Making Likelihood Calculations Fast: Automatic Differentiation Applied to RooFit

Garima Singh[1,2,*], *Jonas* Rembser[1,**], *Lorenzo* Moneta[1], *David* Lange[2], and *Vassil* Vassilev[2]

[1]EP-SFT, CERN, Espl. des Particules 1, 1211 Meyrin, Switzerland
[2]Department of Physics, Princeton University, Princeton, NJ 08544, USA

Abstract. With the growing datasets of current and next-generation High-Energy and Nuclear Physics (HEP/NP) experiments, statistical analysis has become more computationally demanding. These increasing demands elicit improvements and modernizations in existing statistical analysis software. One way to address these issues is to improve parameter estimation performance and numeric stability using Automatic Differentiation (AD). AD's computational efficiency and accuracy are superior to the preexisting numerical differentiation techniques, and it offers significant performance gains when calculating the derivatives of functions with a large number of inputs, making it particularly appealing for statistical models with many parameters. For such models, many HEP/NP experiments use RooFit, a toolkit for statistical modeling and fitting that is part of ROOT.

In this paper, we report on the effort to support the AD of RooFit likelihood functions. Our approach is to extend RooFit with a tool that generates overhead-free C++ code for a full likelihood function built from RooFit functional models. Gradients are then generated using Clad, a compiler-based source-code-transformation AD tool, using this C++ code. We present our results from applying AD to the entire minimization pipeline and profile likelihood calculations of several RooFit and HistFactory models at the LHC-experiment scale. We show significant reductions in calculation time and memory usage for the minimization of such likelihood functions. We also elaborate on this approach's current limitations and explain our plans for the future.

1 Introduction

RooFit [1] is a powerful statistical modeling and fitting library widely used in particle physics, particularly at CERN (European Organization for Nuclear Research), for performing complex statistical analyses of experimental data. It is built on top of the ROOT [2] data analysis framework and provides tools specifically designed for constructing and fitting probability density functions (PDFs) to experimental data. RooFit plays a crucial role in a wide range of physics analyses, particularly in experiments at the Large Hadron Collider (LHC), where it is used to extract valuable information about particle properties, interactions, and potential new phenomena. As such, it is crucial to expand the capabilities of RooFit to accommodate

*e-mail: garima.singh@cern.ch
**e-mail: jonas.rembser@cern.ch

the increasing intricacy of data analyses spanning various areas of physics, including the emerging experiments at the LHC.

In the usual studies conducted with RooFit, one has to minimize log-likelihoods with up to thousands of free parameters spread over up to hundreds of likelihood components, each representing a different measurement channel. Performing this minimization iteratively, one generally has to know the likelihood gradient with respect to all free parameters. Naively, the gradient can be found numerically by varying one parameter at a time and reevaluating the full likelihood. In most cases, however, it is unnecessary to reevaluate the full mathematical expression when only one parameter is changed, which is why RooFit includes a sophisticated caching mechanism. Here, individual mathematical components of the model are represented as separate objects (or groups thereof) that cache their previous results. These results are then used to selectively reevaluate only the parts of the model that are affected by a change in the input parameters during the numerical gradient calculation.

While the caching approach works well for reducing the overall numeric gradient calculation time, it still has plenty of shortcomings that can incur significant overhead. For example, it requires lots of bookkeeping of values, virtual function calls, redundant revaluation if the caching granularity is not high enough, etc. Our work aims to remove these shortcomings by deviating from the caching paradigm and instead translating the full expression graph to overhead-free, independent C++ code. This C++ code representation of the statistical model can then be used with any third-party tool to provide high-performance Automatic Differentiation (AD) based gradients that can be used to replace numerical gradient calculations at various places in RooFit [3].

