July 3, 2019

zfit: scalable model fitting in Python using TensorFlow

Master thesis of
Jonas Eschle

Supervised by
Prof. Nicola Serra
Dr. Albert Puig Navarro

Abstract

Fitting a model to data is an essential part in most High Energy Physics analyses. Several frameworks to perform this action exist in C++, but no powerful enough counterpart exists in Python, a language recently becoming more and more popular for analyses. With the recent success of deep learning, frameworks in Python such as TensorFlow came up, offering a high level interface for efficient, parallelised computing on modern architectures. In this thesis, `zfit`, a library for model fitting in HEP implemented in pure Python and based on top of TensorFlow, is presented. It offers a well structured model fitting workflow allowing to build composite models from a variety of shapes. A high level of customisation is possible due to well specified interfaces and convenient base classes allowing to easily replace any part in the workflow with a custom implementation. Together with the flexibility and scalability of TensorFlow, `zfit` extends its functionality well beyond what current model fitting libraries offer. An overview over the current status of model fitting libraries and the HEP requirements will be discussed followed by the structure of `zfit` and its implementation. Finally, examples which quantify the performance and demonstrate the feasibility of `zfit` for a whole range of real world applications are shown and an additional library for phasespace generation, `phasespace`, is introduced.

Contents

1 Introduction

The Standard Model (SM) of Particle Physics describes the most fundamental particles in the universe and their interactions. According to it, all matter is made up of fermions: quarks and leptons. They appear in different flavours and generations as depicted in Fig. 1. There are additionally four gauge bosons that allow the particles to interact via their exchange: the photon is the electromagnetic force carrier and couples to particles with an electric charge, such as the electron and the quarks. W and Z bosons are the carrier of the weak force and couple to all fermions. Unlike other forces, the strength of the strung interaction increases with the distance of two particles. As a consequence, quarks do not appear alone in nature. Instead, they form composite particles existing of multiple quarks, the hadrons. While there are hundreds of hadrons, nearly all of them have lifetimes under nano seconds and decay to lighter particles. The only stable particles are the well known neutron, proton, electron and neutrino, which together make up the visible matter in our universe. Finally, all particles in the SM (except neutrinos) acquire mass through their interaction with the Higgs Field by the exchange of Higgs Bosons.

With the recent discovery of the Higgs boson, the last missing piece of the SM has been found. It provides a complete description of nearly all observations. And yet it does not seem to be the final answer since there are phenomena that remain unexplained, such as the existence of dark matter that interacts gravitationally and significantly determines the dynamics of galaxies does not appear in the SM, and the fact that neutrinos have mass since they oscillate does not coincide with the predictions of the SM. With larger amounts of data collected, more precise measurements need to be made in order to look for further inconsistencies of the SM that can guide us to a new theory.

In the scientific context, what is called an observations is in fact an answer extracted from nature by asking the right question and using statistics to analyse the response of the data. A question in this sense is an experimental setup and a scientific hypothesis is a proposed explanation for an observed phenomena which can be tested. Different methods can be used to verify a hypothesis, but all of them make use of a test statistic that needs a single value to quantify their agreement with the observations. Given strong enough evidence, the null hypothesis may be rejected in favour of an alternate hypothesis. As an example, this can be used for the discovery of a new particle where the background only hypothesis acts as the null and the alternate is the background and signal combined.

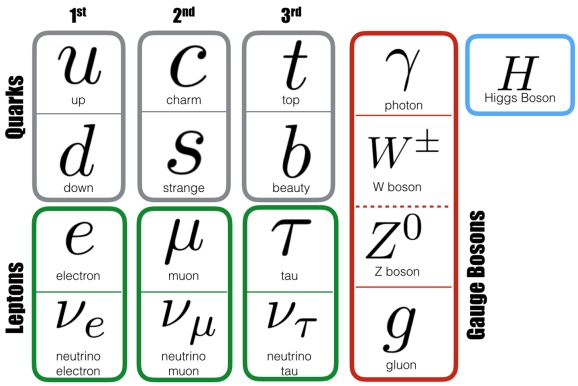


Figure 1: The particles of the SM.

As already mentioned, observations from experiments are needed. To study the fundamental particles of the SM, high enough energies are required to produce them. High Energy Physics (HEP) experiments all over the world accelerate light particles such as electrons or protons and let them collide. The Large Hadron Collider (LHC) at CERN accelerates protons to energies up to 6.5 TeV[1], which is currently the frontier in high energies. Around the collider, there are four large experiments: the general purpose detectors ATLAS and CMS, ALICE, an experiment specialized on lead interactions and LHCb, a detector focused on the study of heavy flavour decays. These experiments are situated at collision points around the LHC where 40 million collisions occur per second. Due to the high concentration of energy on collision, heavier particles are created and decay immediately to lighter particles. The experiments measure the tracks and properties of the decay products that pass through the different detectors. The raw readout of those is forwarded to a computer farm, where events of interest are marked and kept whereby the rest of the events are not recorded. This reduces the stream of data to a frequency that allows it to be stored on persistent storage. From there it can be retrieved and used for further, offline analysis.

Performing a full analysis to measure physics observables from the data involves several steps. This includes among others cleaning the samples by applying selection criteria, reweighting to correct for systematic effects or creating new features that better describe the event. The sample can then be used to directly infer unknown parameters by using physically motivated models and performing a fit to the data. All of these analysis steps require convenient, reliable and fast libraries together with enough computing resources. To accomplish this, a lot of code is being written and stacked upon each other. To cope with the ever increasing amounts of data, both in real-time event filtering as well as offline analysis, it is mandatory to keep the computing level all in all at the state of the art.

Computing is still a comparably young, fast moving field. Hundreds of general programming languages exist, most of them do not stay for a long time or only exist in a specialist community. Even longer lived languages have both advantages as well as shortcomings. This leads to different fields adopting a few, or even one, main languages that specifically well serve their purpose. The field of scientific computing mainly involves either simulation of systems or data analysis. In both cases languages that are fast to do number crunching are required. Among the most popular languages for heavy computations are Fortran and C/C++. The former is over six decades old and still used up to these days. It was designed for numerical processing and contains optimizations that still outperform other languages. C and its superset C++, the most popular language in HEP, date back about four decades and are built for more general usage than Fortran is. Although being fast and a powerful general programming language, it does not offer the convenient abstractions that scripting languages offer. It allows but also requires to manually handle certain resources, such as memory allocation, and is limited in terms of flexibility because it is a statically compiled language. While the latter feature allows for high performant execution, interpreted scripting languages such as Python offer additional comfort and flexibility. While the execution of actual Python code can be significantly slower than comparable static compiled languages when it comes to pure number crunching, the huge Python package ecosystem offers a lot of libraries that implement time consuming mathematical operations in a more efficient language, such as Fortran or C++. This

[1]Natural units with $\hbar = c = 1$ are used throughout.

makes Python a higher level library that abstracts away the handling of computation demanding operations through external function calls. Together with an especially clean syntax style, Python code is expressive and natural to read, giving in only a small penalty for the performance from the overhead of the external calls.

This combination and a fast growing open source-community has established Python as the most popular language for data analysis. With the recent advances in Machine Learning and the rising popularity of Big Data analysis among industry leaders, the size and quality of the scientific Python ecosystem has made a huge leap forward. Topics like deep learning, which require highly optimised code due to the abundance of vectorised matrix multiplications, have lead to the appearance of frameworks designed for this kind of massive, parallel computations and supported by large companies such as Google's TensorFlow [1] (TF) or Facebook's PyTorch [2]. With large economical interests coming into play, these frameworks also focus on the efficient use of specialized hardware, such as Graphical Processing Units (GPU) which are by design optimized for vectorized computations and therefore fit the need of the Deep Learning community. These frameworks are optimized both in terms of performance as well as in ease of use, dealing with the burden of incorporating the parallelization.

Next to all these developments there is also a trend inside the HEP community to move towards a more Python-oriented software stack. In recent surveys [3], its usage surpasses C++ in collaborations such as CMS. The existence of the scientific Python ecosystem offers the possibility of sharing some of the effort with the data analysis industry and open-source community, allowing to perform a significant number of the analysis steps in HEP within Python out of the box. This leaves to the HEP community only the burden of developing the field-specific tools required to fit into the ecosystem. While some of the existing frameworks in C++ offer Python bindings, they are usually not well integrated with the Python language and the whole ecosystem. Several model fitting libraries in pure Python have already been developed in order to fill parts of this gap, though none of them offers the complete feature set that would be desired for HEP analysis and are hard to extend. Therefore, while large advances have been made on this front, a viable alternative to the existing, mature model fitting libraries in C++ is still missing. Nonetheless, some of these libraries have proved the feasibility of using deep learning frameworks as computing backends for model fitting.

Summarizing, HEP has

- the need for scalable, flexible model fitting;
- a strong movement towards Python with its huge data analysis ecosystem;
- the lack of a sufficiently strong model fitting library in pure Python.

Furthermore, modern high performance computing frameworks from deep learning arose and their feasibility for computational backends in model fitting was demonstrated in several projects.

With these ideas in mind, the `zfit` package has been developed with the goal to provide this need by creating a pure Python based library built on a deep learning framework. This requires the formalisation of the fitting procedure, the establishment of a stable API and the usage of current knowledge from similar libraries. The following Section 2 will

expand on model fitting in HEP incensing a discussion on already existing libraries. With this knowledge, the usage of `zfit` and its basic concepts, the formalisation of the model fitting workflow, and the choice of the backend and its capabilities are outlined in Sec. 3. The individual components of `zfit` will be discussed in more detail in Sec. 4. Afterwards, the performance and scalability is evaluated with examples in Sec. 5. The extension of `zfit` to more than the default model fitting is discussed in Sec. 6 by using its capabilities to implement an amplitude fit. Lastly a brief overview of the future plans of the `zfit` library and its ecosystem are given in Sec. 7.

2 Model fitting

In HEP, observations can be quantified by mathematical models which originate from hypotheses or theories and make assumptions about the underlying behaviour of nature. Often, these models have free parameters that we want to measure. A single model may describes only parts of the observations and combinations and compositions of models may be needed to build a model that describes the full data sample. Creating these models in a convenient and correct way and finding the values of the parameters to maximise the agreement with respect to the data is what "model fitting" refers to.

2.1 Maximum Likelihood

At the very heart of model fitting is the need to quantify the agreement, or rather the disagreement, of a model with the data. This function of the parameters and data is known as the loss. It is the very definition of the problem and mathematically fully defines the solution. In HEP analysis losses are mostly based on the likelihood of the model under the data, whereby the model typically depends on free parameters. In the following, an introduction to the method of maximum likelihood is given. A more detailed explanation and derivation can be found in Appendix A.

A likelihood can be defined by the following: given a model parametrised by θ and a dataset x, the likelihood describes the odds that an event happened under θ

$$\mathcal{L}(\theta) = P(x|\theta). \tag{1}$$

The likelihood as shown in Eq. 1 is the quantity to be maximised in order to achieve the maximal $P(\theta|x)$. To build this likelihood, we need the model $f_\theta(x)$ to be a probability density function (PDF), i.e. it's normalised to 1. Especially in HEP, it is often the case that the PDF is zero outside of certain boundaries, for example because points outside a specified domain are removed, in which case

$$\int_l^u f_\theta(x)\mathrm{d}x = 1, \tag{2}$$

where l and u define the lower and upper boundaries of the domain, respectively. This also extends to higher dimensions. It follows directly that any function $g_\theta(x)$[2] can be normalised and therefore used as a PDF $f_\theta(x)$

$$f_\theta(x) = \frac{g_\theta(x)}{\int_l^u g_\theta(x)\mathrm{d}x}. \tag{3}$$

A likelihood can be a product of likelihoods of independent events

$$\mathcal{L} = \prod_i \mathcal{L}_i, \tag{4}$$

and therefore the likelihood of dataset x can be written as the joint probability of each event

$$\mathcal{L}(x|\theta) = \prod_i f_\theta(x_i),$$

[2]This is about the small subset of modelling functions in physics *without* pretending mathematical correctness in a general way. This includes functions $f : \mathbb{R}^n \rightarrowtail \mathbb{R}$ that are positive, l^1 and (piecewise) C^1.

with x_i a single event from the dataset x.

The calculation of $\mathcal{L}(x|\theta)$ involves the product of many small numbers, which is not possible to perform using a normal computer given its limited precision. To solve this issue, a log transformation can be applied. In addition, the log-likelihood is usually negated, thus changing the target of finding the maximum to finding a minimum and ending up with a negative log likelihood (NLL).

A maximum likelihood estimate using the transformation above is therefore given by finding the minimum of the NLL

$$NLL = -\sum_i \ln(f(\theta|x_i)) \tag{5}$$

This maximises therefore the agreement between data and model, i.e. the *probability of the model given the data*.

As seen in Eq. 4, the combination of likelihoods is quite versatile and not only limited to a model shape matching the data shape. Often, a combination of several of the following likelihoods is built

Simultaneous Multiple models can share parameters. To fit them simultaneously to different datasets, their likelihoods can be combined (summed).

Extended While a PDF is normalised, we can add an absolute scale as an additional term to the likelihood to reflect the number of events contained in this model. Given the data, we know the number of events and can add a Poisson term to account for them.

Prior For some parameters, a prior distribution is known. This describes the knowledge obtained from other measurements and influences the likelihood if the parameters spread is in the same order of magnitude as the sensitivity of the fit to the parameter. A prior, or constraint, is a probability depending directly on the parameter value and can also be added to the likelihood.

Regardless of the complexity of the model, we end up with a single number, the loss, that can be used to compare the agreement between different models or parametrizations and the data. When fitting a model, the loss is minimised by adjusting the parameters. While the absolute value of the loss is usually not important, the ratio of losses from different models can often be useful in further statistic tests.

2.2 Requirements

Some features are crucial in order to implement a model fitting library. An important part of model fitting is the model building itself, but a library should also offer a convenient, transparent creation of the loss and the minimisation. Especially in HEP, the following features are essential:

- PDFs are by definition normalised over a certain range. In most other libraries and fields, the domain is assumed to be $(-\infty, \infty)$. In HEP, this is basically never the case and a finite normalisation range is used.

- Fits in HEP are often more than one-dimensional. The framework should therefore naturally extend to higher dimensions.
- Building and combining models from basic shapes like Gaussian or exponential functions only suffices for simpler cases, but this is often not enough to build more complicated or specific models.Therefore, a convenient way to implement custom models has to be provided.
- Reasonable scaling with the data size and the model complexity is a key criteria. This is often especially hard to achieve in combination with the ability of specifying custom models, since the latter usually requires to have the parallelization implemented by the user.
- While the minimisation of the loss yields an optimal value for each parameter, it is crucial in HEP to also know the uncertainty of the value. This requires the library to have a transparent way of handling the parameters and their uncertainties as well as to provide the flexibility to perform an advanced statistics treatment.

2.3 Existing libraries

Model fitting itself is nothing new. In fact there are already a lot of model fitting libraries available. Some of these libraries are also written in Python and cover a similar scope as `zfit`. Building a new fitting library from scratch sounds therefore like reinventing the wheel and should be avoided if not necessary. But as already discussed in Sec. 1, fundamental changes in the computing architecture are leading to vectorized paradigms. Additionally, the needs in HEP for larger and more complicated while still flexible fits require to keep up with the state-of-the-art in computing. And this sometimes requires a re-invention.

However, it is an imperative to make sure that no existing library already fulfils the needs or can be extended to. And even if concluding that a new library is the way to go, as much as possible should be learned and taken from any existing library in order to reinvent as few as necessary. In the following an overview of already existing libraries is given.

2.3.1 General fitting

Fitting models to data is a task that is performed in a variety of fields independent of HEP. Different general fitting libraries exist in Python, but they often contain functionality not actually needed in HEP, such as mean, variance, survival function, and lack central features like a custom normalisation range or the extension to more than one dimension.

- Scipy [4] is the go-to library for scientific calculations in Python and provides an extensive toolbox for statistical and numerical methods. There is a module with distributions that have proven to be stable and work well. Downsides of the package include a non-optimized implementation in terms of parallelisation and lack of support for composite models.
- lmfit [5] shares a lot of its design in terms of naming and concept to `zfit`. It is built for model fitting, has parameters, minimisers, fit results and more. It lacks

more advanced features like the possibility of normalisation ranges for PDFs or good scalability, since it is built on top of numpy, a fast numerical library in Python, and scipy, which strongly limits the ability for massive parallelisation.

- TensorFlow Probability [6] provides a library for statistical reasoning. Its focus is on analytical functions and only marginally extends to numerical and Monte Carlo methods, which limits its application to analytically integrable functions. Interestingly, it contains a lot of of features that can be used inside or together with `zfit`, such as Bayesian inference with MCMC sampler and analytic functions with integrals already implemented in TF.

2.3.2 HEP specific

A wide range of specialised fitters exist in HEP. The overview here is limited to general purpose fitters which can be used from Python.

RooFit [7] is the de-facto standard tool for fitting in HEP. Models are built using classes and provide automatic normalisation and integration. There is support for binned as well as unbinned fits. RooFit itself extends beyond that and offers also an extensive plotting and statistics module. While the library has proven itself in numerous analyses over the years, and the model building part of `zfit` is actually inspired by the core of RooFit, there are several shortcomings which are meant to be addressed with `zfit`:

- RooFit is not a native Python library but can only be accessed through the Python bindings to Root. Since RooFit manages its own memory in C++ and Python uses a garbage collection as well, this can lead to memory leaks and completely undefined behaviour.
- Since the Python interface is barely a wrapper around the C++ classes, it does not integrate well to the scientific Python stack.
- In terms of flexibility, RooFit offers the possibility to be extended up to a certain degree with custom classes in pure C++. But especially when used from Python, it does not provide a convenient way to define custom PDFs.
- While there are improvements in the pipeline, it is not natively optimized to run vectorized on multiple cores or even accelerators like GPUs.
- Since the usage requires Root, the installation and setup is typically not lightweight.

probfit [8] is a fitting library written in pure Python that mainly uses Cython to perform the heavy computations. This is a limitation in terms of performance and custom PDF implementations that makes a possible extension hard. Since it does provide limited features only, a large extension would be needed together with a major conceptual overhaul to be able to include new features.

pyhf [9] is a re-implementation of HistFactory from ROOT in Python. It makes use of TensorFlow and other libraries including PyTorch and Numpy as a backend. It is purely designed to do binned template fits and does not extend its functionality beyond that point.

The CMS Combine Tool [10] contains a subpart that implements template fits in TF. It does not extend its functionality further and is currently not available as a standalone package. Several useful parts like likelihood profiling or a minimiser in pure TF have been implemented there.

TensorFlow Analysis [11] is a library with a simple, functional approach to built the loss with TF and use Minuit [12] directly inside[3] to find the minimum. It offers a lot of physics content to create a model. While the lightweight approach comes with a lot of flexibility, the library also leaves quite some work to the user. For example it does not offer anything close to model composition with automatic normalisation. Notably, in its current state, the library lacks Python 3 support. However, its importance has to be stressed since it demonstrated the feasibility of using TF for unbinned likelihood fits with complex models and was a major inspiration for the development of `zfit`.

