zfit Documentation

Release 0.3.3

zfit

May 15, 2019

Contents

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

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

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

CHAPTER 1

Getting started with zfit

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> import numpy as np
>>> mu_true = 0
>>> sigma_true = 1
>>> data_np = np.random.normal(mu_true, sigma_true, size=10000)
>>> data = zfit.data.Data.from_numpy(obs=obs, array=data_np)
>>> print(data)
<zfit.core.data.Data object at 0x7f90537f4748>
```

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.MinuitMinimizer()

>>> # 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 `MinuitMinimizer` 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: ^{+0.009981041418415555}_{-0.009981515893414316}
sigma: ^{+0.007099472590970696}_{-0.007016265476493973 4}
```

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

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

(continues on next page)

(continued from previous page)

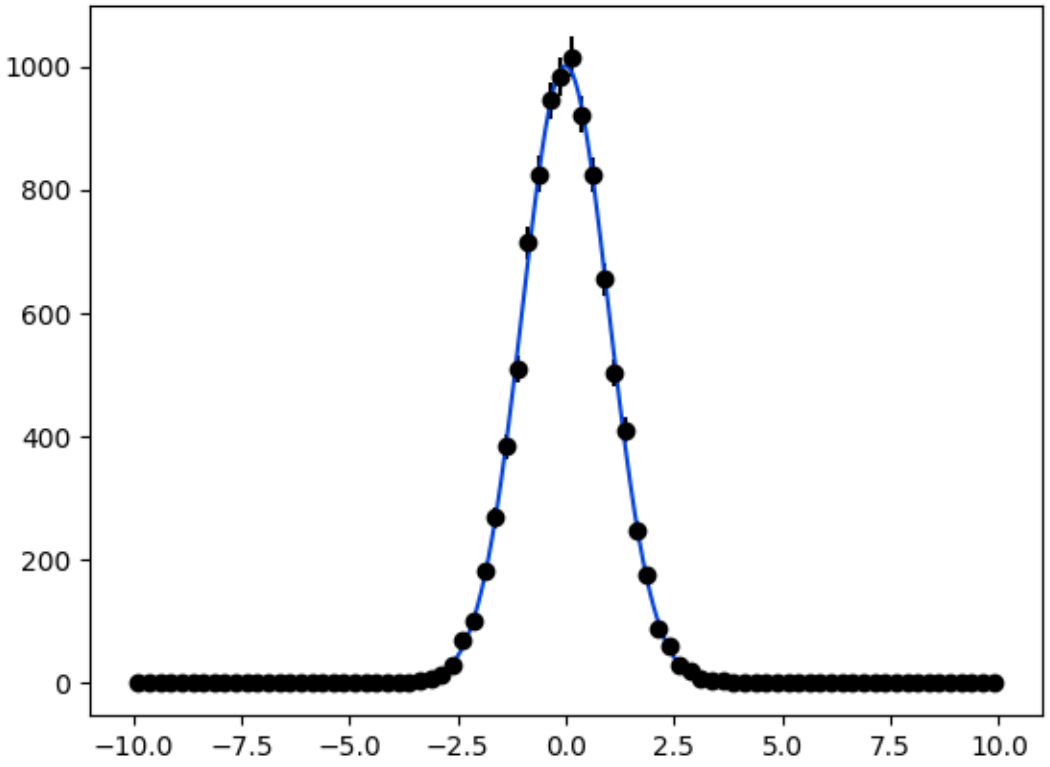
```
'has_made_posdef_covar': False, 'hesse_failed': False, 'has_covariance': True, 'is_
↪above_max_edm': False, 'has_reached_call_limit': False}}
```

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(ztf.constant(consts), norm_range=(-np.infty, np.infty))
```

(continues on next page)

(continued from previous page)

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

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

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

CHAPTER 2

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
- pandas, temporarily
- numpy
- uproot
- iminuit
- typing
- pep487
- 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 is 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

- 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.

CHAPTER 3

Contributing

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

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

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

3.1 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.

CHAPTER 4

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 <-> 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.

CHAPTER 5

Parameter

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

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

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])
```

`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. Parameters are therefore floating by default, but their value can be fixed by setting the attribute `floating` to `False`.

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
>>> 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_auto()  # 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.

CHAPTER 6

Building a model

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

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

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

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', 'SimpleFunctorPDF']
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.MinuitMinimizer()

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

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

> values when the sampler was created.

To give another, though not very useful example:

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

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

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

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

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

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

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

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

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

CHAPTER 7

Data

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

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

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.Data.from_root(root_file,
...                                 root_tree,
...                                 branches)
```

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

CHAPTER 8

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 `tf.Tensor` object 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 adding a gaussian constraint on the `mu` parameter of the previous model, we would write:

```
>>> my_loss = zfit.loss.UnbinnedNLL(model_cb,
>>>                                 data,
>>>                                 fit_range=(-10, 10),
>>>                                 constraints=zfit.constraint.nll_
↪gaussian(params=mu,
```

(continues on next page)

(continued from previous page)

```
>>>                                                                    mu=5279.,
>>>                                                                    sigma=10.
↪))
```

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 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 and fit ranges:

```
>>> # Adding a list of models, data and observable ranges
>>> my_loss_sim = zfit.loss.UnbinnedNLL(model=[models], data=[datasets], fit_
↪range=[obsRange])
```

CHAPTER 9

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.MinuitMinimizer()
>>> # Scipy minimizer
>>> minimizer_scipy = zfit.minimize.ScipyMinimizer()
>>> # Adam's Tensorflow minimizer
>>> minimizer_adam = zfit.minimize.AdamMinimizer()
```

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
```

CHAPTER 10

zfit API documentation

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