University of Zurich[UZH]

Bachelor Thesis

Monte Carlo integration with TensorFlow for unbinned likelihood fits at LHCb

written by
Charlie Rohrbach

Supervised by
Jonas Eschle
Prof. Dr. Nicola Serra

5th April 2023

Abstract

Model fitting using an unbinned maximum likelihood estimation is a common technique for analysing data in High Energy Physics. In the process, a significant fraction of computing power is used for normalising complex model functions through numerical integration. In this thesis, we studied the performance of the advanced Monte Carlo integration algorithm VEGAS to test whether its usage can improve the efficiency of the model fitting library `zfit`. For this purpose, we used an existing implementation with TensorFlow, the computational backend of `zfit`. The accuracy and runtime of the algorithm was measured for the angular distributions of a $B^0 \to K^{*0}\mu^+\mu^-$ decay and a set of multidimensional toy functions. It was found that VEGAS has a better accuracy and a higher runtime overhead compared to a simple Monte Carlo method and excels for functions with narrow peaks. For a given accuracy, VEGAS is therefore superior for non-flat or computationally expensive functions.

Contents

1 Introduction

The Standard Model (SM) of particle physics is a theory that describes the elementary particles and their interactions. It encompasses our current understanding of physics on a subatomic level and has proven successful in predicting and explaining a wide range of phenomena. However, since it does not account for various elements such as gravity, dark matter or neutrino oscillations, it remains an incomplete theory. Though more measurements and data are needed, the discovery of elementary particle processes that are inconsistent with the SM could eventually guide us to a new theory.

Elementary particles and their interactions are studied in various collider experiments around the world. Stable, charged particles, typically protons or electrons, are accelerated to high enough energies such that particles of interest can be produced in the subsequent collisions. To date, the Large Hadron Collider (LHC) at CERN is the highest-energy particle accelerator with a nominal energy of 6.5 TeV per proton. These protons are made to collide at four distinct interaction points, where the particle detectors are located. During the operation of the collider, particles originating from the collisions pass through the detectors, where their tracks and properties are measured. The recorded signals are selected by a trigger system in real-time to reduce the data volume and save only the most interesting events for offline analysis.

The LHCb experiment in particular focuses on the study of rare decays involving b quarks. One example is the decay $B^0 \to K^{*0}\mu^+\mu^-$, which is forbidden in the SM at tree level. Aaij et al. [1] perform a full analysis of the angular distribution of the decay products from this process using $3\,\mathrm{fb}^{-1}$ of integrated luminosity from LHC Run 1. This involves determining values for a set of angular observables. By comparing the results to the SM predictions, one can gauge the possibility of any contributions from physics beyond the SM.

Data analysis is essential for any experiment in order to extract a quantifiable result from the measurement data, which can be compared to the theory. A common method to determine values for physical observables is fitting a model to the data by means of maximum likelihood estimation (MLE). Thereby, the model f is a probability density function (PDF) that is hypothesised to describe the distribution of measured events. It is built taking into account theoretical physics as well as experimental properties and contains free parameters. Fitting a model to the data thus refers to determining values for the free parameters that minimise the discrepancies between the measured and theoretical distribution. For maximum likelihood estimation, the agreement of the data with the model is quantified by the

likelihood L. As the name suggests, the goal is to maximise the likelihood in order to find estimates for the parameters. The maximisation is usually an iterative process, where L is evaluated for a different set of parameter values in each step, essentially probing the parameter space in search of the extremum. Each evaluation of the likelihood also entails evaluations of the model function for the same parameter constellation. To ensure that f fulfils the conditions of a PDF, it needs to be normalised beforehand, which involves the determination of an integral. In case the integral is not known analytically, the integration is performed numerically, which is a computationally expensive task. As a result, the repeated integration during every step can become a performance bottleneck of the model fitting process. Moreover, the accuracy of the integration results directly affects the outcome of the MLE. For this reason, fast and accurate integration methods are essential for analysing the large quantities of data collected in High Energy Physics (HEP). Since the mathematical models are oftentimes multivariate functions, we require algorithms that are compatible with multidimensional integrands. The classical quadrature rules available for one-dimensional problems generally do not scale well to higher dimensions. Instead, we consider a class of algorithms known as Monte Carlo methods, which are based on random number generation and are applicable to integrals of arbitrary dimension.

Data analysis in HEP has traditionally been performed in C++. At CERN in particular, a framework called ROOT has been developed and is currently being used. However, the usage of Python has been perpetually rising in recent years. This popularity can be attributed in part to the existence of a vast Python ecosystem for scientific computing and its ease of use. Although existing C++ libraries can be accessed from Python through Python bindings, this is accompanied by issues with integration in Python and extendibility among other things. Consequently, pure-Python libraries are crucial for a complete transition of HEP analysis to the Python ecosystem. This premise also motivated the development of the model fitting library `zfit` [2]. It is designed with the HEP-specific requirements for model fitting in mind, thereby aiming to serve as an alternative to the RooFit module from ROOT. Moreover, it is implemented using the computational backend TensorFlow [3], which provides additional benefits including operation caching and the ability to run computations on different hardware devices like CPUs and GPUs. The resulting boost in performance allows `zfit` to compete with implementations in compiled languages like C++, which otherwise tend to be faster than interpreted Python code.

As established above, efficient integration is essential for data analysis and model fitting. However, the numerical integration capabilities of `zfit` are currently limited to the simplest Monte Carlo method. In order to improve

the library in this regard, more advanced algorithms need to be implemented. A highly promising integration method, which we will examine in this thesis, is the VEGAS algorithm developed by G.P. Lepage in 1978 [4]. Over the years, the algorithm has been used and implemented in a multitude of programming languages, including a Python version based on TensorFlow in the VegasFlow library [5]. We will take advantage of this implementation in order to investigate the performance of the VEGAS algorithm in comparison to a simple Monte Carlo method within the TensorFlow framework. In the end, we will use our findings to answer the question, whether VEGAS is worth including in `zfit`.