TensorProb is a model fitting library in Python that uses TF as the backend. In general it was built with a similar goal in mind as `zfit`, providing a model fitting library in Python using TF, but using more an experimental approach. It offers models that can also integrate and sample. The content is based on older TF versions and the library is strongly limited in functionality. Most importantly though, the project never grew out of its experimental status and has been discontinued. It recommends now to use `zfit` instead.

While the discussed model fitting libraries have different strengths and weaknesses, no single one fully fulfil the needs of HEP. However it is worth pointing out that their demonstration of concepts, designs and even certain functionality that can be used directly with `zfit` are essential pieces in the development of `zfit`.

[3] This also requires to have the ROOT package installed.

3 zfit introduction

zfit was created in order to fill the gap of a model fitting library in pure Python for HEP. We will now have a look at it, how it is structured and supposed to fill this gap. Model fitting as implemented in zfit is split into five essential parts. To introduce them and zfit itself, an example with a sum of a Gaussian and an exponential PDF will be implemented. This example can be thought of as a fit to an invariant mass distribution of a signal and a background component.

Let's assume we are interested in an observable x within a range from 5 to 10. In zfit this is expressed with a Space defining our domain

```
limits = zfit.Space(obs="x", limits=(5, 10))
```

zfit can handle data from a variety of different sources. In this case, we load the data data_np from a numpy array

```
data = zfit.Data.from_numpy(array=data_np, obs=limits)
```

Since the data was specified with the limits as its observables, it will be cut automatically to be only within the limits range. In this context, we can think of the observable x as of the column of the data frame.

Next the model needs to be built. We create the Gaussian PDF with two free parameters, mu and sigma. Using 7 and 1.5 as initial values, respectively, this is done as

```
mu = zfit.Parameter("mu", 7)
sigma = zfit.Parameter("sigma", 1.5)
```

Creating the Gaussian in the observable x and using the parameters from before as

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

Equivalently the exponential PDF is created. A fixed value of -0.1 is used for the exponential parameter λ as in $e^{\lambda x}$ and can directly be given to the PDF[4]

```
exponential = zfit.pdf.Exponential(obs=limits, lambda_=-0.1)
```

To build the sum, an additional free parameter is used to describe the fraction of the first PDF. It is initialised with 0.5 and limited between 0 and 1

```
frac = zfit.Parameter("fraction", 0.5, 0, 1)
model = zfit.pdf.SumPDF(pdfs=[gauss, exponential], fracs=frac)
```

Now that the model is built, we can define the loss by combining it with the data. Here an unbinned NLL will be used

[4] Alternatively, a Parameter with the argument floating set to False can be created.

```
nll = zfit.loss.UnbinnedNLL(model=model, data=data)
```

which needs to be minimised in order to find the optimal parameters. To achieve this goal, a minimiser such as Minuit is needed. Once it is created, we use its `minimize` method in order to minimise the previously built loss

```
minimizer = zfit.minimize.MinuitMinimizer()
result = minimizer.minimize(nll)
```

The outcome of this minimisation is stored in a `FitResult` object. Whether the convergence was successful can be checked with

```
has_converged = result.converged
```

The free parameters of the model are updated in-place with the values obtained from the minimisation. This implies that the shape of the model has changed now, since it depends on the parameters. While a parameter can change again, the `result` stores the values from the minimisation as immutable numbers. They can be accessed like[5]

```
mu_value = result.params[mu]["value"]
```

The value of the parameter is incomplete without an estimate of its uncertainty. For an accurate estimation, we can use `error`, an advanced method that takes all correlations among the parameters into account

```
errors = result.error()
```

This simple yet complete example demonstrates how model fitting in `zfit` works. All parts contain a lot more functionality than just seen, but the structure of the workflow as shown in Fig. 2 into five independent parts remains the same, no matter how complicated a fit may be.

Model building The construction of models is the core of `zfit` and involves Functions and PDFs. The difference between them is that the latter is normalised to one over a certain domain. Building the model includes a set of convenient base classes that allow to easily create a custom model as explained in Sec. 4.3. Furthermore, composed models involving sum, products and more are available.

Data Any kind of data needs to be loaded and transferred into a well defined `zfit` format. The `Data` class takes care of this and offers several formats to load from, which then can be used by models. The aim here is to provide a simple way of loading data from different formats into `zfit` and applying some cuts.

[5]Notice that `mu`, the parameter itself, and not `"mu"`, the name of it, is used as the key.

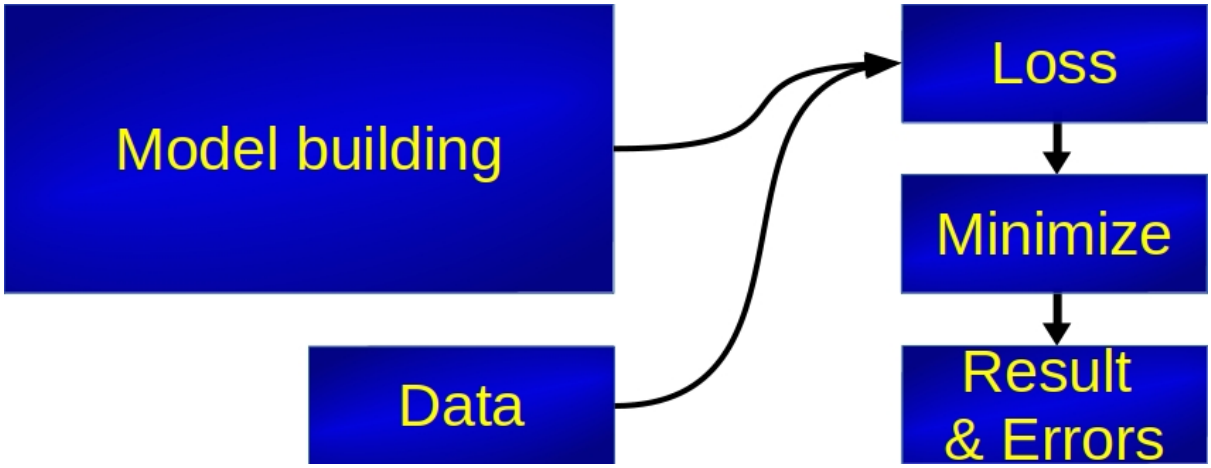


Figure 2: Fitting workflow in `zfit`. Model building is the largest part. Models combined with data can be used to create a loss. A minimiser finds the optimal values and returns them as a result. Estimations on the parameters uncertainties can then be made.

Loss This is the core definition of the problem. It uses the model and data objects to calculate a single number that quantifies the discrepancy from the model and the data. Typically, a binned or unbinned NLL or a χ^2 is used, but `zfit` offers the freedom to implement any desired loss that is not available in a straightforward way. From this step onward, it is irrelevant what data or models are *actually* used. Only the number and the gradients with respect to the models parameters matter.

Minimisation Given a loss, the minimiser minimises its value with respect to the free parameters of the models. In `zfit`, several algorithms are implemented by wrapping existing minimisation libraries.

Result and Errors After each minimisation, a `FitResult` object is created. It stores all the information about the minimisation process and allows ,amongst other things, to check if the convergence was successful. The result also includes the parameters and their values at the minimum. Furthermore, the loss and the minimiser itself are also stored in the result. Using both of them, an estimation on the uncertainty of the parameters can be made. For this purpose, some simple algorithms are provided by `zfit`, but any more sophisticated uncertainty estimation can be made by using the objects made available by the `FitResult`.

This formalisation is a powerful approach: the separation of the model fitting into these fine building blocks allows to improve and maintain the individual parts almost independently. Most importantly, it reveals a surprising similarity to the field of deep learning: apart from the last step, the workflow is *exactly* the same. Using a deep learning framework as the backend for a model fitting library therefore seems like an obvious choice to consider.

3.1 TensorFlow backend

Deep learning has recently gained a lot of attention as it has been quite successful as a tool in big data analysis and predictive statistics. In its core, the idea is to extract correlations

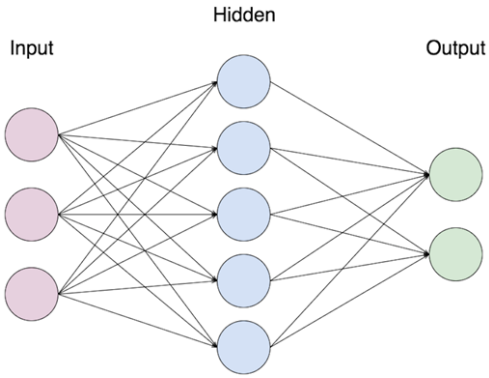


Figure 3: A schematic view of a DNN function. Input takes the data and has the dimension of an event. Each end-point of a line is multiplied by a weight, a free parameter, and added with the other lines. At a node, a non-linear function is then applied.

from data by training a neural network on them in order to make accurate predictions on unknown samples. This can be the classification of images, the prediction of stock markets etc. More interestingly, the typical deep learning workflow can be summarised also by Fig. 2 by replacing "model" with "neural network", [6] "minimisation" with "training" and removing the last block "Result & Error". Moreover, deep learning and HEP model fitting both use large data samples and build complex models. This similarity inspired the implementation of `zfit` with a deep learning framework as the backend.

While we have just shown how the two workflows look incredibly similar at first glance, there are some hidden, crucial differences. Knowing them is essential in order to understand the advantages but also limitations of this approach. In the following we will have a simplified at the core of deep learning and compare then to model fitting.

- A Deep Neural Network (DNN) is simply a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The input space is the number of observables of a single event. The output of the function, the prediction, is notably m dimensional. Contrary to this, a model with PDFs outputs into the one dimensional space of a normalised probability. DNN outputs, if used for classification, correspond to pseudo-probabilities. While they are not normalised, there exists a monotonic transformation to a probability.

  Consequence: normalisation over a certain range is specific to model fitting and no explicit tool, e.g. for numerical integration, exists in the deep learning frameworks.

- In model fitting, the composition of shapes is motivated by previous knowledge and an underlying theory is built. There is often a meaning behind each part and the shape of the model is restricted to a specific problem. This specific shape, coming from assumptions and previous knowledge, is what keeps the number of parameters low: typically, no more than a dozen for simple fits and a maximum of a few hundreds for the most complicated fits are used. Contrary to that, the structure of a DNN is basically agnostic to the problem and depends mostly on its complexity. It therefore incorporates a minimum of pre-knowledge and assumptions about the correlations in the data. This huge versatility is what makes deep learning such a

[6] Neural networks in deep learning are also called models. The term "model" will here solely be used for "classical" model fitting as in `zfit`.

successful field but comes at a price: a great number of free parameters is needed, starting at thousands for really simple DNNs, typically being around hundreds of thousands and going to tens of millions. DNNs are a structure consisting of layers with nodes as shown in Fig. 3. Each of the nodes adds an additional parameter for every incoming connection, resulting in a large number of parameters.

Consequence: While dozens of DNN building libraries like Keras[7], PyTorch and more offer great capabilities in building DNNs, their abstractions into layers are of no use to model fitting.

- A DNN is essentially matrix multiplications scaled by the free parameters, with an additional simple, non-linear activation function applied. In model fitting, the shape can have an arbitrary complexity and contain a whole range of elementary functions. Furthermore control flow elements and complex number are often used as well. Not only is the function itself more complicated, also the dependency of parameters can be highly non-trivial.

  There is an additional difference in the precision of the floating point operations. In model fitting, the precision required is higher, because the values of likelihoods and the changes are larger compared with neural networks often having values varying between -1 and 1. Also the Quasi-Newton methods as described below build an approximate second order derivatives which needs a high enough precision.

  Consequence: While both fields do heavy computations, the focus of the optimizations in a fitting library is slightly different and requires for example to always explicitly specify float64 as data type.

- Minimising a loss is a non-trivial task. Algorithms usually start at a certain point and use local information to make forward steps. The gradient and sometimes higher order derivatives, usually up to the second order, are used to help finding the minimum. In particular, which order is usable strongly depends on the number of parameters: the Hessian matrix of n parameters has n^2 entries rendering its calculation unfeasible for more than a few hundred of parameters; this restricts the minimisation of DNNs to only use the first order derivatives at the cost of more required steps.

  Consequence: On one side, minimisers designed for DNNs and optimised to work with the framework are not suitable for model fitting. On the other side the analytic gradients that are provided by th frameworks for their minimisers can be easily extended to higher orders and come in very handy for model fitting minimisers.

- Fitting a model has the goal to find the parameters for which the model matches the data best. In terms of a loss function, this is equivalent to finding its *global* minimum. Being stuck in a local minimum is a problem and requires careful treatment. Contrary in DNNs the global minimum is not found but also not desired. The DNN has to approximate an arbitrary data sample *good enough* and a local minimum is usually found, it is in fact preferred over the global minimum since, due to the high degrees of freedom of a DNN, this typically entails a huge over-fit[8] and a bad generalization.

[7] Keras is an API specification only, a reference implementation exists.

[8] Basically remembering every noise in the data.

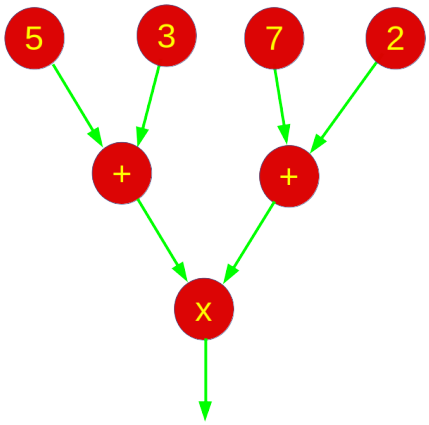


Figure 4: Example of a graph representing $result = (5 + 3) * (7 + 2)$.

Consequence: While finding the global minimum is crucial in model fitting, deep learning is interested to find a local minimum.

- Finding the global minimum in the model fitting case involves the evaluation of the loss over the whole data sample at every step. For the training of a DNN, variations of a technique called stochastic gradient descent (SGD) are used: they work by only evaluating a small mini-batch of the data, typically around 32 events, and then taking a step towards the negative gradient direction.

  Consequence: Both fields are pushing the limits of handling big data and samples too large for the memory are common. Deep learning is optimized to loop through a data set with very small batch sizes to take a minimisation step but to do that millions of times. Model fitting needs to loop through the whole data sample *once* for a single step but no more then a few thousand times.

As seen, while there are differences, the core of the problem is still the same: build a complicated model, use data to get a value from it and tune parameters to optimize the loss.

To accomplish this task efficiently, deep learning frameworks use a declarative paradigm by building a computation graph as seen in Fig. 4. This allows to perform optimisations and define the parallelisation *before* the actual execution of the computation. Furthermore, it also allows to get an analytic expression for the gradient by consecutively applying the chain rule within the graph.

To implement model fitting in `zfit`, the TensorFlow library was chosen, restricting the implementation to static graphs as explained in more detail in Appendix B.2. The main motivation for this decision comes from the fact that models in model fitting *usually* don't change their logic but are rather built once and then minimised. While the same is true for most DNNs, more advanced fields in deep learning like reinforcement learning can heavily rely on dynamic models. The main advantages of a static graph are the additional, potential speedup and the immutability, which leads to less unexpected behaviour. The restriction that *anything built will remain like that and not change* allows for obviously more efficient optimizations compared to a graph where any part could change anytime and a re-analysis of the graph is required.

Working with graphs leads to difficulties and unexpected behaviours in comparison with more traditional, non-graph based code, such as the one used in RooFit. In `zfit`, most of these complications are hidden from the user and the library offers a similar user experience as the one found in other model fitting libraries. This requires some extra care taking behind the scenes, as explained in Appendix B.2.

4 `zfit` implementation

Things should not be easy or hard. But consistent.

In Sec. 3 an example of a complete fit with `zfit` was shown. While this was a rather simple example, `zfit` generalizes to more complicated, custom model or higher dimensional fits in a consistent manner. The guiding principles that facilitate this generalization are

One thing The library is meant to do one thing: model fitting. Additional capabilities like advanced plotting, statistics etc. are intentionally left to other libraries.

Simplicity As often stated, easy cases should be easy or at least straightforward and not hard. For simple, one dimensional models and fits, no extended knowledge about advanced concepts or pitfalls should be necessary. Making mistakes can only happen when clearly leaving the known grounds.

Consistency When going from the simple to the more complex case, it should not be necessary to learn a whole new set of rules and behaviour. Instead, enough generality should be contained also in the simple case, even if it means to slightly heighten the knowledge required for it. The overall difficulty should gradually increase, and expert knowledge should only be required for expert problems.

Flexibility No library can ever cover everything, so special cases will need to be implemented specially, but most elements of the library should nonetheless be usable and only the special part has to be implemented specially.

These requirements can be conflicting. Especially flexibility and simplicity are hard to reconcile. A conflict arises with the potential of creating bugs: since not every case is the most complicated one, not every user will know about all peculiarities of the library and may uses the freedom in the wrong way. To avoid mistakes due to the lack of knowledge of the library but also incorporate the flexibility, `zfit` follows the pythonic way, a term used to describe the philosophy behind Python and its usage.

While the Python language itself, as any other language, consist of syntax specifications, a non-neglectable part of Python consists of its philosophy, the Zen of Python. A commonly used phrase in this context is the expression *"We're all adults here"*. It refers for example to the absence of enforced private class attributes in Python. In its essence it means that anyone should be able to change anything, which offers a great flexibility.

A complementary strong guidance is given by *"There should be one – and preferably only one – obvious way to do it"*. This encourages to discuss problems appearing in the Python community and finding one "best" way to solve a specific problem. And even if there is no "best" solution, at least a convention should be agreed on. Combining these two ideas, `zfit` provides a lot of freedom and allows to implement any part of the fit flow. To counter unintentional mistakes, clear guidelines and examples on *how* to accomplish something specific are provided. This is especially important for the simpler cases, in which one clear way is shown.

In the following, we will go through a few essential pieces of the library that allow to extend the workflow naturally to more complex fits and discuss them in more detail.

4.1 Spaces and Dimensions

The extension to more than one dimension when fitting a model contains some ambiguities. The data, limits and the models all have corresponding axes or columns that should match. To deal with that in a simple, yet consistent manner, `zfit` has a `Space`. The responsibility of this class is to define and handle the axes or dimensions and limits. To understand the concept and the `Space` itself, three definitions need to be made:

Observables The observables are the *named axes of a coordinate system*. A single observable is a string and a list of strings acts as several observables describing a higher dimensional system. In the context of data this is equivalent to columns in a data frame. Observables allow a named, *inter-object* identification of axes. Therefore, working with observables allows to work independently of the underlying data ordering.

Axes Axes are integers and are used to *enumerate the axes of a coordinate system*. This corresponds to indices of an array and provides the fully order-based mapping necessary for *intra-object* manipulations. For example, a `Data` with three columns has three axes, 0, 1, and 2, which can though be reordered so that the corresponding observables match the order of some other observables.

Limits A description of boundaries that can be used to define any kind of limits of the axes. Currently only rectangular limits are supported but arbitrary shaped limits will be provided in the future.

A `Space` can be initialized with observables and limits to define a domain. When it's assigned to an object, it automatically connects the axes of the object with the observables from the `Space`. More details on the implementation and use cases as well as additional functionality for dimensional handling can be found in Appendix C.1.

