.pdf.Chebyshev</code> method), 499
<code>numeric_integrate()</code> (<code>zfit.models.functions.SimpleFunc</code> method), 237	<code>numeric_integrate()</code> (<code>zfit.pdf.Chebyshev2</code> method), 516
<code>numeric_integrate()</code> (<code>zfit.models.functions.SumFunc</code> method), 243	<code>numeric_integrate()</code> (<code>zfit.pdf.CrystalBall</code> method), 449
<code>numeric_integrate()</code> (<code>zfit.models.functions.ZFunc</code> method), 249	<code>numeric_integrate()</code> (<code>zfit.pdf.DoubleCB</code> method), 458
<code>numeric_integrate()</code> (<code>zfit.models.functor.BaseFunctor</code> method), 257	<code>numeric_integrate()</code> (<code>zfit.pdf.Exponential</code> method), 441
<code>numeric_integrate()</code> (<code>zfit.models.functor.ProductPDF</code> method), 265	<code>numeric_integrate()</code> (<code>zfit.pdf.Gauss</code> method), 466
<code>numeric_integrate()</code> (<code>zfit.models.functor.SumPDF</code> method), 273	<code>numeric_integrate()</code> (<code>zfit.pdf.Hermite</code> method), 524
<code>numeric_integrate()</code> (<code>zfit.models.physics.CrystalBall</code> method), 281	<code>numeric_integrate()</code> (<code>zfit.pdf.Laguerre</code> method), 532
<code>numeric_integrate()</code> (<code>zfit.models.physics.DoubleCB</code> method), 290	<code>numeric_integrate()</code> (<code>zfit.pdf.Legendre</code> method), 507
<code>numeric_integrate()</code> (<code>zfit.models.polynomials.Chebyshev</code> method), 299	<code>numeric_integrate()</code> (<code>zfit.pdf.ProductPDF</code> method), 548
<code>numeric_integrate()</code> (<code>zfit.models.polynomials.Chebyshev2</code> method), 307	<code>numeric_integrate()</code> (<code>zfit.pdf.RecursivePolynomial</code> method), 540
<code>numeric_integrate()</code> (<code>zfit.models.polynomials.Hermite</code> method),	<code>numeric_integrate()</code> (<code>zfit.pdf.SimpleFunctorPDF</code> method), 580
	<code>numeric_integrate()</code> (<code>zfit.pdf.SimplePDF</code> method), 572
	<code>numeric_integrate()</code> (<code>zfit.pdf.SumPDF</code> method), 557
	<code>numeric_integrate()</code> (<code>zfit.pdf.TruncatedGauss</code> method), 482
	<code>numeric_integrate()</code> (<code>zfit.pdf.Uniform</code> method), 474
	<code>numeric_integrate()</code> (<code>zfit.pdf.WrapDistribution</code>

method), 490
 numeric_integrate() (zfit.pdf.ZPDF method), 564
 numpy() (zfit.core.parameter.ComposedResourceVariable method), 120
 numpy() (zfit.core.parameter.Parameter method), 129
 numpy() (zfit.core.parameter.TFBaseVariable method), 138
 numpy() (zfit.Parameter method), 36

O

obs (zfit.core.basefunc.BaseFunc attribute), 50
 obs (zfit.core.basemodel.BaseModel attribute), 56
 obs (zfit.core.basepdf.BasePDF attribute), 65
 obs (zfit.core.data.Data attribute), 76
 obs (zfit.core.data.SampleData attribute), 80
 obs (zfit.core.data.Sampler attribute), 83
 obs (zfit.core.dimension.BaseDimensional attribute), 85
 obs (zfit.core.integration.PartialIntegralSampleData attribute), 88
 obs (zfit.core.interfaces.ZfitData attribute), 89
 obs (zfit.core.interfaces.ZfitDimensional attribute), 90
 obs (zfit.core.interfaces.ZfitFunc attribute), 91
 obs (zfit.core.interfaces.ZfitModel attribute), 93
 obs (zfit.core.interfaces.ZfitPDF attribute), 95
 obs (zfit.core.interfaces.ZfitSpace attribute), 97
 obs (zfit.core.limits.Space attribute), 102
 obs (zfit.core.sample.EventSpace attribute), 146
 obs (zfit.data.Data attribute), 387
 obs (zfit.func.BaseFunc attribute), 391
 obs (zfit.func.ProdFunc attribute), 397
 obs (zfit.func.SimpleFunc attribute), 409
 obs (zfit.func.SumFunc attribute), 403
 obs (zfit.models.basefunc.FunctorMixin attribute), 163
 obs (zfit.models.basic.CustomGaussOLD attribute), 170
 obs (zfit.models.basic.Exponential attribute), 178
 obs (zfit.models.dist_tfp.ExponentialTFP attribute), 186
 obs (zfit.models.dist_tfp.Gauss attribute), 194
 obs (zfit.models.dist_tfp.TruncatedGauss attribute), 203
 obs (zfit.models.dist_tfp.Uniform attribute), 211
 obs (zfit.models.dist_tfp.WrapDistribution attribute), 219
 obs (zfit.models.functions.BaseFunctorFunc attribute), 226
 obs (zfit.models.functions.ProdFunc attribute), 232
 obs (zfit.models.functions.SimpleFunc attribute), 238
 obs (zfit.models.functions.SumFunc attribute), 244
 obs (zfit.models.functions.ZFunc attribute), 249
 obs (zfit.models.functor.BaseFunctor attribute), 257
 obs (zfit.models.functor.ProductPDF attribute), 265
 obs (zfit.models.functor.SumPDF attribute), 273
 obs (zfit.models.physics.CrystalBall attribute), 282
 obs (zfit.models.physics.DoubleCB attribute), 291
 obs (zfit.models.polynomials.Chebyshev attribute), 299
 obs (zfit.models.polynomials.Chebyshev2 attribute), 307
 obs (zfit.models.polynomials.Hermite attribute), 316
 obs (zfit.models.polynomials.Laguerre attribute), 324
 obs (zfit.models.polynomials.Legendre attribute), 333
 obs (zfit.models.polynomials.RecursivePolynomial attribute), 341
 obs (zfit.models.special.SimpleFunctorPDF attribute), 350
 obs (zfit.models.special.SimplePDF attribute), 358
 obs (zfit.models.special.ZPDF attribute), 366
 obs (zfit.pdf.BaseFunctor attribute), 433
 obs (zfit.pdf.BasePDF attribute), 425
 obs (zfit.pdf.Chebyshev attribute), 499
 obs (zfit.pdf.Chebyshev2 attribute), 516
 obs (zfit.pdf.CrystalBall attribute), 450
 obs (zfit.pdf.DoubleCB attribute), 458
 obs (zfit.pdf.Exponential attribute), 441
 obs (zfit.pdf.Gauss attribute), 466
 obs (zfit.pdf.Hermite attribute), 524
 obs (zfit.pdf.Laguerre attribute), 533
 obs (zfit.pdf.Legendre attribute), 507
 obs (zfit.pdf.ProductPDF attribute), 548
 obs (zfit.pdf.RecursivePolynomial attribute), 541
 obs (zfit.pdf.SimpleFunctorPDF attribute), 580
 obs (zfit.pdf.SimplePDF attribute), 573
 obs (zfit.pdf.SumPDF attribute), 557
 obs (zfit.pdf.TruncatedGauss attribute), 483
 obs (zfit.pdf.Uniform attribute), 475
 obs (zfit.pdf.WrapDistribution attribute), 491
 obs (zfit.pdf.ZPDF attribute), 565
 obs (zfit.Space attribute), 45
 obs_axes (zfit.core.limits.Space attribute), 102
 obs_axes (zfit.core.sample.EventSpace attribute), 146
 obs_axes (zfit.Space attribute), 45
 ObsIncompatibleError, 376
 ObsNotSpecifiedError, 376
 op (zfit.core.parameter.ComposedResourceVariable attribute), 120
 op (zfit.core.parameter.Parameter attribute), 129
 op (zfit.core.parameter.TFBaseVariable attribute), 138
 op (zfit.Parameter attribute), 36
 OverdefinedError, 376

P

Parameter (class in zfit), 31
 Parameter (class in zfit.core.parameter), 124
 Parameter.SaveSliceInfo (class in zfit), 31
 Parameter.SaveSliceInfo (class in zfit.core.parameter), 124
 params (zfit.ComplexParameter attribute), 41
 params (zfit.ComposedParameter attribute), 40
 params (zfit.constraint.GaussianConstraint attribute), 385
 params (zfit.constraint.SimpleConstraint attribute), 383

- params (*zfit.core.basefunc.BaseFunc* attribute), 50
- 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), 70
- params (*zfit.core.constraint.DistributionConstraint* attribute), 71
- params (*zfit.core.constraint.GaussianConstraint* attribute), 72
- params (*zfit.core.constraint.SimpleConstraint* attribute), 74
- params (*zfit.core.interfaces.ZfitFunc* attribute), 91
- params (*zfit.core.interfaces.ZfitModel* attribute), 93
- params (*zfit.core.interfaces.ZfitNumeric* attribute), 94
- params (*zfit.core.interfaces.ZfitParameter* attribute), 97
- params (*zfit.core.interfaces.ZfitPDF* attribute), 95
- params (*zfit.core.parameter.BaseComposedParameter* attribute), 111
- params (*zfit.core.parameter.BaseParameter* attribute), 112
- params (*zfit.core.parameter.BaseZParameter* attribute), 113
- params (*zfit.core.parameter.ComplexParameter* attribute), 114
- params (*zfit.core.parameter.ComposedParameter* attribute), 115
- params (*zfit.core.parameter.ConstantParameter* attribute), 124
- params (*zfit.core.parameter.Parameter* attribute), 129
- params (*zfit.core.parameter.ZfitParameterMixin* attribute), 142
- params (*zfit.func.BaseFunc* attribute), 391
- params (*zfit.func.ProdFunc* attribute), 398
- params (*zfit.func.SimpleFunc* attribute), 410
- params (*zfit.func.SumFunc* attribute), 404
- params (*zfit.minimizers.fitresult.FitResult* attribute), 152
- params (*zfit.minimizers.interface.ZfitResult* attribute), 153
- params (*zfit.models.basefunc.FunctorMixin* attribute), 163
- params (*zfit.models.basic.CustomGaussOLD* attribute), 170
- params (*zfit.models.basic.Exponential* attribute), 178
- params (*zfit.models.dist_tfp.ExponentialTFP* attribute), 186
- params (*zfit.models.dist_tfp.Gauss* attribute), 194
- params (*zfit.models.dist_tfp.TruncatedGauss* attribute), 203
- params (*zfit.models.dist_tfp.Uniform* attribute), 211
- params (*zfit.models.dist_tfp.WrapDistribution* attribute), 219
- params (*zfit.models.functions.BaseFunctorFunc* attribute), 226
- params (*zfit.models.functions.ProdFunc* attribute), 232
- params (*zfit.models.functions.SimpleFunc* attribute), 238
- params (*zfit.models.functions.SumFunc* attribute), 244
- params (*zfit.models.functions.ZFunc* attribute), 250
- params (*zfit.models.functor.BaseFunctor* attribute), 257
- params (*zfit.models.functor.ProductPDF* attribute), 265
- params (*zfit.models.functor.SumPDF* attribute), 273
- params (*zfit.models.physics.CrystalBall* attribute), 282
- params (*zfit.models.physics.DoubleCB* attribute), 291
- params (*zfit.models.polynomials.Chebyshev* attribute), 299
- params (*zfit.models.polynomials.Chebyshev2* attribute), 308
- params (*zfit.models.polynomials.Hermite* attribute), 316
- params (*zfit.models.polynomials.Laguerre* attribute), 324
- params (*zfit.models.polynomials.Legendre* attribute), 333
- params (*zfit.models.polynomials.RecursivePolynomial* attribute), 341
- params (*zfit.models.special.SimpleFunctorPDF* attribute), 350
- params (*zfit.models.special.SimplePDF* attribute), 358
- params (*zfit.models.special.ZPDF* attribute), 366
- params (*zfit.param.ConstantParameter* attribute), 421
- params (*zfit.Parameter* attribute), 36
- params (*zfit.pdf.BaseFunctor* attribute), 433
- params (*zfit.pdf.BasePDF* attribute), 425
- params (*zfit.pdf.Chebyshev* attribute), 499
- params (*zfit.pdf.Chebyshev2* attribute), 516
- params (*zfit.pdf.CrystalBall* attribute), 450
- params (*zfit.pdf.DoubleCB* attribute), 458
- params (*zfit.pdf.Exponential* attribute), 441
- params (*zfit.pdf.Gauss* attribute), 466
- params (*zfit.pdf.Hermite* attribute), 524
- params (*zfit.pdf.Laguerre* attribute), 533
- params (*zfit.pdf.Legendre* attribute), 508
- params (*zfit.pdf.ProductPDF* attribute), 548
- params (*zfit.pdf.RecursivePolynomial* attribute), 541
- params (*zfit.pdf.SimpleFunctorPDF* attribute), 580
- params (*zfit.pdf.SimplePDF* attribute), 573
- params (*zfit.pdf.SumPDF* attribute), 557
- params (*zfit.pdf.TruncatedGauss* attribute), 483
- params (*zfit.pdf.Uniform* attribute), 475
- params (*zfit.pdf.WrapDistribution* attribute), 491
- params (*zfit.pdf.ZPDF* attribute), 565
- partial_analytic_integrate()
 - (*zfit.core.basefunc.BaseFunc* method), 50
- 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), 391
`partial_analytic_integrate()`
 (`zfit.func.ProdFunc` method), 398
`partial_analytic_integrate()`
 (`zfit.func.SimpleFunc` method), 410
`partial_analytic_integrate()`
 (`zfit.func.SumFunc` method), 404
`partial_analytic_integrate()`
 (`zfit.models.basefunctor.FunctorMixin`
 method), 163
`partial_analytic_integrate()`
 (`zfit.models.basic.CustomGaussOLD` method),
 170
`partial_analytic_integrate()`
 (`zfit.models.basic.Exponential` method), 178
`partial_analytic_integrate()`
 (`zfit.models.dist_tfp.ExponentialTFP` method),
 186
`partial_analytic_integrate()`
 (`zfit.models.dist_tfp.Gauss` method), 194
`partial_analytic_integrate()`
 (`zfit.models.dist_tfp.TruncatedGauss` method),
 203
`partial_analytic_integrate()`
 (`zfit.models.dist_tfp.Uniform` method), 211
`partial_analytic_integrate()`
 (`zfit.models.dist_tfp.WrapDistribution` method),
 219
`partial_analytic_integrate()`
 (`zfit.models.functions.BaseFunctorFunc`
 method), 226
`partial_analytic_integrate()`
 (`zfit.models.functions.ProdFunc` method),
 232
`partial_analytic_integrate()`
 (`zfit.models.functions.SimpleFunc` method),
 238
`partial_analytic_integrate()`
 (`zfit.models.functions.SumFunc` method),
 244
`partial_analytic_integrate()`
 (`zfit.models.functions.ZFunc` method), 250
`partial_analytic_integrate()`
 (`zfit.models.functor.BaseFunctor` method),
 257
`partial_analytic_integrate()`
 (`zfit.models.functor.ProductPDF` method),
 265
`partial_analytic_integrate()`
 (`zfit.models.functor.SumPDF` method), 273
`partial_analytic_integrate()`
 (`zfit.models.physics.CrystalBall` method),
 282
`partial_analytic_integrate()`
 (`zfit.models.physics.DoubleCB` method),
 291
`partial_analytic_integrate()`
 (`zfit.models.polynomials.Chebyshev` method),
 299
`partial_analytic_integrate()`
 (`zfit.models.polynomials.Chebyshev2` method),
 308
`partial_analytic_integrate()`
 (`zfit.models.polynomials.Hermite` method),
 316
`partial_analytic_integrate()`
 (`zfit.models.polynomials.Laguerre` method),
 324
`partial_analytic_integrate()`
 (`zfit.models.polynomials.Legendre` method),
 333
`partial_analytic_integrate()`
 (`zfit.models.polynomials.RecursivePolynomial`
 method), 341
`partial_analytic_integrate()`
 (`zfit.models.special.SimpleFunctorPDF`
 method), 350
`partial_analytic_integrate()`
 (`zfit.models.special.SimplePDF` method),
 358
`partial_analytic_integrate()`
 (`zfit.models.special.ZPDF` method), 366
`partial_analytic_integrate()`
 (`zfit.pdf.BaseFunctor` method), 433
`partial_analytic_integrate()`
 (`zfit.pdf.BasePDF` method), 425
`partial_analytic_integrate()`
 (`zfit.pdf.Chebyshev` method), 499
`partial_analytic_integrate()`
 (`zfit.pdf.Chebyshev2` method), 516
`partial_analytic_integrate()`
 (`zfit.pdf.CrystalBall` method), 450
`partial_analytic_integrate()`
 (`zfit.pdf.DoubleCB` method), 458
`partial_analytic_integrate()`
 (`zfit.pdf.Exponential` method), 441
`partial_analytic_integrate()` (`zfit.pdf.Gauss`
 method), 466
`partial_analytic_integrate()`
 (`zfit.pdf.Hermite` method), 524
`partial_analytic_integrate()`
 (`zfit.pdf.Laguerre` method), 533
`partial_analytic_integrate()`
 (`zfit.pdf.Legendre` method), 508
`partial_analytic_integrate()`
 (`zfit.pdf.ProductPDF` method), 549
`partial_analytic_integrate()`