2 Theory

2.1 Maximum Likelihood Estimation

Maximum likelihood estimation is a method for estimating unknown parameters in a mathematical model f depending on a set of measured events $\vec{x}_1, \ldots, \vec{x}_n$. The resulting constellation of parameter values characterises the version of f that yields the highest probability of measuring $\vec{x}_1, \ldots, \vec{x}_n$ in an experiment. This is accomplished by finding the maximum of the likelihood function.

For a single measured event $\vec{x}_i$, the likelihood of the parameters $\theta_1, \ldots, \theta_k$ is defined as

$$\mathcal{L}(\theta_1, \ldots, \theta_k \,|\, \vec{x}_i) = P(\vec{x}_i \,|\, \theta_1, \ldots, \theta_k) \tag{1}$$

where $P(\vec{x}_i \,|\, \theta_1, \ldots, \theta_k)$ is the conditional probability of measuring $\vec{x}_i$ given the parameter values $\theta_1, \ldots, \theta_k$. Since the probability distribution of $\vec{x}_i$ is described by f, it follows that

$$\mathcal{L}(\theta_1, \ldots, \theta_k \,|\, \vec{x}_i) = f(\vec{x}_i \,|\, \theta_1, \ldots, \theta_k) \tag{2}$$

The main difference between the PDF and the likelihood function is which quantities are considered variables and which are considered fixed parameters. The PDF is a function of $\vec{x}$ given fixed parameter values for $\theta_1, \ldots, \theta_k$, whereas the likelihood is a function of $\theta_1, \ldots, \theta_k$ given a fixed measurement value $\vec{x}_i$.

For multiple independent events, the likelihood is the product of the individual likelihoods of each event.

$$\mathcal{L}(\theta_1, \ldots, \theta_k \,|\, \vec{x}_1, \ldots, \vec{x}_n) = \prod_{i=1}^{n} \mathcal{L}(\theta_1, \ldots, \theta_k \,|\, \vec{x}_i) \tag{3}$$

Thus, the likelihood function of all events in the dataset, usually referred to as L, is given as

$$L = \mathcal{L}(\theta_1, \ldots, \theta_k \,|\, \vec{x}_1, \ldots, \vec{x}_n) = \prod_{i=1}^{n} f(\vec{x}_i \,|\, \theta_1, \ldots, \theta_k) \tag{4}$$

The estimates for the unknown parameters of f correspond to the argument values for which L is maximal.

$$\hat{\theta}_1, \ldots, \hat{\theta}_k = \operatorname{argmax}(L) \tag{5}$$

While some likelihood functions are simple enough that the maximum can be found analytically, which results in exact values for the parameters, this is not possible in the general case. Instead, iterative algorithms are used that sample the parameter space in order to find approximate numerical values for the model parameters.

Further information about likelihood and maximum likelihood estimation can be found in [6] and [7].

2.2 Normalisation of the PDF

Since the model function f is a PDF, it is non-negative and integrates to 1 over the considered space Ω of possible events.

$$\int_{\Omega} f\left(\vec{x} \,|\, \theta_1, \ldots, \theta_k\right) \mathrm{d}\vec{x} = 1 \tag{6}$$

It may be rewritten as the following quotient

$$f(\vec{x} \,|\, \theta_1, \ldots, \theta_k) = \frac{p\left(\vec{x} \,|\, \theta_1, \ldots, \theta_k\right)}{\int_{\Omega} p\left(\vec{x} \,|\, \theta_1, \ldots, \theta_k\right) \mathrm{d}\vec{x}} \tag{7}$$

where p is the kernel of the PDF that describes the shape of the distribution and the denominator is a normalisation factor that ensures eq. (6) holds.

The normalisation factor can only be given as an explicit function of the parameters $\theta_1, \ldots, \theta_k$ if the integration of the kernel can be performed analytically. Otherwise, it needs to be computed using numerical integration methods, which employ algorithms to compute an approximation of the numerical value of the integral.

The accuracy of the integral value as a normalisation factor directly affects the accuracy of the PDF and consequently of L. Using inaccurate values for the likelihood in the maximisation may lead to inaccurate parameter estimates or even failure to find the true maximum. Therefore, it is crucial for the parameter estimation to obtain numerical integration results that are as precise as possible and at least in the order of magnitude of the desired accuracy.

2.3 Numerical Integration

For one-dimensional problems, there are efficient quadrature rules that require evaluating the function at evenly spaced points, such as the Simpson's rule or the trapezoidal rule. The same rules may be used to compute higher

dimensional integrals by phrasing them as a product of one-dimensional integrals. However, assuming a fixed number of function evaluations per 1D integral, this causes the total number of function evaluations and thus the computation time to grow exponentially with the number of dimensions.

A more attractive option for multidimensional integrals are the so-called Monte Carlo methods that make use of random number generation for determining the points where the function should be evaluated. In contrast to the quadrature rules, they are easy to apply to any integrand with an accuracy that is independent of the number of dimensions. As a result, they may yield better results for an equal number of function evaluations, especially for high-dimensional integrals.

2.4 Crude Monte Carlo

The simplest Monte Carlo method for estimating the integral value I of a function f is also called Crude Monte Carlo.

$$I = \int_\Omega f(\vec{x})\,\mathrm{d}\vec{x} \tag{8}$$

For a sample size N, one draws N random points $\vec{x}_i$ from a uniform distribution over the integration space Ω. The estimate of the integral is then

$$\hat{I} = V\frac{1}{N}\sum_{i=1}^{N} f(\vec{x}_i) \tag{9}$$

where V is the volume of Ω. The corresponding uncertainty on the integral value is

$$\sigma = \sqrt{\mathrm{Var}(\hat{I})} = \frac{V}{\sqrt{N}}\sqrt{\mathrm{Var}(f)} \tag{10}$$

Since the exact value for the variance of f contains the unknown integral value I, one can use an unbiased sample variance to compute the uncertainty estimator.

$$\begin{aligned}\mathrm{Var}(f) &= \frac{1}{N-1}\sum_{i=1}^{N}\left(f(\vec{x}_i) - \frac{1}{N}\sum_{i=1}^{N} f(\vec{x}_i)\right)^2 \\ &= \frac{1}{N-1}\left(\sum_{i=1}^{N} f^2(\vec{x}_i) - \frac{1}{N}\left(\sum_{i=1}^{N} f(\vec{x}_i)\right)^2\right)\end{aligned} \tag{11}$$

According to the Law of large numbers, the estimators will converge to the true values as N tends to infinity.

