

What the new RooFit can do for your analysis

Stephan Hageboeck^{a,*}

^a*CERN,*

1, Esplanade des Particules, Meyrin, Switzerland

E-mail: stephan.hageboeck@cern.ch

RooFit is a toolkit for statistical modelling and fitting, and together with RooStats it is used for measurements and statistical tests by most experiments in particle physics. Since one year, RooFit is being modernised. In this talk, improvements already released with ROOT will be discussed, such as faster data loading, vectorised computations and more standard-like interfaces. These allow for speeding up unbinned fits by several factors, and make RooFit easier to use from both C++ and Python.

*40th International Conference on High Energy physics - ICHEP2020
July 28 - August 6, 2020
Prague, Czech Republic (virtual meeting)*

*Speaker

1. Introduction

RooFit [1] is a C++ package for statistical modelling distributed with ROOT [2]. With RooFit, users can define likelihood models using observables, parameters, functions and PDFs¹, which can be fit to data, plotted or be used for statistical tests. The tools for performing such tests are provided by the RooStats package, and the HistFactory package provides tools to create RooFit models from collections of ROOT histograms.

RooFit was started in the year 2000 in the BaBar collaboration. Since then, RooFit has been a reliable tool for many experiments in high-energy physics at B factories or the Large Hadron Collider. Due to its long history, it is nonetheless time to modernise and optimise RooFit for today's hardware, enabling researchers to analyse larger datasets, to devise more elaborate statistical models and to solve challenging research questions.

2. Improving the Usability of RooFit

2.1 Extending RooFit with more Stable and Faster Built-in PDFs

In modernising RooFit, the Johnson [3] (ROOT 6.18+, fig. 1) and Hypatia2 [4] distributions (ROOT 6.20+) were added. Although RooFit can interpret any formula using ROOT's cling interpreter [5], built-in PDFs are usually more stable, and can be optimised to evaluate faster [6].

In ROOT 6.20+, another convenience PDF was added, RooWrapperPdf. Since RooFit treats functions and PDFs differently (the former are just evaluated while the latter are evaluated and normalised automatically), users are sometimes forced to decide whether they should implement an object as a function, as a PDF or both. With the addition of RooWrapperPdf, only the function implementation has to be provided. The function can be used as a PDF by wrapping it into RooWrapperPdf. That is, the function is augmented with automatic numerical normalisation, and toy data can be sampled from it.

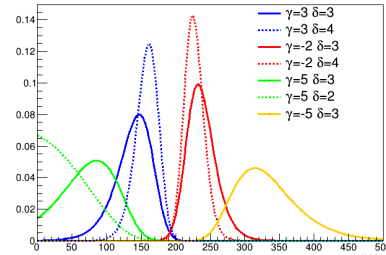


Figure 1: Johnson [3] distribution for various values of the parameters γ and δ

2.2 Unbiased Binned Fits

A long-known problem in RooFit discussed during the conference are binned fits with PDFs that have high curvature². To save computation time in binned fits, RooFit evaluates PDFs only in the centre of a bin, and uses this as an approximation for the probability of an entire bin. If a function has a high curvature, this is not correct, and can lead to biases as shown in fig. 2a. These biases can be reduced by using more bins, but this is not always an option for users given available data statistics. In ROOT 6.24+, the class RooBinSamplingPdf was added to integrate a continuous PDF over each bin. It converts continuous into binned PDFs as shown in fig. 2b, and evaluating it in the bin centre, in fact, anywhere in a bin, yields correct probability densities. Residuals are reduced, fits converge more reliably and fit results are not biased, any more.

¹Probability Density Functions

²That is non-zero second derivatives. These are models that cannot easily be approximated with piece-wise linear functions.