[\(zfit.pdf.RecursivePolynomial method\), 541](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.SimpleFuncPDF method\), 580](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.SimplePDF method\), 573](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.SumPDF method\), 557](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.TruncatedGauss method\), 483](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.Uniform method\), 475](#)
[partial_analytic_integrate\(\)](#)
[\(zfit.pdf.WrapDistribution method\), 491](#)
[partial_analytic_integrate\(\)](#) [\(zfit.pdf.ZPDF method\), 565](#)
[partial_integrate\(\)](#)
[\(zfit.core.basefunc.BaseFunc method\), 50](#)
[partial_integrate\(\)](#)
[\(zfit.core.basemodel.BaseModel method\), 56](#)
[partial_integrate\(\)](#) [\(zfit.core.basepdf.BasePDF method\), 66](#)
[partial_integrate\(\)](#) [\(zfit.core.interfaces.ZfitFunc method\), 91](#)
[partial_integrate\(\)](#)
[\(zfit.core.interfaces.ZfitModel method\), 93](#)
[partial_integrate\(\)](#) [\(zfit.core.interfaces.ZfitPDF method\), 95](#)
[partial_integrate\(\)](#) [\(zfit.func.BaseFunc method\), 392](#)
[partial_integrate\(\)](#) [\(zfit.func.ProdFunc method\), 398](#)
[partial_integrate\(\)](#) [\(zfit.func.SimpleFunc method\), 410](#)
[partial_integrate\(\)](#) [\(zfit.func.SumFunc method\), 404](#)
[partial_integrate\(\)](#)
[\(zfit.models.basefunc.FunctorMixin method\), 163](#)
[partial_integrate\(\)](#)
[\(zfit.models.basic.CustomGaussOLD method\), 171](#)
[partial_integrate\(\)](#)
[\(zfit.models.basic.Exponential method\), 179](#)
[partial_integrate\(\)](#)
[\(zfit.models.dist_tfp.ExponentialTFP method\), 187](#)
[partial_integrate\(\)](#) [\(zfit.models.dist_tfp.Gauss method\), 195](#)
[partial_integrate\(\)](#)
[\(zfit.models.dist_tfp.TruncatedGauss method\), 203](#)
[partial_integrate\(\)](#)
[\(zfit.models.dist_tfp.Uniform method\), 211](#)
[partial_integrate\(\)](#)
[\(zfit.models.dist_tfp.WrapDistribution method\), 219](#)
[partial_integrate\(\)](#)
[\(zfit.models.functions.BaseFuncPDF method\), 226](#)
[partial_integrate\(\)](#)
[\(zfit.models.functions.ProdFunc method\), 232](#)
[partial_integrate\(\)](#)
[\(zfit.models.functions.SimpleFunc method\), 238](#)
[partial_integrate\(\)](#)
[\(zfit.models.functions.SumFunc method\), 244](#)
[partial_integrate\(\)](#) [\(zfit.models.functions.ZFunc method\), 250](#)
[partial_integrate\(\)](#)
[\(zfit.models.functor.BaseFunc method\), 257](#)
[partial_integrate\(\)](#)
[\(zfit.models.functor.ProductPDF method\), 265](#)
[partial_integrate\(\)](#)
[\(zfit.models.functor.SumPDF method\), 274](#)
[partial_integrate\(\)](#)
[\(zfit.models.physics.CrystalBall method\), 282](#)
[partial_integrate\(\)](#)
[\(zfit.models.physics.DoubleCB method\), 291](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.Chebyshev method\), 300](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.Chebyshev2 method\), 308](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.Hermite method\), 316](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.Laguerre method\), 325](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.Legendre method\), 333](#)
[partial_integrate\(\)](#)
[\(zfit.models.polynomials.RecursivePolynomial method\), 341](#)
[partial_integrate\(\)](#)
[\(zfit.models.special.SimpleFuncPDF method\), 351](#)
[partial_integrate\(\)](#)
[\(zfit.models.special.SimplePDF method\),](#)

359

`partial_integrate()` (`zfit.models.special.ZPDF method`), 366

`partial_integrate()` (`zfit.pdf.BaseFunc`
`method`), 434

`partial_integrate()` (`zfit.pdf.BasePDF method`), 426

`partial_integrate()` (`zfit.pdf.Chebyshev method`), 500

`partial_integrate()` (`zfit.pdf.Chebyshev2 method`), 516

`partial_integrate()` (`zfit.pdf.CrystalBall method`), 450

`partial_integrate()` (`zfit.pdf.DoubleCB method`), 459

`partial_integrate()` (`zfit.pdf.Exponential method`), 442

`partial_integrate()` (`zfit.pdf.Gauss method`), 467

`partial_integrate()` (`zfit.pdf.Hermite method`), 525

`partial_integrate()` (`zfit.pdf.Laguerre method`), 533

`partial_integrate()` (`zfit.pdf.Legendre method`), 508

`partial_integrate()` (`zfit.pdf.ProductPDF method`), 549

`partial_integrate()` (`zfit.pdf.RecursivePolynomial method`), 541

`partial_integrate()` (`zfit.pdf.SimpleFuncPDF method`), 581

`partial_integrate()` (`zfit.pdf.SimplePDF method`), 573

`partial_integrate()` (`zfit.pdf.SumPDF method`), 557

`partial_integrate()` (`zfit.pdf.TruncatedGauss method`), 483

`partial_integrate()` (`zfit.pdf.Uniform method`), 475

`partial_integrate()` (`zfit.pdf.WrapDistribution method`), 491

`partial_integrate()` (`zfit.pdf.ZPDF method`), 565

`partial_numeric_integrate()` (`zfit.core.basefunc.BaseFunc method`), 51

`partial_numeric_integrate()` (`zfit.core.basemodel.BaseModel method`), 57

`partial_numeric_integrate()` (`zfit.core.basepdf.BasePDF method`), 66

`partial_numeric_integrate()` (`zfit.func.BaseFunc method`), 392

`partial_numeric_integrate()` (`zfit.func.ProdFunc method`), 399

`partial_numeric_integrate()` (`zfit.func.SimpleFunc method`), 411

`partial_numeric_integrate()` (`zfit.func.SumFunc method`), 405

`partial_numeric_integrate()` (`zfit.models.basefunc.Func`
`method`), 164

`partial_numeric_integrate()` (`zfit.models.basic.CustomGaussOLD method`), 171

`partial_numeric_integrate()` (`zfit.models.basic.Exponential method`), 179

`partial_numeric_integrate()` (`zfit.models.dist_tfp.ExponentialTFP method`), 187

`partial_numeric_integrate()` (`zfit.models.dist_tfp.Gauss method`), 195

`partial_numeric_integrate()` (`zfit.models.dist_tfp.TruncatedGauss method`), 204

`partial_numeric_integrate()` (`zfit.models.dist_tfp.Uniform method`), 212

`partial_numeric_integrate()` (`zfit.models.dist_tfp.WrapDistribution method`), 220

`partial_numeric_integrate()` (`zfit.models.functions.BaseFunc`
`method`), 227

`partial_numeric_integrate()` (`zfit.models.functions.ProdFunc method`), 233

`partial_numeric_integrate()` (`zfit.models.functions.SimpleFunc method`), 239

`partial_numeric_integrate()` (`zfit.models.functions.SumFunc method`), 245

`partial_numeric_integrate()` (`zfit.models.functions.ZFunc method`), 251

`partial_numeric_integrate()` (`zfit.models.func`
`method`), 258

`partial_numeric_integrate()` (`zfit.models.func`
`method`), 266

`partial_numeric_integrate()` (`zfit.models.func`
`method`), 274

`partial_numeric_integrate()` (`zfit.models.physics.CrystalBall method`), 283

`partial_numeric_integrate()` (`zfit.models.physics.DoubleCB method`), 292

`partial_numeric_integrate()` (`zfit.models.polynomials.Chebyshev method`), 300

`partial_numeric_integrate()`
 (`zfit.models.polynomials.Chebyshev2` method), 309
`partial_numeric_integrate()`
 (`zfit.models.polynomials.Hermite` method), 317
`partial_numeric_integrate()`
 (`zfit.models.polynomials.Laguerre` method), 325
`partial_numeric_integrate()`
 (`zfit.models.polynomials.Legendre` method), 334
`partial_numeric_integrate()`
 (`zfit.models.polynomials.RecursivePolynomial` method), 342
`partial_numeric_integrate()`
 (`zfit.models.special.SimpleFunctorPDF` method), 351
`partial_numeric_integrate()`
 (`zfit.models.special.SimplePDF` method), 359
`partial_numeric_integrate()`
 (`zfit.models.special.ZPDF` method), 367
`partial_numeric_integrate()`
 (`zfit.pdf.BaseFunctor` method), 434
`partial_numeric_integrate()`
 (`zfit.pdf.BasePDF` method), 426
`partial_numeric_integrate()`
 (`zfit.pdf.Chebyshev` method), 500
`partial_numeric_integrate()`
 (`zfit.pdf.Chebyshev2` method), 517
`partial_numeric_integrate()`
 (`zfit.pdf.CrystalBall` method), 451
`partial_numeric_integrate()`
 (`zfit.pdf.DoubleCB` method), 459
`partial_numeric_integrate()`
 (`zfit.pdf.Exponential` method), 442
`partial_numeric_integrate()` (`zfit.pdf.Gauss` method), 467
`partial_numeric_integrate()` (`zfit.pdf.Hermite` method), 525
`partial_numeric_integrate()`
 (`zfit.pdf.Laguerre` method), 534
`partial_numeric_integrate()`
 (`zfit.pdf.Legendre` method), 509
`partial_numeric_integrate()`
 (`zfit.pdf.ProductPDF` method), 549
`partial_numeric_integrate()`
 (`zfit.pdf.RecursivePolynomial` method), 542
`partial_numeric_integrate()`
 (`zfit.pdf.SimpleFunctorPDF` method), 581
`partial_numeric_integrate()`
 (`zfit.pdf.SimplePDF` method), 574
`partial_numeric_integrate()`
 (`zfit.pdf.SumPDF` method), 558
`partial_numeric_integrate()`
 (`zfit.pdf.TruncatedGauss` method), 484
`partial_numeric_integrate()`
 (`zfit.pdf.Uniform` method), 476
`partial_numeric_integrate()`
 (`zfit.pdf.WrapDistribution` method), 492
`partial_numeric_integrate()` (`zfit.pdf.ZPDF` method), 566
`PartialIntegralSampleData` (class in `zfit.core.integration`), 88
`pdf()` (`zfit.core.basepdf.BasePDF` method), 66
`pdf()` (`zfit.core.interfaces.ZfitPDF` method), 96
`pdf()` (`zfit.models.basic.CustomGaussOLD` method), 171
`pdf()` (`zfit.models.basic.Exponential` method), 179
`pdf()` (`zfit.models.dist_tfp.ExponentialTFP` method), 188
`pdf()` (`zfit.models.dist_tfp.Gauss` method), 196
`pdf()` (`zfit.models.dist_tfp.TruncatedGauss` method), 204
`pdf()` (`zfit.models.dist_tfp.Uniform` method), 212
`pdf()` (`zfit.models.dist_tfp.WrapDistribution` method), 220
`pdf()` (`zfit.models.functor.BaseFunctor` method), 258
`pdf()` (`zfit.models.functor.ProductPDF` method), 266
`pdf()` (`zfit.models.functor.SumPDF` method), 275
`pdf()` (`zfit.models.physics.CrystalBall` method), 283
`pdf()` (`zfit.models.physics.DoubleCB` method), 292
`pdf()` (`zfit.models.polynomials.Chebyshev` method), 300
`pdf()` (`zfit.models.polynomials.Chebyshev2` method), 309
`pdf()` (`zfit.models.polynomials.Hermite` method), 317
`pdf()` (`zfit.models.polynomials.Laguerre` method), 326
`pdf()` (`zfit.models.polynomials.Legendre` method), 334
`pdf()` (`zfit.models.polynomials.RecursivePolynomial` method), 342
`pdf()` (`zfit.models.special.SimpleFunctorPDF` method), 352
`pdf()` (`zfit.models.special.SimplePDF` method), 359
`pdf()` (`zfit.models.special.ZPDF` method), 367
`pdf()` (`zfit.pdf.BaseFunctor` method), 434
`pdf()` (`zfit.pdf.BasePDF` method), 427
`pdf()` (`zfit.pdf.Chebyshev` method), 500
`pdf()` (`zfit.pdf.Chebyshev2` method), 517
`pdf()` (`zfit.pdf.CrystalBall` method), 451
`pdf()` (`zfit.pdf.DoubleCB` method), 460
`pdf()` (`zfit.pdf.Exponential` method), 442
`pdf()` (`zfit.pdf.Gauss` method), 468
`pdf()` (`zfit.pdf.Hermite` method), 526
`pdf()` (`zfit.pdf.Laguerre` method), 534
`pdf()` (`zfit.pdf.Legendre` method), 509
`pdf()` (`zfit.pdf.ProductPDF` method), 550
`pdf()` (`zfit.pdf.RecursivePolynomial` method), 542