The estimated uncertainty on the obtained integration result is inversely correlated to the square root of the sample size. This means that while it is possible to increase the number of function evaluations to obtain more accurate results, eventually one reaches a point of diminishing returns. Therefore, to obtain the best results with a given number of evaluations, it is essential to use smart algorithms that employ additional strategies to improve the accuracy.

2.5 Stratification

Stratification or stratified sampling refers to the strategy of dividing the integration volume Ω into a set of smaller volumes Ω_i, for which the numerical integrals are then computed individually. Due to the linearity of integration, the results can be summed up to find the total integral estimate.

$$\int_\Omega f(\vec{x})\,\mathrm{d}\vec{x} = \sum_{i=1}^{k} \int_{\Omega_i} f(\vec{x})\,\mathrm{d}\vec{x} \tag{12}$$

Since the integral estimates from the different sections are independent, they can be considered uncorrelated random variables. Therefore, one can sum up the variances of each section to find the total variance. This results in the following error estimate

$$\sigma = \sqrt{\sum_{i=1}^{N} \mathrm{Var}(\hat{I}_i)} \tag{13}$$

The variance of each $\hat{I}_i$ is related to the variance of f within the volume Ω_i. The spread of the function values of f within a section of the total volume is either the same or more likely smaller than over all of Ω. That means, disregarding volume factors, the variance of f on that section is also smaller. Thus, by computing the smaller integrals individually and combining them, it is possible to reduce the error on the integral without changing the sample size.

Stratification can be applied without prior knowledge about f by dividing the integration volume into evenly sized sections. However, there are also adaptive algorithms that attempt to optimise the division depending on f in order to minimise σ.

2.6 Importance Sampling

The idea of importance sampling is to draw less samples in the areas of Ω where the function values $f(\vec{x})$ are low and more samples in the areas

where they are high and thus contribute more to the integral value. Since more samples leads to a higher accuracy, this means spending the computing power proportionately to where it has the largest impact. In practice, this is achieved by sampling the $\vec{x}_i$ from a density function p with a shape similar to that of f instead of a uniform distribution.

One can think of the integral over f as being an integral over a new function f/p weighted by the density function p.

$$I = \int_\Omega f(\vec{x})\,\mathrm{d}\vec{x} = \int_\Omega \frac{f(\vec{x})}{p(\vec{x})}\,p(\vec{x})\,\mathrm{d}\vec{x} \tag{14}$$

The Monte Carlo estimate then becomes

$$\hat{I} = V\frac{1}{N}\sum_{i=1}^{N}\frac{f(\vec{x}_i)}{g(\vec{x}_i)} \tag{15}$$

which is analogous to eq. (9) with f/p instead of f, hence the uncertainty follows according to eq. (10). Since f and p are similar, the variance of f/p is smaller than the variance of f and therefore the resulting uncertainty obtained with importance sampling is smaller.

This technique is especially useful for peaky functions, where a large fraction of the integral comes from a small area of the integration space. Unlike stratification, it can not be applied blindly. The probability density p should be a function that can be sampled from efficiently and finding a suitable one requires prior knowledge about f.

2.7 VEGAS

The VEGAS algorithm is a Monte Carlo algorithm for multidimensional integration that employs adaptive importance sampling. The used probability density p is a step function defined by a coordinate grid, which is adapted to the integrand over multiple iterations. In the following, the algorithm will be outlined in one dimension. For multiple dimensions, the one-dimensional strategy is applied to each axis individually. A more detailed explanation of the algorithm can be found in [4].

At first, the integration range is divided into M uniform intervals $[x_i,\ x_{i+1}]$. The step function p is defined as being inversely proportional to the interval width $\Delta x_i = x_{i+1} - x_i$.

$$p(x) = \frac{1}{M\Delta x_i} \tag{16}$$

Thus, starting from an equidistant grid, the x_i are changed iteratively in order to adapt the density p to the integrand. In each iteration, Monte Carlo integration with N samples is performed. Then, each interval is divided further into m_i subintervals, where m_i is proportional to the interval's contribution to the integral. Finally, the initial number of intervals is restored by merging subintervals in a way that attempts to even out the integral contributions. In the next iteration, the new grid is used for the integration step and refined further. Eventually, one reaches an optimum where the grid only fluctuates minimally due to the random nature of the procedure.

The total integral estimate and its corresponding error estimate can be computed from the results I_j and σ_j of each iteration as follows.

$$\bar{I} = \sigma_{\bar{I}}^2 \sum_j \frac{I_j}{\sigma_j^2} \tag{17}$$

$$\sigma_{\bar{I}} = \left(\sum_j \frac{1}{\sigma_j^2} \right)^{-\frac{1}{2}} \tag{18}$$

The results are weighted by the inverse of their variance, which means more accurate values contribute more to the final result. In practice, it may also make sense to exclude the first few iterations from the computation, because the results can be unreliable when the grid is not well established.

2.8 VEGAS Enhanced

The enhanced VEGAS or VEGAS+ algorithm is an improved version of the original VEGAS algorithm that combines adaptive importance sampling with adaptive stratification. Instead of taking an equal amount of samples in each area Ω_i, adaptive stratification attempts to distribute the fixed number of samples across the areas in a way that minimizes σ. This additional step leads to improved accuracy of the integral that is most noticeable in lower dimensions, while adding only a negligible amount of computational cost. This modification also helps the algorithm deal with multiple peaks or structures aligned with the diagonals, which are difficult for the original VEGAS algorithm to cope with. A detailed explanation of the algorithm can be found in [8].