2 Background

2.1 Automatic Differentiation

Automatic Differentiation (AD), or Algorithmic Differentiation, is a set of mathematical techniques that can efficiently calculate accurate derivatives of computer programs. AD breaks down a complex program into an elementary set of instructions and then applies the chain rule of differential calculus over those instructions to propagate the derivatives through the program. AD can also handle functions with multiple inputs and outputs, making it particularly suitable for applications with high-dimensional parameter spaces, like RooFit.
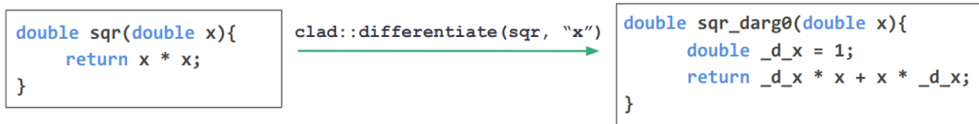
AD majorly defines two modes of operation - forward accumulation mode and reverse accumulation mode. For an arbitrarily complex function $y = f(g(x))$ that can be split into intermediate steps as follows:

$$w_0 = x \quad w_1 = g(x) \quad w_2 = f(g(x)) = y \tag{1}$$

The forward and reverse accumulation modes are defined as follows:

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}} * \frac{\partial w_{i-1}}{\partial x} \quad \text{and} \quad \frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} * \frac{\partial w_{i+1}}{\partial w_i} \tag{2}$$

AD has many advantages over numerical differentiation, the mode of gradient calculation used in RooFit. Firstly, unlike numerical differentiation, which only calculates an estimate, AD calculates the exact derivative of a program. AD is also more suited to problems with a large number of parameters because it does not require a full reevaluation of the target function every time a gradient with respect to a different parameter is requested.

```
double sqr(double x){          clad::differentiate(sqr, "x")      double sqr_darg0(double x){
    return x * x;                                                     double _d_x = 1;
}                                                                     return _d_x * x + x * _d_x;
                                                                  }
```

Figure 1. An example function being forward-differentiated by Clad. Here the RHS represents the derivative code generated by Clad.

2.2 Source Code Transformation based AD

While there are multiple flavors of AD, the two most popular are operator overloading AD and Source Code Transformation AD. Operator Overloading (OO) AD typically defines a custom data type that augments the basic arithmetic operations with derivative generation logic. This allows the tool to track derivatives as the program executes these overloaded operations. Some examples of OO tools include PyTorch [4], CoDiPack [5], etc. On the other hand, Source Code Transformation (SCT) synthesizes the derivative code by analyzing the source code of the target program. The derivative code can then be compiled and executed as regular code. Examples of SCT tools are Clad [6], Enzyme [7], etc.

While the OO AD approach is suitable for smaller and newer projects, it is hard to use OO on larger and more complex applications because it often requires handwriting annotations and changing data types. On the other hand, as long as the AD tool supports the constructs used in the target application, SCT requires no additional changes. Moreover, since the SCT approach generates code that can be compiled and subject to compiler optimizations, it is usually more performant than the OO approach.

For these reasons, we use Clad [6], a compiler-based SCT tool for programs written in C/C++. Clad inspects the internal compiler representation of the target function to generate its derivative. Fig. 1 showcases an example program transformation Clad performs on the target function to generate its derivative. Clad's proximity to the compiler allows for more control over the derivative synthesis, allowing us to also insert RooFit-specific optimizations while building the derivative code. Moreover, Clad has good support for modern C++ constructs and has off-the-shelf integration with Cling [8] (the C++ interpreter used in ROOT), making it an ideal choice for our work.

3 Design and Implementation

While the AD techniques and tools discussed in the previous section work very well for most applications, they struggle to correctly differentiate applications with side effects (such as classes and other object-oriented programming constructs). AD tools rely on access to the mathematical operations and dependencies between variables in a program to compute derivatives accurately. When these details are hidden or abstracted in a way that is not easily accessible, either through high-level object-oriented design principles or abstraction at lower levels, it becomes challenging to integrate AD into such frameworks.

RooFit is a prime example of such a framework as it relies heavily on the use of high-level abstractions, often hiding the mathematical meaning of the different components, to promote ease of use. One example of such an abstraction in RooFit is illustrated in Fig. 2, where a common mathematical formula is represented by a C++ object of the same name. RooFit has many such mappings between different mathematical notations and "*Roo*" objects. While these mappings are crucial to keep RooFit from becoming over-complicated and hard to use, they often obfuscate important details required by AD tools for differentiation.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \longrightarrow$$

```
//Obj represents f(x) here
RooGaussian obj(x, mu, sigma);
```

Gaussian Probability
Distribution Function (PDF)

Equivalent Code in C++ with RooFit

Figure 2. An example of data abstraction implemented by RooFit. In RooFit, whenever a user wants to use a Gaussian PDF, they have to use an object of type *RooGaussian* instead. While this connection is obvious to the user or programmer, it is not so obvious to an AD tool like Clad.