- pdf () (*zfit.pdf.SimpleFuncPDF* method), 582
pdf () (*zfit.pdf.SimplePDF* method), 574
pdf () (*zfit.pdf.SumPDF* method), 558
pdf () (*zfit.pdf.TruncatedGauss* method), 484
pdf () (*zfit.pdf.Uniform* method), 476
pdf () (*zfit.pdf.WrapDistribution* method), 492
pdf () (*zfit.pdf.ZPDF* method), 566
PDFCompatibilityError, 376
pdfs_extended (*zfit.models.functor.BaseFunc* attribute), 258
pdfs_extended (*zfit.models.functor.ProductPDF* attribute), 266
pdfs_extended (*zfit.models.functor.SumPDF* attribute), 275
pdfs_extended (*zfit.models.special.SimpleFuncPDF* attribute), 352
pdfs_extended (*zfit.pdf.BaseFunc* attribute), 435
pdfs_extended (*zfit.pdf.ProductPDF* attribute), 550
pdfs_extended (*zfit.pdf.SimpleFuncPDF* attribute), 582
pdfs_extended (*zfit.pdf.SumPDF* attribute), 558
poisson () (in module *zfit.sample*), 584
poly_complex () (in module *zfit.core.math*), 109
pop () (*zfit.util.container.DotDict* method), 372
popitem () (*zfit.util.container.DotDict* method), 372
pow () (in module *zfit.ztf.wrapping_tf*), 381
ProdFunc (class in *zfit.func*), 395
ProdFunc (class in *zfit.models.functions*), 229
ProductPDF (class in *zfit.models.functor*), 261
ProductPDF (class in *zfit.pdf*), 544
- ## R
- raise_error_if_norm_range () (in module *zfit.models.special*), 369
random_normal () (in module *zfit.ztf.wrapping_tf*), 381
random_poisson () (in module *zfit.ztf.wrapping_tf*), 381
random_uniform () (in module *zfit.ztf.wrapping_tf*), 381
randomize () (*zfit.core.parameter.Parameter* method), 129
randomize () (*zfit.Parameter* method), 36
read_value () (*zfit.ComplexParameter* method), 41
read_value () (*zfit.ComposedParameter* method), 40
read_value () (*zfit.core.parameter.BaseComposedParameter* method), 111
read_value () (*zfit.core.parameter.BaseZParameter* method), 113
read_value () (*zfit.core.parameter.ComplexParameter* method), 114
read_value () (*zfit.core.parameter.ComposedParameter* method), 115
read_value () (*zfit.core.parameter.ComposedResourceVariable* method), 120
read_value () (*zfit.core.parameter.ComposedVariable* method), 123
read_value () (*zfit.core.parameter.ConstantParameter* method), 124
read_value () (*zfit.core.parameter.Parameter* method), 129
read_value () (*zfit.core.parameter.TFBaseVariable* method), 138
read_value () (*zfit.param.ConstantParameter* method), 421
read_value () (*zfit.Parameter* method), 36
real (*zfit.ComplexParameter* attribute), 42
real (*zfit.core.parameter.ComplexParameter* attribute), 114
RecursivePolynomial (class in *zfit.models.polynomials*), 336
RecursivePolynomial (class in *zfit.pdf*), 536
register () (*zfit.core.integration.AnalyticIntegral* method), 87
register_additional_repr () (*zfit.core.basefunc.BaseFunc* class method), 51
register_additional_repr () (*zfit.core.basemodel.BaseModel* class method), 57
register_additional_repr () (*zfit.core.basepdf.BasePDF* class method), 67
register_additional_repr () (*zfit.func.BaseFunc* class method), 393
register_additional_repr () (*zfit.func.ProdFunc* class method), 399
register_additional_repr () (*zfit.func.SimpleFunc* class method), 411
register_additional_repr () (*zfit.func.SumFunc* class method), 405
register_additional_repr () (*zfit.models.basefunctor.FunctorMixin* class method), 164
register_additional_repr () (*zfit.models.basic.CustomGaussOLD* class method), 172
register_additional_repr () (*zfit.models.basic.Exponential* class method), 180
register_additional_repr () (*zfit.models.dist_tfp.ExponentialTFP* class method), 188
register_additional_repr () (*zfit.models.dist_tfp.Gauss* class method), 196
register_additional_repr () (*zfit.models.dist_tfp.TruncatedGauss* class method), 196

[method](#)), 204
[register_additional_repr\(\)](#)
 ([zfit.models.dist_tfp.Uniform](#) class method),
 212
[register_additional_repr\(\)](#)
 ([zfit.models.dist_tfp.WrapDistribution](#) class
 method), 220
[register_additional_repr\(\)](#)
 ([zfit.models.functions.BaseFunc](#) class
 method), 227
[register_additional_repr\(\)](#)
 ([zfit.models.functions.ProdFunc](#) class method),
 233
[register_additional_repr\(\)](#)
 ([zfit.models.functions.SimpleFunc](#) class
 method), 239
[register_additional_repr\(\)](#)
 ([zfit.models.functions.SumFunc](#) class method),
 245
[register_additional_repr\(\)](#)
 ([zfit.models.functions.ZFunc](#) class method),
 251
[register_additional_repr\(\)](#)
 ([zfit.models.funcutor.BaseFunc](#) class method),
 259
[register_additional_repr\(\)](#)
 ([zfit.models.funcutor.ProductPDF](#) class method),
 267
[register_additional_repr\(\)](#)
 ([zfit.models.funcutor.SumPDF](#) class method),
 275
[register_additional_repr\(\)](#)
 ([zfit.models.physics.CrystalBall](#) class method),
 283
[register_additional_repr\(\)](#)
 ([zfit.models.physics.DoubleCB](#) class method),
 292
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.Chebyshev](#) class
 method), 301
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.Chebyshev2](#) class
 method), 309
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.Hermite](#) class
 method), 317
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.Laguerre](#) class
 method), 326
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.Legendre](#) class
 method), 334
[register_additional_repr\(\)](#)
 ([zfit.models.polynomials.RecursivePolynomial](#)
 class method), 342
[register_additional_repr\(\)](#)
 ([zfit.models.special.SimpleFunc](#) class
 method), 352
[register_additional_repr\(\)](#)
 ([zfit.models.special.SimplePDF](#) class method),
 360
[register_additional_repr\(\)](#)
 ([zfit.models.special.ZPDF](#) class method),
 367
[register_additional_repr\(\)](#)
 ([zfit.pdf.BaseFunc](#) class method), 435
[register_additional_repr\(\)](#) ([zfit.pdf.BasePDF](#)
 class method), 427
[register_additional_repr\(\)](#)
 ([zfit.pdf.Chebyshev](#) class method), 501
[register_additional_repr\(\)](#)
 ([zfit.pdf.Chebyshev2](#) class method), 517
[register_additional_repr\(\)](#)
 ([zfit.pdf.CrystalBall](#) class method), 451
[register_additional_repr\(\)](#)
 ([zfit.pdf.DoubleCB](#) class method), 460
[register_additional_repr\(\)](#)
 ([zfit.pdf.Exponential](#) class method), 443
[register_additional_repr\(\)](#) ([zfit.pdf.Gauss](#)
 class method), 468
[register_additional_repr\(\)](#) ([zfit.pdf.Hermite](#)
 class method), 526
[register_additional_repr\(\)](#) ([zfit.pdf.Laguerre](#)
 class method), 534
[register_additional_repr\(\)](#) ([zfit.pdf.Legendre](#)
 class method), 509
[register_additional_repr\(\)](#)
 ([zfit.pdf.ProductPDF](#) class method), 550
[register_additional_repr\(\)](#)
 ([zfit.pdf.RecursivePolynomial](#) class method),
 542
[register_additional_repr\(\)](#)
 ([zfit.pdf.SimpleFunc](#) class method),
 582
[register_additional_repr\(\)](#)
 ([zfit.pdf.SimplePDF](#) class method), 574
[register_additional_repr\(\)](#) ([zfit.pdf.SumPDF](#)
 class method), 558
[register_additional_repr\(\)](#)
 ([zfit.pdf.TruncatedGauss](#) class method),
 484
[register_additional_repr\(\)](#) ([zfit.pdf.Uniform](#)
 class method), 476
[register_additional_repr\(\)](#)
 ([zfit.pdf.WrapDistribution](#) class method),
 492
[register_additional_repr\(\)](#) ([zfit.pdf.ZPDF](#)
 class method), 566

<code>register_analytic_integral()</code> (<code>zfit.core.basefunc.BaseFunc</code> class method), 51	<code>register_analytic_integral()</code> (<code>zfit.models.functions.SimpleFunc</code> class method), 239
<code>register_analytic_integral()</code> (<code>zfit.core.basemodel.BaseModel</code> class method), 57	<code>register_analytic_integral()</code> (<code>zfit.models.functions.SumFunc</code> class method), 245
<code>register_analytic_integral()</code> (<code>zfit.core.basepdf.BasePDF</code> class method), 67	<code>register_analytic_integral()</code> (<code>zfit.models.functions.ZFunc</code> class method), 251
<code>register_analytic_integral()</code> (<code>zfit.core.interfaces.ZfitFunc</code> class method), 91	<code>register_analytic_integral()</code> (<code>zfit.models.functor.BaseFunctor</code> class method), 259
<code>register_analytic_integral()</code> (<code>zfit.core.interfaces.ZfitModel</code> class method), 93	<code>register_analytic_integral()</code> (<code>zfit.models.functor.ProductPDF</code> class method), 267
<code>register_analytic_integral()</code> (<code>zfit.core.interfaces.ZfitPDF</code> class method), 96	<code>register_analytic_integral()</code> (<code>zfit.models.functor.SumPDF</code> class method), 275
<code>register_analytic_integral()</code> (<code>zfit.func.BaseFunc</code> class method), 393	<code>register_analytic_integral()</code> (<code>zfit.models.physics.CrystalBall</code> class method), 284
<code>register_analytic_integral()</code> (<code>zfit.func.ProdFunc</code> class method), 399	<code>register_analytic_integral()</code> (<code>zfit.models.physics.DoubleCB</code> class method), 292
<code>register_analytic_integral()</code> (<code>zfit.func.SimpleFunc</code> class method), 411	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.Chebyshev</code> class method), 301
<code>register_analytic_integral()</code> (<code>zfit.func.SumFunc</code> class method), 405	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.Chebyshev2</code> class method), 309
<code>register_analytic_integral()</code> (<code>zfit.models.basefunctor.FunctorMixin</code> class method), 164	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.Hermite</code> class method), 318
<code>register_analytic_integral()</code> (<code>zfit.models.basic.CustomGaussOLD</code> class method), 172	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.Laguerre</code> class method), 326
<code>register_analytic_integral()</code> (<code>zfit.models.basic.Exponential</code> class method), 180	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.Legendre</code> class method), 334
<code>register_analytic_integral()</code> (<code>zfit.models.dist_tfp.ExponentialTFP</code> class method), 188	<code>register_analytic_integral()</code> (<code>zfit.models.polynomials.RecursivePolynomial</code> class method), 343
<code>register_analytic_integral()</code> (<code>zfit.models.dist_tfp.Gauss</code> class method), 196	<code>register_analytic_integral()</code> (<code>zfit.models.special.SimpleFunctorPDF</code> class method), 352
<code>register_analytic_integral()</code> (<code>zfit.models.dist_tfp.TruncatedGauss</code> class method), 205	<code>register_analytic_integral()</code> (<code>zfit.models.special.SimplePDF</code> class method), 360
<code>register_analytic_integral()</code> (<code>zfit.models.dist_tfp.Uniform</code> class method), 213	<code>register_analytic_integral()</code> (<code>zfit.models.special.ZPDF</code> class method), 368
<code>register_analytic_integral()</code> (<code>zfit.models.dist_tfp.WrapDistribution</code> class method), 221	<code>register_analytic_integral()</code> (<code>zfit.pdf.BaseFunctor</code> class method), 435
<code>register_analytic_integral()</code> (<code>zfit.models.functions.BaseFunctorFunc</code> class method), 227	<code>register_analytic_integral()</code>
<code>register_analytic_integral()</code> (<code>zfit.models.functions.ProdFunc</code> class method), 233	

(*zfit.pdf.BasePDF class method*), 427
 register_analytic_integral() (*zfit.pdf.Chebyshev class method*), 501
 register_analytic_integral() (*zfit.pdf.Chebyshev2 class method*), 518
 register_analytic_integral() (*zfit.pdf.CrystalBall class method*), 451
 register_analytic_integral() (*zfit.pdf.DoubleCB class method*), 460
 register_analytic_integral() (*zfit.pdf.Exponential class method*), 443
 register_analytic_integral() (*zfit.pdf.Gauss class method*), 468
 register_analytic_integral() (*zfit.pdf.Hermite class method*), 526
 register_analytic_integral() (*zfit.pdf.Laguerre class method*), 534
 register_analytic_integral() (*zfit.pdf.Legendre class method*), 509
 register_analytic_integral() (*zfit.pdf.ProductPDF class method*), 550
 register_analytic_integral() (*zfit.pdf.RecursivePolynomial class method*), 542
 register_analytic_integral() (*zfit.pdf.SimpleFunctorPDF class method*), 582
 register_analytic_integral() (*zfit.pdf.SimplePDF class method*), 574
 register_analytic_integral() (*zfit.pdf.SumPDF class method*), 559
 register_analytic_integral() (*zfit.pdf.TruncatedGauss class method*), 485
 register_analytic_integral() (*zfit.pdf.Uniform class method*), 476
 register_analytic_integral() (*zfit.pdf.WrapDistribution class method*), 492
 register_analytic_integral() (*zfit.pdf.ZPDF class method*), 566
 register_cacher() (*zfit.ComplexParameter method*), 42
 register_cacher() (*zfit.ComposedParameter method*), 40
 register_cacher() (*zfit.constraint.GaussianConstraint method*), 385
 register_cacher() (*zfit.constraint.SimpleConstraint method*), 383
 register_cacher() (*zfit.core.basefunc.BaseFunc method*), 52
 register_cacher() (*zfit.core.basemodel.BaseModel method*), 58
 register_cacher() (*zfit.core.baseobject.BaseNumeric method*), 60
 register_cacher() (*zfit.core.basepdf.BasePDF method*), 67
 register_cacher() (*zfit.core.constraint.BaseConstraint method*), 70
 register_cacher() (*zfit.core.constraint.DistributionConstraint method*), 71
 register_cacher() (*zfit.core.constraint.GaussianConstraint method*), 72
 register_cacher() (*zfit.core.constraint.SimpleConstraint method*), 74
 register_cacher() (*zfit.core.data.Data method*), 77
 register_cacher() (*zfit.core.data.SampleData method*), 80
 register_cacher() (*zfit.core.data.Sampler method*), 83
 register_cacher() (*zfit.core.loss.BaseLoss method*), 105
 register_cacher() (*zfit.core.loss.CachedLoss method*), 106
 register_cacher() (*zfit.core.loss.ExtendedUnbinnedNLL method*), 107
 register_cacher() (*zfit.core.loss.SimpleLoss method*), 108
 register_cacher() (*zfit.core.loss.UnbinnedNLL method*), 109
 register_cacher() (*zfit.core.parameter.BaseComposedParameter method*), 111
 register_cacher() (*zfit.core.parameter.BaseZParameter method*), 113
 register_cacher() (*zfit.core.parameter.ComplexParameter method*), 114
 register_cacher() (*zfit.core.parameter.ComposedParameter method*), 115
 register_cacher() (*zfit.core.parameter.ConstantParameter method*), 124
 register_cacher() (*zfit.core.parameter.Parameter method*), 129
 register_cacher() (*zfit.core.parameter.ZfitParameterMixin*