3 Python Libraries

3.1 zfit and TensorFlow

`zfit` is a newer Python library for advanced model building and fitting intended mainly for data analysis in High Energy Physics. Unlike other libraries that serve a similar purpose, it is written in pure Python. Additionally, it uses the machine learning library TensorFlow as a computational backend, which comes with specific advantages but also imposes restrictions on the applications.

While pure Python code is interpreted, TensorFlow can be used to just-in-time compile functions when they are first called in the code. This constructs a directional computational graph, where each node represent an operation and each edge represents a tensor. In this case, a tensor is an object in TensorFlow that corresponds to an arbitrarily dimensional array, whose size and data type need to be specified upon creation and are immutable. The graph is built using placeholder values with the respective size and type of the input parameters, such that a function evaluation becomes equivalent to executing the graph using specific values. Due to the compilation, repeated calls of the same function are faster. In addition, the framework allows performing computations on different devices – e.g. GPUs or CPU cores – and even distributing them across multiple devices. Depending on the task and the used hardware, a certain degree of parallelisation can be achieved, which speeds up the computation. Another benefit of using TensorFlow is the support for automatic gradient computation from a graph. This is particularly helpful for finding the maximum of the likelihood function during model fitting.

The main drawback of using TensorFlow is that functions may not utilise regular Python logic like conditionals or loops if they are to be compiled. Instead, one is limited to using operations provided by TensorFlow, which places restrictions on what kind of functions may be built.

3.2 vegas and VegasFlow

The vegas library [9] is an implementation of the VEGAS+ algorithm by the original author. In order to achieve performances that are similar to compiled languages, it is written in cython and uses numpy functions. This means it makes use of functions that are compiled in C and accessible through Python.

The VegasFlow library implements the VEGAS algorithm using TensorFlow. It is built on a structure for general Monte Carlo integration techniques, which facilitates extending the library with additional integration al-

gorithms. At the time of writing, VEGAS+ and crude Monte Carlo are already available. The usage of TensorFlow allows VegasFlow to take advantage of all the above mentioned advantages, such as hardware acceleration, without the need of handling the implementation details.

4 Methods

Crude Monte Carlo provides a perfectly parallelisable algorithm that is easy to implement and currently the default in `zfit`. It therefore serves as the baseline that VEGAS is compared against. For a fair comparison, the integration with both algorithms is performed using TensorFlow with the classes `PlainFlow` and `VegasFlow` respectively from the VegasFlow module. The integrands used to study the algorithm properties include physical model functions from HEP analysis and different toy functions. The S-Wave and P-Wave angular distributions of the final state particles of $B^0 \to K^{*0}\mu^+\mu^-$ decays as found in [1] serve as the examples of the physical models. The toy functions consist of polynomial and trigonometric functions as well as Gaussian distributions and are representatives of different properties. These include symmetric and asymmetric functions, varying levels of peakiness and different dimensionalities. The main focus is on multivariate functions with small dimensions, hence the considered integrands are at most five-dimensional.

4.1 Accuracy

Firstly, we are interested in the reported accuracy of the integration results when using the same amount of effort for both algorithms. For this purpose, it is assumed that the function evaluations dominate the computational cost and that it is therefore sufficient to use the same number of sample points and iterations irrespective of the actual runtime. Performing integration using the above-mentioned classes from the VegasFlow library yields the estimated integral value and the corresponding uncertainty. We repeat the integration of a function using different numbers of samples per iteration so that we can plot the accuracy against the sample size and study their correlation. By including the resulting datasets for both algorithms in the same graph, we can compare not only the magnitude of the errors, but also if and how the correlation differs. In order to draw comparisons between the results of different integrands, we use relative errors, i.e. the uncertainty divided by the integral value.

4.2 Runtime

For the first part, it was assumed that the computational cost of the integration depends primarily on the number of function evaluations. This idealisation would predict that crude Monte Carlo and VEGAS have a similar runtime when using the same number of iterations and samples. However, this neglects the fact that the runtime also depends on the algorithm and

the specific implementation. The VEGAS algorithm is more complex and contains more computation steps for performing the integration. Consequently, one would expect this to result in a longer runtime. In order to test this hypothesis, we measure the time it takes to compute an integral estimate for each algorithm with the same number of samples and iterations. Since the first iteration of any run will generally take longer due to the compilation of the TensorFlow functions, we exclude it from the measurement. Additionally, by also measuring the time needed for evaluating the function, the computational overhead of the integration part can be determined from the difference between the total runtime and the evaluation time. As is common for timing functions, we take an average of numerous repetitions in order to even out statistical fluctuations arising from the randomness of the process. Since TensorFlow is capable of running computations on different hardware, a further step would be to compare the relative runtime the integration requires on the CPU and the GPU.

For practical purposes, it might be most interesting to directly measure the runtime of the algorithms for computing an integral estimate with a given maximal uncertainty. This would show explicitly if one algorithm is preferable in terms of speed for equally accurate results. Unfortunately, using a target accuracy as an input value and letting the number of samples and iterations be chosen automatically is a rather complex task. Although it has been requested[1], there is no implementation of this in VegasFlow at the time of writing.

4.3 Limits of VEGAS

After investigating the properties of VEGAS for well behaved functions, we are also interested in what its limits are. Namely, we want to study how far we can change the function properties or the input parameters of the algorithm such that it fails to produce a consistent result. Failure of the algorithm does not mean that it throws an error, but rather that the result is unreliable. This can be the result of the grid not adapting to the function properly, such as not being able to find a peak. However, without knowledge of the expected integration result, this can be hard to identify. While a large relative uncertainty is a strong indicator of an underlying problem, it is not a necessary criterion. In the example where a peak is missed completely, the resulting values might look reasonable while being completely wrong.

[1] `https://github.com/N3PDF/vegasflow/issues/72`

One property that affects the results strongly is the peakiness of the function. In HEP, data may consist of a slowly varying background and a sharp signal peak, which is the main area of interest. Thus, it is important that the integration algorithm does not miss this peak. To investigate the effect of the peakiness, we consider a superposition of two multivariate Gaussian distributions. We choose one peak to be relatively wide with a fixed width and perform integration for varying widths of the second peak. The advantage of using Gaussians is that they are defined in arbitrary dimension and that the expected integral value is known. By comparing our integration results to the expected values, we can identify when the algorithms have successfully captured both peaks. In order to explore the boundaries of the algorithm, we may also vary the dimension of the multivariate Gaussians while keeping the sample size constant and vice versa.