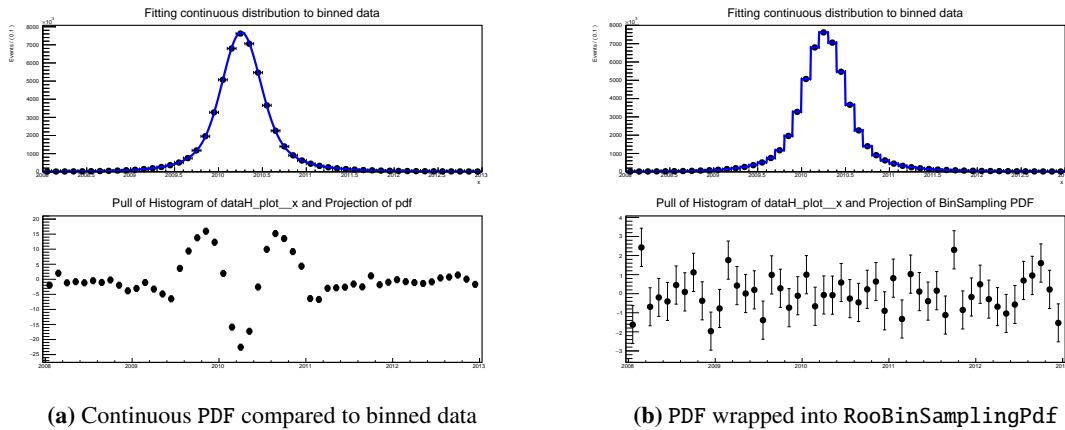


Figure 2: Comparison of pulls with and without `RooBinSamplingPdf`. The pulls are computed by comparing event counts with the plotted curves at the bin centres. NB: The y-axis is zoomed 6-fold in (b).

2.3 Recovery from Evaluation Errors

When `RooFit` evaluates a mathematical model, it relies on the fact that all model parameters are within the domains of the functions involved. If this assumption is violated, computations might yield the value `NaN`³. Starting from ROOT 6.24, `RooFit` will warn users if they set up parameters that might violate the domain of a function. Not all invalid parameters can be prevented by checking parameter limits, though. If the minimiser chooses a set of parameters that violates the domain of a function, the model cannot be evaluated, and no gradient can be computed to continue the minimisation. The minimiser will try to change each parameter, but it may only slowly, sometimes never, find a set of valid parameters to continue the minimisation.

Starting from ROOT 6.24, some of these fit failures can be avoided. `RooFit` can pass information to the minimiser by packing it into the mantissa of a `NaN`. If, for example, a PDF evaluates to a negative value, which is disallowed, the magnitude of the undershoot is packed into a `NaN`. Since IEEE-compliant floating-point operations leave the mantissa unaffected, this information can propagate through all computations. Before the log-likelihood is passed to the minimiser, this information is unpacked and the magnitudes of all violations are summed. This is converted into a penalty term, which is passed to the minimiser. From the magnitude of the penalty term, the minimiser can compute a gradient, and use it to step away from disallowed parameters. Notoriously unstable PDFs such as `RooPolynomial`⁴ were found to fit to data much faster, and fits that would previously fail were found to converge more reliably.

2.4 Modernisation of Interfaces

In `RooFit`, any collection of mathematical entities such as parameters, observables, functions or PDFs are saved or passed to functions using the classes `RooArgSet` and `RooArgList`. To operate `RooFit`, users have to manipulate these collections, for example, for querying the values of fit parameters. Until ROOT 6.18, these collections were based on a linked list shipped with

³“Not a Number”. For a Gaussian distribution, for example, this happens when setting the parameter $\sigma \leq 0$.

⁴Since polynomials of uneven order inevitably evaluate to negative values if x is large or small enough, very careful tuning of model parameters is required to keep the polynomial positive across the fit range of the observable.

RooFit. Iterating through such a collection was cumbersome and not efficient. To allow for the use of range-based **for** loops and to speed up iterations, RooFit’s collections were converted to `std::vector`-based collections in ROOT 6.18. This results in simpler code:

ROOT 6.18+	ROOT 6.16 and before
<pre> 1 // No variables outside loop required 2 for (auto p : *pdf.getParameters(obs)) 3 p->Print(); 4 // No danger of memory leak </pre>	<pre> 1 TIterator* it = 2 pdf.getParameters(obs)->createIterator(); 3 RooAbsArg* p; 4 while ((p=(RooAbsArg*)it->Next())) { 5 p->Print(); 6 } 7 delete it; </pre>

Iterating proved to be 25% faster, and the code is significantly simpler. Typical workflows in RooFit are sped up from 5% to 21% [7], depending on how many iterations through collections are required. The old interface remains supported, however, so users are not forced to rewrite their code.