`method)`, 142
`register_cacher()` (`zfit.data.Data` `method`), 388
`register_cacher()` (`zfit.func.BaseFunc` `method`), 394
`register_cacher()` (`zfit.func.ProdFunc` `method`), 400
`register_cacher()` (`zfit.func.SimpleFunc` `method`), 412
`register_cacher()` (`zfit.func.SumFunc` `method`), 406
`register_cacher()` (`zfit.loss.BaseLoss` `method`), 415
`register_cacher()` (`zfit.loss.ExtendedUnbinnedNLL` `method`), 413
`register_cacher()` (`zfit.loss.SimpleLoss` `method`), 416
`register_cacher()` (`zfit.loss.UnbinnedNLL` `method`), 414
`register_cacher()` (`zfit.minimize.Minuit` `method`), 419
`register_cacher()` (`zfit.minimizers.minimizer_minuit.Minuit` `method`), 154
`register_cacher()` (`zfit.models.basefunctor.FunctorMixin` `method`), 165
`register_cacher()` (`zfit.models.basic.CustomGaussOLD` `method`), 172
`register_cacher()` (`zfit.models.basic.Exponential` `method`), 180
`register_cacher()` (`zfit.models.dist_tfp.ExponentialTFP` `method`), 189
`register_cacher()` (`zfit.models.dist_tfp.Gauss` `method`), 197
`register_cacher()` (`zfit.models.dist_tfp.TruncatedGauss` `method`), 205
`register_cacher()` (`zfit.models.dist_tfp.Uniform` `method`), 213
`register_cacher()` (`zfit.models.dist_tfp.WrapDistribution` `method`), 221
`register_cacher()` (`zfit.models.functions.BaseFunc` `method`), 228
`register_cacher()` (`zfit.models.functions.ProdFunc` `method`), 234
`register_cacher()` (`zfit.models.functions.SimpleFunc` `method`), 240
`register_cacher()` (`zfit.models.functions.SumFunc` `method`), 246
`register_cacher()` (`zfit.models.functions.ZFunc` `method`), 252
`register_cacher()` (`zfit.models.functor.BaseFunc` `method`), 259
`register_cacher()` (`zfit.models.functor.ProductPDF` `method`), 267
`register_cacher()` (`zfit.models.functor.SumPDF` `method`), 276
`register_cacher()` (`zfit.models.physics.CrystalBall` `method`), 284
`register_cacher()` (`zfit.models.physics.DoubleCB` `method`), 293
`register_cacher()` (`zfit.models.polynomials.Chebyshev` `method`), 301
`register_cacher()` (`zfit.models.polynomials.Chebyshev2` `method`), 310
`register_cacher()` (`zfit.models.polynomials.Hermite` `method`), 318
`register_cacher()` (`zfit.models.polynomials.Laguerre` `method`), 327
`register_cacher()` (`zfit.models.polynomials.Legendre` `method`), 335
`register_cacher()` (`zfit.models.polynomials.RecursivePolynomial` `method`), 343
`register_cacher()` (`zfit.models.special.SimpleFunc` `method`), 353
`register_cacher()` (`zfit.models.special.SimplePDF` `method`), 360
`register_cacher()` (`zfit.models.special.ZPDF` `method`), 368
`register_cacher()` (`zfit.param.ConstantParameter` `method`), 421
`register_cacher()` (`zfit.Parameter` `method`), 36
`register_cacher()` (`zfit.pdf.BaseFunc` `method`), 435
`register_cacher()` (`zfit.pdf.BasePDF` `method`), 428
`register_cacher()` (`zfit.pdf.Chebyshev` `method`), 501
`register_cacher()` (`zfit.pdf.Chebyshev2` `method`), 518
`register_cacher()` (`zfit.pdf.CrystalBall` `method`), 452

`register_cacher()` (*zfit.pdf.DoubleCB method*), 461
`register_cacher()` (*zfit.pdf.Exponential method*), 443
`register_cacher()` (*zfit.pdf.Gauss method*), 469
`register_cacher()` (*zfit.pdf.Hermite method*), 527
`register_cacher()` (*zfit.pdf.Laguerre method*), 535
`register_cacher()` (*zfit.pdf.Legendre method*), 510
`register_cacher()` (*zfit.pdf.ProductPDF method*), 551
`register_cacher()` (*zfit.pdf.RecursivePolynomial method*), 543
`register_cacher()` (*zfit.pdf.SimpleFunctorPDF method*), 583
`register_cacher()` (*zfit.pdf.SimplePDF method*), 575
`register_cacher()` (*zfit.pdf.SumPDF method*), 559
`register_cacher()` (*zfit.pdf.TruncatedGauss method*), 485
`register_cacher()` (*zfit.pdf.Uniform method*), 477
`register_cacher()` (*zfit.pdf.WrapDistribution method*), 493
`register_cacher()` (*zfit.pdf.ZPDF method*), 567
`register_cacher()` (*zfit.util.cache.Cachable method*), 371
`register_cacher()` (*zfit.util.cache.ZfitCacheable method*), 371
`register_inverse_analytic_integral()` (*zfit.core.basefunc.BaseFunc class method*), 52
`register_inverse_analytic_integral()` (*zfit.core.basemodel.BaseModel class method*), 58
`register_inverse_analytic_integral()` (*zfit.core.basepdf.BasePDF class method*), 68
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitFunc class method*), 91
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitModel class method*), 94
`register_inverse_analytic_integral()` (*zfit.core.interfaces.ZfitPDF class method*), 96
`register_inverse_analytic_integral()` (*zfit.func.BaseFunc class method*), 394
`register_inverse_analytic_integral()` (*zfit.func.ProdFunc class method*), 400
`register_inverse_analytic_integral()` (*zfit.func.SimpleFunc class method*), 412
`register_inverse_analytic_integral()` (*zfit.func.SumFunc class method*), 406
`register_inverse_analytic_integral()` (*zfit.models.basefunctor.FunctorMixin class method*), 165
`register_inverse_analytic_integral()` (*zfit.models.basic.CustomGaussOLD class method*), 172
`register_inverse_analytic_integral()` (*zfit.models.basic.Exponential class method*), 181
`register_inverse_analytic_integral()` (*zfit.models.dist_tfp.ExponentialTFP class method*), 189
`register_inverse_analytic_integral()` (*zfit.models.dist_tfp.Gauss class method*), 197
`register_inverse_analytic_integral()` (*zfit.models.dist_tfp.TruncatedGauss class method*), 205
`register_inverse_analytic_integral()` (*zfit.models.dist_tfp.Uniform class method*), 213
`register_inverse_analytic_integral()` (*zfit.models.dist_tfp.WrapDistribution class method*), 221
`register_inverse_analytic_integral()` (*zfit.models.functions.BaseFunctorFunc class method*), 228
`register_inverse_analytic_integral()` (*zfit.models.functions.ProdFunc class method*), 234
`register_inverse_analytic_integral()` (*zfit.models.functions.SimpleFunc class method*), 240
`register_inverse_analytic_integral()` (*zfit.models.functions.SumFunc class method*), 246
`register_inverse_analytic_integral()` (*zfit.models.functions.ZFunc class method*), 252
`register_inverse_analytic_integral()` (*zfit.models.functor.BaseFunctor class method*), 259
`register_inverse_analytic_integral()` (*zfit.models.functor.ProductPDF class method*), 267
`register_inverse_analytic_integral()` (*zfit.models.functor.SumPDF class method*), 276
`register_inverse_analytic_integral()` (*zfit.models.physics.CrystalBall class method*), 284
`register_inverse_analytic_integral()` (*zfit.models.physics.DoubleCB class method*), 293
`register_inverse_analytic_integral()` (*zfit.models.polynomials.Chebyshev class method*), 302
`register_inverse_analytic_integral()` (*zfit.models.polynomials.Chebyshev2 class method*), 310

`register_inverse_analytic_integral()`
 (*zfit.models.polynomials.Hermite class method*), 318
`register_inverse_analytic_integral()`
 (*zfit.models.polynomials.Laguerre class method*), 327
`register_inverse_analytic_integral()`
 (*zfit.models.polynomials.Legendre class method*), 335
`register_inverse_analytic_integral()`
 (*zfit.models.polynomials.RecursivePolynomial class method*), 343
`register_inverse_analytic_integral()`
 (*zfit.models.special.SimpleFunctorPDF class method*), 353
`register_inverse_analytic_integral()`
 (*zfit.models.special.SimplePDF class method*), 360
`register_inverse_analytic_integral()`
 (*zfit.models.special.ZPDF class method*), 368
`register_inverse_analytic_integral()`
 (*zfit.pdf.BaseFunctor class method*), 436
`register_inverse_analytic_integral()`
 (*zfit.pdf.BasePDF class method*), 428
`register_inverse_analytic_integral()`
 (*zfit.pdf.Chebyshev class method*), 502
`register_inverse_analytic_integral()`
 (*zfit.pdf.Chebyshev2 class method*), 518
`register_inverse_analytic_integral()`
 (*zfit.pdf.CrystalBall class method*), 452
`register_inverse_analytic_integral()`
 (*zfit.pdf.DoubleCB class method*), 461
`register_inverse_analytic_integral()`
 (*zfit.pdf.Exponential class method*), 444
`register_inverse_analytic_integral()`
 (*zfit.pdf.Gauss class method*), 469
`register_inverse_analytic_integral()`
 (*zfit.pdf.Hermite class method*), 527
`register_inverse_analytic_integral()`
 (*zfit.pdf.Laguerre class method*), 535
`register_inverse_analytic_integral()`
 (*zfit.pdf.Legendre class method*), 510
`register_inverse_analytic_integral()`
 (*zfit.pdf.ProductPDF class method*), 551
`register_inverse_analytic_integral()`
 (*zfit.pdf.RecursivePolynomial class method*), 543
`register_inverse_analytic_integral()`
 (*zfit.pdf.SimpleFunctorPDF class method*), 583
`register_inverse_analytic_integral()`
 (*zfit.pdf.SimplePDF class method*), 575
`register_inverse_analytic_integral()`
 (*zfit.pdf.SumPDF class method*), 559
`register_inverse_analytic_integral()`
 (*zfit.pdf.TruncatedGauss class method*), 485
`register_inverse_analytic_integral()`
 (*zfit.pdf.Uniform class method*), 477
`register_inverse_analytic_integral()`
 (*zfit.pdf.WrapDistribution class method*), 493
`register_inverse_analytic_integral()`
 (*zfit.pdf.ZPDF class method*), 567
`reorder_by_indices()` (*zfit.core.limits.Space method*), 102
`reorder_by_indices()`
 (*zfit.core.sample.EventSpace method*), 146
`reorder_by_indices()` (*zfit.Space method*), 45
`resample()` (*zfit.core.data.Sampler method*), 83
`rescale_minus_plus_one()` (in module *zfit.models.polynomials*), 345
`reset()` (*zfit.util.execution.RunManager method*), 377
`reset_cache()` (*zfit.ComplexParameter method*), 42
`reset_cache()` (*zfit.ComposedParameter method*), 40
`reset_cache()` (*zfit.constraint.GaussianConstraint method*), 385
`reset_cache()` (*zfit.constraint.SimpleConstraint method*), 383
`reset_cache()` (*zfit.core.basefunc.BaseFunc method*), 52
`reset_cache()` (*zfit.core.basemodel.BaseModel method*), 58
`reset_cache()` (*zfit.core.baseobject.BaseNumeric method*), 60
`reset_cache()` (*zfit.core.basepdf.BasePDF method*), 68
`reset_cache()` (*zfit.core.constraint.BaseConstraint method*), 70
`reset_cache()` (*zfit.core.constraint.DistributionConstraint method*), 71
`reset_cache()` (*zfit.core.constraint.GaussianConstraint method*), 72
`reset_cache()` (*zfit.core.constraint.SimpleConstraint method*), 74
`reset_cache()` (*zfit.core.data.Data method*), 77
`reset_cache()` (*zfit.core.data.SampleData method*), 80
`reset_cache()` (*zfit.core.data.Sampler method*), 84
`reset_cache()` (*zfit.core.loss.BaseLoss method*), 105
`reset_cache()` (*zfit.core.loss.CachedLoss method*), 106
`reset_cache()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 107
`reset_cache()` (*zfit.core.loss.SimpleLoss method*), 108
`reset_cache()` (*zfit.core.loss.UnbinnedNLL method*), 109
`reset_cache()` (*zfit.core.parameter.BaseComposedParameter method*), 111