4.4 Iterations vs. Sample Size

In comparison to other algorithms that might take an accuracy threshold or the number of samples as input parameters, VEGAS requires the specification of both a number of iterations i and a number of samples per iteration N. Choosing appropriate values is non-trivial as increasing i or N both result in a higher computational cost, but affect the integration results differently. According to [8], the grid of the VEGAS algorithm usually converges within 5 to 20 iterations. However, there is no general guideline as to how many iterations one should choose even within these boundaries. We want to explore the question if it is preferable to increase one parameter over the other in order to improve the integration result. For this purpose, we choose a fixed value for one and perform the integration repeatedly for varying values of the other parameter and compare the results.

5 Results

5.1 Accuracy

For both plain and VEGAS, the relative uncertainty of the integral estimate as reported by the algorithm was measured for a selection of different functions and sample sizes. For each integrand, the two data sets were plotted against the sample size in the same graph as shown in fig. 1.

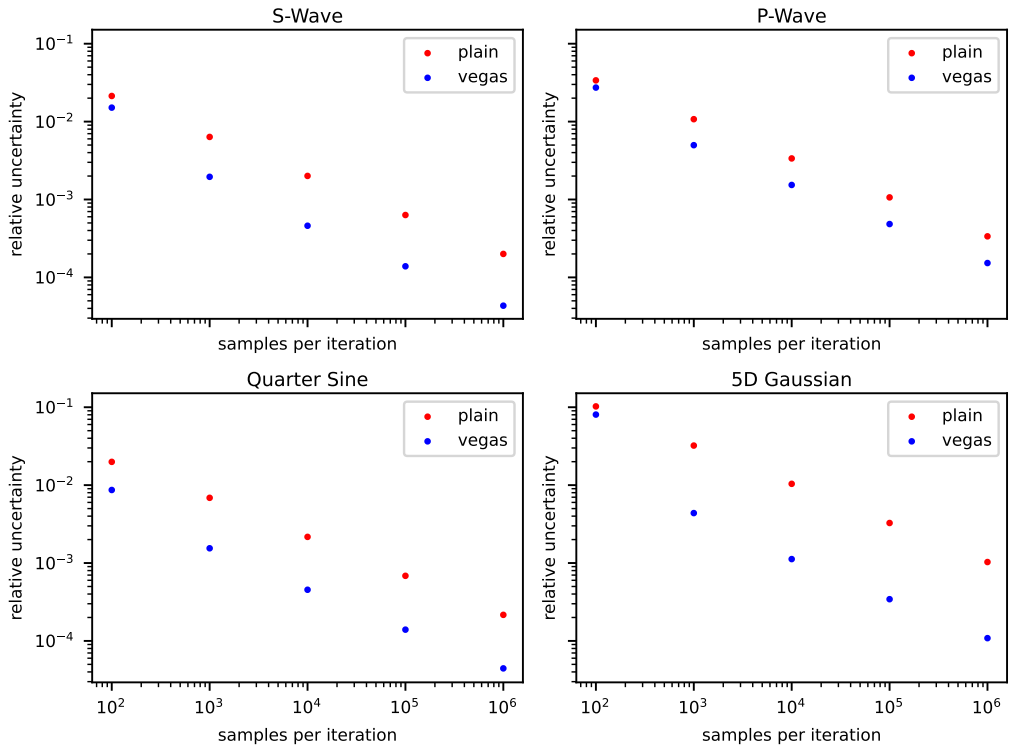


Figure 1: Reported relative uncertainties of the VEGAS and plain algorithm after 5 iterations for different sample sizes

Across all investigated functions, a larger sample size correlates with a smaller uncertainty. In a double logarithmic plot, an approximately linear correlation with a negative slope between the uncertainty and the number of samples N can be observed.

The comparison between the algorithms shows that VEGAS outperforms plain Monte Carlo in terms of accuracy given the same number of function evaluations. In other words, VEGAS needs less samples than plain Monte Carlo in order to reach the same accuracy. While the difference in magnitude

of the uncertainties varies between different integrands, VEGAS is better up to an order of magnitude. This can be seen in fig. 1, where the data points for VEGAS are generally located below those for plain Monte Carlo.

These observations are consistent with the expectation according to eq. (10). When a logarithm is applied to both sides of this equation, the product can be rewritten as a sum by applying logarithmic identities.

$$\log(\sigma) = -\frac{1}{2}\log(N) + \frac{1}{2}\log(\mathrm{Var}(f)) + \log(V) \tag{19}$$

The result is a linear correlation between the logarithms of the observed quantities with a negative slope. In addition, the y-axis shift depends on the variance of the integrand. Since the importance sampling used in the VEGAS algorithm aims to reduce the variance of f compared to crude Monte Carlo, one would expect to see a smaller shift.

5.2 Runtime

The runtime of both the VEGAS and plain algorithm as well as the mere function evaluation was measured for the same functions and sample sizes as in the previous section. For each integrand, the three quantities were plotted against the sample size in a common graph as shown in fig. 2.

