Making RooFit Ready for Run 3

S Hageböck[1] **and L Moneta**[1]

[1] CERN, 1211 Geneva 23, Switzerland

E-mail: stephan.hageboeck@cern.ch

Abstract. RooFit and RooStats, the toolkits for statistical modelling in ROOT, are used in most searches and measurements at the Large Hadron Collider. The data to be collected in Run 3 will enable measurements with higher precision and models with larger complexity, but also require faster data processing.

In this work, first results on modernising RooFit's collections, restructuring data flow and vectorising likelihood fits in RooFit will be discussed. These improvements will enable the LHC experiments to process larger datasets without having to compromise with respect to model complexity, as fitting times would increase significantly with the large datasets to be expected in Run 3.

1. Introduction

RooFit [1] is a C++ package for statistical modelling distributed with ROOT [2]. RooFit allows to define computation graphs connecting observables, parameters, functions and PDFs in order to compute likelihoods and perform fits to data. An example of such a computation graph is shown in figure 1. Every node of the graph can be evaluated to a real value (*i.e.*, real-valued number), which for a PDF denotes the probability to find a data event with the given values of observables, for a given model and its set of parameters.

RooStats is a collection of statistical tools to perform statistical tests with RooFit models (*e.g.* Toy Monte Carlo, setting limits). Further, HistFactory provides tools to create RooFit models from a collection of ROOT histograms.

RooFit was originally developed for the BaBar collaboration, but later picked up by many others. Its central parts were designed for single-core processors, and were neither optimised for large caches nor SIMD computations. This work stands at the beginning of efforts to modernise RooFit to speed up fits, and make it more accessible from both C++ and Python. The features to be described in the following two sections will be released in ROOT 6.18.

2. Modernising RooFit's Internal Collections

In RooFit, computation graphs and sets/lists of functions, PDFs, observables and parameters are saved with the help of RooAbsCollection, the base class of RooFit's main collections RooArgSet and RooArgList. Internally, these were using a linked list with optional hash lookup.

The most common operation on these collections during fitting, *i.e.*, repeated evaluation of the computation graph, is iteration. Less frequent operations are appending and finding elements, and collections are very rarely sorted or modified. This favours array-like data structures, and indeed the linked lists were identified as a bottleneck for fits.

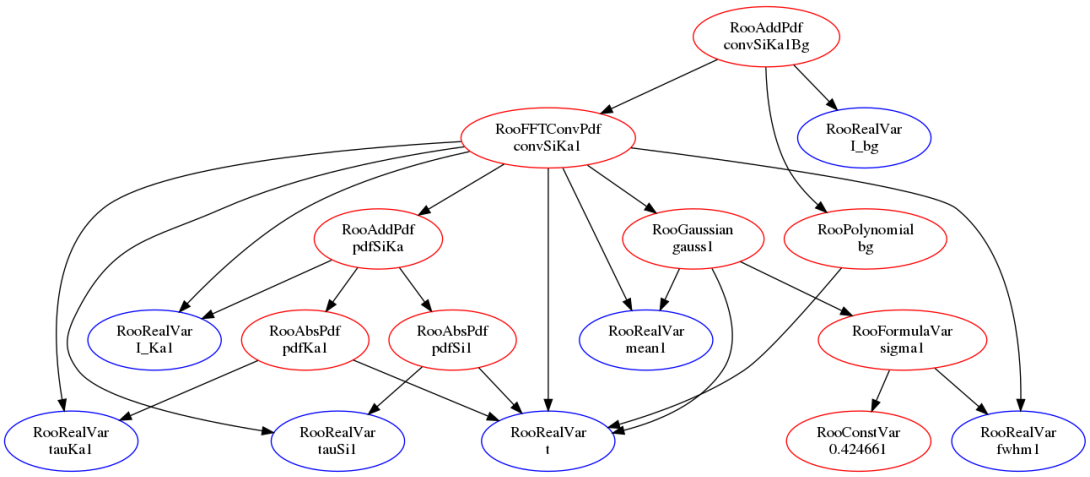

Figure 1. A `RooFit` likelihood model[2]. This model represents the sum of a signal and background PDF, where the former is a convolution of other PDFs, the latter is a polynomial distribution. Likelihood models are implemented as tree-like structures of nodes that can be evaluated to real values. Blue nodes represent parameters or observables, red nodes, which depend on the values of other nodes, represent functions or PDFs.