4.1.1 Limits

Limits are used in many instances, be it in sampling limits, integration or data limits. A `Space` not only defines the observables but typically also has limits associated with it. In one-dimensional fits, limits as seen in the example in Sec. 3 are needed. Simple limits consist of a tuple for each observable with lower and upper limits. For example a `Space` in one observable `x` from -5 to 3 can be created like

```
limits = zfit.Space(obs="x", limits=(-5, 3))
```

This is the simplest way of specifying limits and rather a special case. For anything more sophisticated, such as multiple limits or multiple observables, either a composition of `Space`s or the more general format as explained in Appendix C.2 has to be used. For example when blinding a region, a `Space` with multiple limits can be used.

While the general format is fully specified independent of a `Space` and can therefore be useful programmatically, a `Space` with multiple limits can be built unambiguously from `Space`s with simple limits by adding them, either through a dedicated `zfit` function or using the addition operator in Python. In this way, multiple limits can be created through simple composition and without the need of using the more general format.

As an example, let's assume a `Space` should be created with the observables `x, y` in the two domains $l_0 1$ and $l_2 3$

$$l_0 1 = \{(x, y) | x_0 < x < x_1, y_0 < y < y_1\}$$
$$l_2 3 = \{(x, y) | x_2 < x < x_3, y_2 < y < y_3\}.$$

We start out creating the domains by specifying the limits in the x observable

```
limit_x_01 = zfit.Space(obs="x", limits=(x0, x1))
limit_x_23 = zfit.Space(obs="x", limits=(x2, x3))
limits_x = limit_x_01 + limit_x_23
```

Equivalently `limits_y` can be composed. Since going to higher dimensions is unambiguous with two limits in each space, this can be done using the multiplication operator in Python or the function `combine`.[9]

```
limits_xy = limits_x * limits_y
limits_yx = limits_y * limits_x
```

The difference between `limits_xy` and `limits_yx` is the order of the observables. In the first case, it's `["x", "y"]` while for the latter it's `["y", "x"]`. In order to ensure consistency, if the two `Space`s already have observables in common, the limits in this observables have to be the same. [10] Reordering the `Space` is possible as well as extracting a subspace, a `Space` only defined in subset of the dimensions. More details can be found in Appendix C.1.

4.2 Data handling

Data to fit can come from different sources but it should be handled uniformly inside `zfit`. To ease this, the `Data` object is responsible for loading, ordering and simple preprocessing of data, which can have weights assigned to it. Furthermore, this abstraction layer with `Data` potentially allows for more advanced use cases such as batched, out-of-core computations of the likelihood.

The `Data` class supports a variety of data files and structures. While adding additional loading capabilities is not difficult, the focus is on the following formats: the default HEP format ROOT[11], Numpy arrays and Pandas DataFrames, and pure Tensors. More details can be found in Appendix C.3

Each `Data` has a `Space` with observables it is defined in. This assigns an observable to each column of the data, so the observables here act like columns from spreadsheets or DataFrames. This allows to retrieve a subset or different ordering of the `Data` by specifying the observables explicitly in the method that returns the data as a Tensor.

Once instantiated, a `Data` object appears like a lightweight wrap of the Tensor class and can be directly used as such. It is possible therefore to simply operate on a `Data`

[9] If a different number of limits were defined, an error would be thrown.

[10] This exact behaviour of the multiplication and observables is the same if models are multiplied.

[11] Even though ROOT files are supported, the ROOT library is not needed thanks to the uproot package as explained in Appendix C.3.

object with any operation that would also accept a pure Tensor. While this is convenient for certain contexts where the correct ordering of the data is guaranteed such as inside a model, the preferred way of using the `Data` is to access the columns by names using the `unstack_x` method. The `Data` class can also handle data generated on the fly and not fitting into memory, see C.4.

`Data` objects can be ordered in-place as opposed to `Space`, the reordering of which returns a new instance. This is heavily used together with a context manager inside models if a `Data` is given as an argument in order to match the order of its observables with the order of the models observables.

4.3 Model

Building models is the core competence of `zfit`. As seen in the example in Sec. 3, this can be done in a simple manner by using already implemented models and possibly combining them, but the models can also be completely custom built. Within `zfit`, there are two basic types of models to cover most cases: Functions and Probability Density Functions (PDF).

The basic features of a model include

- Each model is defined inside a `Space`. Its dimension are "observables", simple string identifiers as previously discussed.
- A model implements a function that returns its value depending on some data. This is either `pdf` for a PDF or `func` for a Func.
- Full as well as partial integration over a model is possible. This includes numerical as well as analytical integration, if available.
- Generating a sample following the models shape using numerical or analytical methods, the latter only if available.

The value function as well as the integration and sampling are implemented to return pure Tensors. Depending on the task, higher-level methods providing either a more intuitive, imperative behaviour or a significantly more performant execution for certain cases such as repetitive pseudo-experiments are also implemented.

The main differences between a PDF and a Function concern the normalisation and the output dimensionality. This leads to a few subtle differences.

PDF A PDF $f(x)$ is only well-defined with a given normalisation range. This defines the normalisation constant so that, with the limits from *lower* to *upper*, the integral over the PDF equals to one as

$$\sum_i \int_{l_i}^{u_i} f(x)dx = 1 \tag{6}$$

with i indexing all the limits that make up the domain, l_i and u_i are the lower respectively upper limit.

A PDF object has three special attributes, which are

- the probability density function `pdf`. It returns always a rank one Tensor, i.e. a simple vector, with the length number of data points.
- the probability density function *without* the normalisation constant, `unnormalised_pdf`. In cases where only the shape is needed, using this function is less expensive, especially if the normalisation has to be computed numerically.
- the limits that define the normalisation constant of Eq. 6. They can be set using the `set_norm_range` method.

Function A Function is in a way more simple and general than a PDF. It takes the same values as a PDF but returns something with dimensionality $\mathbb{R}^m$. It can be used to transform values or to use it as a building block for more complex expressions.

It has the method `func` to evaluate its value and no other special attributes.

4.3.1 Parametrization

Models can be parametrized by `Parameter` objects which can be used in the implementation of the shape function like any other Tensor-like object when building the model. In the following we will first have a look at the `Parameter` itself. Afterwards we will see how they are exactly used with a model.

There is a distinction between dependent and independent parameters as only the latter can be changed directly and have limits while the former are any arbitrary combination of them. An independent `Parameter`

- has a name with purely descriptive purpose;
- has an initial value;
- maybe has lower and upper limits;
- is either floating or not, independent of whether limits were specified;
- has a step size indicating the order of magnitude of the parameter which can be given. Otherwise, it is automatically inferred. A well chosen step size improves the minimization process and can be critical, mostly in the absence of limits and with a weak dependence on the model, so that large steps will be required to change the model.

Currently, the shape of parameters is implicitly restricted to a scalar.[12] As a consequence, a parameter cannot simply be treated like data as it is possible in RooFit. A function `Parameter` which depends on the data itself will most likely be available in the future.

The name of a parameter, as any other name for a single object in `zfit`, has purely descriptive character. There is purposely[13] no direct way provided to access parameters

[12] This comes from the fact that the PDF will have different sized data as input which is not controlled by the user, such as when doing numerical integration. Any parameter therefore has to be able to broadcast seamlessly.

[13] Contrary to the Root framework. If the need ever arises, adding this as an additional feature is relatively simple.

by name: instead of using names the actual parameter object is passed around. This has the advantage of avoiding double bookkeeping the parameters, since then a reference on an object as well as on a string would be needed. The name is mandatory for parameters, as opposed to other objects in `zfit`, since matching the value of a parameter to the right name is a critical task during and after a minimization, when reading off the right value.

Composed parameters are dependent parameters. In general they can be any Tensor, that is a result of any kind of operation. They can depend on zero, one or more independent parameters, as composed parameters are arbitrary functions; therefore, operations such as shifting and scaling are included as a trivial subset. For more details on the actual implementation and the dependency management see Appendix C.5. Composed parameters can neither be floating nor have limits currently, since the independent parameters a composed parameter may depends on can have arbitrary relations and have to be restricted themselves. In order to restrict a parameter, arbitrary constraints can be given to the loss instead.

Every model can depend on multiple parameters, both dependent and independent. Each parameter that parametrises the model has a name specific to the model and is given on instantiation. For example a `Gauss` has parameters named `mu` and `sigma` as in the example in Sec. 3. They are stored in a mapping attribute named `params`.

```
mu_param = gauss.params["mu"]
sigma_param = gauss.params["sigma"]
```

Notice that this is not contradictory to the statement above that single objects *cannot* be accessed by *their* name. Each object has a unique identifier, the name, but objects can have names for their constituents that are *not* unique, like `mu`. This simply describes a part of the PDF and any `Gauss` will have a parameter `mu`.

4.3.2 Implementing a custom PDF

An essential feature of `zfit` is the ability to simply create custom models. There is a large freedom in building models from the `BaseModel` class, since it takes care of most boilerplate and has well defined entry points than can be customized. The full implementation details and possibilities for customizations are described in Appendix C.6.

For the most common use cases though, there exists a simple way of creating a custom model. The `ZPDF`, basically a more user friendly wrapper around `BasePDF`, can be be used as a base class in these simple cases. The following function will be used as the PDF shape

$$f(x, y) = a \cdot x^2 + b \cdot y^4. \tag{7}$$

To implement this function, `_unnormalized_pdf` has to be overwritten. For the vast majority of custom models, this is the only method to be overwritten. Changing other methods, especially `_pdf`, is an advanced feature and only needed in special cases. For more details on the customization and the possibility of hooking into the calls see Appendix C.6.

This is a two dimensional PDF with two parameters. To implement it in `zfit`, a new class is created

```
class EvenPolyPDF(zfit.pdf.ZPDF):
  """Implementation of f(x, y) = a*x^2 + b*y^4"""
   _PARAMS = ['a', 'b']
   _N_OBS = 2  # since two dimensional

  def _unnormalized_pdf(x):
    xdata, ydata = x.unstack_x()
    a = self.params['a']
    b = self.params['b']
    return a * xdata ** 2 + b * ydata ** 4
```

Note that we only need the *shape* of the function but do not need to take care of the normalisation, as numerical methods are already implemented in the base class.

We can see the advantage of the preprocessing that is done by the base class, especially the reordering of the data `x`. It is a `Data` object and calling the `unstack_x` method returns a list[14] containing a Tensor for each column sorted according to the models observables. The length of the list has to coincide with the specified `_N_OBS`, the number of observables. It is not mandatory to specify this field in a Model and is sometimes not possible to know previously, in which case it can simply be left away.

The naming of the parametrization of the function is defined with `_PARAMS`. These exact names have to be used when creating an instance of the model. The parameters are then stored in the `params` dictionary and extracting them is usually the first step inside `_unnormalized_pdf`.

Documentation plays an important role here: it defines the name of the parameters that have to be used and what they represent in the function, but it is also crucial to communicate the ordering of the data. In this case, a user can see that the first observable and the corresponding column in the data will be used as x in the function.

This PDF is already complete and works out of the box. We can create an instance and use its methods. As an example, the observables of the instance will be called "xobs" and "yobs" with the normalisation range going from 0 to 10 and from 0 to 5, respectively.

```
param_a = zfit.Parameter("a", 1.)
param_b = zfit.Parameter("b", 2.)
x_obs = zfit.Space("xobs", limits=(0, 10))
y_obs = zfit.Space("yobs", limits=(0, 5))
obs = x_obs * y_obs
poly_model = EvenPolyPDF(a=param_a, b=param_b, obs=obs)

prob = poly_model.pdf(...)  # some data needed

x_limits = zfit.Space("xobs", limits=(3, 5))
y_limits = zfit.Space("yobs", limits=(1, 3))
integral_limits = x_limits * y_limits
integral = poly_model.integrate(integral_limits)

sample = poly_model.sample(n=1000)
```

[14] Or a single Tensor fo the one dimensional case if not specified differently in the arguments.

Using the `integrate` method, we obtain the integral i of our normalized PDF f_{norm}

$$i = \int_3^5 \mathrm{d}x \int_1^3 \mathrm{d}y \; f_{normed}(x, y) \tag{8}$$

$$\overset{(3)}{=} \frac{\int_3^5 \mathrm{d}x \int_1^3 \mathrm{d}y \; f(x, y)}{\int_0^{10} \mathrm{d}x \int_0^5 \mathrm{d}y \; f(x, y)} \tag{9}$$

with x and y corresponding to the observables `x` and `y`, respectively, and $f(x, y)$ is the unnormalised PDF as defined in Eq. 7. Using Eq. 3 we end up with a precise formulation of what is *actually* executed in `zfit`. In other words, the integral was calculated over the limits 3 to 5 and 1 to 3 respectively and normalised over the normalisation range `obs`. The latter is defined on initialisation and taken as the default normalisation range. As mentioned before, the limits could also be multiplied in a different order resulting in `limits` having the order ("yobs", "xobs"). The model takes care internally (see also Appendix C.6.1) that the right limits are at the right place.

The `sample` method is used to generate 1000 points from the model. With the instance created, also probability densities `prob` for points can be calculated by calling the method with some `Data` object called `data` as follows

```
probs = poly_model.pdf(data)
```

It is worth noting that none of the operations are executed yet and what is returned by the methods are Tensors, as described in Sec. 3.1. To run the actual computations, it is necessary to call `zfit.run(...)` on any Tensor. Running

```
probs_np = zfit.run(probs_np)
integral_np = zfit.run(integral)
```

returns an actual numpy array for `probs_np` and a Python float for `integral_np`.

For further, advanced customization of the PDF, methods can be overridden as described in Appendix C.6.1. However, in most use cases there exist better ways: for example, to add an analytic integral to the model, overwriting `_analytic_integrate` should be the last resort. Instead, integrals defined over a specific range only or the whole range and over all dimensions or just partially can be registered with a model. A priority attribute allows to specify preferences on one over other methods.

A typical use case for this feature is a special integral that is known exactly, for example the value of the integral over the full space of a Gaussian shaped model[15] An integral over both dimensions is registered but partial integrals could be added as well

$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} f(x, y) \mathrm{d}x \mathrm{d}y = (1/3 \cdot a \cdot x^3 + 1/5 \cdot b \cdot y^5)\Big|_{x_0}^{x_1}\Big|_{y_0}^{y_1}. \tag{10}$$

In case a full integral is requested but no analytic integral over all dimensions is available, the fallback of `integrate` looks for partial integrals. If available, it uses them to numerically integrate over the remaining dimensions instead of using the unnormalised PDF.

The implementation of the integral with `zfit` is done by registering it to the PDF

[15] As already hinted in Eq. 8, it is important to note that the integral *to be implemented* is over the unnormalised PDF as implemented in the `_unnormalized_pdf` method.

```
def integral_full(x, limits, norm_range, params, model):
  lower, upper = limits.limit1d
  a = params['a']
  b = params['b']

  lower = ztf.convert_to_tensor(lower)
  upper = ztf.convert_to_tensor(upper)
  def indef_integral(limit):
    return 1 / 3 * a ** 3 + 1 / 5 * b * y ** 5

  return indef_integral(upper) - indef_integral(lower)

lower = ((zfit.Space.ANY_LOWER, zfit.Space.ANY_LOWER),)
upper = ((zfit.Space.ANY_UPPER, zfit.Space.ANY_UPPER),)
limits = zfit.Space.from_axes(axes=(0, 1), limits=(lower, upper))

EvenPolyPDF.register_analytic_integral(func=integral_full,
                                       limits=limits)
```

Since the observables that will be assigned to each axis are unknown, the `Space` has to be defined with axes, not with observables. This follows the principle of using axes inside the model as explained in 4.1. Instead of `ANY_LOWER` and `ANY_UPPER`, it could also be defined over a specific domain by using plain Python floats. The PDF will now automatically use this integral if possible. Otherwise, as for example when creating a partial integral, it will silently fall back to numerical integration.

This example demonstrates how to implement a shape that does only depend on the data and parameters. Models, that also depend on other models, such as the `SumPDF` from Sec. 3, require to be built from a `Functor`. This implies as a minor change only an additional argument to the base class, the depending models, in order to track the dependencies correctly. This is described in more details in Appendix C.6.6.

4.3.3 Sampling

Sampling from a model can be done in two different ways to cover two distinct use cases. On one hand, the method `sample` returns a pure Tensor, which behaves as any other sampling algorithm in TF, for example `tf.random.uniform`. This can be useful for specific and mostly advanced cases, but he overall behaviour can be unintuitive for inexperienced users, as discussed in Appendix B.2.

On the other hand, sampling from a model is typically used to perform toy studies, which consists of the generation of events according to the model and afterwards a fit of the model with randomly initialized parameters to the sampled data. With `create_sampler`, a `Data`-like object which handles the correct storage and the sampling from the model is created. As any other `Data`, it can be used to build a loss. The actual data is sampled by invoking `resample` and stays unchanged until it is invoked again. This sampler keeps a dependency on the original model and uses it to sample, keeping the original parameter values it was created with. If desired, any parameter that has changed in between will be at that new value when `resample` is called, effectively changing the sampling shape. [16]

[16] Just to be clear: the default behaviour is that the sampler samples from the exact distribution that it was created with *in terms of parameter values*.

To illustrate the use of the toy sampler, let's look at a toy study example. Note that the actual (large) *objects* needed such as the model, minimiser, loss etc. are created *outside* of the loop and only the necessary *methods* are called *inside*. In general, this removes boilerplate code from the additional object creation but is especially important when working with a graph based backend. We assume to have previously built a model already and a minimizer as for example shown in Sec. 3, and we generate 1000 events according to our model.

```
sampler = model.create_sampler(n=1000)
nll = zfit.loss.UnbinnedNLL(model, sampler)
for _ in range(n_toys):
  sampler.resample()  # now the sampling happens
  for param in model.get_dependents(only_floating=True):
    param.set_value(np.random.normal())  # any initial value
  result = minimizer.minimize(nll)

  if result.converged:  # check if the fit was successful
    ...  # safe results in a list or similar
```

The sampling is implemented with the accept-reject method that works with arbitrary shapes. If an analytic inverse integral is registered, this will be used, providing a more efficient way of sampling. More details about the different techniques and the use of importance sampling are described in Appendix C.7.

4.3.4 Extended PDFs

In addition to the shape, a PDF can carry more information, namely a yield, in which case we refer to it as an Extended PDF. The yield is a scale that describes the magnitude of the PDF, typically reflecting the number of events in the data sample. It can be used to multiply the output of `pdf`, the normal probability density function, which results in a number probability. This implies that when multiplying the integral with the yield, the number of events is retrieved instead of the probability over a certain range.[17]. To count for example how many signal particles are in the sample, a composite PDF existing of a background and a signal component, both extended, can be fitted to the data. The `SumPDF` used to build this composition does not need a fraction if the PDFs are already extended, but uses the yields, normalised to one, as fractions. In this case, integrating the PDF that represents the signal shape over the whole range returns the number of signal events.

To create an extended PDF, `create_extended` can be used. This method returns a copy of the PDF and adds a yield to it.

4.4 Loss

As discussed in Sec. 2, the loss is the core of the problem specification, since it describes the disagreement between a model and the corresponding data. Additionally, it can contain constraints which help further specify the problem.

[17] *Currently*, the integral of an extended PDF is multiplied by the yield *by default* but this behaviour is meant to change in a future version

Having the loss as an independent part of the whole workflow is a crucial design feature in `zfit`: it is the connection between the model, data, and their relation on one side and the minimisation process on the other side. Having the loss as an extra step accomplishes decoupling the former from the latter: a minimiser can take a loss and minimise it *without* knowing anything about the underlying models, data or the actual definition of the loss. Therefore, it is important that the loss knows everything that is needed for a minimisation, as listed in detail in Appendix C.8.