In the investigated range of at most 10^6 samples per iteration, there appears to be a linear correlation between the measured time and the sample size for all three quantities. Consequently, as the difference between two quantities with a linear dependence on the sample size, the overhead of the algorithm also exhibits a linear behaviour. This becomes more apparent when the data is shown using linearly scaled axes as in fig. 3, which depicts the overhead for different integrands in the same graph. In theory, the scale factor of the linear correlation depends on the dimensionality of the integration space, but not the actual integrand. Any direct dependency on the integrand would result from function evaluations, which do not contribute to the overhead as per its definition. However, the cost of operations performed on points from the integration space – such as those found in the grid refinement process – scales with the size of the array and therefore the dimensionality. The overhead of VEGAS depicted in fig. 3b reflects these expectations perfectly, as the observed scale factor is larger for higher-dimensional integrands and almost identical for the two physical functions with the same dimensionality. In comparison, the differences between the scale factors are relatively minor for the overhead of plain Monte Carlo shown in fig. 3a. This observation might

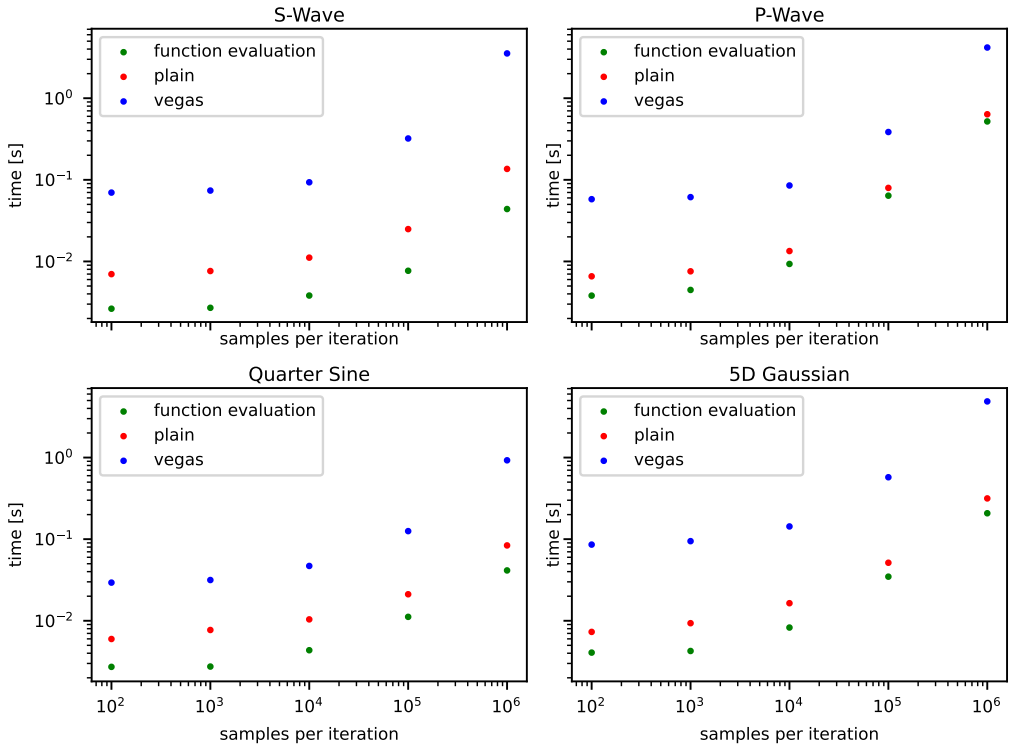


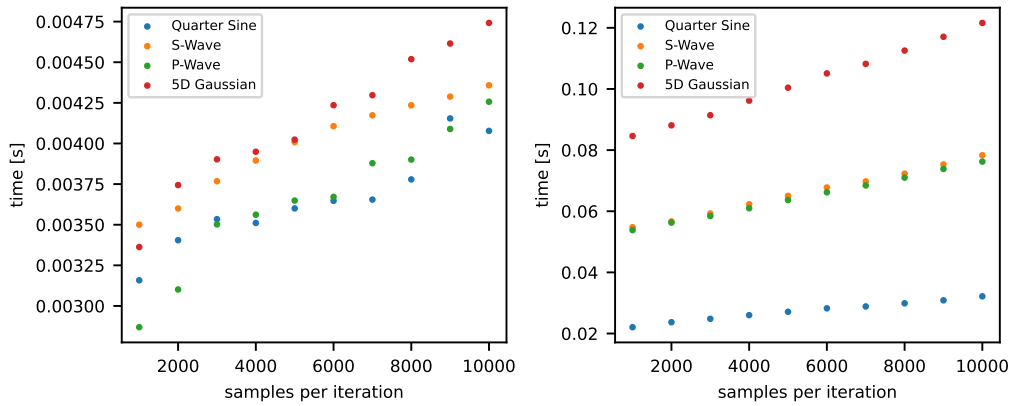
Figure 2: Runtimes for VEGAS, plain and the function evaluation for different sample sizes, averaged over 100 repetitions of 5 iterations each

be explained by the simplicity of the algorithm, which lacks any extensive array computations that would result in a pronounced dependency on the dimensionality.

Figures 2 and 3 clearly show that the VEGAS algorithm has a significantly larger overhead. Due to its complexity, more operations are performed per event in comparison to plain, which results in a larger scale factor on the sample size. Therefore, the observation is in line with our expectations.

The comparison between the evaluation time and the algorithm runtimes indicates that the assumption of the computational cost being dominated by the function evaluation does not hold for VEGAS. In our measurements, the function evaluation accounted for less than 10% of the total runtime in most cases. While the contribution is more significant for plain, the overhead of the algorithm can still not be considered negligible.

However, in the example of the P-Wave function, we find that the evaluation time makes up an increasingly larger fraction of the total runtime as



(a) Plain Monte Carlo (b) VEGAS algorithm

Figure 3: Overhead of the VEGAS and plain algorithm for a set of linearly spaced sample sizes, averaged over 100 repetitions of 5 iterations each

the sample size grows. This diminishing importance of the overhead is expected when a function is computationally intensive enough that its evaluation time scales more strongly with the sample size than the algorithm overhead. Given that the evaluation of the P-Wave is by far the costliest among those we measured, it seems reasonable that we observe the effect for this function.