Therefore, the linked list in `RooAbsCollection` was replaced by a `std::vector`, mostly to speed up iterating. `RooAbsCollection` was further provided with an STL-like interface (`size, begin, end`) to enable range-based for loops. This speeds up forward iteration by about 20 %, and random access completes in constant time. The figures 2 and 3 compare the old and new interface. The STL-like interface allows to reduce heap allocations, and replaces while loops with non-local variables by range-based for loops. This reduces code clutter and the danger of memory leaks, dangling pointers and variable shadowing. It will also facilitate generating `ROOT`'s automatic Python bindings to iterate through `RooFit`'s collections in Python. Figure 4 compares run times for typical `RooFit` workflows[3] between `ROOT 6.16` and `6.18`. Depending on how often collections are iterated, the speed up varies between 5 and 21 %. Tests with an ATLAS likelihood model [3] yielded a speed up of 19 %.

```
TIterator* paramIter = paramList.createIterator() ;
RooAbsArg* param ;
while((param = (RooAbsArg*)paramIter->Next())) {
  _paramList.add(*param) ;
}

delete paramIter ;
```

Figure 2. Iterating through a `RooFit` collection with old interface.

```
for (const auto param : paramList) {
  _paramList.add(*param) ;
}
```

Figure 3. Iterating through a `RooFit` collection with new interface.

2.1. Backward Compatibility

The old interface of `RooAbsCollection` and its subclasses exposed three kinds of iterators to users, one of which is shown in figure 2. All three are being used in `RooFit`, and it is likely that `RooFit`'s users also use all three. Removing any of these would therefore break user code.

To provide backward compatibility, these legacy iterators were re-implemented to also work with the new collections. Since the old `RooFit` collections were based on a linked list, the legacy

[2] This figure was obtained using the `graphVizTree()` export supported by all nodes in a `RooFit` graph.
[3] These are a selection of representative `RooFit` tutorials.

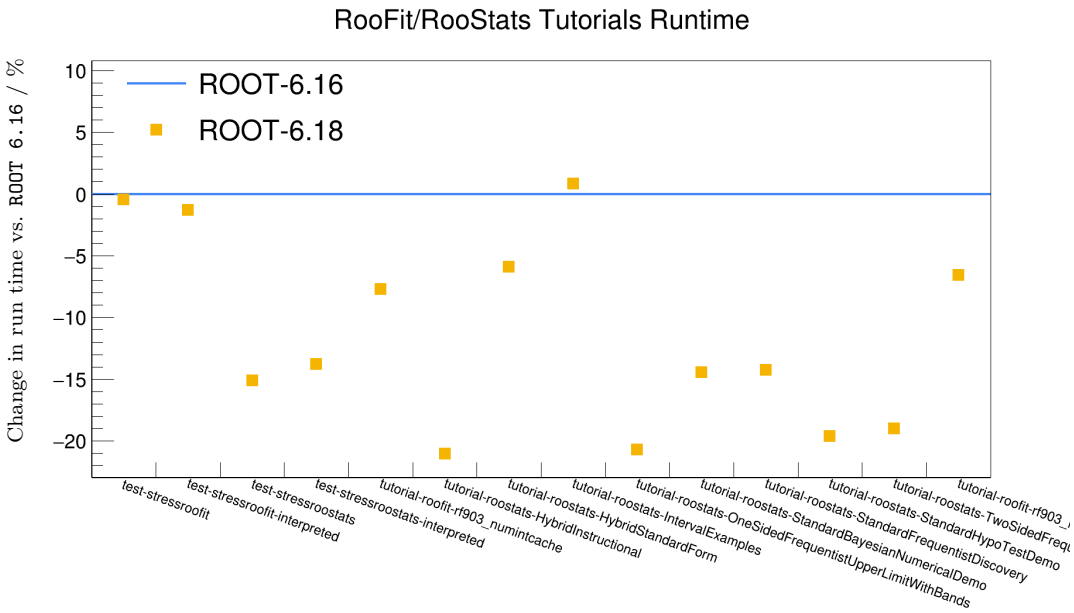

Figure 4. Comparison of run times for typical `RooFit` workflows between `ROOT 6.16` and `6.18`. Frequent use of STL-like iterators leads to a speed up of 20 %, heavy use of legacy iterators to a slow down.

iterators need to remain valid also when reallocations happen. Therefore, the legacy iterators hold a reference to the new collection, and use index access to iterate through it. This makes them tolerant against reallocations, but inserting or deleting elements *before* the current position is not supported, unlike for linked lists. Not a single instance of such usage was found in `RooFit`, though. Nevertheless, run-time checks were added to warn users if they insert/delete before the current iterator. These are performed only if `ROOT` is compiled with assertions enabled, and only for legacy iterators.