Basic loss implementations like the `UnbinnedNLL` use a PDF and data to calculate the loss according to Eq. 5. With extended PDFs, an additional term is taken into account if using `ExtendedUnbinnedNLL` instead, as derived in Eq. 26. Furthermore, additional terms can be added to express prior knowledge about parameters as in Eq. 29. To keep the flexibility, *any* Tensor can be added as a constraint to the loss with the method `add_constraint`. Alternatively, a custom constraint can be implemented by using the base class `BaseConstraint`. `zfit` implements the most often used constraints to improve usability: in the example from Sec. 3 a Gaussian constraint for the parameter μ could be applied by adding the following line after the creation of the `nll`.

```
mu = ...  # parameter
nll = zfit.loss.UnbinnedNLL(gauss, data)
mu_constr = zfit.constraint.nll_gaussian(params=mu, mu=6.8, sigma=0.4)
nll.add_constraint(mu_constr)
```

Typically, some parameters are shared between different fits to different data samples. These can be obtained through a simultaneous fit of all the datasets by creating multiple PDFs with some of the `Parameter` objects being the same. Since this corresponds to a simple addition of the losses as seen in Eq. 25, `zfit` allows to perform precisely this operation. As an example we create two Gaussians and assume to already have their data. Here the μ is shared while the σ is not. Limits and the data are created as in the example of Sec. 3:

```
mu = zfit.Parameter("mu", 7)
sigma1 = zfit.Parameter("sigma", 1.1)
sigma2 = zfit.Parameter("sigma", 1.5)

gauss1 = zfit.pdf.Gauss(mu=mu, sigma=sigma1, obs=limits)
gauss2 = zfit.pdf.Gauss(mu=mu, sigma=sigma2, obs=limits)

nll1 = zfit.loss.UnbinnedNLL(gauss1, data1)
nll2 = zfit.loss.UnbinnedNLL(gauss2, data2)
simul_nll = nll1 + nll2
```

Alternatively, a list of models and their corresponding data can be given to create the loss

```
simul_nll = zfit.loss.UnbinnedNLL([gauss1, gauss2], [data1, data2])
```

There is a special loss available in `zfit` that gives a flexibility rarely found in other fitting packages: the `SimpleLoss`. This object lightly wraps any Tensor, which allows to build *any* kind of loss. No dependency on the data structure or the model layout is

required and it is completely up to the user. This allows to create not yet implemented losses, such as binned ones, and allows other libraries which build loss functions with TF to simply hook in with this mechanism. Because the `SimpleLoss` can then be used with the next steps such as the minimisation and error estimation, an other library can therefore use the whole available tools in `zfit` as well as any library that builds on top of it.

4.5 Minimisation

The optimisation of functions is a large topic by itself and a lot of implementations of different algorithms exist. They usually need a function that returns a value, such as the loss, which depends on the parameter values that they use as arguments. Since the computation of the loss is efficiently implemented in `zfit` thanks to TF and this usually is the heavy part of the minimisation, in practice any minimiser can be wrapped easily. In `zfit` several minimiser algorithms are implemented by wrapping existing libraries and giving them a common API. An important part of the design is that the creation of a minimiser object does neither execute any minimiser function nor tie itself to a specific loss, it's simply the configuration of the minimiser. This implies that the minimiser is stateless, and to actually invoke it, the `minimise` method needs to be called. The information about the minimisation procedure and the parameter values are collected in a `FitResult` and returned by the minimiser.

4.5.1 Different optimisations

While a whole variety of algorithms exists, not all are equally feasible to be used for model fitting as done in `zfit`. There are various distinctions that influence the choice of a certain minimiser.

Derivative Some optimisers use the derivatives and others don't. In general, using the derivative provides a huge advantage since it tells about the local shape of the function. This requires though that the function to minimise be continuous, which is not always the case. For model fitting, functions are continuous and, thanks to TF, `zfit` uses an analytic expression for them.

Global/Local Some optimisers are better in finding a global minimum by doing a variation of a large grid search. Others focus on a local minimum by using a starting point and going along a path to the next minimum. The latter is more accurate and faster *if* a good initial parameter estimation is given, so that the minimiser does not start far from the true minimum. In model fitting, a reasonable estimate can often be made.

Dimensionality The number of parameters that have to be tweaked in order to minimise the loss has an impact on the strategy. While the Hessian matrix can help greatly with the minimisation, it has n^2_{params} entries. This renders its usage impossible for hundreds of thousands of parameters as used in deep learning. For model fitting with a maximum of a few hundreds and typically around a dozen of parameters, using the Hessian is feasible.

In HEP model fitting, mostly local, second order derivative minimisers are used. An important algorithm is the Newton method of minimisation: leaving the mathematical details away, the algorithm performs a second order Taylor approximation of the function using the exact Hessian. Assuming that the target is a saddle point with the derivate equal to zero, the equation is solved and the algorithm jumps to the estimated minimum. Since this solution is only the true minimum if the second order approximation were exact, this step is repeated until some convergence criteria are fulfilled.

Newton's method is often not directly used since the computation of the Hessian and its inverse can be computationally expensive. Instead an approximation of the inverse Hessian is calculated and updated on every minimisation step, giving rise to a family of methods called Quasi-Newton methods. Since these updates bring some ambiguity in higher dimensions, different methods use different assumptions on the updates. A prominent example is the BFGS algorithm and its low memory variant L-BFGS, which only stores the most recent steps. L-BFGS is also the method implemented in Minuit, the most common used minimiser in frameworks like RooFit or Root.

Another important point is the stopping criteria. While Minuit has its own stopping criteria and in general good criteria have to be found with experience, this tuning for other minimisers is still ongoing work in `zfit`. It is though simply possible to define and change this criteria for a minimiser.

A special mention has to be made about TF optimisers. As discussed above, for deep learning first order algorithms are used relying on the so called "gradient descent" technique. This algorithm simply follows the negative gradient iteratively. Advanced variations alter the step size based on heuristics but are still largely inferior for low dimensional problems in comparison with Quasi-Newton methods. A wrapper for any TF optimiser is available in `zfit`, an example is provided using the implementation of the Adam [13] optimiser. Simple benchmarks though yield an order of magnitude increase in the number of minimization steps compared to Quasi-Newton methods, they are not competitive. to the

4.6 Results and uncertainties

After every minimisation, the information about the process as well as the results are returned as a `FitResult`. This object does not only store information but is also capable of performing uncertainty estimations of the parameters.

The most important information that it collects is:

- Information about the parameters, including the values at the function minimum, information about their limits and uncertainties calculated using different methods.
- General information about the minimisation itself, including
  - A flag that indicates whether the minimisation was successful or not.
  - The minimum of the loss function that was determined by the minimiser.
  - An estimation made by the minimiser of how far away the minimum value is from the actual true minimum.
  - All the additional information produced by a minimiser. This can highly vary depending on which minimiser was used.

- The instance of the minimiser that was used to perform minimisation. Since minimisers are stateless, no information is stored in it.[18]
- The instance of the loss that was minimised. Since a loss keeps references to the model and data it was built with, it is possible to thereby retrieve all information regarding this minimisation.

As the last of five steps in the minimisation workflow, the `FitResult` serves again as an additional abstraction layer. A lot of different statistical quantities such as limits, confidence intervals and more can be calculated using the result, loss and minimiser. Since they are all bundled together in the `FitResult`, no other object is required for advanced statistical treatment of fit results.

4.6.1 Parameter uncertainties

The values of the parameters at the minimum are important, but they are meaningless without an uncertainty estimate. Therefore, the `FitResult` provides two ways of calculating it:

- For a fast, approximative and symmetric estimate, the `hesse` method can be invoked. It provides an estimation based on the Hessian matrix, assuming a second order approximation of the loss around the minimum value of the parameter.
- If there are high non-linearities in the loss and the parameter correlations, the actual uncertainties differ strongly from what `hesse` returns. Good estimates can be retrieved by creating a profile: fixing the parameter at a certain value and run a complete minimisation. The calculation is invoked by using the `error` method. If the Minuit minimiser was used to perform the minimisation, the `minos` method can be invoked in this way. Currently, no other error estimation is implemented, but any custom statistical method can be easily applied thanks to the information contained in the fit result.

To only calculate the uncertainties with respect to specific parameters, the desired parameters can be given as arguments to `hesse` and `error`. The results for each parameter are cached and won't be recomputed on an additional call.

[18] Ideally. Some minimisers like iminuit have a state and can be accessed like this. A copy of the actual minimiser is stored in the `FitResult`

5 Performance

In the previous Sections, the structure and logic of model fitting in `zfit` was elaborated. Apart from the functionality, model fitting involves a lot of number crunching and thus state-of-the-art performance is a hard requirement.

The performance of code depends on several factors and can often be divided into a serial and a parallel part. The efficient implementation of parallelised parts of code are as important as deciding *when* to actually run in parallel, as discussed in Appendix D.2. While this part is mostly left to TF, the fact that some pieces of code simply *are not* parallelisable, and therefore cause a bottleneck, depends on the nature of the problem itself. In real code, the two are not clearly separated and a mixture of both influence the actual performance.

In this Section we will focus on quantifying the performance of `zfit` by measuring the execution time of the whole process, mixing together the performance of TF and `zfit` with the bottlenecks coming inherently from model fitting. While this limits the ability to draw conclusions about bottlenecks and remove them, it reflects the performance of the library as experienced in real life usage.

In the following, two studies are presented: on one hand, a more artificial but well scalable example consisting of Gaussian models, and on the other hand, a more realistic case, namely a fit to an angular distribution is performed. In both cases the performance of toy studies is measured and compared. The hardware specifications can be found in Appendix D.1.

5.1 Gaussian models

There are three quantities of interest to describe the scaling of the library: the complexity of the model, the number of free parameters and the size of the data sample used to fit. In this study, a sum of Gaussian PDFs is used as the model. The fractions are constant and the mean and width are parameters, each shifted by a different constant, and either all depending on only two parameters or scaling with the number of Gaussians.

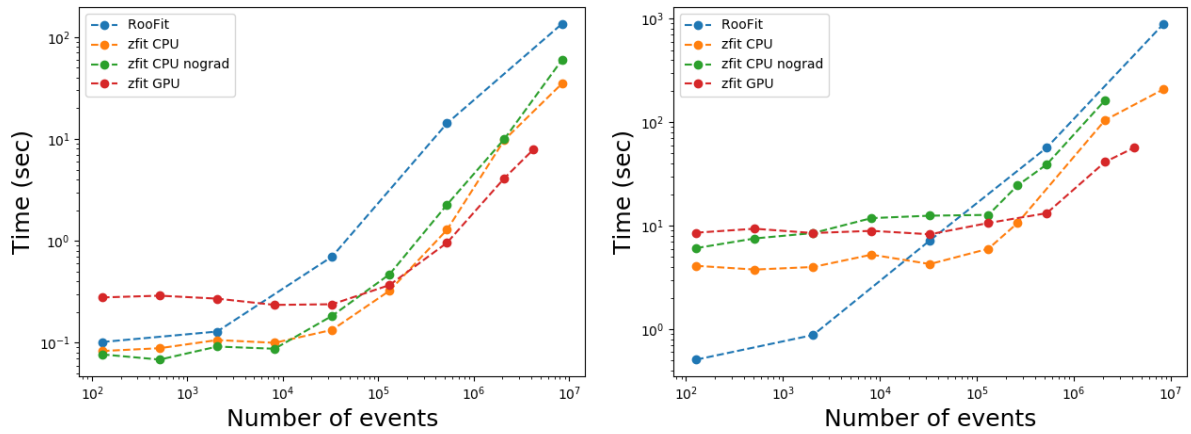
This model is used to perform toy studies. First a sample from the model is created using a fixed, initial parameter setup. Then the parameters are randomized and a fit to the sampled data is performed. The implementation of sampling and the minimisation are two distinct steps and for fits to data, only the latter is actually invoked. Since this combines two execution time measurements making reasoning even harder and there is a known, temporary[19] bottleneck in `zfit` sampling, only the execution time of the minimisation is measured. A approximate comparison of the current performance with sampling can be found in Appendix D.3.

For each setup, twenty toys are run ranging from 2 up to 20 Gaussians with sample sizes from 128 to 8 million events (except for GPU, where it goes only up to 4 million[20]). A comparison with an implementation in RooFit is performed, although only ranging from 2 to 9 Gaussians.[21] To have a fair comparison, both use the Minuit algorithm for

[19]This problem is resolved now, though in this thesis the old measurements are still used since the conclusions are the same.

[20]The GPU used has a comparably small memory. Performing larger-then-memory computations and multi-GPU is still work in progress.

[21]Due to technical problems using a sum of more than 9 pdfs with RooFit that were overcame only



(a) 2 free parameters

(b) 18 free parameters

Figure 5: A sum of 9 Gaussian PDFs with shared (left) or individual (right) mu and sigma. On the y axes, the time for a single fit is shown, averaged over 20 fits. It is plotted against the number of events that have been drawn per toy.

minimisation.

In the following, 4 cases are considered:

- an implementation in `zfit`, labelled as "zfit CPU", using the analytic gradient provided by TF. In this case, an initial run is done to remove the graph compile time. While not significant, this provides a more realistic estimation.
- The same implementation as above is used but the analytic gradients provided by TF are disabled, denoted by the addition "nograd". Instead, the Minuit minimiser calculates a numerical approximation of the gradients internally.
- The same as "zfit CPU" but run on a GPU and labelled as "zfit GPU".
- An implementation in RooFit using the Python bindings with PyROOT. The parallelisation is done when invoking the `fitTo` method and equals the number of cores available.

In Fig. 5, the time per toy is measured and plotted against the number of events used in the toy. While `zfit` on CPU outperforms RooFit in the left plot with only two free parameters, in the right plot with 18 free parameters, RooFit performs significantly faster with a low number of events. This comes from the fact that while the Minuit minimiser is used in both measurements, their are tweaked differently: the version used by RooFit is configured to better cope with a bumpy likelihood as given with a high number of parameters and a low number of events, which results in a vastly superior performance for complicated, low statistics cases while reducing the performance in simpler cases. It is to note that tweaking the minimiser is still ongoing effort in `zfit` and the performance behaviour is likely to change in the future. Different tuning cannot simply be quantified and compared, since not only the number of evaluations of the loss function, but also the

after the measurements were done.

gradient and the (very expensive) Hessian computation is part of the strategy. This limits the conclusions that can be drawn from the comparison of the performance with a low number of events. Furthermore, the minimiser for 18 free parameters using `zfit` was not able to converge for more than a million of events *without the automatic gradient from TF*, therefore no data points are available there.[22]

The comparison in Fig. 5 still reveals a few important things that agree very well with the expectations.

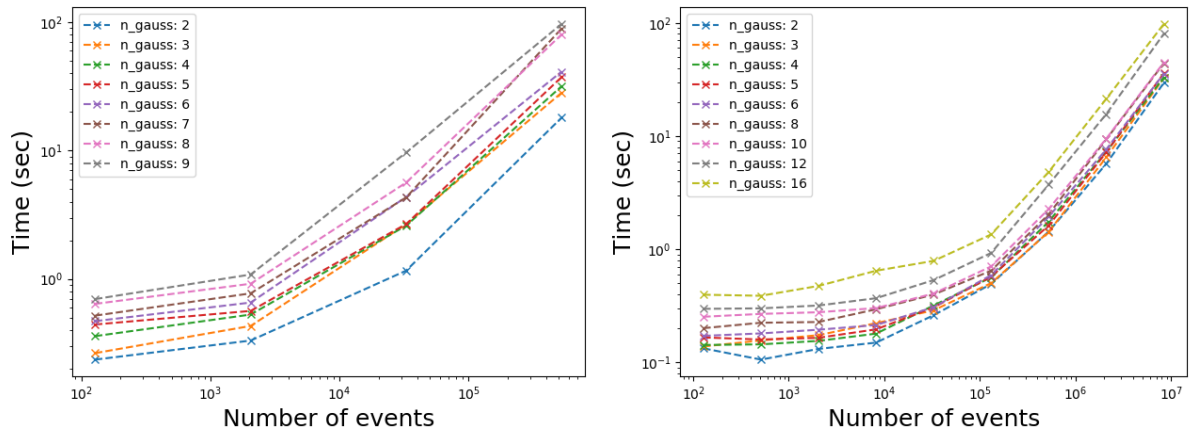
- As the number of events increase, the execution time of RooFit monotonically increases.
- There is no advantage using parallelisation for very few calculations since the overhead of splitting and collecting the results is dominant. With increasing number of events this gets negligible and as an effect the execution time of `zfit` increases way slower than for RooFit. This also comes from the fact that more events mean a more stable loss shape, so the minimiser used in `zfit` performs better.
- The GPU is highly efficient in computing thousands of events in parallel. For only a few data points though, the overhead of moving data back and forth dominates strongly, making it unfeasible for only a small number of events. A continuous decrease of the time up to 10'000 in the left plot of Fig. 5 events confirms that and even shows a drop in the computation time.

As a conclusion, the speed for very small examples around 100 events of `zfit` is *marginally* slower than the corresponding RooFit example. For larger fits, the speedup of `zfit` is up to a factor of ten at around a million events. For more complicated fits and a small number of events, RooFit is an order of magnitude faster because of its currently better tuned minimiser, though there is ongoing work in `zfit`. The GPU delivers in larger examples a similar performance as the multicore setup. Finally, not using the TF gradient yields only a minor penalty for large fits and can even be faster for small ones, experiments have shown that the effect of an increased time can be larger for complicated, real use cases and helps reducing the number of steps required to take by the minimiser.

In Fig.6, a comparison of what can be achieved within a certain time frame is shown. The fits scale with the number of events for different number of added Gaussians, having only two free parameters *in total*. The observation matches the intuitive behaviour of scaling with complexity and size of the sample. Compared to RooFit, the fitting time of `zfit` increases an order of magnitude less; note that both time scales end at 100 seconds. In this time, `zfit` fits 8 millions of events with 16 Gaussians, RooFit does half a million with 9 Gaussians. For this setup, 8 CPUs on a shared cluster were used, leaving slight ambiguity about the results due to the unknown configuration and actual workload on the machine. However, the order of magnitude matches with the results in Fig. 5, which were performed in a clean environment.

Scaling the number of free parameters with the number of Gaussians added is shown in Fig.7. We see clearly that for a large number of parameters having a higher number of events can actually be more performant on an absolute scale, at least up to a certain threshold, since this reduces the number of steps to be taken for the minimisation.

[22] This indicates numerical instabilities due to limited precision and there are ways to circumvent them which are planned to be implemented in `zfit`.



(a) RooFit (b) `zfit` CPU

Figure 6: Measurement of the computation time with a sum of *n_gauss* Gaussians and in total two free parameters. Notice that the y-scale is the same for both plots but the x-axis for `zfit` goes an order of magnitude higher. Also, `zfit` sums up to 16 Gaussians whereas RooFit only goes to 9.