5.3 Limits of VEGAS

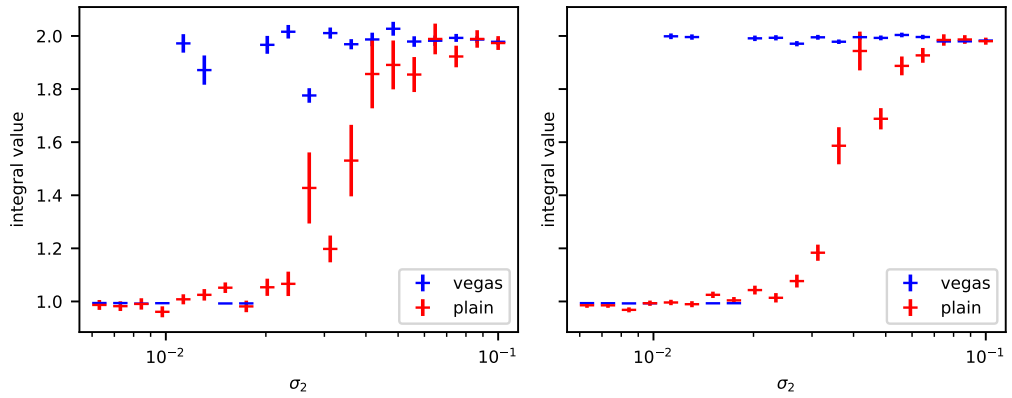


Figure 4: Integral values of the sum of two 5-dimensional Gaussians for varying standard deviations of the second peak. The left side shows the results after 5 iterations, the right side after 20.

In fig. 4, one can see the results obtained from integrating the sum of two multidimensional Gaussians using varying widths for one peak, while keeping the other variables constant. There are a few things that can be observed from this.

Firstly, for large values of the width σ_2, the obtained integral values are around 2, while for small values they are around 1. Since each Gaussian should integrate to approximately 1, this means both algorithms eventually fail to find the second peak as it becomes too narrow. However, this seems to happen faster with plain compared to VEGAS. Secondly, the values for VEGAS are mostly in the area around either 1 or 2, while the values are more widespread for plain.

These observations can be explained by the adaptive importance sampling employed by the VEGAS algorithm. If any sample points land in the area of the peak in the first few iterations, the grid is adapted such that more points are sampled in this area in subsequent iterations. Thus, with enough iterations, VEGAS is able to find an approximation for the integral including the narrow peak. On the contrary, if no points land in the area of the peak towards the beginning, it is more likely to be left out completely, because the grid is adapted to focus on the area of the wider Gaussian. Even if some points land there in later iterations, the effect will likely be small. This is due to the grid adaption being damped, which means that the changes made to the grid become smaller with increasing iterations in order to avoid rapid fluctuations. As a consequence, the results obtained with VEGAS are mostly split between the integral value of both peaks and the one of the wider peak only.

In contrast, the integral values obtained with plain depend more strongly on how many sample points land in the area of the narrow peak. The extreme ends of the spectrum are the same as for VEGAS, where a good estimate of both peaks is obtained with plenty of points and the narrow peak is practically ignored with too few points. However, there is also a range in between, where there are enough sample points in the area of the peak to significantly affect the integral value, yet not enough to obtain an accurate estimate. In that case, decreasing the peak width reduces the number of points that statistically land in the peak area and consequently their contribution to the integral estimate. Therefore, we observe values that are more widespread and change gradually with the peak width σ_2.

It is further notable, that the errors for plain are larger in this intermediate value range, thereby hinting at the inaccuracy of the result. In contrast, the values on the lower end show the smallest error estimates. This is likely due to the fact that missing the narrow peak practically reduces the integrand

to the wider Gaussian, which has a lower variance. It also illustrates that one should not rely solely on the reported accuracy to assess the quality of the results. In practice, it is best to have a good understanding of the integrand, especially for peaky functions. Approximately knowing what integral value to expect certainly helps with judgement of the result. In addition, having a rough idea of the space covered by the peak within the integration volume can guide us in deciding how many sample points to use. For peaks that occupy a very small fraction of the volume, it may be better to use stratification in order to prevent VEGAS from reducing the sample points in certain areas too drastically. If the position of the peak is known, the surrounding subvolume could even be selected manually and integrated separately from the rest in order to focus more sample points on the peak to capture its integral.

5.4 Iterations vs. Sample Size

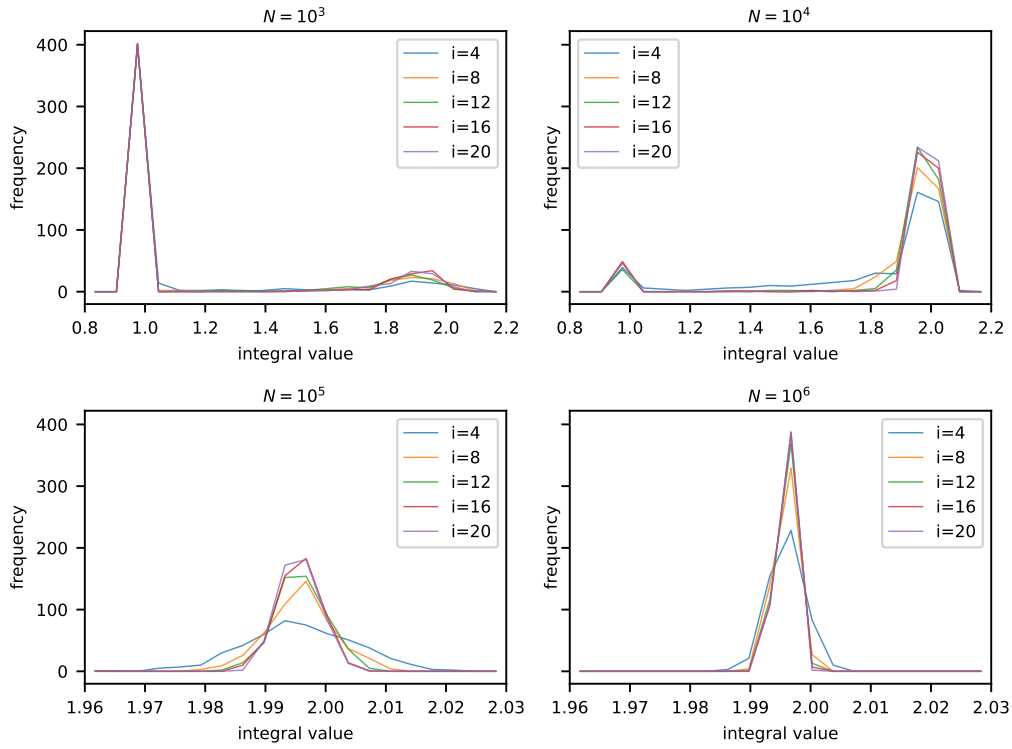


Figure 5: Distribution of the integral values obtained with VEGAS for varying sample sizes and numbers of iteration. The integrand is the sum of a wide and a narrow three-dimensional Gaussian.