All legacy iterators further need to work both with the original `RooLinkedList` and with the STL-based collections[4]. They were therefore implemented as adapters to a polymorphic iterator interface, which supports both the `RooLinkedList` and the counting iterator. This means that all legacy iterators will work irrespective of reallocations and the actual type of the collection, but they are slower than the fastest original `RooFit` iterator, because they require a heap allocation to polymorphically switch between different backends. The slow down observed for one workflow in figure 4 is caused by this.

Nevertheless, instances of slow legacy iterators are easily found using profiling tools, and can be replaced by STL iterators by changing the code as shown in figures 2 and 3, which leads to a speed up of 20 %. The most critical iterators in `RooFit` have already been replaced, and less critical iterators will be replaced as the modernisation of `RooFit` continues.

3. Faster `HistFactory` Models

`HistFactory` [4] is a toolkit to create `RooFit` models from a collection of histograms. It supports multiple channels, multiple signal and background samples, sample scale factors, systematic uncertainties for shape and normalisation differences in histograms, and allows to implement combined measurements of parameters such as a signal strength.

[4] The `RooLinkedList` will be deprecated, but not all instances of its usage have been removed. It might further be used in user code.

To parametrise systematic uncertainties using histograms, users supply three histograms of the same distribution: the nominal distribution and two histograms representing the $\pm 1\,\sigma$ uncertainty. Given that these have to be evaluated for multiple (sometimes hundreds) of systematic uncertainties, for multiple samples and multiple channels, several thousands of histograms might have to be analysed.

When `HistFactory` was implemented, move semantics or shared pointers were not available. The authors therefore resorted to copying histograms, leading to a large overhead of copying and deleting histograms. Performance-critical sections of the `HistFactory` code were therefore revisited, and move semantics as well as shared pointers implemented. This speeds up creating a likelihood model for an ATLAS measurement [3] by more than ten times, with identical results. This model comprises 10 832 histograms, 28 channels and 253 systematic uncertainties, and was constructed in 150 s instead of 1 800 s. Fits using this model furthermore converged 20 % faster because of the optimisations discussed in section 2.

4. Batched Likelihood Computations

A bottleneck for likelihood computations in `RooFit` is the repeated evaluation of the computation graph such as the one shown in figure 1. To compute a likelihood, the probability of observing *each* event in the dataset has to be computed. `RooFit` achieves this by loading the values of the observables into the leaves of the computation graph for a *single* event, and evaluating the probability of the top node. Each node caches its last value, and therefore constant branches of the computation graph (*e.g.* branches that only depend on parameters) are only computed once. Yet, all branches that depend on observables have to be recomputed for each entry in the dataset. In figure 1, for example, the node "t" in the centre of the graph is the observable, whereas other leaves are parameters. This means that the majority of nodes has to be recomputed for each entry in the dataset. Evaluating a node involves virtual function calls, and the number of such calls is proportional both to the number of entries in the dataset and to the number of (non-constant) nodes in the graph, $N_{\text{Data}} \cdot N_{\text{Nodes}}$. For a small graph of 10 nodes and one million events, this already amounts to considerably more than 10 million function calls because additional calls for the normalisation of PDFs and for invalidating (node-local) caches are necessary.

Furthermore, loading single values into the nodes of the computation graph is hostile to CPU caches. There is a high likelihood that when a node is being revisited to load the next entry, data have been thrashed from the highest-level cache(s). This means that `RooFit` runs inefficient on modern CPUs because of poor data locality and inefficient memory access patterns.

4.1. Batch Computations for Higher Data Locality

To demonstrate that a likelihood evaluation can be sped up by loading batches of data, a preliminary interface using `std::span` was implemented for a few selected PDFs (Gaussian, Poisson and exponential distribution, summation of PDFs). Instead of computing only one probability per node, all probabilities for all entries in the data set were computed in a *single* function call for each node. Since such computations operate on array-like structures, data locality, caching and prefetching improve. Figure 5 shows that run times for fitting simple models decrease by a factor 2 to 3.5. The speed up is expected to be even larger for larger models.