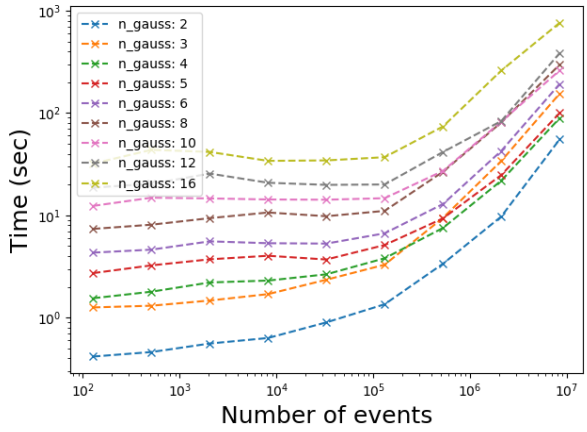


Figure 7: Time of a single toy in dependence of the number of events used. Plotted for a sum of *n_gauss* Gaussians and two free parameters *per Gaussian*.

5.2 Angular analysis

In order to study the scalability of `zfit` in a challenging, realistic setting, a toy study from an ongoing analysis involving `zfit` is used. The example, which consists in the angular analysis of $B^0 \to K^*(\to K^+\pi^-)\ell^+\ell^-$ with ℓ being either e or μ, is an important legacy analysis and the fact that it was implementable in `zfit` is itself already an achievement. The model is from the folded angular analysis. The folding to measure the P5' parameter

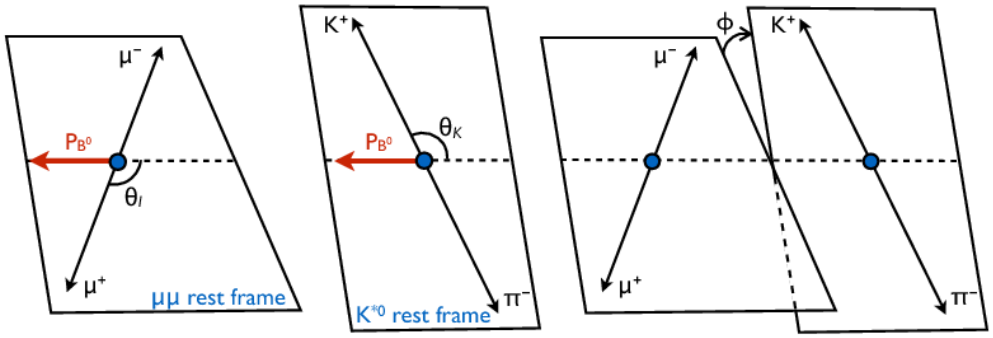


Figure 8: Schematic view of the angles of the $B^0 \to K^*(\to K^+\pi^-)\ell^+\ell^-$ decay. The image was taken from [15]

as defined in [14] is implemented. The resulting model is given by

$$\begin{aligned} f_{angular}(\theta_K, \theta_\ell, \phi; F_L, A_T^{(2)}, P_5') = \; & \frac{3}{4}(1 - F_L)\sin^2\theta_K + F_L\cos^2\theta_K \\ & + \frac{1}{4}(1 - F_L)\sin^2\theta_K\cos 2\theta_\ell \\ & - F_L\cos^2\theta_K\cos 2\theta_\ell \\ & + S_3\sin^2\theta_K\sin^2\theta_\ell\cos 2\phi \\ & + S_5\sin 2\theta_K\sin\theta_\ell\cos\phi \end{aligned} \tag{11}$$

with

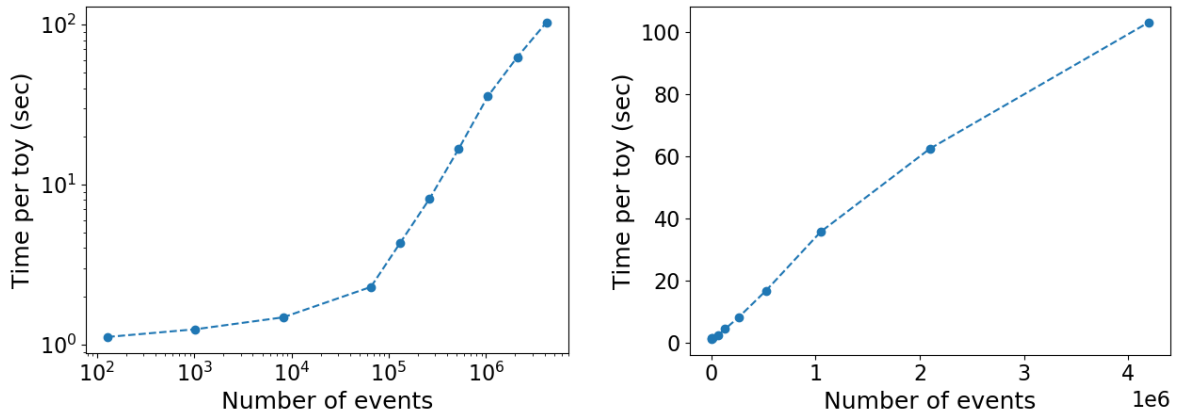
$$\begin{aligned} S_3 &= \frac{1}{2}A_T^{(2)}(1 - F_L)(1 - F_L) \\ S_5 &= P_5'\sqrt{F_L(1 - F_L)}. \end{aligned}$$

The model is depends on three angles, shown in Fig. 8, where

- ϕ is the angle between the plane spanned by the flight direction of the two leptons with the plane spanned by the kaon and pion,
- θ_K is the angle between the kaon and the negative flight direction of the B^0 and
- θ_ℓ is the angle between the ℓ^+ (ℓ^-) and the negative flight direction of the B^0.

The model is extended to four dimensions by adding a description of the B invariant mass distribution by building the product with the angular part as defined in Eq. 11. The model used for the mass shape is a combination of two Gaussians, each with a powerlaw tail.

The implementation of the angular part is straight forward and follows the example in Sec. 3. Using the implemented `DoubleCB` for the mass, the four dimensional model can be built as



(a) Double log scale

(b) Linear scale

Figure 9: The figures show the time needed per toy for the four dimensional P5' folded angular distribution. 25 toys are produced for each number of events.

```
limit_thetak = zfit.Space('thetak', limits=...)
limit_thetal = zfit.Space('thetal', limits=...)
limit_phi = zfit.Space('phi', limits=...)
limit_bmass = zfit.Space('bmass', limits=...)

angular_obs = limit_thetak * limit_thetal * limit_phi
angular = AngularPDF(obs=angular_obs, ...)
mass = zfit.pdf.DoubleCB(obs=limit_bmass, ...)

model = angular * mass
```

With this model a set of toy studies is performed, varying in each of them the number of events while keeping the number of free parameters fixed to nine.

Fig. 9 shows the performance of the toys on a shared cluster with 8 CPUs requested and we can see that the time per toy increases slightly sublinearly with the number of events. An interesting number is the average time for toys with around 1000 events, the expected number of events to be seen in the data and therefore actually used in the real toy studies, which is around one second per toy. Both go well together with the expectations of a model fitting library. The whole example demonstrates the suitability of `zfit` to be used in non-trivial, real world analyses.

6 Beyond standard fitting

In the previous Sections, we have seen the capabilities of `zfit` for fitting a model to data. The inherent flexibility and the powerful functionality of the core parts allow for the library to be extended beyond the usual feature set of general model fitting libraries in HEP. To perform more specialized fits, the components of `zfit` can be used as building blocks of a higher level fitter. In order to keep the size and features of `zfit` at a reasonable level, the whole project is split into multiple packages that are tightly coupled to each

other. In this Section, we will have a closer look at the different packages that extend `zfit` beyond the simple model fitting we have seen so far.

The main package is the core library `zfit`, as described in the thesis up to now, that provides all the fundamental building blocks and is in itself a self-contained fitting library. Content-wise, `zfit` offers a more field agnostic selection and does not contain models and tools too specific for HEP. The focus is on a stable and solid implementation together with the API and format definitions.

More field specific content can be added in other libraries such as `zfit-physics`. This repository is meant to be the place for community contributions focusing on HEP-specific content. Guidelines, examples and automated tests are planned to be in place in order to lower the threshold for contributions. Furthermore, it will also contain functions dealing with physical quantities such as kinematics, which can serve as building blocks for models. Higher level interfaces that use `zfit` to build specific models are also intended to be placed in this repositories, for example a whole amplitude analysis framework currently under development which will be explained in the following Section.

Additionally, the project also contains libraries that `zfit` depends on but which themselves are self-contained and which are factored out of the core- and extension libraries to standalone packages. This allows other projects than `zfit` to make use of them as well. An example is the implementation of `phasespace`, which generates four momenta of particles from a decay taking into account the correct kinematics. The package will be explained in more detail in Sec. 6.2.

6.1 Amplitude fits

`zfit-amplitude` is a higher level fitting library, which is currently under active development and in its early stage, built with elements from `zfit` in order to perform amplitude analyses, including Dalitz. In the following, the extension within the scope of the `zfit` project is discussed by showing an implementation example of the decay $D^0 \to K^+\pi^-\pi^0$. As part of this, an additional library will be introduced, `phasespace`, which covers an essential part of being able to generate amplitude fits.

In order to perform amplitude fits, `zfit-amplitude` contains

- shapes such as the Breit-Wigner distribution that can be used to model resonances;
- helper classes to build PDFs such as some that efficiently cache intermediate results specific to amplitude fits, and
- a higher level interface to build amplitude fits in a transparent manner, similar to other specialised amplitude fitters in HEP. This removes the need of low level handling *but* keeps the flexibility to replace any part of it by a custom object.

In order to perform toy studies as described in 5, an efficient way of sampling from a model is needed. While accept-reject is an universal and good working way, the efficiency can be very low in higher dimensional spaces and/or with peaky model shapes. To avoid inefficiencies, samples can be produced by using importance sampling as described in Appendix C.7, which requires a distribution approximately resembling the shape of the model. In `zfit-amplitude` the approach is to use the kinematic constituents of the decay particles. They are then transformed to the desired variables that are used in the model.

This is a very general approach that allows all sorts of variables to be constructed from the kinematics. On the downside, in order to produce the particles with realistic kinematics, a phasespace generator is needed. While there is an implementation in ROOT called the TGenPhasespace class, no equivalent is available in pure Python. Therefore, a library porting the above with an extension to real world experiment kinematics and implemented using TF has been created.

More details on the `zfit-amplitude` library, especially on the higher level interface, will be provided in a future paper and goes beyond the scope of this thesis.

6.2 phasespace

The kinematics of a particle are a four-dimensional tuple representing its four-momentum. In a decay of a parent particle to lighter children particles, their kinematics are constrained by the fundamental physical laws of momentum and energy conservation. With

$$p^{0,1,2} = p^{x,y,z} \qquad p^3 = \sqrt{p^2 + m^2 c^2}$$

this reads as

$$p^{\mu}_{parent} = \sum_i p^{\mu}_i \tag{12}$$

with p_i being the four-momenta of all children particles. For a two-body decay $A \to BC$, there are six free variables from the momenta of the two children. Using Eq. 12 there are two degrees of freedom left in the decay kinematics that lead to a distribution. Furthermore, the mass is only fixed for stable particles but is a distribution for resonances. To sample from the phasespace within the `zfit` project a package named `phasespace` was created. The purpose of it is to generate arbitrary decays obeying the physical constraints discussed before and expressed in Eq. 12. Since it uses TF as the backend, it integrates seamlessly into `zfit`.

The algorithm used in `phasespace` is the Raubold and Lynch method for n-body events as described in [16]. In principle, the algorithm builds a tree where every node has two leaves representing a two-body decay. It calculates the available kinematic energy that will be assigned to the particles from the difference of the parent rest mass and the sum of all the children rest masses. The latter are either fixed or drawn from the mass distribution in case of short-lived particles

$$E_{kin} = m_{parent} - \sum_i f_i^{mass}(m_i^{min}, m_i^{max})$$

with f_i^{mass} being the mass distribution of the particle i, m_i^{min} the minimum mass recursively determined from the children masses of particle i and m_i^{max} the available energy from the top particle minus the other parent particles of particle i. The remaining E_{kin} is randomly split into fractions along all the decaying particles in the tree where each fraction is the kinetic energy for the boost of this particle.

Given the mass of each particle in the tree and its kinetic energy, particles are recursively generated starting from the top of the tree, *i.e.*, with the lightest particles. Each parent particle randomly generates the two childrens in the available phasespace. Then the whole decay tree is boosted to the parent momentum. This continues until the top particle is reached, which is not boosted by default. However, an additional argument to the

generation method allows to also boost it. This can be used to reproduce the physics happening in actual colliders.

Usage

The library has a main class `GenParticle` that describes the particles mass, either fixed or as a distribution, and has a method to set the children particle it decays to. This allows to build an arbitrary decay chain in an object-oriented way. As an example, the decay of $D^0 \to K^*(892)(\to K^+\pi^-)\pi^0$, which will be implemented as an amplitude fit in the next Section, would be implemented as following

```
import phasespace as phsp

kplus = phsp.GenParticle("K+", mass=KPLUS_MASS)
piplus = phsp.GenParticle("Pi+", mass=PIPLUS_MASS)
piminus = phsp.GenParticle("Pi-", mass=PIMINUS_MASS)
kstar = phsp.GenParticle("K*", mass=kstar_mass_func)
dzero = phsp.GenParticle("D0", mass=DZERO_MASS)

kstar.set_children(kplus, piminus)
dzero.set_children(kstar, piplus)
```

Here, `kstar_mass_func` is a function sampling from the mass distribution of a K^*. This is a custom function that can be implemented by the user.

Having specified the decay chain, two methods can be used to generate the actual particles: Either `generate_tensor` which returns the four momenta as a Tensor and can be used directly inside `zfit`, or alternatively, `generate` returns the same information but as a numpy ndarray. Internally, a TF session is called and the computation is run.

```
weights, particles = dzero.generate(1000)  # generate 1000 particles
```

The weights correspond to every single event and quantify the probability of the generated event. The returned `particles` is a dictionary containing the momentum of each particle. For example

```
kstar_kinematics = particles['K*']
kstar_x = kstar_kinematics[:, 0]
kstar_mass = kstar_kinematics[:, 3]
```

and the format of the kinematic is (number of events, components).

As a shortcut for simple decays, a high level function `generate_decay` is available which allows to describe a decay with stacked lists of masses.

6.3 Dalitz implementation

As an example of an amplitude fit, a Dalitz analysis of the decay $D^0 \to K^+\pi^-\pi^0$ is implemented using some parts of `zfit-amplitude` together with `zfit` and `phasespace`. This implementation of the resonances and their shapes is done following the analysis in Ref. [17]. The amplitude is built using the isobar approach, which calculates the coherent

sum of the individual contributions, the intermediate resonances. The implemented resonances are $\rho(770)$, $\rho(1700)$, $K^{*0}(892)$, $K^{*+}(892)$, $K^{*0}(1430)$, $K^{*+}(1430)$ and $K_2^*(1430)$

For amplitude analyses, dedicated fitting libraries exist and using a general purpose fitter like `zfit` for this case is rather special. In the following, we will go through the elements that are needed from a top-down approach and see how the problem can be separated into smaller pieces using the functionality of `zfit`. Although unimportant technicalities are left away and the example is a sketch only of the actual implementation, it is going to be rather advanced and involves functionality only explained in Appendices.

The observables in this case are the invariant masses of pairs of the different children particles, which are $m_{K^+\pi^-}$, $m_{K^+\pi^0}$ and $m_{\pi^-\pi^0}$. The whole PDF f_{tot} is of the form

$$f_{tot}(m_{K^+\pi^-}, m_{K^+\pi^0}, m_{\pi^-\pi^0}) = \left\| \sum_i^{n_{amp}} c_i A_i(m_{K^+\pi^-}, m_{K^+\pi^0}, m_{\pi^-\pi^0}) \right\|^2 \tag{13}$$

with c_i being the complex coefficient of each amplitude A_i. Expanding this term contains cross ($i \neq j$) and square ($i = j$) terms b_{ij} of the form

$$b_{ij} = c_i c_j^* A_i A_j^*$$

To implement this in `zfit`, a custom class is created that takes the coefficients, amplitudes and a `Func` called `AmplitudeProduct` which calculates the above products b_{ij}. Before we will look at these three components, we can build the global PDF as defined in Eq. 13

```
class SumAmplitudeSquaredPDF(zfit.pdf.BaseFunctor):
  def __init__(self, obs, coefs, amps,...)
    self._cross_terms = [AmplitudeProduct(c1, c2, a1, a2)
              for (c1, a1), (c2, a2)
              in combinations(coefs, amps)]

    self._square_terms = [AmplitudeProduct(coef, coef, amp, amp)
                for coef, amp in zip(coefs, amps)]
    ...

  def _unnormalized_pdf(self, x):
    value = tf.reduce_sum([2. * amp.func(x=x) for amp in self.
                                    _cross_terms]
                  +[amp.func(x=x) for amp in self._squared_terms],
                  axis=0)  # over which axis to sum
    return tf.real(value)  # convert it to a real number
```

As the next piece, we need to have the amplitude product. Since this is complex in general, it is again split into smaller parts that represent the real and imaginary parts by a class `AmplitudeProductProjection`.

```
class AmplitudeProduct(zfit.func.BaseFunctorFunc):
  def __init__(self, coef1, coef2, amp1, amp2,...):
    prod_real = AmplitudeProductProjection(amp1, amp2, proj=tf.math.
                                           real)
    prod_img = AmplitudeProductProjection(amp1, amp2, proj=tf.math.imag
                                          )
    super().__init__(funcs=[prod_real, prod_imag],
              params={'coef1': coef1, 'coef2', coef2}, ...)
    ...

  def _func(self, x)
    coef1 = self.params['coef1']
    coef2 = self.params['coef2']
        prod_real, prod_imag = self.funcs
        coeffs = coef1 * tf.conj(coef2)
    return  coeffs * tf.complex(prod_real.func(x), prod_imag.func(x))
```

Splitting the real and imaginary parts has the advantage of keeping both of them real and therefore also their integrals. This allows to make full usage of the `zfit` integration. Since these parts are *independent* of any coefficient, this allows to cache the integral value.

```
class AmplitudeProductProjection(zfit.func.BaseFunctorFunc):
  def __init__(self, amp1: ZfitFunc, amp2: ZfitFunc, proj, ...):
  self.projector = proj
  super().__init__(funcs=[amp1, amp2], ...)
  self._cache_integral = None

  def _func(self, x):
    amp1, amp2 = self.funcs
    return self.projector(amp1.func(x) * tf.conj(amp2.func(x)))

  def _single_hook_integrate(self, limits, norm_range, name):
    integral = self._cache_integral
    if integral is None:
      integral = super()._single_hook_integrate(limits=limits,
                            norm_range=norm_range,
                            name=name)
      integral = zfit.run(integral)  # simplified
      self._cache_integral = integral
    return integral
```

where we used the hook for the integration as described in Appendix C.6.

All that is left now are the coefficients, which are just parameters, and the resonances. Using the Breit-Wigner function from `zfit-physics`,[23] we can build them

```
resonances = [
  'rho(770)': RelativisticBreitWigner(rho_770_plus_mass,
                    obs=zfit.Space('m2pipi', limits...))
  ...
  ]
coeffs = [
  zfit.ComplexParameter.from_polar('c_rho770', 1.0, 0.0),
  ...
  ]
```

[23]The Breit-Wigner is currently an open merge request.