`reset_cache()` (`zfit.core.parameter.BaseZParameter` method), 267
`reset_cache()` (`zfit.core.parameter.ComplexParameter` method), 113
`reset_cache()` (`zfit.core.parameter.ComposedParameter` method), 114
`reset_cache()` (`zfit.core.parameter.ConstantParameter` method), 115
`reset_cache()` (`zfit.core.parameter.Parameter` method), 124
`reset_cache()` (`zfit.core.parameter.ZfitParameterMixin` method), 129
`reset_cache()` (`zfit.data.Data` method), 142
`reset_cache()` (`zfit.func.BaseFunc` method), 388
`reset_cache()` (`zfit.func.ProdFunc` method), 394
`reset_cache()` (`zfit.func.SimpleFunc` method), 400
`reset_cache()` (`zfit.func.SumFunc` method), 406
`reset_cache()` (`zfit.loss.BaseLoss` method), 412
`reset_cache()` (`zfit.loss.ExtendedUnbinnedNLL` method), 413
`reset_cache()` (`zfit.loss.SimpleLoss` method), 415
`reset_cache()` (`zfit.loss.UnbinnedNLL` method), 417
`reset_cache()` (`zfit.minimize.Minuit` method), 419
`reset_cache()` (`zfit.minimizers.minimizer_minuit.Minuit` method), 154
`reset_cache()` (`zfit.models.basefunctor.FunctorMixin` method), 165
`reset_cache()` (`zfit.models.basic.CustomGaussOLD` method), 173
`reset_cache()` (`zfit.models.basic.Exponential` method), 181
`reset_cache()` (`zfit.models.dist_tfp.ExponentialTFP` method), 189
`reset_cache()` (`zfit.models.dist_tfp.Gauss` method), 197
`reset_cache()` (`zfit.models.dist_tfp.TruncatedGauss` method), 205
`reset_cache()` (`zfit.models.dist_tfp.Uniform` method), 213
`reset_cache()` (`zfit.models.dist_tfp.WrapDistribution` method), 221
`reset_cache()` (`zfit.models.functions.BaseFunctorFunc` method), 228
`reset_cache()` (`zfit.models.functions.ProdFunc` method), 234
`reset_cache()` (`zfit.models.functions.SimpleFunc` method), 240
`reset_cache()` (`zfit.models.functions.SumFunc` method), 246
`reset_cache()` (`zfit.models.functions.ZFunc` method), 252
`reset_cache()` (`zfit.models.functor.BaseFunctor` method), 259
`reset_cache()` (`zfit.models.functor.ProductPDF` method), 267
`reset_cache()` (`zfit.models.functor.SumPDF` method), 276
`reset_cache()` (`zfit.models.physics.CrystalBall` method), 284
`reset_cache()` (`zfit.models.physics.DoubleCB` method), 293
`reset_cache()` (`zfit.models.polynomials.Chebyshev` method), 302
`reset_cache()` (`zfit.models.polynomials.Chebyshev2` method), 310
`reset_cache()` (`zfit.models.polynomials.Hermite` method), 318
`reset_cache()` (`zfit.models.polynomials.Laguerre` method), 327
`reset_cache()` (`zfit.models.polynomials.Legendre` method), 335
`reset_cache()` (`zfit.models.polynomials.RecursivePolynomial` method), 343
`reset_cache()` (`zfit.models.special.SimpleFunctorPDF` method), 353
`reset_cache()` (`zfit.models.special.SimplePDF` method), 361
`reset_cache()` (`zfit.models.special.ZPDF` method), 368
`reset_cache()` (`zfit.param.ConstantParameter` method), 421
`reset_cache()` (`zfit.Parameter` method), 36
`reset_cache()` (`zfit.pdf.BaseFunctor` method), 436
`reset_cache()` (`zfit.pdf.BasePDF` method), 428
`reset_cache()` (`zfit.pdf.Chebyshev` method), 502
`reset_cache()` (`zfit.pdf.Chebyshev2` method), 518
`reset_cache()` (`zfit.pdf.CrystalBall` method), 452
`reset_cache()` (`zfit.pdf.DoubleCB` method), 461
`reset_cache()` (`zfit.pdf.Exponential` method), 444
`reset_cache()` (`zfit.pdf.Gauss` method), 469
`reset_cache()` (`zfit.pdf.Hermite` method), 527
`reset_cache()` (`zfit.pdf.Laguerre` method), 535
`reset_cache()` (`zfit.pdf.Legendre` method), 510
`reset_cache()` (`zfit.pdf.ProductPDF` method), 551
`reset_cache()` (`zfit.pdf.RecursivePolynomial` method), 543
`reset_cache()` (`zfit.pdf.SimpleFunctorPDF` method), 583
`reset_cache()` (`zfit.pdf.SimplePDF` method), 575
`reset_cache()` (`zfit.pdf.SumPDF` method), 559
`reset_cache()` (`zfit.pdf.TruncatedGauss` method), 485
`reset_cache()` (`zfit.pdf.Uniform` method), 477
`reset_cache()` (`zfit.pdf.WrapDistribution` method), 493
`reset_cache()` (`zfit.pdf.ZPDF` method), 567
`reset_cache()` (`zfit.util.cache.Cachable` method), 371

`reset_cache()` (*zfit.util.cache.ZfitCachable method*), 371
`reset_cache_self()` (*zfit.ComplexParameter method*), 42
`reset_cache_self()` (*zfit.ComposedParameter method*), 40
`reset_cache_self()` (*zfit.constraint.GaussianConstraint method*), 385
`reset_cache_self()` (*zfit.constraint.SimpleConstraint method*), 383
`reset_cache_self()` (*zfit.core.basefunc.BaseFunc method*), 52
`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*), 68
`reset_cache_self()` (*zfit.core.constraint.BaseConstraint method*), 70
`reset_cache_self()` (*zfit.core.constraint.DistributionConstraint method*), 71
`reset_cache_self()` (*zfit.core.constraint.GaussianConstraint method*), 72
`reset_cache_self()` (*zfit.core.constraint.SimpleConstraint method*), 74
`reset_cache_self()` (*zfit.core.data.Data method*), 77
`reset_cache_self()` (*zfit.core.data.SampleData method*), 80
`reset_cache_self()` (*zfit.core.data.Sampler method*), 84
`reset_cache_self()` (*zfit.core.loss.BaseLoss method*), 105
`reset_cache_self()` (*zfit.core.loss.CachedLoss method*), 106
`reset_cache_self()` (*zfit.core.loss.ExtendedUnbinnedNLL method*), 107
`reset_cache_self()` (*zfit.core.loss.SimpleLoss method*), 108
`reset_cache_self()` (*zfit.core.loss.UnbinnedNLL method*), 109
`reset_cache_self()` (*zfit.core.parameter.BaseComposedParameter method*), 112
`reset_cache_self()` (*zfit.core.parameter.BaseZParameter method*), 113
`reset_cache_self()` (*zfit.core.parameter.ComplexParameter method*), 114
`reset_cache_self()` (*zfit.core.parameter.ComposedParameter method*), 115
`reset_cache_self()` (*zfit.core.parameter.ConstantParameter method*), 124
`reset_cache_self()` (*zfit.core.parameter.Parameter method*), 129
`reset_cache_self()` (*zfit.core.parameter.ZfitParameterMixin method*), 142
`reset_cache_self()` (*zfit.data.Data method*), 388
`reset_cache_self()` (*zfit.func.BaseFunc method*), 394
`reset_cache_self()` (*zfit.func.ProdFunc method*), 400
`reset_cache_self()` (*zfit.func.SimpleFunc method*), 412
`reset_cache_self()` (*zfit.func.SumFunc method*), 406
`reset_cache_self()` (*zfit.loss.BaseLoss method*), 415
`reset_cache_self()` (*zfit.loss.ExtendedUnbinnedNLL method*), 413
`reset_cache_self()` (*zfit.loss.SimpleLoss method*), 417
`reset_cache_self()` (*zfit.loss.UnbinnedNLL method*), 414
`reset_cache_self()` (*zfit.minimize.Minuit method*), 419
`reset_cache_self()` (*zfit.minimizers.minimizer_minuit.Minuit method*), 155
`reset_cache_self()` (*zfit.models.basefunctor.FunctorMixin method*), 165
`reset_cache_self()` (*zfit.models.basic.CustomGaussOLD method*), 173
`reset_cache_self()` (*zfit.models.basic.Exponential method*), 181
`reset_cache_self()` (*zfit.models.dist_tfp.ExponentialTFP method*), 189
`reset_cache_self()` (*zfit.models.dist_tfp.Gauss method*), 197
`reset_cache_self()`

(*zfit.models.dist_tfp.TruncatedGauss* method), 205

reset_cache_self() (*zfit.models.dist_tfp.Uniform* method), 213

reset_cache_self() (*zfit.models.dist_tfp.WrapDistribution* method), 221

reset_cache_self() (*zfit.models.functions.BaseFunc* method), 228

reset_cache_self() (*zfit.models.functions.ProdFunc* method), 234

reset_cache_self() (*zfit.models.functions.SimpleFunc* method), 240

reset_cache_self() (*zfit.models.functions.SumFunc* method), 246

reset_cache_self() (*zfit.models.functions.ZFunc* method), 252

reset_cache_self() (*zfit.models.functor.BaseFunc* method), 259

reset_cache_self() (*zfit.models.functor.ProductPDF* method), 267

reset_cache_self() (*zfit.models.functor.SumPDF* method), 276

reset_cache_self() (*zfit.models.physics.CrystalBall* method), 284

reset_cache_self() (*zfit.models.physics.DoubleCB* method), 293

reset_cache_self() (*zfit.models.polynomials.Chebyshev* method), 302

reset_cache_self() (*zfit.models.polynomials.Chebyshev2* method), 310

reset_cache_self() (*zfit.models.polynomials.Hermite* method), 318

reset_cache_self() (*zfit.models.polynomials.Laguerre* method), 327

reset_cache_self() (*zfit.models.polynomials.Legendre* method), 335

reset_cache_self() (*zfit.models.polynomials.RecursivePolynomial* method), 343

reset_cache_self() (*zfit.models.special.SimpleFunc* method), 353

reset_cache_self() (*zfit.models.special.SimplePDF* method), 361

reset_cache_self() (*zfit.models.special.ZPDF* method), 368

reset_cache_self() (*zfit.param.ConstantParameter* method), 421

reset_cache_self() (*zfit.Parameter* method), 36

reset_cache_self() (*zfit.pdf.BaseFunc* method), 436

reset_cache_self() (*zfit.pdf.BasePDF* method), 428

reset_cache_self() (*zfit.pdf.Chebyshev* method), 502

reset_cache_self() (*zfit.pdf.Chebyshev2* method), 518

reset_cache_self() (*zfit.pdf.CrystalBall* method), 452

reset_cache_self() (*zfit.pdf.DoubleCB* method), 461

reset_cache_self() (*zfit.pdf.Exponential* method), 444

reset_cache_self() (*zfit.pdf.Gauss* method), 469

reset_cache_self() (*zfit.pdf.Hermite* method), 527

reset_cache_self() (*zfit.pdf.Laguerre* method), 535

reset_cache_self() (*zfit.pdf.Legendre* method), 510

reset_cache_self() (*zfit.pdf.ProductPDF* method), 551

reset_cache_self() (*zfit.pdf.RecursivePolynomial* method), 543

reset_cache_self() (*zfit.pdf.SimpleFunc* method), 583

reset_cache_self() (*zfit.pdf.SimplePDF* method), 575

reset_cache_self() (*zfit.pdf.SumPDF* method), 559

reset_cache_self() (*zfit.pdf.TruncatedGauss* method), 485

reset_cache_self() (*zfit.pdf.Uniform* method), 477

reset_cache_self() (*zfit.pdf.WrapDistribution* method), 493

reset_cache_self() (*zfit.pdf.ZPDF* method), 567

reset_cache_self() (*zfit.util.cache.Cachable* method), 371

reset_cache_self() (*zfit.util.cache.ZfitCacheable* method), 371

run_no_nan() (in module *zfit.ztf.zextension*), 381

RunManager (class in *zfit.util.execution*), 377

S

- `safe_where()` (in module `zfit.ztf.zextension`), 381
- `sample()` (`zfit.constraint.GaussianConstraint` method), 385
- `sample()` (`zfit.constraint.SimpleConstraint` method), 383
- `sample()` (`zfit.core.basefunc.BaseFunc` method), 52
- `sample()` (`zfit.core.basemodel.BaseModel` method), 58
- `sample()` (`zfit.core.basepdf.BasePDF` method), 68
- `sample()` (`zfit.core.constraint.BaseConstraint` method), 70
- `sample()` (`zfit.core.constraint.DistributionConstraint` method), 71
- `sample()` (`zfit.core.constraint.GaussianConstraint` method), 73
- `sample()` (`zfit.core.constraint.SimpleConstraint` method), 74
- `sample()` (`zfit.core.interfaces.ZfitFunc` method), 91
- `sample()` (`zfit.core.interfaces.ZfitModel` method), 94
- `sample()` (`zfit.core.interfaces.ZfitPDF` method), 96
- `sample()` (`zfit.func.BaseFunc` method), 394
- `sample()` (`zfit.func.ProdFunc` method), 400
- `sample()` (`zfit.func.SimpleFunc` method), 412
- `sample()` (`zfit.func.SumFunc` method), 406
- `sample()` (`zfit.models.basefuncor.FunctorMixin` method), 165
- `sample()` (`zfit.models.basic.CustomGaussOLD` method), 173
- `sample()` (`zfit.models.basic.Exponential` method), 181
- `sample()` (`zfit.models.dist_tfp.ExponentialTFP` method), 189
- `sample()` (`zfit.models.dist_tfp.Gauss` method), 197
- `sample()` (`zfit.models.dist_tfp.TruncatedGauss` method), 205
- `sample()` (`zfit.models.dist_tfp.Uniform` method), 213
- `sample()` (`zfit.models.dist_tfp.WrapDistribution` method), 221
- `sample()` (`zfit.models.functions.BaseFuncorFunc` method), 228
- `sample()` (`zfit.models.functions.ProdFunc` method), 234
- `sample()` (`zfit.models.functions.SimpleFunc` method), 240
- `sample()` (`zfit.models.functions.SumFunc` method), 246
- `sample()` (`zfit.models.functions.ZFunc` method), 252
- `sample()` (`zfit.models.functor.BaseFuncor` method), 260
- `sample()` (`zfit.models.functor.ProductPDF` method), 268
- `sample()` (`zfit.models.functor.SumPDF` method), 276
- `sample()` (`zfit.models.physics.CrystalBall` method), 284
- `sample()` (`zfit.models.physics.DoubleCB` method), 293
- `sample()` (`zfit.models.polynomials.Chebyshev` method), 302
- `sample()` (`zfit.models.polynomials.Chebyshev2` method), 310
- `sample()` (`zfit.models.polynomials.Hermite` method), 318
- `sample()` (`zfit.models.polynomials.Laguerre` method), 327
- `sample()` (`zfit.models.polynomials.Legendre` method), 335
- `sample()` (`zfit.models.polynomials.RecursivePolynomial` method), 343
- `sample()` (`zfit.models.special.SimpleFuncorPDF` method), 353
- `sample()` (`zfit.models.special.SimplePDF` method), 361
- `sample()` (`zfit.models.special.ZPDF` method), 368
- `sample()` (`zfit.pdf.BaseFuncor` method), 436
- `sample()` (`zfit.pdf.BasePDF` method), 428
- `sample()` (`zfit.pdf.Chebyshev` method), 502
- `sample()` (`zfit.pdf.Chebyshev2` method), 518
- `sample()` (`zfit.pdf.CrystalBall` method), 452
- `sample()` (`zfit.pdf.DoubleCB` method), 461
- `sample()` (`zfit.pdf.Exponential` method), 444
- `sample()` (`zfit.pdf.Gauss` method), 469
- `sample()` (`zfit.pdf.Hermite` method), 527
- `sample()` (`zfit.pdf.Laguerre` method), 535
- `sample()` (`zfit.pdf.Legendre` method), 510
- `sample()` (`zfit.pdf.ProductPDF` method), 551
- `sample()` (`zfit.pdf.RecursivePolynomial` method), 543
- `sample()` (`zfit.pdf.SimpleFuncorPDF` method), 583
- `sample()` (`zfit.pdf.SimplePDF` method), 575
- `sample()` (`zfit.pdf.SumPDF` method), 559
- `sample()` (`zfit.pdf.TruncatedGauss` method), 485
- `sample()` (`zfit.pdf.Uniform` method), 477
- `sample()` (`zfit.pdf.WrapDistribution` method), 493
- `sample()` (`zfit.pdf.ZPDF` method), 567
- `SampleData` (class in `zfit.core.data`), 77
- `Sampler` (class in `zfit.core.data`), 81
- `scatter_add()` (`zfit.core.parameter.ComposedResourceVariable` method), 120
- `scatter_add()` (`zfit.core.parameter.Parameter` method), 129
- `scatter_add()` (`zfit.core.parameter.TFBaseVariable` method), 138
- `scatter_add()` (`zfit.Parameter` method), 36
- `scatter_nd_add()` (`zfit.core.parameter.ComposedResourceVariable` method), 120
- `scatter_nd_add()` (`zfit.core.parameter.Parameter` method), 130
- `scatter_nd_add()` (`zfit.core.parameter.TFBaseVariable` method), 138
- `scatter_nd_add()` (`zfit.Parameter` method), 36