4.2. SIMD Computations

When computations on array-like structures are performed, single-instruction-multiple-data computations (SIMD) can be used. The automatic vectorisation optimisation of the clang compiler was used to vectorise computations for an `AVX2` architecture for the Gaussian and exponential distributions, as well as the addition of PDFs and the normalisation of the three. This requires auto-vectorisable mathematical functions, which need to be inlinable, and not

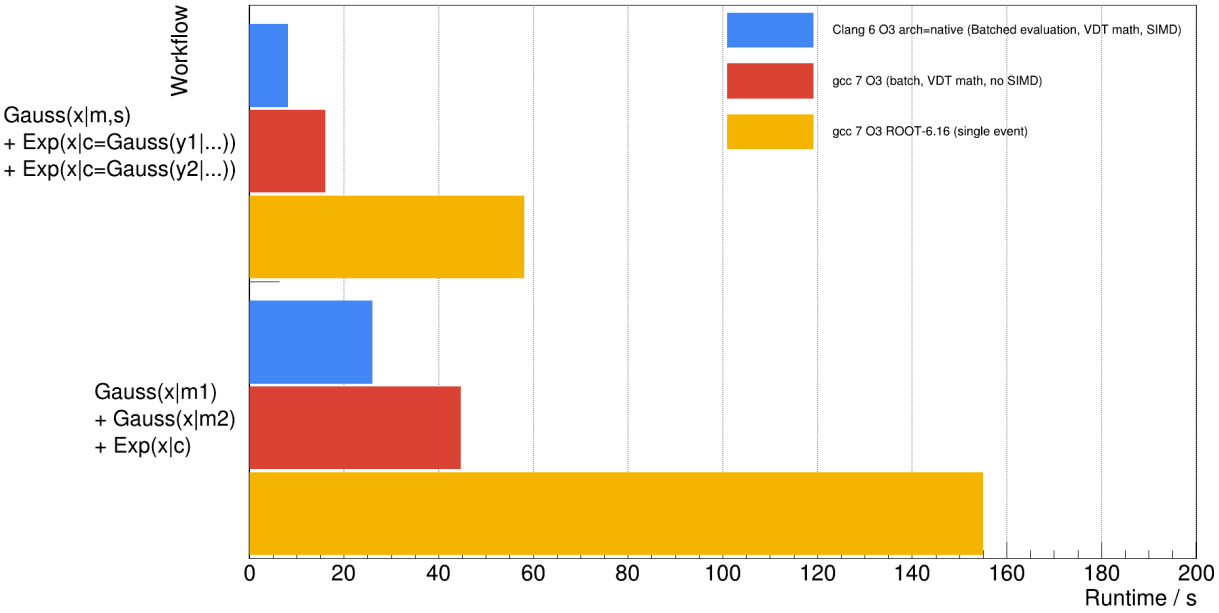

Figure 5. Run time for fitting different likelihood models to datasets of two million events, Intel i7-4790. **Top**: Using `AVX2` SIMD instructions and batch data processing increases the speed by 7×. **Middle**: Batch data processing leads to a speed up of 3.5×. **Bottom**: Current `RooFit`.

have any data dependencies between elements of the underlying arrays. Such functions are provided by the `VDT` [5] package, of which the logarithm and exponential function were used. Further, computations were rewritten such that there are no data dependencies, that they can be executed entirely in registers, and that they use only limited branches, no (non-inlinable) functions and only simple reductions.

Auto vectorisation for these selected PDFs with `AVX2` instructions increased the speed up to 6× to 7×.

5. Summary

The improvements discussed in sections 2 and 3 will be released in `ROOT 6.18`, and the work on batched and vectorised computations will continue. The batch interface will be refined to work with any PDF, and a fall-back implementation for PDFs that have not been modified will be provided. Auto-vectorisable batch computations will be implemented for a growing number of PDFs to increase the single-thread performance of `RooFit` by several factors without requiring code changes on the user side. In conjunction with work an parallelising computations [6], a speed up by more than an order of magnitude can be expected.

References

[1] Verkerke W and Kirkby D P 2003 *econf* **C0303241** MOLT007 (*Preprint* `physics/0306116`)
[2] Brun R and Rademakers F 1997 *Nucl. Instrum. Methods Phys. Res* 81–86 URL `https://root.cern`
[3] ATLAS Collaboration 2015 *JHEP* **01** 069 (*Preprint* `1409.6212`)
[4] Cranmer K, Lewis G, Moneta L, Shibata A and Verkerke W 2012 Histfactory: A tool for creating statistical models for use with roofit and roostats Tech. Rep. CERN-OPEN-2012-016 CERN URL `https://cds.cern.ch/record/1456844`
[5] Piparo D, Innocente V and Hauth T 2014 *J. Phys. Conf. Ser.* **513** 052027
[6] Bos P *et al.* 2019 *J. Phys. Conf. Ser.* **(This volume)**