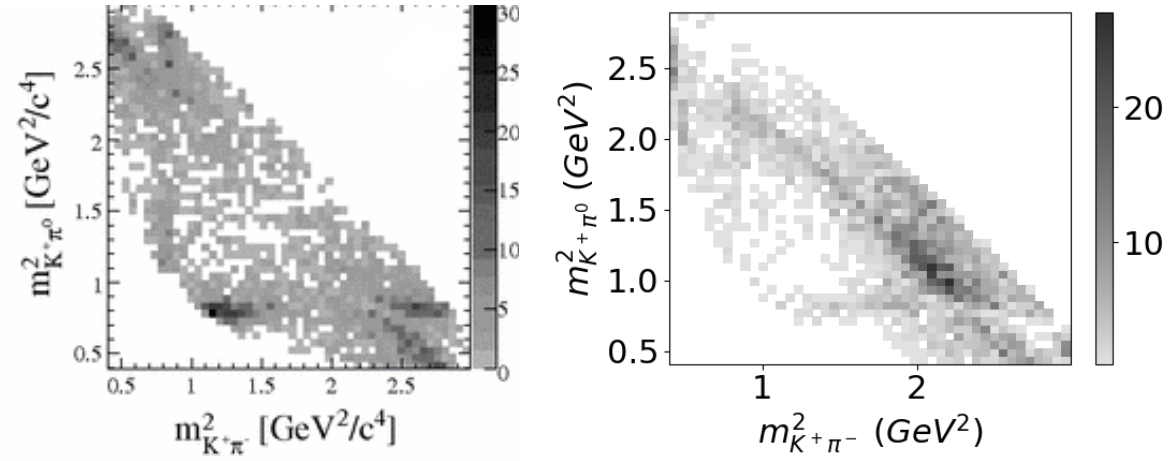


Figure 10: Dalitz plot of the invariant mass $K^+\pi^-$ and $K^+\pi^0$. The left histogram is taken from the paper while the one on the right was sampled from a distribution built with `zfit` and using `phasespace` as sampler.

and combining all of the above we can finally create the whole model

```
obs = zfit.Space(['m2kpim', 'm2kpi0', 'm2pipi', limits=...])
pdf = SumAmplitudeSquaredPDF(obs=obs, coeffs=coeffs, amps=resonances,
                                      ...)
```

This PDF is quite an accomplishment: while the implementation compared to all previous examples is more extensive, none of the above parts is *really* complicated to implement and uses basic `zfit` functionalities. Nonetheless, with this few lines of code, `zfit` extends its functionality into the world of amplitude fitters, where usually only dedicated tools exist.

A missing part is the sampling: since the kinematics of the particles is not encoded into the PDF, it is needed inside the sampling. For that, we can create a decay as shown in examples above in Sec. 6.2 with the `phasespace` package. For simplicity, the Breit-Wigner distribution is used to model the mass shape. Building the decay and sample from it can be registered with the PDF as importance sampling.[24]

In order to compare the `zfit` implementation with the measured data, a sample is drawn from the PDF. It is to note that the right plot in Fig. **??**, drawn with `zfit`, is currently not representative, since an instability in the sampling can vary the samples strongly depending on a single proposed event.[25]. While the two distributions do not agree, it is notable that resonances are taken into account and that the phasespace is sampled with correct borders. This unveils another current limitation in `zfit`, that currently only rectangular limits are possible. As a future extension, arbitrary limits for `Space` will be implemented.

What was shown in this example are the low level components to build an amplitude analysis using `zfit`. Most of the work above is straight forward but cumbersome and

[24]Since this feature is not yet fully public in `zfit`, no explicit example is shown. The mechanism will be similar to the registration of an integral.

[25]As described in C.7, weights going to zero as occurring in the phasespace can cause large problems.

can be hidden behind a higher level interface that takes care of the right assignment of observables, resonances and compositions. Furthermore, and adding an additional layer around the resonances, the `phasespace` decays can be kept with the actual resonances which allows the phasespace to be an integral part of the amplitude. A higher level interface, which leaves the user with the specification of the resonances and coefficients but still offers the possibility to replace any part that was shown above, is currently under work in `zfit-amplitude`.

7 Conclusion and outlook

`zfit` is a versatile library that fills the gap of model fitting in Python for HEP. Built on top of the deep learning framework TensorFlow, it has shown great advantages including a remarkable speedup for parallelisation. The formalisation into five loosely coupled parts and an extensive base class for custom models extend its scope far beyond the usual feature set of HEP fitting libraries.

The project has been successful so far, and is being used in several high-impact analyses. In addition, it has been shown how the `zfit` design allows to build extremely complex analyses, namely amplitude analyses, in a reasonably simple way, unlike most general fitting libraries. However, a lot of work remains to be done on the way to establish a stable, reliable fitting library. The main features to be improved on in the near future are:

Binned fits Fits are sometimes performed in bins of data, mostly to speed up the computation. Furthermore, the shape of a model can be too complicated to be described analytically and has to be deduced from simulation or data samples by creating a template PDF. Currently, there is no native support yet for binned fits or template PDFs. This is right now under active development and will be added to `zfit` in the future.

Optimisation Model fitting can be a numbers game: in order to estimate uncertainties of parameters or to study the sensitivity of a fit with toys, a large number of repeated fits have to be performed. To keep this feasible in terms of time and computing resources, performance matters. There are currently still various places where the computation can be optimised. This includes the caching of computations and more efficient numerical integration by using advanced Monte Carlo techniques or other numerical methods.

Serialisation Models are currently built within a script. Often, a model needs to be stored and used again later on or in a modified version, which is not well achieved by just dumping the code. To actually define a model, for most cases no code is actually needed but a configuration file with the model description is sufficient. This allows to change certain parts of a model and rather inexperienced users to safely build a model. Therefore, a complete serialisation of a model into a human readable format is planned for `zfit`.

Content In HEP, there are quite a few different shapes and possibilities of combinations that models are built with in order to describe the observables correctly. This includes angles, masses, incorporating smearing effects and more. `zfit` and its extensions currently don't contain a lot of different models or losses. The essential parts are contained but more are planed to come in the future. It is expected for them to be continuously added, also depending on the needs that may arise. Furthermore, with `zfit-physics` a repository especially created for content and simple community contributions is available.

Large scale Fits in HEP can be large, both in terms of data as well as in the complexity of the fitting model. With future experiment upgrades an increased amount of data is expected and a fitting library has to cope with that. Complex models and more precise measurements also increase the need for a reliable normalisation, achieved

by a higher number of random samples drawn for the integration. While scalability to medium scales is already available with `zfit`, the software should not be the limit in terms of scaling, the computing infrastructure should be. This requires that on-the-fly normalisation computation can be performed. The extension to huge data samples with out-of-core computations and to use multiple nodes as well as GPUs is also a requirement. TF supports this quite well, it was designed for that, but the explicit implementation inside `zfit` is not yet there.

Most of these current shortcomings have been foreseen and make it into the idea of `zfit` to become a stable library; a clean implementation with a minimal maintenance effort is preferred over quantitative content. Additionally, the flexibility and available base classes allow the user to add these features on top of `zfit` as they are required.

Another future challenge is provided by a significant change of the backend. TensorFlow 2.0 is currently in beta stage and expected to appear somewhere during Summer/Fall of 2019. A complete restructuring of the library is expected, including a lot of clean up. While a lot will remain the same, some work will be needed to adjust `zfit` to it.

In summary, `zfit` started not only filling an open gap in the HEP Python ecosystem but also extends its functionality through the formalisation and flexibility far beyond of what traditional fitting frameworks are able to do. While still under heavy development, the current library is already well suitable for a diversity of simple to advanced analysis.

Acknowledgements

I would like to first express my gratitude to Professor Nicola Serra for letting me do this thesis in his group and being very supportive and trusting overall.

A huge thanks goes to Dr. Albert Puig Navarro, who was not only a great supervisor but also a core member of the `zfit` project, a co-author, code reviewer, advertiser as well as main author of `phasespace` and `zfit-amplitude`, both libraries that are very closely interconnected with `zfit`. Without all the discussions about the large and small details of the library and use cases, the pure coding and reviewing contributions, this project would not be on a comparable level and may never grew over a small collection of scripts.

I would like to greatly thank Dr. Rafael Silva Coutinho who is also part of the `zfit` core team and part of bringing the project to life. His contributions in discussions with a more user sided view as well as extensive usage with real use cases is, next to code and documentation contributions, a great support in designing and creating the library.

There are a lot of members in my group that I would like to thank as well for a variety of things. Be it for discussions about the development of `zfit` and code snippets I like to specially thank Dr. Abhijit Mathad, Dr. Julian Garcias and Dr. Oliver Lantwin. For the usage and trial of the library including new features where I like to thank Davide Lancierini, Sascha Liechti and Martina Ferrillo. And last but not least for the idea of the library, the technical and user sided experience with an already existing tool and helpful advices I would like to thank Dr. Andrea Mauri and Michele Atzeni.

A specially thank goes to Prof. Anton Poluektov for several discussions and whose library `TensorFlow Analysis` was a major inspiration to build `zfit`.

Furthermore, I would also like to thank the scikit-hep community for various design discussions, especially Dr. Eduardo Rodrigues and Chris Burr.

A great thanks goes to the University of Zurich and CERN for offering the opportunities to do this kind of research by providing the adequate resources.

Many thanks also go to Google and the TensorFlow development team for open-sourcing the library and thereby allow libraries like `zfit` to be built.

Last but not least I'd like to thank two persons not only professionally for their support but also privately for various things, too many as they could even remotely be summarised here: Simone Steinbrüchel and Patrik Eschle

Appendices

A Likelihood

A reasonable loss is to have a quantity that expresses the *probability of the model given the data*, which can then be maximised. A form of Bayes theorem about probability can be used in the setting of a model parametrised under θ and data x to express the above

$$P(\theta|x) = \frac{P(x|\theta)P(\theta)}{P(x)} \tag{14}$$

It states that *the probability of θ under our observed data equals to the likelihood of finding our data given θ times the prior of θ over the prior of our data*. While $P(data|\theta)$ is a probability, fixing the data and varying θ is called the likelihood of θ and denoted as $\mathcal{L}(\theta)$. If there is no previous knowledge about θ, we can assume a uniform prior, this is usually the case. The same is done for the data, whose marginal probability acts as a normalising constant. This means they both introduce constants into our term since they do not depend on our parametrization θ.

Note that the following derivations rely on a discrete probability distribution as opposed to a continuous one. This simplifies the argumentation. It can be shown that the results are equivalently valid for the latter, therefore no strict distinction is made. We start out with a form of Bayes theorem, which states that the combined probability of two events are independent of their order.

$$P(A \cap B) = P(B \cap A) \tag{15}$$

The probability of two events to happen can be rewritten in terms of the conditional $P(A|B)$, read as *the probability of A given B*, and the marginal probability $P(B)$ as

$$P(A \cap B) = P(A|B)P(B) \tag{16}$$

Combining 15 with 16 and rearranging we get

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{17}$$

that is usually in this form famously known as Bayes theorem. It states that *the probability of A under B equals the likelihood of B given A times the prior of A over the prior of B*. While $P(B|A)$ is a probability, fixing B and varying A is called the likelihood of A and denoted as $\mathcal{L}(A)$. If there is no previous knowledge about A, we can assume a uniform prior. The same is done for B, whose marginal probability usually acts as a normalizing constant. This means they both introduce constants into our term and are therefore not of interest later on. They will be left away from now on

$$\mathcal{L}(B) = P(A|B) \tag{18}$$

Given a hypothesis H_0 and a dataset x, the likelihood is the probability that an event happened under a certain hypothesis

$$\mathcal{L}(H_0) = P(x|H_0). \tag{19}$$

While the probability itself is normalized over x, a likelihood is not. Notice that the likelihood of H_0 is a function in H_0, namely the probability of x under H_0 *with* H_0 *changing*. So the difference between likelihood and probability is the distinction of what is the free parameter and what is the parametrization. We assume now our hypothesis is described by a model parametrised with θ, a PDF as defined in 2.

There is actually a distinction between parameters of interest (POI) and nuisance parameters when speaking of parametrisation. The latter describe parts of the models shape but are not of direct interest in the sense that they do not appear in the hypothesis but rather describe well known effects like the width of some smearing. Since they also have to be inferred in the same way, we restrict ourselves in the derivation of the likelihood to models with exclusively POI. The distinction becomes apparent later in the hypothesis testing, which goes beyond the scope of this thesis.

While the likelihood denotes the odds of the model parametrised with θ under the observations x, it is, as seen above, equal to the probability density of finding x given the parametrisation θ

$$\mathcal{L}(\theta|x) = f_\theta(x) \tag{20}$$

Since the likelihood under data x is the product of likelihoods under each independent data point, we can write the likelihood as a product of probability densities as in 4

Using this expression, we can get a maximum likelihood estimate of our parametrisation θ

$$\hat{\theta} = argmax(\mathcal{L}(\theta|x)) \tag{21}$$

In practice, finding the maximum is done using numerical methods. Eq. 4 involves the product of many small numbers <1, not feasible for most computers given their limited precision on numbers. The monotony of the logarithm can be used to transform the probability densities to log densities whereby the multiplication becomes an addition

$$argmax(f(\theta|x)) = argmax(ln(f(\theta|x))) \tag{22}$$

It is often more convenient to find a minimum instead of a maximum, so that our likelihood takes the form as in 5

which is our NLL. Therefore, our likelihood estimation becomes

$$\hat{\theta} = argmin(-\sum_{i}^{n} ln(f(\theta|x_i))) \tag{23}$$

This estimation is valid for n measured data points with weight one. We can generalize this expression, including event weights. They introduce a power factor but can be taken out of the logarithm to have a simple multiplication

$$\hat{\theta} = argmin(-\sum_{i}^{n} w_i \cdot ln(f(\theta|x_i))) \tag{24}$$

with w_i being the weight of the i^{th} event. This generalization is not only useful when using data points directly to build an unbinned NLL but allows straightforward to bin

the data in the first place and use the bin height as the event weights by choosing an appropriate x_i for each bin. Binning can speed up the computation significantly but looses slight precision[26].

We considered now the likelihood of a model fit to a dataset. The same parameters often occur in more than one model, for example the invariant mass of a mother particle. To take this into account, a simultaneous fit can be performed, which is simply the multiplication of several likelihoods.

$$\mathcal{L}_{f(x)}(\theta|data_0, data_1, ..., data_n) = \prod_i \mathcal{L}(\theta_i, data_i) \tag{25}$$

where θ_i is a subset of parameters of θ used for the model fit to $data_i$.

The likelihood can be multiplied by constraint terms, most notably an extended likelihood estimator (EML) and parameter constraints.

If a parameter changes both shape and the overall normalization of the pdf, an EML fit is superior. This situation typically arises if there is a sum of several shapes as when adding the background and the signal shape. So for example, taking the model $N_{sig} * PDF_{sig} + N_{bkg} * PDF_{bkg}$ and assuming a Poisson distribution of the number of events in the data, we can multiply the likelihood by a term

$$\mathcal{L}_{extended} = poiss(N_{tot}, N_{data}) \tag{26}$$

$$= N_{data}^{N_{tot}} \frac{e^{-N_{data}}}{N_{tot}!} \tag{27}$$

where $N_{tot} = N_{sig} + N_{bkg}$. Therefore the EML looks like

$$\hat{\theta} = argmin(-\sum_i \cdot ln(f_i(\theta|data_i)) - \sum_i ln(poiss(N_i, N_{data_i})) \tag{28}$$

where $data_i$ are, as before, different datasets. The weights are inside $data_i$ and taken into account as described above. The total number of events is, in general, the sum of the weights of all the events.

For certain parameters, prior knowledge is available. If the order of magnitude of the knowledges uncertainty is within the expected fitting sensitivity, a constraint term can be used to incorporate this. This is nothing else than an additional term the likelihood is multiplied with

$$\mathcal{L}(\theta) = \mathcal{L}_{unconstrained} \prod_i f_{constr_i}(\theta) \tag{29}$$

$$= \mathcal{L}_{unconstrained} \mathcal{L}_{constr} \tag{30}$$

as an example, a parameter θ_i that is Gaussian constraint with μ_{constr} and $\sigma constr$ looks like this

[26]Currently, only unbinned losses are implemented in `zfit` but the extension to binned is already tested and will be there in the future.

$$constr_i = Gauss(\theta_i; \mu_{constr}, \sigma constr) \tag{31}$$

Combining equations 25, 26 and 29 yields for the likelihood in general

$$\mathcal{L} = \mathcal{L}_{f(x)} \cdot \mathcal{L}_{extended} \cdot \mathcal{L}_{constr} \tag{32}$$

The absolute magnitude of the likelihood itself does not have any specific meaning, but ratios and scanning over the parameter space of θ allow for statistical interpretation. This however goes beyond the scope of `zfit` and is intentionally left to other packages like lauztat [18].

As mentioned in the beginning, a likelihood as described in 32 is not the only possibility to define a loss, though the most common used one. A prominent example is the χ^2 loss, which is equivalent to a likelihood in the limiting case if each point is normally distributed.

B Backend

Choosing the right computational backend for model fitting is crucial. In the following Sections, the different paradigms and backend designs are introduced. Furthermore, the TF library and how it is wrapped inside `zfit` is explained.

B.1 HPC and paradigms

High performance computing (HPC) solves an entirely different problem than high level programming languages do. While the flexibility offered by Python is nearly unlimited and an incredible strong and comfortable feature, it is only possible due to the dynamic nature of the language. For the interpreter of the language that actually translates and executes the code to machine level, nearly no assumptions can be made about any object. Neither about their size or type nor about their future in the code. This means that no previous optimization is possible. For HPC this is the crucial key to success: the more is known *previously* of the actual computation, the better the performance will be. The less flexibility is available, the more assumptions can be made. This way the layout of arrays, parallelisation of computation, caching and more can be efficiently achieved. Since less flexibility is not an overall desired characteristics, a good compromise has to be found. The distinction between two fundamental paradigm are of importance

Imperative Most code is run imperatively: a statement is executed when the line is hit. Python, C++ and many more work that way. The advantage is that the state changes as the code runs. An addition of two numbers for example will be executed right when the command appears. The disadvantage is that it's impossible to optimize over more than one step, since the next line of code is not known yet.

Declarative/Graph When a statement is supposed to be executed, it is *actually* not yet run but somehow remembered. It runs either explicitly when asked for it or implicitly if it is needed as a dependency. The latter could also be described as "lazy evaluation". The disadvantage is that the state of the code may looks quite undetermined since for example an addition of two numbers is not yet actually

executed when the line is hit. Instead, an object that actually calculates the computation is usually returned. The paradigm can also be described as a "graph" based approach, since an execution graph is built. In the special case of HPC, this can be a computational graph displaying data input, operations and outputs. While a declarative paradigm is better in terms of raw performance and optimizations in general, it is also more limited in functionality and cannot offer the flexibility that the imperative paradigm offers. The workflow is divided into *compilation* or *graph construction* time and *run* time.

In reality, a mixture of this approaches is usually applied on different levels which some languages and frameworks tending strongly toward one or the other. For example, any ahead-of-time compilation as used in a C/C++ compiler is kind of a declarative step. But the language C/C++ itself is imperative. For deep learning frameworks, there is a strong tendency towards a declarative paradigm, more specific a graph is built. To ease the difficulty of debugging and making experimentation simpler, an immediate execution of a graph can make a graph based approach behave imperatively. Some frameworks therefore offer a mix with this immediate execution. For the maximal performance, a graph based approach will though be superior for decent complex problems. There are three major advantages of using a graph.