Starting from ROOT 6.22, RooFit uses modernised category classes, which behave map-like. Defining and printing category states compares as follows:

ROOT 6.22+	ROOT 6.20
<pre> 1 RooCategory cat("cat", "Lep. mult."); 2 cat["0Lep"] = 0; 3 cat["1Lep"] = 1; 4 for (const auto& name_idx : cat) { 5 std::cout << name_idx.first << ", " 6 << name_idx.second << std::endl; 7 } </pre>	<pre> 1 RooCategory cat("cat", "Lep. mult."); 2 cat.defineType("0Lep", 0); 3 cat.defineType("1Lep", 1); 4 TIterator* typeIt = cat.typeIterator(); 5 RooCatType* catType; 6 while ((catType = 7 dynamic_cast<RooCatType*>(typeIt->Next())) 8 != nullptr) { 9 std::cout << catType.GetName() << ", " 10 << catType.getVal() << std::endl; 11 } 12 delete typeIt; </pre>

The new categories use 4 instead of 288 bytes of memory per entry in a dataset, and can better be integrated into batch computations (see section 3). Also here, old interfaces remain supported.

The modernisation of C++ interfaces is also beneficial for using RooFit from Python. Since ROOT ships with the C++ interpreter `cling` [5], it can automatically generate Python bindings for C++ objects (“PyROOT” [8]). Before ROOT 6.18, Python users would have had to imitate the C++ code at the beginning of this section. Now, the equivalent loop reads:

```

1 for p in pdf.getParameters(obs):
2   p.Print()

```

In addition to the automatically generated interfaces, PyROOT features so-called “Pythonisations”, short Python code that helps steer C++. For example, while an import function for RooFit objects reads “`workspace.import(object)`” in C++, Python users were required to use the workaround “`getattr(workspace, 'import')(object)`”, since `import` is a reserved keyword. Starting from ROOT 6.22, users can use the more intuitive “`workspace.Import(object)`”.

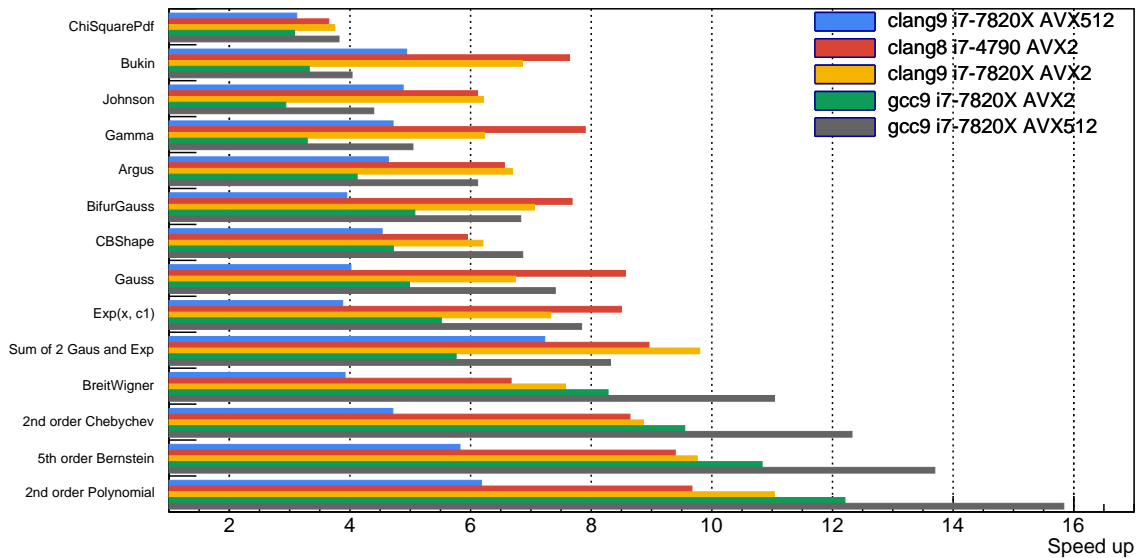


Figure 3: Speed up for computing the likelihoods of datasets of 100 000 to 300 000 events for various likelihood models. The fast batch interface in ROOT 6.20 is timed against the normal single-value computations. Depending on compiler, CPU capabilities and workflow, a speed up of 3x to 16x can be expected.