`scatter_nd_sub()` (`zfit.core.parameter.ComposedResourceVariable` attribute), 121
`scatter_nd_sub()` (`zfit.core.parameter.Parameter` attribute), 130
`scatter_nd_sub()` (`zfit.core.parameter.TFBaseVariable` attribute), 139
`scatter_nd_sub()` (`zfit.Parameter` method), 37
`scatter_nd_update()` (`zfit.core.parameter.ComposedResourceVariable` method), 121
`scatter_nd_update()` (`zfit.core.parameter.Parameter` method), 131
`scatter_nd_update()` (`zfit.core.parameter.TFBaseVariable` method), 140
`scatter_nd_update()` (`zfit.Parameter` method), 38
`scatter_sub()` (`zfit.core.parameter.ComposedResourceVariable` method), 122
`scatter_sub()` (`zfit.core.parameter.Parameter` method), 132
`scatter_sub()` (`zfit.core.parameter.TFBaseVariable` method), 140
`scatter_sub()` (`zfit.Parameter` method), 38
`scatter_update()` (`zfit.core.parameter.ComposedResourceVariable` method), 122
`scatter_update()` (`zfit.core.parameter.Parameter` method), 132
`scatter_update()` (`zfit.core.parameter.TFBaseVariable` method), 141
`scatter_update()` (`zfit.Parameter` method), 39
`Scipy` (class in `zfit.minimize`), 419
`Scipy` (class in `zfit.minimizers.minimizers_scipy`), 156
`ScipyMinimizer` (in module `zfit.minimize`), 417
`ScipyOptimizerInterface` (class in `zfit.minimizers.tf_external_optimizer`), 158
`sess` (`zfit.core.data.Data` attribute), 77
`sess` (`zfit.core.data.SampleData` attribute), 80
`sess` (`zfit.core.data.Sampler` attribute), 84
`sess` (`zfit.core.parameter.Parameter` attribute), 132
`sess` (`zfit.data.Data` attribute), 388
`sess` (`zfit.minimize.Adam` attribute), 418
`sess` (`zfit.minimize.Minuit` attribute), 419
`sess` (`zfit.minimize.Scipy` attribute), 420
`sess` (`zfit.minimize.WrapOptimizer` attribute), 417
`sess` (`zfit.minimizers.base_tf.WrapOptimizer` attribute), 149
`sess` (`zfit.minimizers.baseminimizer.BaseMinimizer` attribute), 150
`sess` (`zfit.minimizers.fitresult.FitResult` attribute), 152
`sess` (`zfit.minimizers.minimizer_minuit.Minuit` attribute), 155
`sess` (`zfit.minimizers.minimizer_tfp.BFGSMinimizer` attribute), 155
`sess` (`zfit.minimizers.optimizers_tf.Adam` attribute), 157
`sess` (`zfit.Parameter` attribute), 39
`sess` (`zfit.util.execution.RunManager` attribute), 377
`sess` (`zfit.util.execution.SessionHolderMixin` attribute), 377
`SessionHolderMixin` (class in `zfit.util.execution`), 377
`set_data_range()` (`zfit.core.data.Data` method), 77
`set_data_range()` (`zfit.core.data.SampleData` method), 80
`set_data_range()` (`zfit.core.data.Sampler` method), 84
`set_data_range()` (`zfit.data.Data` method), 388
`set_n_cpu()` (`zfit.util.execution.RunManager` method), 377
`set_norm_range()` (`zfit.core.basepdf.BasePDF` method), 68
`set_norm_range()` (`zfit.core.interfaces.ZfitPDF` method), 96
`set_norm_range()` (`zfit.models.basic.CustomGaussOLD` method), 173
`set_norm_range()` (`zfit.models.basic.Exponential` method), 181
`set_norm_range()` (`zfit.models.dist_tfp.ExponentialTFP` method), 189
`set_norm_range()` (`zfit.models.dist_tfp.Gauss` method), 198
`set_norm_range()` (`zfit.models.dist_tfp.TruncatedGauss` method), 206
`set_norm_range()` (`zfit.models.dist_tfp.Uniform` method), 214
`set_norm_range()` (`zfit.models.dist_tfp.WrapDistribution` method), 222
`set_norm_range()` (`zfit.models.functor.BaseFunctor` method), 260
`set_norm_range()` (`zfit.models.functor.ProductPDF` method), 268
`set_norm_range()` (`zfit.models.functor.SumPDF` method), 276
`set_norm_range()` (`zfit.models.physics.CrystalBall` method), 285
`set_norm_range()` (`zfit.models.physics.DoubleCB` method), 294
`set_norm_range()` (`zfit.models.polynomials.Chebyshev` method), 302
`set_norm_range()` (`zfit.models.polynomials.Chebyshev2` method), 311
`set_norm_range()` (`zfit.models.polynomials.Hermite` method), 319
`set_norm_range()` (`zfit.models.polynomials.Laguerre` method), 327
`set_norm_range()` (`zfit.models.polynomials.Legendre`

- method*), 336
- `set_norm_range()` (*zfit.models.polynomials.RecursivePolynomials* *method*), 344
- `set_norm_range()` (*zfit.models.special.SimpleFunctorPDF* *method*), 353
- `set_norm_range()` (*zfit.models.special.SimplePDF* *method*), 361
- `set_norm_range()` (*zfit.models.special.ZPDF* *method*), 369
- `set_norm_range()` (*zfit.pdf.BaseFunctor* *method*), 436
- `set_norm_range()` (*zfit.pdf.BasePDF* *method*), 428
- `set_norm_range()` (*zfit.pdf.Chebyshev* *method*), 502
- `set_norm_range()` (*zfit.pdf.Chebyshev2* *method*), 519
- `set_norm_range()` (*zfit.pdf.CrystalBall* *method*), 453
- `set_norm_range()` (*zfit.pdf.DoubleCB* *method*), 461
- `set_norm_range()` (*zfit.pdf.Exponential* *method*), 444
- `set_norm_range()` (*zfit.pdf.Gauss* *method*), 470
- `set_norm_range()` (*zfit.pdf.Hermite* *method*), 527
- `set_norm_range()` (*zfit.pdf.Laguerre* *method*), 536
- `set_norm_range()` (*zfit.pdf.Legendre* *method*), 511
- `set_norm_range()` (*zfit.pdf.ProductPDF* *method*), 552
- `set_norm_range()` (*zfit.pdf.RecursivePolynomial* *method*), 544
- `set_norm_range()` (*zfit.pdf.SimpleFunctorPDF* *method*), 584
- `set_norm_range()` (*zfit.pdf.SimplePDF* *method*), 576
- `set_norm_range()` (*zfit.pdf.SumPDF* *method*), 560
- `set_norm_range()` (*zfit.pdf.TruncatedGauss* *method*), 486
- `set_norm_range()` (*zfit.pdf.Uniform* *method*), 478
- `set_norm_range()` (*zfit.pdf.WrapDistribution* *method*), 494
- `set_norm_range()` (*zfit.pdf.ZPDF* *method*), 568
- `set_seed()` (in module *zfit.settings*), 584
- `set_sess()` (*zfit.core.data.Data* *method*), 77
- `set_sess()` (*zfit.core.data.SampleData* *method*), 80
- `set_sess()` (*zfit.core.data.Sampler* *method*), 84
- `set_sess()` (*zfit.core.parameter.Parameter* *method*), 132
- `set_sess()` (*zfit.data.Data* *method*), 388
- `set_sess()` (*zfit.minimize.Adam* *method*), 418
- `set_sess()` (*zfit.minimize.Minuit* *method*), 419
- `set_sess()` (*zfit.minimize.Scipy* *method*), 420
- `set_sess()` (*zfit.minimize.WrapOptimizer* *method*), 417
- `set_sess()` (*zfit.minimizers.base_tf.WrapOptimizer* *method*), 149
- `set_sess()` (*zfit.minimizers.baseminimizer.BaseMinimizer* *method*), 150
- `set_sess()` (*zfit.minimizers.fitresult.FitResult* *method*), 152
- `set_sess()` (*zfit.minimizers.minimizer_minuit.Minuit* *method*), 155
- `set_sess()` (*zfit.minimizers.minimizer_tfp.BFGSMinimizer* *method*), 155
- `set_sess()` (*zfit.minimizers.minimizers_scipy.Scipy* *method*), 156
- `set_sess()` (*zfit.minimizers.optimizers_tf.Adam* *method*), 157
- `set_sess()` (*zfit.Parameter* *method*), 39
- `set_sess()` (*zfit.util.execution.SessionHolderMixin* *method*), 377
- `set_shape()` (*zfit.core.parameter.ComposedResourceVariable* *method*), 122
- `set_shape()` (*zfit.core.parameter.Parameter* *method*), 132
- `set_shape()` (*zfit.core.parameter.TFBaseVariable* *method*), 141
- `set_shape()` (*zfit.Parameter* *method*), 39
- `set_value()` (*zfit.core.parameter.Parameter* *method*), 132
- `set_value()` (*zfit.Parameter* *method*), 39
- `set_verbosity()` (in module *zfit.settings*), 584
- `set_weights()` (*zfit.core.data.Data* *method*), 77
- `set_weights()` (*zfit.core.data.SampleData* *method*), 80
- `set_weights()` (*zfit.core.data.Sampler* *method*), 84
- `set_weights()` (*zfit.data.Data* *method*), 388
- `setdefault()` (*zfit.util.container.DotDict* *method*), 372
- `setup_function()` (in module *zfit.core.testing*), 148
- `shape` (*zfit.core.parameter.ComposedResourceVariable* *attribute*), 122
- `shape` (*zfit.core.parameter.Parameter* *attribute*), 132
- `shape` (*zfit.core.parameter.TFBaseVariable* *attribute*), 141
- `shape` (*zfit.Parameter* *attribute*), 39
- `ShapeIncompatibleError`, 376
- `SimpleConstraint` (class in *zfit.constraint*), 382
- `SimpleConstraint` (class in *zfit.core.constraint*), 73
- `SimpleFunc` (class in *zfit.func*), 407
- `SimpleFunc` (class in *zfit.models.functions*), 235
- `SimpleFunctorPDF` (class in *zfit.models.special*), 346
- `SimpleFunctorPDF` (class in *zfit.pdf*), 576
- `SimpleLoss` (class in *zfit.core.loss*), 107
- `SimpleLoss` (class in *zfit.loss*), 416
- `SimpleModelSubclassMixin` (class in *zfit.core.basemodel*), 59
- `SimplePDF` (class in *zfit.models.special*), 354
- `SimplePDF` (class in *zfit.pdf*), 568
- `sort_by_axes()` (*zfit.core.data.Data* *method*), 77
- `sort_by_axes()` (*zfit.core.data.SampleData* *method*), 150

- method*), 80
- `sort_by_axes()` (*zfit.core.data.Sampler method*), 84
- `sort_by_axes()` (*zfit.core.integration.PartialIntegralSampleData method*), 88
- `sort_by_axes()` (*zfit.core.interfaces.ZfitData method*), 89
- `sort_by_axes()` (*zfit.data.Data method*), 388
- `sort_by_obs()` (*zfit.core.data.Data method*), 77
- `sort_by_obs()` (*zfit.core.data.SampleData method*), 80
- `sort_by_obs()` (*zfit.core.data.Sampler method*), 84
- `sort_by_obs()` (*zfit.core.integration.PartialIntegralSampleData method*), 88
- `sort_by_obs()` (*zfit.core.interfaces.ZfitData method*), 89
- `sort_by_obs()` (*zfit.data.Data method*), 388
- `Space` (class in *zfit*), 42
- `Space` (class in *zfit.core.limits*), 99
- `space` (*zfit.core.basefunc.BaseFunc attribute*), 52
- `space` (*zfit.core.basemodel.BaseModel attribute*), 58
- `space` (*zfit.core.basepdf.BasePDF attribute*), 68
- `space` (*zfit.core.data.Data attribute*), 77
- `space` (*zfit.core.data.SampleData attribute*), 80
- `space` (*zfit.core.data.Sampler attribute*), 84
- `space` (*zfit.core.dimension.BaseDimensional attribute*), 85
- `space` (*zfit.core.integration.PartialIntegralSampleData attribute*), 88
- `space` (*zfit.core.interfaces.ZfitData attribute*), 89
- `space` (*zfit.core.interfaces.ZfitDimensional attribute*), 90
- `space` (*zfit.core.interfaces.ZfitFunc attribute*), 92
- `space` (*zfit.core.interfaces.ZfitModel attribute*), 94
- `space` (*zfit.core.interfaces.ZfitPDF attribute*), 96
- `space` (*zfit.data.Data attribute*), 388
- `space` (*zfit.func.BaseFunc attribute*), 394
- `space` (*zfit.func.ProdFunc attribute*), 400
- `space` (*zfit.func.SimpleFunc attribute*), 412
- `space` (*zfit.func.SumFunc attribute*), 406
- `space` (*zfit.models.basefunc.FunctorMixin attribute*), 166
- `space` (*zfit.models.basic.CustomGaussOLD attribute*), 173
- `space` (*zfit.models.basic.Exponential attribute*), 181
- `space` (*zfit.models.dist_tfp.ExponentialTFP attribute*), 189
- `space` (*zfit.models.dist_tfp.Gauss attribute*), 198
- `space` (*zfit.models.dist_tfp.TruncatedGauss attribute*), 206
- `space` (*zfit.models.dist_tfp.Uniform attribute*), 214
- `space` (*zfit.models.dist_tfp.WrapDistribution attribute*), 222
- `space` (*zfit.models.functions.BaseFuncatorFunc attribute*), 228
- `space` (*zfit.models.functions.ProdFunc attribute*), 234
- `space` (*zfit.models.functions.SimpleFunc attribute*), 240
- `space` (*zfit.models.functions.SumFunc attribute*), 246
- `space` (*zfit.models.functions.ZFunc attribute*), 252
- `space` (*zfit.models.functor.BaseFuncator attribute*), 260
- `space` (*zfit.models.functor.ProductPDF attribute*), 268
- `space` (*zfit.models.functor.SumPDF attribute*), 276
- `space` (*zfit.models.physics.CrystalBall attribute*), 285
- `space` (*zfit.models.physics.DoubleCB attribute*), 294
- `space` (*zfit.models.polynomials.Chebyshev attribute*), 302
- `space` (*zfit.models.polynomials.Chebyshev2 attribute*), 311
- `space` (*zfit.models.polynomials.Hermite attribute*), 319
- `space` (*zfit.models.polynomials.Laguerre attribute*), 327
- `space` (*zfit.models.polynomials.Legendre attribute*), 336
- `space` (*zfit.models.polynomials.RecursivePolynomial attribute*), 344
- `space` (*zfit.models.special.SimpleFuncatorPDF attribute*), 353
- `space` (*zfit.models.special.SimplePDF attribute*), 361
- `space` (*zfit.models.special.ZPDF attribute*), 369
- `space` (*zfit.pdf.BaseFuncator attribute*), 436
- `space` (*zfit.pdf.BasePDF attribute*), 428
- `space` (*zfit.pdf.Chebyshev attribute*), 502
- `space` (*zfit.pdf.Chebyshev2 attribute*), 519
- `space` (*zfit.pdf.CrystalBall attribute*), 453
- `space` (*zfit.pdf.DoubleCB attribute*), 461
- `space` (*zfit.pdf.Exponential attribute*), 444
- `space` (*zfit.pdf.Gauss attribute*), 470
- `space` (*zfit.pdf.Hermite attribute*), 527
- `space` (*zfit.pdf.Laguerre attribute*), 536
- `space` (*zfit.pdf.Legendre attribute*), 511
- `space` (*zfit.pdf.ProductPDF attribute*), 552
- `space` (*zfit.pdf.RecursivePolynomial attribute*), 544
- `space` (*zfit.pdf.SimpleFuncatorPDF attribute*), 584
- `space` (*zfit.pdf.SimplePDF attribute*), 576
- `space` (*zfit.pdf.SumPDF attribute*), 560
- `space` (*zfit.pdf.TruncatedGauss attribute*), 486
- `space` (*zfit.pdf.Uniform attribute*), 478
- `space` (*zfit.pdf.WrapDistribution attribute*), 494
- `space` (*zfit.pdf.ZPDF attribute*), 568
- `SpaceIncompatibleError`, 376
- `sparse_read()` (*zfit.core.parameter.ComposedResourceVariable method*), 122
- `sparse_read()` (*zfit.core.parameter.Parameter method*), 132
- `sparse_read()` (*zfit.core.parameter.TFBaseVariable method*), 141
- `sparse_read()` (*zfit.Parameter method*), 39
- `spec` (*zfit.core.parameter.ComposedResourceVariable.SaveSliceInfo attribute*), 116
- `spec` (*zfit.core.parameter.Parameter.SaveSliceInfo attribute*), 125