Distributed computing Moore's law states since over 40 years that the processing power, which originally equals to the clock speed, of CPUs will double approximately every 1.5 years. While there is no theoretical foundation for this statement, reality holds up to it. Since the beginning of the 2000's, CPUs have reached a clock speed around gigahertz and started hitting a limit: the thermal dissipation of a CPU goes with the third power of the clock speed. As a consequence most CPUs today don't have a higher clock rate in order to stay efficient. Nonetheless, CPU power keeps increasing at a similar rate by using more than one CPU unit in parallel. Multicore systems became common on almost every machine. It is an imperative therefore that any numerical intensive library makes a good use of distributed computing. This includes shared memory machines to large scale clusters with multiple nodes. Moreover, with the recent success of ever growing DNNs and frameworks built to make easy use of them, vectorial computing devices like GPUs can be used easily as an alternative to CPUs.

Mathematical optimization Since there is not only one step for each computation available but the whole computation at once, the possibility for mathematical optimization arise. Most notable, an analytic expression for the derivative is available through automatic differentiation. This subsequently applies the chain rule to any operation.

Caching A throughout analysis of the computation graph allows for various computational optimizations. Common sub-expression elimination identifies if the same operation is executed in several places and substitutes the nodes for a single computation.

The disadvantage of building a graph first is the overhead to build, analyse and store the graph. And the additional indirectness, since the main principle is to build the

computation graph once and run it several times. This implies a non-intuitive behaviour for users coming from an imperative background.

While the graph based approach seems very feasible, there are two fundamental different assumptions that need be made with graphs.

Static A static graph is built once and *cannot* be altered. So if we have for example a complicated function taking as input a uniform value generator, we build the graph once and execute it several times. In order to change the input to a normal distributed random generator, this requires to build the whole graph again with one operation, the random input generator, changed.

Dynamic Dynamic graphs are mutable and can be used if there are a lot of actual modifications to the graph. Any part can be changed, its not restricted to append-only.

As an example, TensorFlow offers a static graph while PyTorch uses a dynamic graph.

B.2 Working with TensorFlow

Graph based approaches imply another level of indirectness and need some wrapping in order to avoid unexpected or inefficient behaviour. In `zfit` this is mostly hidden, exposing some of it but also removing the obstacles for most use cases.

As an example, we'll look at a task of several random number generations in a row. In the imperative style this would look like this, where numpy is used to generate a uniform distribution

```
for _ in range(n_samples):
  sample = np.random.uniform(...)
  # do something with the sample
```

In a declarative approach as with TF, the same is achieved by *first* building the actual operation of the graph and *then* executing the computation

```
sample_op = tf.random.uniform(...)
for _ in range(n_samples):
  sample = zfit.run(sample_op)
  # do something with the sample
```

where `zfit.run(...)` is just the command to execute a certain part of the graph. The object `sample` is in both cases a numpy array. Note that in the declarative approach, we created the operation only once but *execute* it multiple times. This indirection can be surprising for the unaware and a typical mistake is to implement it this way

```
# BAD EXAMPLE
for _ in range(n_samples):
  sample_op = tf.random.uniform(...)
  sample = zfit.run(sample_op)
  # do something with the sample
```

which just fills up the graph with additional operations that actually all do the same thing. In order to hide this inconvenience to the user, a caching system is in place in `zfit` to prevent an accidental re-creation of the operation. In the above example, we can think of wrapping a function `uniform` (*fictive example*), create the operation on the first call and use the cached operation afterwards. In pseudo code, this would look like the following

```
cached_op = None
def uniform(...):
  global cache_op  # make "cache_op" assignable
  if cached_op is None:
    cached_op = tf.random.uniform(...)  # create op
  return cache_op
```

This function can then be used a mixed stile

```
for _ in range(n_samples):
  sample_op = uniform(...)  # as created above
  sample = zfit.run(sample_op)
  # do something with the sample
```

This hides some of the difficulties. While this example of just random number generation may seems artificial, a prominent example is the loss that is typically built using models and data. Calling `value` builds the operation, adds it to the graph and stores it, so that in subsequent calls, the stored operation is returned instead of a new one created. While this is very convenient for the user combining the good of two worlds, it comes with an additional burden for `zfit` of having to cache the operations efficiently.

B.2.1 Caching

In HPC, sometimes certain parts of a computation do not need to be recomputed and storing the results in a cache for later reuse can improve the performance significantly. Since `zfit` uses a declarative graph based approach, there are two different caches: For the operations built on construction time as discussed above and only appearing due to the graph based approach. Furthermore, intermediate calculations can be cached in general, which corresponds to the objects built on run time. The latter is commonly used also in imperative approaches.

As seen before, creating operations with a declarative approach adds each of them to a global graph, a collection of operations. And since these are only instructions and not actual computations, there is no need to rebuild the operations and stitching them together again but rather re-use the previously built instruction.

Therefore `zfit` has a possibility to cache Tensors after they have been built for the first time inside the object that the Tensor was created in. Since the graph is static and cannot change, an even small modification to a part of it requires a complete rebuild[27]. This can be as less as removing an single, additional term in the model. Any object that may needs to perform such a modification is therefore registered within the caching object and can notify the cacher in order to invalidate its cache and rebuild any Tensor.

[27]There are ways of changing the *value* but not the logic, the computation flow.

Numerical results of computations inside the graph can remain the same during several executions of an operation which can therefore be cached. Opposed to the construction time caching, numerical caches do also invalidate if a number inside the graph changes, not only its structure. Caching in a declarative approach is not straightforward since this implies to replace a node, which can be a large subgraph by itself, with a single value. Since the graph cannot be changed, this would require a complete rebuild of the graph, rendering the caching way less efficient. There are two ways to circumvent this problem

feed_dict The TF sessions `run` method offers the possibility to overwrite specific nodes with a value using a feed_dict. This means that cached values can simply be stored and overriden on runtime. At the current stage, `zfit` is not designed to always control the execution but leaves the freedom to the user to invoke the `Session.run` method themselves.

Variable Since all TF object so far are just operations but not numbers, storing a value between the runtimes requires a special object. A TF `Variable` can be used for this case. Its purpose is to store a value but also act as a node in the graph with a read operation that return the current value. Its value can be changed between the runtimes in case it has to be updated. This though can have side effects on other object that still use this cache. The disadvantage here is that the initialization of the Variable can be tricky if it is created inside a control flow such as `tf.while` or `tf.cond`.

In `zfit` currently a feed_dict independent implementation is being tested, but that is likely to change in the future. One main problem with this kind of caching is the gradient. If a value is not supposed to change, then its simple since there is no gradient. But if there is a value that *can* change, the gradients value also needs to be cached. A likely solution is to wrap the ordinary Tensor class from TF and provide a caching Tensor, that automatically takes care of gradient caching as well. Also the feed_dict seems like an efficient solutions, at least for cases without a gradient.

C Implementation

C.1 Spaces definition

A `Space` is either initialized through observables *or* axes and *maybe* also has limits. It *can* have both, observables and axes, which means there is an order-based, bijective mapping defined between the two. In general, a `Space` is immutable and adding or changing anything will always return a copy.

When a user creates a `Space`, observables are used and define with that the coordinate system. Once a dimensional object, as a model or data, is created with a `Space`, the order of the observables matters. Since the `Space` at this point only has observables and does not yet have any axes, when the dimensional object is instantiated, the axis are created by filling up from zero to $n_{obs} - 1$. This step is crucial and defines the mapping of internal axes of this dimensional object to the externally used observables. In other words, every dimensional object has *implicitly* defined axes by counting up to $n_{obs} - 1$, assigning observables creates a mapping by basically enumerating the observables.

For example we assume a model was instantiated with a `Space` consisting of some observables and data with the same observables but in a different order. Now the assignment of observables to the model and the data columns are fixed, therefore it is well defined how the data has to be reordered if it is passed to the model.

While we used data and a model in the example above, the same is true for limits that can be used to specify the bounds of integration and more. Since limits can be part of a `Space`, the reordering is done automatically if the order of the observables or axes is changed, not in-place but in the returned copy of it.

To help with the accounting of dimensions, `Space` can return a copy of itself with differently ordered observables. Internally of the `Space`, the axes, if given, and the limits, if given, are reordered as well. This is crucial in input preprocessing for any dimensional object since with that each `Space` is ordered in the same way as the observables of that object.

A subspace can be created from a `Space`. This is a subset of the dimensions of the `Space`. It can be used for example if a model is composed of lower dimensional models. This is often the case for functors such as a product PDF as described in Sec. C.6.6.

C.2 General limits

Simple limits are just tuples. But a more general format is needed to express multiple limits and higher dimensions in a straight forward way. Multiple limits are technically done with multiple tuples of lower and upper limits for each observable. The `Space` is handled as *one* domain. So the area of a `Space` is the sum of all the areas of each simple limit, the integral over a `Space` is the sum of integrals over each simple limit and so on. Multiple limits are defined separately and are not built from the projections of all limits. The format is therefore to specify the lower limits and the upper limits in each dimension as a tuple. Multiple limits contain multiple lower limit tuples and multiple upper limit tuples.

As an example, a `Space` is created with the observables `x, y` and the two limits l_1 and l_2

$$l_1 = (x_0, y_0)\ to\ (x_1, y_1)$$
$$l_2 = (x_2, y_2)\ to\ (x_3, y_3)$$

to write the limits in the right order, the upper and lower limits have to be separated and concatenated, so that the lower and upper limits look like this

$$lower = ((x_0, y_0), (x_2, y_2))$$
$$upper = ((x_1, y_1), (x_3, y_3))$$

By definition of the format, *lower* and *upper* have to have the same length. The limits for the `Space` is the tuple $(lower, upper)$. Creating this `Space` with `zfit` is done as follows

```
lower = ((x0, y0), (x2, y2))
upper = ((x1, y1), (x3, y3))
limits = (lower, upper)
multiple_limits = zfit.Space(obs=["x", "y"], limits=limits)
```

The same format is returned by the property `Space.limits`. This is a quite general format that covers the needs for rectangular shaped limits. However, more advanced shapes may be necessary, see also 6.1, which will most probably be provided in the future.

Since the order of observable matters and `limits_xy` and `limits_yx` as used in Sec. 4.1 define the same domain (apart from the order of the axis), they can be converted into each other using the `Space.with_obs` method

```
limits_xy_resorted = limits_yx.with_obs(limits_xy.obs)
limits_xy == limits_xy_resorted  # -> True
```

Notice that this created a new `Space` and left `limits_yx` untouched. This method can also be used to only select a subspace

```
limits_x == limits_xy.with_obs("x")  # -> True
```

and therefore go to lower dimensions again.

C.3 Data formats

Currently, the following formats can be read by `Data`.

ROOT The standard file format used in HEP analysis. It efficiently stores data samples and is the native format of the ROOT library. Due to the recent development of `uproot` [19], it is possible to load these files in Python *without* an installation of ROOT.

Pandas DataFrame Pandas [20] is the most extensive data container in Python used for data analysis. It provides DataFrames that offer an extensive set of data analysis tools going from plotting to feature creation and selections. It is the de-facto standard in Python and has the ability to load from a variety of data formats including hdf5, csv and more. The possibility to load them directly into `zfit` is therefore a powerful feature because it allows to do any preprocessing in DataFrames. `Data` can also be converted *to* DataFrames, which allows to load for example from ROOT files into `Data`, convert to a DataFrame, apply some preprocessing steps and then load again into `Data`.

Numpy Numpy [21] is the standard computing library in Python that has been around since a long time. Several libraries, including TF, are inspired by its API and behaviour design. The numpy arrays are the default way to handle any vectorized data and are also returned by TF as a result of computations.

Tensors `Data` can also take a pure Tensor as input. While this may seem at first glance the obvious thing to do, it is trickier: a Tensor is, compared to the other data types *not* fixed per se, since it is only an instruction to compute a certain quantity. While constants for example behave straight forward and will always return the same, a `Data` initialized with a random Tensor will produce different data every time it is called. Therefore, special care has to be taken for this case, from the developer as well as from the user site.

C.4 Data batching

Small datasets are internally simply converted to a Tensor and attached to the graph. Large datasets though, which either exceed the memory limit of the computing device or the limit of the graph size (which is 2 GB), need to make use of batched out-of-core computation. TF has a data handling class `Dataset`, which provides a performant way to do batched computations. It incorporates the loading of the batches from disk into the whole graph as several operations. This allows the runtime to split the execution in order to asynchronously load a batch of the data and run the graph of an already loaded data batch.

Another way of on-the-fly computation can be more generally be done with Tensors, since they can be used to instantiate `Data`. For example, `Data` has a subclass `Sampler`, which is specialised on this. It allows to evaluate the Tensor and store its value. This way, the Tensor is only re-evaluated when requested. The `Sampler` acts for example as the returned `Data` when sampling from a model. This data depends on the model but can be used like a normal data for example to construct a loss.

C.5 Dependency management

The graph built by TF can be fully accessed, the parent operations of any operation is available. This enables to detect any dependency by walking along the graph. Or as TF does internally, to create the gradient. Using this in general can be risky though since for example caching with a `Variable` changes how the *actual* graph looks like. Furthermore, it is also time consuming on a larger scale. Inside `zfit`, it is used as an additional feature to figure out dependencies automatically of certain subgraphs. There is one type of independents that can change as also described in 4.3.1, `Parameter`, that other objects can depend on, directly or indirectly. To have a dependency structure that is independent of the graph, `zfit` has a `BaseDependentsMixin`. A subclass implements a method of returning the dependents for itself. This can then be done recurrently up to the independent parameters (see also 4.3.1) that return themselves. Any major base class implements the appropriate functions but requires for example to have a distinction between a model that depends only on parameters or also on other models. Both of them are `Dependents` but the model has to be aware to not only extract dependents from the parameters but also from the models.

To get the dependents from any object, `get_dependents` can be used, which returns a set of independent `Parameters`. This is fundamentally different from the `params` that each model has. The former will return all the *independent* parameters that the model depends on. This can be any number of *independent* parameters. The latter returns the exact parameters that are used in the function defining the shape of the model. For

fitting, when tuning the parameters, the `get_dependents` should be used, since the actual changeable parameters matter. When reading off a value from a model, like the mean by accessing `mu`, the `params` has to be used. Using the latter for fitting can easily result in an error if not all of the parameters are independents, since the value of dependents (including constant) parameters *cannot be set*.

C.6 Base Model

From all the classes, the `BaseModel` and therefore also its subclasses, `BaseFunc` and `BasePDF`, contain the most logic. The implementation has a few peculiarities that will be highlighted. It is meant to be used as a base to implement custom models providing flexibility but ease of use at the same time, as discussed in Sec. 4. Therefore in this class a structure is provided where everything can be directly controlled *but does not have to be*. The class takes care of anything that is *unambiguous* but maybe cumbersome to do while leaving the full control to the user. The intended usage as a base class leaves it to the user to implement himself any method he wants to control directly. Namely the class provides the guarantee that any of the main methods can be overriden by changing the implementation of `_method`, the same method name but with a leading underscore. Furthermore, any direct control can always be given back to the class by simply raising a `NotImplementedError`. The class acts as if the overriden method was never called. Furthermore, the base class takes care of some unambiguous handling of arguments (see below e.g. C.6.3, C.6.4). It is mandatory to decorate each _method with the `supports()` decorator. This allows to specify if the method can handle certain things, like normalization ranges. By default, this filters the more advanced arguments and handles them automatically. It is meant to provide a way to still allow specific implementations and workarounds.

C.6.1 Public methods

The internal logic of the methods `pdf` (and to some extend `func` and `unnormalized_pdf`), `sample` and all the `integrate` have a very similar layout. Their logic is split into a public part and more internal methods. We'll start with the outermost method, the public methods, and follow the subsequent method calls. There is a strict order on what will be called after which method, we follow the same order here.

A public method is a method starting without an underscore. It serves as the entry point to a model, asking it to do something. It is supposed to provide a clean API to the user as well as appropriate documentation of the method. The functional responsibility of the method is to clean the input which mostly means to take care of the ordering of dimensions and automatically convert certain input to a more general format. The following cleaning is done to the input

limits Limits for norm_range or for sampling and integration can be given as arguments to certain methods. First, the limits are automatically converted to a `Space` *if it's save to do so*. Namely a simple limit consisting of a tuple in a one dimensional case is converted, anything else raises an error. Second, the method takes care of sorting the space in the right way. Since each model has a `Space` with observables assigned to it with a given order, the given or auto converted `Space` is sorted accordingly. This assures that the limits are internally in the right order.

data As for limits, the data is first converted to the right format, a `Data` *if possible*. This will convert any input data that coincides with the dimensionality of the model. While this is convenient since it allows to directly feed numpy arrays and Tensors to a model, it relies on the correct order of both the data as well as the model observables. Since this can silently lead to mistakes by using the wrong order, probably in the future this won't be possible anymore, given that a simple conversion to `Data` can always be made.

As a second step, the `Data` is sorted according to the observables of the model, just like the limits. This ordering is done using context managers that revert the reordering once the data exits the method.

The ordering is a crucial element to allow the direct usage of any object inside a model while having the matching ordering guaranteed.

C.6.2 Hooks

After the public method, two hooks follow. Assuming we have a public method named `_method`, the first hook is named `_single_hook_method`. Its purpose is to be directly called from the public method *or* if the model itself needs to call one of it's own methods. The second hook called by the first is `_hook_method` and is used for repeated calls by the very same method that was called. This is useful if for example a scaling factor is supposed to be applied at the end of the calculations. If a method calls itself recursively, the scaling is supposed to be applied just once at the very end. The general idea of hooks is to provide a convenient way to change the behaviour of a method *without* altering the public method and its input cleaning. It provides the user the possibility to directly change any input before it is passed further down or any output right before it is returned. So any kind of advanced, model specific pre- or post-processing or setting of internal flags can be handled here. Some implementations inside `zfit` make use of this. For example the implementation of the exponential shape uses the exponential-shift trick[28]. This requires to determine the shift before the actual computation method is called. Whatever modification is applied in a hook, it is important that any hook always calls it's parent method to allow stacking hooks.

C.6.3 Norm range handling

Following the hooks, `_norm_method` is invoked. It only exists if a normalization range is used inside the function. This methods responsibility is to automatically take care of the *normalization logic* if the underlying function does not handle it itself. A `NormRangeNotImplementedError` can be raised by any deeper nested function which is caught here. For example in the case of an integral, the method first calls the subsequent method and if it catches an error, it splits into two calls: it calculates two integrals, each one *without* a normalization range, one over the limits to be integrated over and one over the normalization range. The first is then divided be the second, which is the very definition of a normalized integral.

[28] A normalized exponential shape is invariant under translation. This can be used to increase the numerical stability by avoiding large number calculations and keep it around zero.