In fig. 5, one can see the value distributions obtained from repeatedly integrating the sum of two multidimensional Gaussians with VEGAS for a varying number of iterations. For small enough sample sizes, we observe peaks representing value accumulations around two different values. As in the previous section, the lower values are a result of the smaller Gaussian being missed. With the exception of the lowest number, the graph also shows that increasing the number of iterations hardly affects the distribution. This is in line with our previous statement, that if not enough points land in the peak to significantly contribute to the integral in the first few iterations, this is unlikely to change with more iterations. It is a consequence of the adaptive importance sampling distributing the bulk of the points to other areas and the adaptation being damped. On the other hand, by increasing the sample size per iteration, the lower peak eventually vanishes and the distribution around the correct value becomes narrower.

Generalising these observations, one could say that in order to obtain a good integration result, we need to use a sample size that is large enough for the shape of the integrand to be captured at the beginning. Otherwise, the adaption of the grid is at best inefficient and at worst counterproductive. Additionally, we need to run enough iterations that the grid can adapt to the function properly, from which point on the changes to it will only be minor. According to [8], once a good grid has been found, it is generally better to increase the sample size in order to reduce the statistical uncertainty on the integration result. They also indicate a usual range of $5 - 20$ iterations for the grid to converge. Apart from that, determining how many sample points and iterations are enough to fulfil the aforementioned conditions depends on the integrand and remains largely a matter of trial and error.

5.5 Use Cases of VEGAS and Plain

The efficiency of each algorithm may change depending on factors that vary from case to case such as the integrand shape or the number of samples used. While neither is universally superior, they show certain tendencies regarding their suitability for different use cases. In the following, we will offer some suggestions as to why one might choose one algorithm over the other.

The VEGAS algorithm is especially beneficial for peaky functions, because it yields more accurate results by focusing more points on the relevant areas, provided that they are found initially. The main drawback of VEGAS is the comparatively large overhead of the integration. The contribution of the overhead to the runtime becomes less important the larger the evaluation time of

the integrand is. Thus, VEGAS becomes more appealing to use for computationally intensive functions. Additionally, the importance sampling of VEGAS can be particularly useful for integrating multiple functions that only differ slightly. After training the grid on the first integrand, it can be reused for the remaining integrands to obtain more accurate results compared to uniform sampling without any additional computational cost. Alternatively, it can be used as a starting point to reduce the total number of adaption steps needed.

The plain algorithm is generally preferable for integrands with relatively low variance and no sharp peaks, where the additional accuracy obtained by using VEGAS is minor and does not compensate for the additional runtime required. Similarly, it might be the better choice if the main objective is obtaining an integral estimate quickly without concern for high accuracy, given that no essential parts of the integrand are missed when using uniformly sampled points.

6 Conclusion

In this thesis, we investigated the computational efficiency and numerical accuracy of the VegasFlow library. On one hand, we found that the runtime of the VEGAS algorithm is significantly higher than for the plain algorithm when using the same number of samples. On the other hand, VEGAS generally yields more accurate results and thus requires less samples overall to achieve the same accuracy. Depending on the integrand properties, one algorithm may be preferable over the other. The plain algorithm might be sufficient for functions whose shape is captured reasonably well by uniformly sampled points. In contrast, VEGAS tends to perform better than plain for peaky or computationally expensive functions, as well as for bulk integration of similar functions. Therefore, it would be worth implementing the VEGAS algorithm in the `zfit` library in addition to the currently available simple Monte Carlo.

In the future, we would like to see a wider selection of algorithms used with TensorFlow. Since the VegasFlow library provides a framework for Monte Carlo algorithms to be used with TensorFlow, additional algorithms could be implemented as extensions to the library in order to be tested for performance with less effort. New algorithms or combination of algorithms that are found to outperform VEGAS or specialise on different use cases could then be incorporated into the `zfit` library. One notable candidate is the VEGAS+ algorithm discussed in 2.8, which has been implemented in VegasFlow, but not yet documented. Furthermore, it would be worth investigating how the usage of GPUs for the computations in addition to or instead of CPUs affects the performance of the algorithms.

References

[1] LHCb collaboration, R. Aaij *et al., Angular analysis of the $B^0 \to K^{*0}\mu^+\mu^-$ decay using 3 fb^{-1} of integrated luminosity,* JHEP **2** (2016) 104, arXiv:1512.04442.

[2] J. Eschle, A. Puig Navarro, R. Silva Coutinho and N. Serra, *zfit: Scalable pythonic fitting,* SoftwareX **11** (2020) 100508.

[3] M. Abadi *et al., TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,* (2015). Software available from tensorflow.org.

[4] G. P. Lepage, *A new algorithm for adaptive multidimensional integration,* J. Comp. Phys. **27** (1978) 192–203.

[5] S. Carrazza and J. M. Cruz-Martinez, *VegasFlow: Accelerating Monte Carlo simulation across multiple hardware platforms,* Comput. Phys. Commun. **254** (2020) 107376, arXiv:2002.12921.

[6] L. Lista, *Practical statistics for particle physicists,* CERN Yellow Reports: School Proceedings **5** (2017) 213–257, arXiv:1609.04150.

[7] J. Eschle, *zfit: scalable model fitting in Python using TensorFlow,* MA thesis (University of Zurich, 2019).

[8] G. P. Lepage, *Adaptive multidimensional integration: VEGAS enhanced,* J. Comp. Phys. **439** (2021) 110386, arXiv:2009.05112.

[9] G. P. Lepage, *gplepage/vegas: vegas version 5.1.1* (2022), doi: 10.5281/zenodo.5893494.