- spec (*zfit.core.parameter.TFBaseVariable.SaveSliceInfo* attribute), 134
- spec (*zfit.Parameter.SaveSliceInfo* attribute), 32
- sqrt () (in module *zfit.ztf.wrapping_tf*), 381
- square () (in module *zfit.ztf.wrapping_tf*), 381
- stack_x () (in module *zfit.ztf.zextension*), 382
- status (*zfit.minimizers.fitresult.FitResult* attribute), 152
- step () (*zfit.minimize.Adam* method), 418
- step () (*zfit.minimize.Minuit* method), 419
- step () (*zfit.minimize.Scipy* method), 420
- step () (*zfit.minimize.WrapOptimizer* method), 417
- step () (*zfit.minimizers.base_tf.WrapOptimizer* method), 149
- step () (*zfit.minimizers.baseminimizer.BaseMinimizer* method), 150
- step () (*zfit.minimizers.interface.ZfitMinimizer* method), 153
- step () (*zfit.minimizers.minimizer_minuit.Minuit* method), 155
- step () (*zfit.minimizers.minimizer_tfp.BFGSMinimizer* method), 155
- step () (*zfit.minimizers.minimizers_scipy.Scipy* method), 156
- step () (*zfit.minimizers.optimizers_tf.Adam* method), 157
- step_size (*zfit.core.parameter.Parameter* attribute), 132
- step_size (*zfit.Parameter* attribute), 39
- SubclassingError, 376
- SumFunc (class in *zfit.func*), 401
- SumFunc (class in *zfit.models.functions*), 241
- SumPDF (class in *zfit.models.functor*), 269
- SumPDF (class in *zfit.pdf*), 552
- supports () (in module *zfit*), 47
- supports () (in module *zfit.core.limits*), 103
- synchronization (*zfit.core.parameter.ComposedResourceVariable* attribute), 123
- synchronization (*zfit.core.parameter.Parameter* attribute), 132
- synchronization (*zfit.core.parameter.TFBaseVariable* attribute), 141
- synchronization (*zfit.Parameter* attribute), 39
- ## T
- teardown_function () (in module *zfit.core.testing*), 148
- TemporarilySet (class in *zfit.util.temporary*), 379
- TFBaseVariable (class in *zfit.core.parameter*), 133
- TFBaseVariable.SaveSliceInfo (class in *zfit.core.parameter*), 134
- tfd_analytic_sample () (in module *zfit.models.dist_tfp*), 222
- to_complex () (in module *zfit.ztf.zextension*), 382
- to_pandas () (*zfit.core.data.Data* method), 77
- to_pandas () (*zfit.core.data.SampleData* method), 80
- to_pandas () (*zfit.core.data.Sampler* method), 84
- to_pandas () (*zfit.data.Data* method), 388
- to_proto () (*zfit.core.parameter.ComposedResourceVariable* method), 123
- to_proto () (*zfit.core.parameter.ComposedResourceVariable.SaveSliceInfo* method), 116
- to_proto () (*zfit.core.parameter.Parameter* method), 132
- to_proto () (*zfit.core.parameter.Parameter.SaveSliceInfo* method), 125
- to_proto () (*zfit.core.parameter.TFBaseVariable* method), 141
- to_proto () (*zfit.core.parameter.TFBaseVariable.SaveSliceInfo* method), 134
- to_proto () (*zfit.Parameter* method), 39
- to_proto () (*zfit.Parameter.SaveSliceInfo* method), 32
- to_real () (in module *zfit.ztf.zextension*), 382
- tolerance (*zfit.minimize.Adam* attribute), 418
- tolerance (*zfit.minimize.Minuit* attribute), 419
- tolerance (*zfit.minimize.Scipy* attribute), 420
- tolerance (*zfit.minimize.WrapOptimizer* attribute), 417
- tolerance (*zfit.minimizers.base_tf.WrapOptimizer* attribute), 149
- tolerance (*zfit.minimizers.baseminimizer.BaseMinimizer* attribute), 150
- tolerance (*zfit.minimizers.interface.ZfitMinimizer* attribute), 153
- tolerance (*zfit.minimizers.minimizer_minuit.Minuit* attribute), 155
- tolerance (*zfit.minimizers.minimizer_tfp.BFGSMinimizer* attribute), 155
- tolerance (*zfit.minimizers.minimizers_scipy.Scipy* attribute), 156
- tolerance (*zfit.minimizers.optimizers_tf.Adam* attribute), 157
- ToyStrategyFail (class in *zfit.minimizers.baseminimizer*), 150
- trainable (*zfit.core.parameter.ComposedResourceVariable* attribute), 123
- trainable (*zfit.core.parameter.Parameter* attribute), 133
- trainable (*zfit.core.parameter.TFBaseVariable* attribute), 141
- trainable (*zfit.Parameter* attribute), 39
- TruncatedGauss (class in *zfit.models.dist_tfp*), 198
- TruncatedGauss (class in *zfit.pdf*), 478
- ## U
- UnbinnedNLL (class in *zfit.core.loss*), 108
- UnbinnedNLL (class in *zfit.loss*), 413
- UnderdefinedError, 377
- Uniform (class in *zfit.models.dist_tfp*), 206

Uniform (class in `zfit.pdf`), 470
 UniformSampleAndWeights (class in `zfit.core.sample`), 147
 unnormalized_pdf() (`zfit.core.basepdf.BasePDF` method), 68
 unnormalized_pdf() (`zfit.models.basic.CustomGaussOLD` method), 173
 unnormalized_pdf() (`zfit.models.basic.Exponential` method), 181
 unnormalized_pdf() (`zfit.models.dist_tfp.ExponentialTFP` method), 189
 unnormalized_pdf() (`zfit.models.dist_tfp.Gauss` method), 198
 unnormalized_pdf() (`zfit.models.dist_tfp.TruncatedGauss` method), 206
 unnormalized_pdf() (`zfit.models.dist_tfp.Uniform` method), 214
 unnormalized_pdf() (`zfit.models.dist_tfp.WrapDistribution` method), 222
 unnormalized_pdf() (`zfit.models.functor.BaseFunctor` method), 260
 unnormalized_pdf() (`zfit.models.functor.ProductPDF` method), 268
 unnormalized_pdf() (`zfit.models.functor.SumPDF` method), 276
 unnormalized_pdf() (`zfit.models.physics.CrystalBall` method), 285
 unnormalized_pdf() (`zfit.models.physics.DoubleCB` method), 294
 unnormalized_pdf() (`zfit.models.polynomials.Chebyshev` method), 302
 unnormalized_pdf() (`zfit.models.polynomials.Chebyshev2` method), 311
 unnormalized_pdf() (`zfit.models.polynomials.Hermite` method), 319
 unnormalized_pdf() (`zfit.models.polynomials.Laguerre` method), 327
 unnormalized_pdf() (`zfit.models.polynomials.Legendre` method), 336
 unnormalized_pdf() (`zfit.models.polynomials.RecursivePolynomial` method), 344
 unnormalized_pdf() (`zfit.models.special.SimpleFunctorPDF` method), 353
 unnormalized_pdf() (`zfit.models.special.SimplePDF` method), 361
 unnormalized_pdf() (`zfit.models.special.ZPDF` method), 369
 unnormalized_pdf() (`zfit.pdf.BaseFunctor` method), 436
 unnormalized_pdf() (`zfit.pdf.BasePDF` method), 428
 unnormalized_pdf() (`zfit.pdf.Chebyshev` method), 502
 unnormalized_pdf() (`zfit.pdf.Chebyshev2` method), 519
 unnormalized_pdf() (`zfit.pdf.CrystalBall` method), 453
 unnormalized_pdf() (`zfit.pdf.DoubleCB` method), 461
 unnormalized_pdf() (`zfit.pdf.Exponential` method), 444
 unnormalized_pdf() (`zfit.pdf.Gauss` method), 470
 unnormalized_pdf() (`zfit.pdf.Hermite` method), 527
 unnormalized_pdf() (`zfit.pdf.Laguerre` method), 536
 unnormalized_pdf() (`zfit.pdf.Legendre` method), 511
 unnormalized_pdf() (`zfit.pdf.ProductPDF` method), 552
 unnormalized_pdf() (`zfit.pdf.RecursivePolynomial` method), 544
 unnormalized_pdf() (`zfit.pdf.SimpleFunctorPDF` method), 584
 unnormalized_pdf() (`zfit.pdf.SimplePDF` method), 576
 unnormalized_pdf() (`zfit.pdf.SumPDF` method), 560
 unnormalized_pdf() (`zfit.pdf.TruncatedGauss` method), 486
 unnormalized_pdf() (`zfit.pdf.Uniform` method), 478
 unnormalized_pdf() (`zfit.pdf.WrapDistribution` method), 494
 unnormalized_pdf() (`zfit.pdf.ZPDF` method), 568
 unstack_x() (in module `zfit.ztf.zextension`), 382
 unstack_x() (`zfit.core.data.Data` method), 77
 unstack_x() (`zfit.core.data.SampleData` method), 80
 unstack_x() (`zfit.core.data.Sampler` method), 84
 unstack_x() (`zfit.core.integration.PartialIntegralSampleData` method), 88
 unstack_x() (`zfit.data.Data` method), 388
 update() (`zfit.util.container.DotDict` method), 372

[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](#)), 69
[update_integration_options\(\)](#)
 ([zfit.core.interfaces.ZfitFunc method](#)), 92
[update_integration_options\(\)](#)
 ([zfit.core.interfaces.ZfitModel method](#)), 94
[update_integration_options\(\)](#)
 ([zfit.core.interfaces.ZfitPDF method](#)), 96
[update_integration_options\(\)](#)
 ([zfit.func.BaseFunc method](#)), 394
[update_integration_options\(\)](#)
 ([zfit.func.ProdFunc method](#)), 400
[update_integration_options\(\)](#)
 ([zfit.func.SimpleFunc method](#)), 412
[update_integration_options\(\)](#)
 ([zfit.func.SumFunc method](#)), 406
[update_integration_options\(\)](#)
 ([zfit.models.basefunc.FunctorMixin method](#)), 166
[update_integration_options\(\)](#)
 ([zfit.models.basic.CustomGaussOLD method](#)), 174
[update_integration_options\(\)](#)
 ([zfit.models.basic.Exponential method](#)), 182
[update_integration_options\(\)](#)
 ([zfit.models.dist_tfp.ExponentialTFP method](#)), 190
[update_integration_options\(\)](#)
 ([zfit.models.dist_tfp.Gauss method](#)), 198
[update_integration_options\(\)](#)
 ([zfit.models.dist_tfp.TruncatedGauss method](#)), 206
[update_integration_options\(\)](#)
 ([zfit.models.dist_tfp.Uniform method](#)), 214
[update_integration_options\(\)](#)
 ([zfit.models.dist_tfp.WrapDistribution method](#)), 222
[update_integration_options\(\)](#)
 ([zfit.models.functions.BaseFuncorFunc method](#)), 228
[update_integration_options\(\)](#)
 ([zfit.models.functions.ProdFunc method](#)), 234
[update_integration_options\(\)](#)
 ([zfit.models.functions.SimpleFunc method](#)), 240
[update_integration_options\(\)](#)
 ([zfit.models.functions.SumFunc method](#)), 246
[update_integration_options\(\)](#)
 ([zfit.models.functions.ZFunc method](#)), 252
[update_integration_options\(\)](#)
 ([zfit.models.functor.BaseFuncor method](#)), 260
[update_integration_options\(\)](#)
 ([zfit.models.functor.ProductPDF method](#)), 268
[update_integration_options\(\)](#)
 ([zfit.models.functor.SumPDF method](#)), 277
[update_integration_options\(\)](#)
 ([zfit.models.physics.CrystalBall method](#)), 285
[update_integration_options\(\)](#)
 ([zfit.models.physics.DoubleCB method](#)), 294
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.Chebyshev method](#)), 303
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.Chebyshev2 method](#)), 311
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.Hermite method](#)), 319
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.Laguerre method](#)), 328
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.Legendre method](#)), 336
[update_integration_options\(\)](#)
 ([zfit.models.polynomials.RecursivePolynomial method](#)), 344
[update_integration_options\(\)](#)
 ([zfit.models.special.SimpleFuncorPDF method](#)), 354
[update_integration_options\(\)](#)
 ([zfit.models.special.SimplePDF method](#)), 362
[update_integration_options\(\)](#)
 ([zfit.models.special.ZPDF method](#)), 369
[update_integration_options\(\)](#)
 ([zfit.pdf.BaseFuncor method](#)), 437
[update_integration_options\(\)](#)
 ([zfit.pdf.BasePDF method](#)), 429
[update_integration_options\(\)](#)
 ([zfit.pdf.Chebyshev method](#)), 503
[update_integration_options\(\)](#)
 ([zfit.pdf.Chebyshev2 method](#)), 519
[update_integration_options\(\)](#)
 ([zfit.pdf.CrystalBall method](#)), 453
[update_integration_options\(\)](#)
 ([zfit.pdf.DoubleCB method](#)), 462