To illustrate this behaviour in pseudo code, a function `integrate` that takes the limits and the normalization range as arguments is assumed to exist. `False` as argument to the normalization range means the calculated integral is unnormalized.

```
try:
  integral = integrate(limits, norm_range)  # pseudo method that
                                            integrates
except NormRangeNotImplementedError:
  integral_unnormalized = integrate(limits, False)
  normalization = integrate(norm_range, False)  # integrate over
                                                norm_range
  integral = integral_unnormalized / normalization
```

This allows for a user to implement only a simple integral when overwritting or registering without the need to worry about its normalization. For most integrals this would anyway end in two times calling the integration function itself, basically what was done above. The default behaviour is that the normalization range will be automatically handled, as described in Appendix C.6. But it still leaves room for the possibility to implement a method that handles the integral as well as the normalization of it. There are special cases where this can be achieved with less computations than two times calculating the integral, such as in the case where the normalization range equals the limits.

C.6.4 Multiple limits handling

Next in the call sequence the method `_limits_method` is invoked. It has the responsibility of handling multiple limits which are described in 4.1.1. Equivalently to the way norm_range is handled in C.6.3, multiple limits are caught here as well. Consecutive methods are called inside a `try-except` block in order to catch any `MultipleLimitsNotImpementedError`. If handling multiple limits is well defined, `_limits_method` is supposed to take care of it. As an example, the implementation of integration with a limit of n limits is to split them into n independent `Space`s, each with only one limit, and call the following methods with this new `Space`. It is in then a simple matter of summing the results.

Given some multiple limits as a `Space` to be integrated over by invoking the pseudo `integrate` function, the implementation looks like this

```
try:
  integral = integrate(limits)
except MultipleLimitsNotImpementedError:
  integral = 0
  for limit in limits.iter_limits():
    integral += integrate(limit)
```

C.6.5 Most efficient method

In `_call_method`, the actual functions are invoked. There are usually several choices for which function to invoke depending on availability and efficiency. The order chosen therefore starts with the most efficient implementation. If that raises a `NotImplementedError`, the next method is called.

- First the `_method` is called and returned if successful. This is a method that by default raises a `NotImplementedError` but can be overwritten by the user. It is guaranteed to be executed in this case making the public `method` call seem like a call to `_method`, module input cleaning and limit handling. This asserts the full level of flexibility in a model: any major function can completely be overwritten by this procedure.

- If no explicit method is implemented, closely related alternatives are invoked. For example, if `log_pdf` was called and `_log_pdf` is not overwritten, `_pdf` is invoked and if implemented, its logarithm is returned. As an other example, if `integrate` was called and `_integrate` is not implemented, an analytic integral calculation is performed. If that is not available and also raises an error, more expensive alternatives are called.

- The freedom of `_call_method` is to try simple, but maybe failing alternatives to return the desired. When all of the above fails, the `_fallback_method` is invoked. This is the last resort and may be quite a complex function. The calculations that it returns may be expensive but the method is guaranteed to work. It must not fail if the `Model` is implemented correctly.

C.6.6 Functors

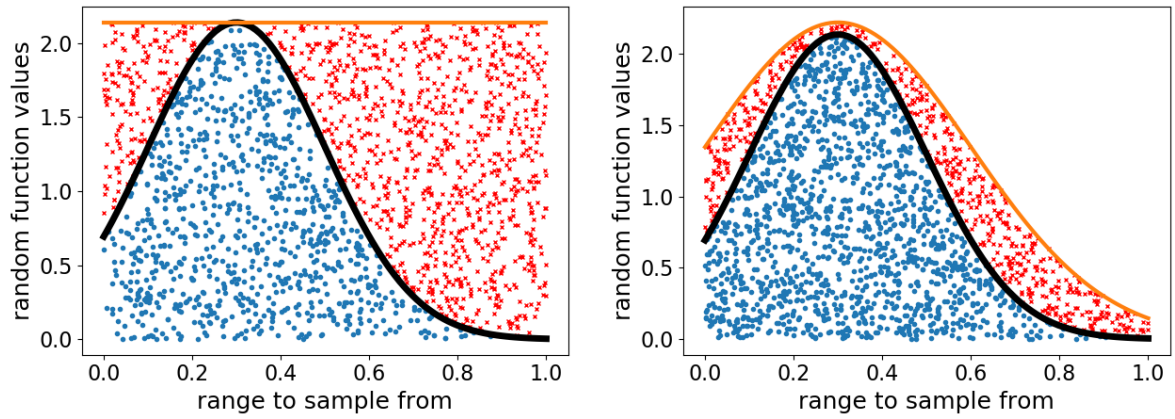
To implement a function just depending on data, the normal base class as described above is sufficient. But a model can also depend on other models. Creating compositions, as for example the `SumPDF` from Sec. 3, requires an additional tweak regarding the dependencies: a functor does not only depend on its own parameters but also on the sub models parameters associated with it. This is automatically taken care of by having a functor base class. Compared to normal models, they often don't need to define their observables but are inferred from the sub models.

C.7 Sampling techniques

Implementing a sampling from an arbitrary distribution is not a straight forward task. The generation of uniformly distributed numbers is comparably simple and is used as a basis for any more advanced sampling technique. Two major ways are often used to sample from a distribution. If the inverse analytic integral function is known, then this can be used to transform a uniform distribution to the desired distribution by treating it as the sample on y. The inverse returns values proportional to the target distribution. This method is very fast and efficient since every drawn event is used. The downside is that it requires the inverse analytic integral, which is not available for most shapes, especially custom ones.

For a model where only the shape and no integral is known, the accept-reject method can be used. Thereby, samples are randomly generated in the model domain and evaluated. A random number is drawn between 0 and the maximum of the target shape for each event. If the value returned by the model is larger than the random number, the event is accepted. Otherwise it is rejected. The technique is illustrated in Fig. 11.

This can be very inefficient though for peaky distributions since all red values are lost. An increase in efficiency can be achieved by sampling from a distribution that follows



(a) (b)

Figure 11: Visualization of the accept reject method. Proposed events are randomly sampled in the valid range. In a) a uniformly sampled y value and in b) a Gaussian shaped y is used to either accept or reject them. The black like is the true shape of the model. The orange line represents the distribution the y were drawn from. Blue values are accepted, red are rejected.

better the target shape, so called "importance sampling", as shown in 11 on the right. It needs to take a little bit more into account, namely

target This is the distribution we want to approximate.

probability The target probability is the function value of the target shape at a given x. While called probability, this does not have to be normalized to anything but be proportional to a real probability.

proposal The proposed sample is the events that will be either accepted or rejected. They are drawn from the proposal distribution.

weights This is the probability (or proportional to it) of an event from the proposal being drawn from the proposal distribution.

rnd A random number drawn uniformly between 0 and 1 to decide whether to accept or reject an event.

To approximate the target, a sample is drawn. To accept or reject samples, the following is checked

$$accept = prob_i < weight_i \cdot rnd \tag{33}$$

This is only unbiased if $all(prob_i < weight_i)$. Otherwise the distribution will be misshaped as shown in Fig. 12

Therefore to be unbiased, the weights have to be scaled enough. This can though lead to problems if the weight is significantly lower (even if only in a single point) than the target probability. Since this requires a large rescaling, the sampling of the rest of the target gets rendered inefficient. Therefore it is important that the proposal distribution matches the target reasonably well. Most importantly the maximum and minimum of the ratios of the two distributions should be as close as possible.

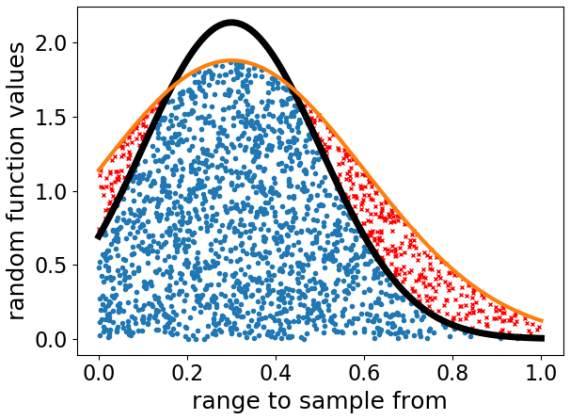


Figure 12: Importance sampling with a wrong scaled weight. The sampled Gaussian distribution (blue) is cut on the top and does not resemble the correct shape.

C.8 Loss defined

The following parts are provided by a loss to enable minimisation

value The actual value of a loss is needed since it is the desired object to be minimised. The method `value` returns the Tensor that can be run with `zfit.run(...)`. This Tensor contains typically the heavy computations.

parameters The loss depends on one or more parameter that is floating. They are associated with models and change the shape thereof. As for a model, all dependents can be retrieved by using `get_dependents`. Changing their values affects the value of the loss.

gradients Calling `gradients` returns the gradients of the loss with respect to the parameters. Gradients provide a helpful tool for the minimisation algorithm.

D Performance studies

D.1 Hardware specification

The measurements are either performed on CPUs only or on an additional GPU. The following hardware and software stack has been used for Fig. 5.

CPU 12 core Intel i7 8850H with 2.60 GHz, 6 cores are virtual using hyper-threading. The available shared memory is 32 GB RAM, which was never even half-way filled. While hyper-threading can be very useful for applications where the bottleneck is not the actual computation by the CPU, for HPC this is often not the case. As the experiments have shown, there is only a minor difference between using 6 or 12 cores. Therefore, only 6 cores, the physical ones, are used in order to quantify the speedup correctly.

GPU Mobile Nvidia P1000 with 4GB RAM. It contains the same processing unit as the consumer GTX 1050 series but is for professional usage and performs more efficient float64 computations.

It is notable that the price of the GPU and the CPUs are roughly the same, which allows for some kind of comparison between them.

For Fig. 6, a cluster server with varying hardware was used. Eight cores were requested for the studies, though the workload of other jobs and the CPU type may have an impact on the results.

The tests were performed with the TensorFlow version 1.13. It was pre-built by anaconda and uses the MKL library. There is also a version with the Eigen library available, tests revealed differing performances for different tasks, around a factor of two in time. For the GPU version, CUDA 10.0 with cudnn was used.

D.2 Profiling TensorFlow

Code consists of parallelised and serialised parts. While the speed of the former scales[29] with the number of cores, the latter does not. The total execution time t is given by

$$t = \sum_i^{n_s} t_s^{(i)} + \sum_i^{n_p} (t_p^{(i)}/n_{cpu} + t_o^{(i)}). \tag{34}$$

where n_s and n_p are the number of *serial* and *parallel* parts, respectively. $t_s^{(i)}$ refers to the execution time of the ith serial part, the $t_p^{(i)}$ for the parallel parts if executed serial and $t_o^{(i)}$ denotes the overhead that is needed for each parallel execution.

The serial part consists of

- Reading in data from disk.
- Setup code such as building a model in `zfit`.
- Global operations such as reductions on all values. For example determining whether a stopping criteria such as the sum of all gradients has gone below threshold is a serial operation.

while the parallel time $t_{parallel}$ contains usually the heavy computations: evaluating a function on data whereby the data can be split amongst the cores. The overhead for the parallel execution time includes

- the overhead of creating a new thread for the parallel execution.
- the time to move data between the CPUs or even to the GPU.

The serial time t_{serial} consists of

- Bottlenecks in I/O or moving data

In order to achieve maximum performance and minimize t

[29] Ideally. In reality, cores

	1 CPU	6 CPU	GPU
1 x problem	1.0 sec	0.27 sec	0.093 sec
12 x problem	13.4 sec	3.3 sec	1.0 sec

Table 1: Execution time measurement of a loss-like function execution. The complexity of the problem is scaled by n times adding the same loss again to the reduce function.

- there should be as few serial code execution time as possible. This is though heavily limited by the logic and a *certain* amount will always be there.
- a minimum of splitting into serial and parallel parts should occur, since each add a constant t_o term.

This two points are often heavily conflicting and end up with the simple equation to describe when to parallelise

$$t_s^{(i)} - t_o^{(i)} > t_p^{(i)}/n_{cpu}$$

which reveals that even for large n_cpu, the overhead can be the decisive term. Furthermore, whether it is suitable to execute a piece of code serial or parallel depends on the n_{cpu}. Together with the difficulty of predicting the overhead time, this leaves *just the decision* of whether a perfectly parallelisable piece of code actually should be run in parallel as a heuristic problem. TF uses as a strategy to find the optimal parallelisation to run a small simulation of the graph, thereby determining the overhead and the number of cores.

Since TF actually executes the computations, any execution time measurement highly reflects the performance of TF for this task. As TF itself is under active development, the performance in general is expected to improve in the future.

To get a reasonable estimate of what TF is capable of and somewhat avoid potential bottlenecks from `zfit`, a dummy test function similar to a loss was written in pure TF. The function creates three times one million of random numbers and does a few operations on them before adding and reducing them to a single number. This is added to the previous calculation in a loop 100 times. There are no I/O bottlenecks and, while not as an optimal example for TF, it seems reasonable to what can be expected in model fitting.

We can see that the speedup is roughly a factor of 2/3 per core compared to the ideal case of 1. For example, the time from 1 CPU to 6 CPUs could be expected to decrease by a factor of 1/6 but does by 1/4 instead. While there are ways of building more efficient code, the example was chosen to reflect an arbitrarily, non-optimized implementation as expected to be found in `zfit`, mostly with custom models.

D.3 Additional profiling

Performance studies have been conducted, not shown in Sec. 5. They are displayed here.

References

[1] M. Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

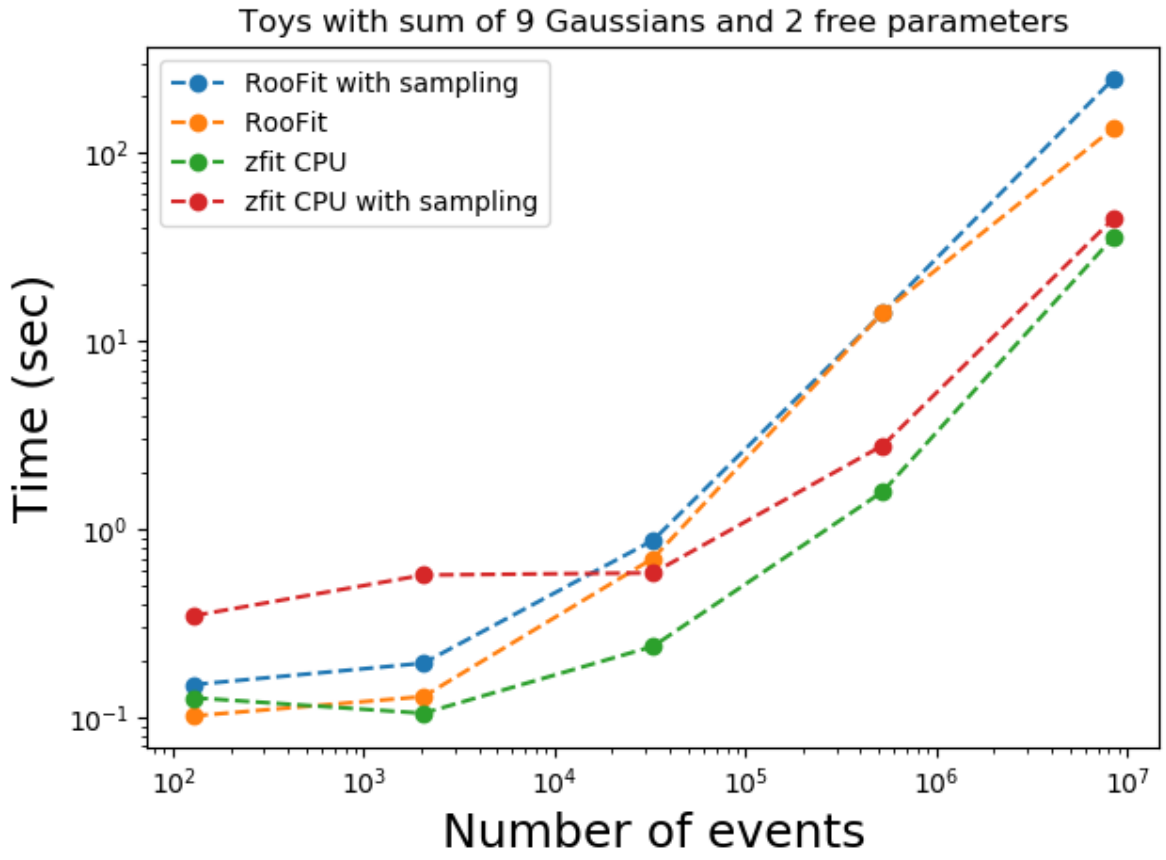


Figure 13: Full toy study with sum of 9 Gaussians and 2 free parameters. We can see that `zfit` s temporary bottleneck in sampling causes an extraordinary increase in execution time mostly for low number of events, but the conclusions and the overall scaling behaviour is still the same as described in Sec. 5.1.

[2] A. Paszke *et al.*, *Automatic differentiation in pytorch*, .

[3] J. Pivarski, *Non-fork repositories of github users who forked cmssw/aliphysics*, `https://github.com/jpivarski/2019-06-10-usatlas-argonne-python/blob/master/01-why-python-in-hep.ipynb`, 2019.

[4] T. E. Oliphant, *Python for scientific computing*, Computing in Science Engineering **9** (2007) 10.

[5] M. Newville *et al.*, *lmfit/lmfit-py 0.9.13*, 2019. doi: 10.5281/zenodo.2620617.

[6] J. V. Dillon *et al.*, *Tensorflow distributions*, CoRR **abs/1711.10604** (2017) `arXiv:1711.10604`.

[7] W. Verkerke and D. P. Kirkby, *The RooFit toolkit for data modeling*, eConf **C0303241** (2003) MOLT007, `arXiv:physics/0306116`, [,186(2003)].

[8] P. Ongmongkolkul *et al.*, *scikit-hep/probfit: 1.1.0*, 2018. doi: 10.5281/zenodo.1477853.

[9] Lukas *et al.*, *diana-hep/pyhf v0.0.15*, 2018. doi: 10.5281/zenodo.1464139.

[10] J. Bendavid, *Higgsanalysis-combinedlimit*, `https://github.com/bendavid/HiggsAnalysis-CombinedLimit/tree/tensorflowfit`, 2016.

[11] A. Poluektov, *Tensorflow analysis*, `https://gitlab.cern.ch/poluekt/TensorFlowAnalysis/`, 2017.

[12] F. James and M. Roos, *Minuit: A System for Function Minimization and Analysis of the Parameter Errors and Correlations*, Comput. Phys. Commun. **10** (1975) 343.

[13] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014.

[14] LHCb, R. Aaij *et al.*, *Angular analysis of the $B^0 \to K^{*0}\mu^+\mu^-$ decay using 3 fb^{-1} of integrated luminosity*, JHEP **02** (2016) 104, `arXiv:1512.04442`.

[15]

[16] F. James, *Monte-Carlo phase space*, .

[17] BABAR Collaboration, B. Aubert *et al.*, *Measurement of $D^0-\overline{d}^0$ mixing from a time-dependent amplitude analysis of $D^0 \to K^+\pi^-\pi^0$ decays*, Phys. Rev. Lett. **103** (2009) 211801.

[18] M. Marinangeli, *marinang/lauztat: v1.1.2*, 2019. doi: 10.5281/zenodo.2648147.

[19] J. Pivarski *et al.*, *scikit-hep/uproot: 3.6.3*, 2019. doi: 10.5281/zenodo.3239529.

[20] W. McKinney, *Data Structures for Statistical Computing in Python*, 2010.

[21] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, 2006–. [Online; accessed ¡today¿].