3. Faster PDF Computations

When RooFit computes likelihoods, one or multiple PDFs have to be evaluated for each entry in a dataset. However, RooFit only evaluates a single event in each function call⁵, and rows of a dataset are read one by one. This strategy is inefficient, because it does not make use of data caches, memory prefetching or vector extensions.

Starting from ROOT 6.20, data access and likelihood computations were reorganised such that data are read from a dataset in batches instead of copying them out of a dataset one by one [6]. Each sub-computation produces multiple values per function call, so less functions have to be called. Data are read in blocks. This mode uses slightly more memory, because intermediate results have to be stored, but it speeds up likelihood computations by about 300 %, which is especially relevant for data-intensive unbinned fits. Note that this speed up multiplies with the speed up from multi-processing.

PDF classes need to implement the interface 'span getValues(...)'⁶ to benefit from this speed up. Most PDFs in RooFit have been updated to support it. External PDFs that do not implement this interface can be used nevertheless, since a fall-back function that calls RooFit's classic evaluation functions in a loop will be used. Since the optimised functions might yield slightly different numbers, the fast batch mode has to be activated by users:

```
pdf.fitTo(data, BatchMode(true)); // Evaluate likelihood using fast batch mode
```

On modern CPU architectures with vector extensions such as SSE or AVX, the batch computation functions can be optimised even further using SIMD computations [6]. Figure 3 shows the speed

⁵A multi-process mode can be used to parallelise computations, but this still computes one event per function call.

⁶In ROOT 6.20 and 6.22, the experimental `getValBatch()` was in use, but it has been superseded by `getValues()`.

up that was achieved in comparison to RooFit's classic single-value computations when vector extensions are used. The speed up ranges from 3x for the "ChiSquared" PDF, which is based on a function from an external library without vectorisation, to 16x with AVX512 extensions.

To use vector extensions, users have to compile ROOT 6.20 and 6.22 themselves, manually enabling the desired extensions. Therefore, pre-compiled ROOT distributions (e.g. provided centrally by collaborations) will mostly benefit from the 3x speed up due to the more efficient data access. ROOT 6.24 and later will ship with multiple versions of a RooFit computation library, which are optimised for different vector extensions. ROOT will inspect the CPU and load the fastest library supported by the hardware. Users will therefore be able to benefit from larger speed ups than 3x.

Unit tests ensure that the optimised computation functions yield the same results as the classic RooFit functions. Computations of likelihoods usually agree to a relative accuracy of 1.0×10^{-14} , and log-likelihoods agree up to 2.0×10^{-14} with a few exceptions. Fit parameters usually agree better than 1.0×10^{-5} , which is orders of magnitude smaller than the statistical error in most fits.

4. Conclusions

In 2019, development in RooFit has been resumed. RooFit's interfaces are being modernised, long-standing problems are solved, and significant speed ups were achieved by better using the capabilities of modern CPUs. Work is underway to use the benefits of the new computation interface for RooFit computations on GPUs. These developments are aimed at providing RooFit's users with a better tool, and the ROOT team encourages users to get in touch with ideas and requests.

References

- [1] W. Verkerke and D.P. Kirkby, *The RooFit toolkit for data modeling*, *econf C0303241* (2003) MOLT007 [[physics/0306116](#)].
- [2] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, *Nucl. Instrum. Methods Phys. Res. A* **389** (1997) 81.
- [3] N.L. Johnson, *Systems of frequency curves generated by methods of translation*, *Biometrika* **36** (1949) 149.
- [4] D.M. Santos and F. Dupertuis, *Mass distributions marginalized over per-event errors*, *Nucl. Instrum. Methods Phys. Res. A* **764** (2014) 150 .
- [5] V. Vasilev, P. Canal, A. Naumann and P. Russo, *Cling – the new interactive interpreter for ROOT 6*, *J. Phys.: Conf. Ser.* **396** (2012) 052071.
- [6] S. Hageboeck, *A Faster, More Intuitive RooFit*, *EPJ Web Conf.* **245** (2020) 06007 [[2003.12875](#)].
- [7] S. Hageboeck and L. Moneta, *Making RooFit Ready for Run 3*, *J. Phys. Conf. Ser.* **1525** (2020) 012114 [[2003.12861](#)].
- [8] M. Galli, E. Tejedor and S. Wunsch, *A New PyROOT: Modern, Interoperable and More Pythonic*, *EPJ Web Conf.* **245** (2020) 06004.