[update_integration_options\(\)](#) ([zfit.pdf.Exponential method](#)), 445
[update_integration_options\(\)](#) ([zfit.pdf.Gauss method](#)), 470
[update_integration_options\(\)](#) ([zfit.pdf.Hermite method](#)), 528
[update_integration_options\(\)](#) ([zfit.pdf.Laguerre method](#)), 536
[update_integration_options\(\)](#) ([zfit.pdf.Legendre method](#)), 511
[update_integration_options\(\)](#) ([zfit.pdf.ProductPDF method](#)), 552
[update_integration_options\(\)](#) ([zfit.pdf.RecursivePolynomial method](#)), 544
[update_integration_options\(\)](#) ([zfit.pdf.SimpleFunctorPDF method](#)), 584
[update_integration_options\(\)](#) ([zfit.pdf.SimplePDF method](#)), 576
[update_integration_options\(\)](#) ([zfit.pdf.SumPDF method](#)), 560
[update_integration_options\(\)](#) ([zfit.pdf.TruncatedGauss method](#)), 486
[update_integration_options\(\)](#) ([zfit.pdf.Uniform method](#)), 478
[update_integration_options\(\)](#) ([zfit.pdf.WrapDistribution method](#)), 494
[update_integration_options\(\)](#) ([zfit.pdf.ZPDF method](#)), 568
[upper](#) ([zfit.core.interfaces.ZfitSpace attribute](#)), 97
[upper](#) ([zfit.core.limits.Space attribute](#)), 102
[upper](#) ([zfit.core.sample.EventSpace attribute](#)), 146
[upper](#) ([zfit.Space attribute](#)), 45
[upper_limit](#) ([zfit.core.parameter.Parameter attribute](#)), 133
[upper_limit](#) ([zfit.Parameter attribute](#)), 39

V

[value\(\)](#) ([zfit.ComplexParameter method](#)), 42
[value\(\)](#) ([zfit.ComposedParameter method](#)), 40
[value\(\)](#) ([zfit.constraint.GaussianConstraint method](#)), 385
[value\(\)](#) ([zfit.constraint.SimpleConstraint method](#)), 384
[value\(\)](#) ([zfit.core.constraint.BaseConstraint method](#)), 70
[value\(\)](#) ([zfit.core.constraint.DistributionConstraint method](#)), 71
[value\(\)](#) ([zfit.core.constraint.GaussianConstraint method](#)), 73
[value\(\)](#) ([zfit.core.constraint.SimpleConstraint method](#)), 74
[value\(\)](#) ([zfit.core.data.Data method](#)), 77
[value\(\)](#) ([zfit.core.data.LightDataset method](#)), 77
[value\(\)](#) ([zfit.core.data.SampleData method](#)), 81
[value\(\)](#) ([zfit.core.data.Sampler method](#)), 84

[value\(\)](#) ([zfit.core.integration.PartialIntegralSampleData method](#)), 88
[value\(\)](#) ([zfit.core.interfaces.ZfitData method](#)), 89
[value\(\)](#) ([zfit.core.interfaces.ZfitLoss method](#)), 92
[value\(\)](#) ([zfit.core.interfaces.ZfitParameter method](#)), 97
[value\(\)](#) ([zfit.core.loss.BaseLoss method](#)), 105
[value\(\)](#) ([zfit.core.loss.CachedLoss method](#)), 106
[value\(\)](#) ([zfit.core.loss.ExtendedUnbinnedNLL method](#)), 107
[value\(\)](#) ([zfit.core.loss.SimpleLoss method](#)), 108
[value\(\)](#) ([zfit.core.loss.UnbinnedNLL method](#)), 109
[value\(\)](#) ([zfit.core.parameter.BaseComposedParameter method](#)), 112
[value\(\)](#) ([zfit.core.parameter.BaseParameter method](#)), 112
[value\(\)](#) ([zfit.core.parameter.BaseZParameter method](#)), 113
[value\(\)](#) ([zfit.core.parameter.ComplexParameter method](#)), 114
[value\(\)](#) ([zfit.core.parameter.ComposedParameter method](#)), 115
[value\(\)](#) ([zfit.core.parameter.ComposedResourceVariable method](#)), 123
[value\(\)](#) ([zfit.core.parameter.ComposedVariable method](#)), 123
[value\(\)](#) ([zfit.core.parameter.ConstantParameter method](#)), 124
[value\(\)](#) ([zfit.core.parameter.Parameter method](#)), 133
[value\(\)](#) ([zfit.core.parameter.TFBaseVariable method](#)), 141
[value\(\)](#) ([zfit.core.parameter.ZfitBaseVariable method](#)), 141
[value\(\)](#) ([zfit.data.Data method](#)), 388
[value\(\)](#) ([zfit.loss.BaseLoss method](#)), 416
[value\(\)](#) ([zfit.loss.ExtendedUnbinnedNLL method](#)), 413
[value\(\)](#) ([zfit.loss.SimpleLoss method](#)), 417
[value\(\)](#) ([zfit.loss.UnbinnedNLL method](#)), 414
[value\(\)](#) ([zfit.param.ConstantParameter method](#)), 421
[value\(\)](#) ([zfit.Parameter method](#)), 39
[values\(\)](#) ([zfit.util.container.DotDict method](#)), 372

W

[weights](#) ([zfit.core.data.Data attribute](#)), 77
[weights](#) ([zfit.core.data.SampleData attribute](#)), 81
[weights](#) ([zfit.core.data.Sampler attribute](#)), 84
[weights](#) ([zfit.core.integration.PartialIntegralSampleData attribute](#)), 88
[weights](#) ([zfit.core.interfaces.ZfitData attribute](#)), 89
[weights](#) ([zfit.data.Data attribute](#)), 388
[WeightsNotImplementedError](#), 377
[with_autofill_axes\(\)](#) ([zfit.core.interfaces.ZfitSpace method](#)), 97
[with_autofill_axes\(\)](#) ([zfit.core.limits.Space method](#)), 102

`with_autofill_axes()` (`zfit.core.sample.EventSpace` method), 146
`with_autofill_axes()` (`zfit.Space` method), 45
`with_axes()` (`zfit.core.interfaces.ZfitSpace` method), 97
`with_axes()` (`zfit.core.limits.Space` method), 102
`with_axes()` (`zfit.core.sample.EventSpace` method), 146
`with_axes()` (`zfit.Space` method), 45
`with_limits()` (`zfit.core.interfaces.ZfitSpace` method), 98
`with_limits()` (`zfit.core.limits.Space` method), 102
`with_limits()` (`zfit.core.sample.EventSpace` method), 146
`with_limits()` (`zfit.Space` method), 46
`with_obs()` (`zfit.core.interfaces.ZfitSpace` method), 98
`with_obs()` (`zfit.core.limits.Space` method), 103
`with_obs()` (`zfit.core.sample.EventSpace` method), 147
`with_obs()` (`zfit.Space` method), 46
`with_obs_axes()` (`zfit.core.limits.Space` method), 103
`with_obs_axes()` (`zfit.core.sample.EventSpace` method), 147
`with_obs_axes()` (`zfit.Space` method), 46
`with_traceback()` (`zfit.minimizers.baseminimizer.FailMinimizeNaN` method), 150
`with_traceback()` (`zfit.util.exception.AlreadyExtendedPDFError` method), 373
`with_traceback()` (`zfit.util.exception.AxesNotSpecifiedError` method), 373
`with_traceback()` (`zfit.util.exception.AxesNotUnambiguousError` method), 373
`with_traceback()` (`zfit.util.exception.BasePDFSubclassingError` method), 373
`with_traceback()` (`zfit.util.exception.ConversionError` method), 373
`with_traceback()` (`zfit.util.exception.DueToLazynessNotImplementedError` method), 374
`with_traceback()` (`zfit.util.exception.ExtendedPDFError` method), 374
`with_traceback()` (`zfit.util.exception.IncompatibleError` method), 374
`with_traceback()` (`zfit.util.exception.IntentionNotUnambiguousError` method), 374
`with_traceback()` (`zfit.util.exception.LimitsIncompatibleError` method), 374
`with_traceback()` (`zfit.util.exception.LimitsNotSpecifiedError` method), 374
`with_traceback()` (`zfit.util.exception.LimitsOverdefinedError` method), 374
`with_traceback()` (`zfit.util.exception.LimitsUnderdefinedError` method), 374
`with_traceback()` (`zfit.util.exception.LogicalUndefinedOperationError` method), 374
`with_traceback()` (`zfit.util.exception.ModelIncompatibleError` method), 375
`with_traceback()` (`zfit.util.exception.MultipleLimitsNotImplementedError` method), 375
`with_traceback()` (`zfit.util.exception.NameAlreadyTakenError` method), 375
`with_traceback()` (`zfit.util.exception.NormRangeNotImplementedError` method), 375
`with_traceback()` (`zfit.util.exception.NormRangeNotSpecifiedError` method), 375
`with_traceback()` (`zfit.util.exception.NoSessionSpecifiedError` method), 375
`with_traceback()` (`zfit.util.exception.NotExtendedPDFError` method), 375
`with_traceback()` (`zfit.util.exception.NotMinimizedError` method), 376
`with_traceback()` (`zfit.util.exception.NotSpecifiedError` method), 376
`with_traceback()` (`zfit.util.exception.ObsIncompatibleError` method), 376
`with_traceback()` (`zfit.util.exception.ObsNotSpecifiedError` method), 376
`with_traceback()` (`zfit.util.exception.OverdefinedError` method), 376
`with_traceback()` (`zfit.util.exception.PDFCompatibilityError` method), 376
`with_traceback()` (`zfit.util.exception.ShapeIncompatibleError` method), 376
`with_traceback()` (`zfit.util.exception.SpaceIncompatibleError` method), 376
`with_traceback()` (`zfit.util.exception.SubclassingError` method), 376
`with_traceback()` (`zfit.util.exception.UnderdefinedError` method), 377
`with_traceback()` (`zfit.util.exception.WeightsNotImplementedError` method), 377
`WrapDistribution` (class in `zfit.pdf`), 486
`WrapOptimizer` (class in `zfit.minimize`), 417
`WrapOptimizer` (class in `zfit.minimizers.base_tf`), 149
`zfit` (module), 31
`zfit.constraint` (module), 382
`zfit.core` (module), 47
`zfit.core.basefunc` (module), 47
`zfit.core.basemodel` (module), 53
`zfit.core.baseobject` (module), 59
`zfit.core.basepdf` (module), 60
`zfit.core.constraint` (module), 69
`zfit.core.data` (module), 74
`zfit.core.dimension` (module), 84

zfit.core.integration (*module*), 86
zfit.core.interfaces (*module*), 89
zfit.core.limits (*module*), 98
zfit.core.loss (*module*), 104
zfit.core.math (*module*), 109
zfit.core.operations (*module*), 109
zfit.core.parameter (*module*), 110
zfit.core.sample (*module*), 143
zfit.core.testing (*module*), 148
zfit.data (*module*), 385
zfit.func (*module*), 388
zfit.loss (*module*), 413
zfit.minimize (*module*), 417
zfit.minimizers (*module*), 149
zfit.minimizers.base_tf (*module*), 149
zfit.minimizers.baseminimizer (*module*), 149
zfit.minimizers.fitresult (*module*), 151
zfit.minimizers.interface (*module*), 153
zfit.minimizers.minimizer_minuit (*module*), 154
zfit.minimizers.minimizer_tfp (*module*), 155
zfit.minimizers.minimizers_scipy (*module*), 156
zfit.minimizers.optimizers_tf (*module*), 156
zfit.minimizers.tf_external_optimizer (*module*), 157
zfit.models (*module*), 160
zfit.models.basefunctor (*module*), 160
zfit.models.basic (*module*), 166
zfit.models.dist_tfp (*module*), 182
zfit.models.functions (*module*), 223
zfit.models.functor (*module*), 253
zfit.models.physics (*module*), 277
zfit.models.polynomials (*module*), 294
zfit.models.special (*module*), 346
zfit.param (*module*), 420
zfit.pdf (*module*), 421
zfit.sample (*module*), 584
zfit.settings (*module*), 584
zfit.util (*module*), 370
zfit.util.cache (*module*), 370
zfit.util.checks (*module*), 371
zfit.util.container (*module*), 372
zfit.util.diverse (*module*), 373
zfit.util.exception (*module*), 373
zfit.util.execution (*module*), 377
zfit.util.graph (*module*), 378
zfit.util.logging (*module*), 378
zfit.util.temporary (*module*), 379
zfit.util.ztyping (*module*), 380
zfit.ztf (*module*), 380
zfit.ztf.random (*module*), 380
zfit.ztf.tools (*module*), 380
zfit.ztf.wrapping_tf (*module*), 380
zfit.ztf.zextension (*module*), 381
ZfitBaseVariable (*class in* zfit.core.parameter), 141
ZfitCachable (*class in* zfit.util.cache), 371
ZfitData (*class in* zfit.core.interfaces), 89
ZfitDependentsMixin (*class in* zfit.core.interfaces), 89
ZfitDimensional (*class in* zfit.core.interfaces), 90
ZfitFunc (*class in* zfit.core.interfaces), 90
ZfitFunctorMixin (*class in* zfit.core.interfaces), 92
ZfitLoss (*class in* zfit.core.interfaces), 92
ZfitMinimizer (*class in* zfit.minimizers.interface), 153
ZfitModel (*class in* zfit.core.interfaces), 92
ZfitNumeric (*class in* zfit.core.interfaces), 94
ZfitObject (*class in* zfit.core.interfaces), 94
ZfitParameter (*class in* zfit.core.interfaces), 96
ZfitParameterMixin (*class in* zfit.core.parameter), 141
ZfitPDF (*class in* zfit.core.interfaces), 94
ZfitResult (*class in* zfit.minimizers.interface), 153
ZfitSpace (*class in* zfit.core.interfaces), 97
ZfitStrategy (*class in* zfit.minimizers.baseminimizer), 150
ZFunc (*class in* zfit.models.functions), 247
ZPDF (*class in* zfit.models.special), 362
ZPDF (*class in* zfit.pdf), 560