To combat this issue, we introduce a "translation" system as a pre-processing step in the AD pipeline. This translation step aims to extract the differentiable properties of each of the abstracted RooFit classes such that they can be differentiated using AD. Since we use Clad in our work, we extract these differentiable properties into a single, stateless C++ function. We do this translation by introducing two functions for each RooFit class – a stateless function describing the underlying mathematical notation of the class as simple C++ code and a "glue" function that enables all of the former stateless functions of the nodes in the compute graph to be "glued" together to make a single free C++ function. In this work, we refer to this pre-processing step as *Code-Squashing*. This "squashed" function represents the full RooFit model and can later be compiled by the C++ interpreter Cling and then be differentiated using Clad. Fig. 3 describes an example of the functions discussed above for the *RooGaussian* class (as seen in Fig. 2). Further, Fig. 4 illustrates an example of the generated *squashed-code* from a simple RooFit workspace description.

Stateless function enabling differentiation of each class.

The "glue" function enabling graph squashing.

```cpp
double ADDetail::gauss(double x, double m, double s)
{
 const double arg = x - m;
 return std::exp(-0.5 * arg * arg / (s * s));
}
```

```cpp
void RooGaussian::translate(...) override {
  return "ADDetail::gauss(" +
                 _x->getRes() + " ,"
               + _mu->getRes() + " ,"
               + _sigma->getRes() + ")";
}
```

Figure 3. An example set of functions that enable translation for the *RooGaussian* class. Here, the former function (function *gauss*) just represents the Gaussian PDF (not normalized) as C++ code, and the latter function (function *translate*) builds a call to *gauss* in the form of an *std :: string*. These strings are concatenated from all the different classes in a model to form the final C++ code that represents the full model.

4 Results

We benchmarked fitting a simple RooFit model composed of the sum of two Gaussians and an exponent term. We benchmarked the regular RooFit fitting pipeline with code-squashing numerical differentiation (numerical differentiation applied to the generated C++ function) and code-squashing AD (AD applied to the squashed function).

Fig. 5 showcases the performance comparison results between the different fitting configurations for up to 700 parameters. We fit this simple model with independent parameters in independent channels, where we arbitrarily scale up the number of channels to increase the number of parameters for our benchmark. Each channel here contains 2 parameters from

```
double myGauss(double *params, double const *obs)
{
  const double sigma_scaled = params[2] * 1.5;      "prod::sigma_scaled(sigma[3.0, 0.01, 10], 1.5)"
  const double mu_shifted = params[0] + params[1];   "sum::mu_shifted(mu[0, -10, 10], shift[1.0, -10, 10])"
  const double gauss_Int_x = ADDetail::gaussianIntegral(-10, 10, mu_shifted, sigma_scaled);
  const double gauss = ADDetail::gauss(params[3], mu_shifted, sigma_scaled);
  const double normGauss = gauss / gauss_Int_x;      "Gaussian::gauss(x[0, -10, 10], mu_shifted, sigma_scaled)"
  return normGauss;
}
```

Figure 4. Example code generated from a simple AD-compatible RooFit workspace. Here, the *params* **and** *obs* **represent the model inputs and observables. Moreover, the lines highlighted in green represent the workspace commands used to generate the model. Each highlighted line is mapped to the corresponding code it generates after a call to** *translate***.**

each Gaussian term, 1 from the exponent term and 2 from their relative weights. In total, each channel contains 7 parameters, and we scale up to 100 channels in the benchmarks from Fig. 5.

We found that while the vanilla numerical-diff applied to the squashed code is no match for RooFit's numerical-diff due to RooFit's superior caching and bookkeeping mechanisms, AD performs significantly better, even for a smaller number of parameters. Specifically, code-squashing AD is up to **18 times** faster than regular RooFit for around 700 parameters. In this benchmark, we report the minimization time excluding the *seeding step* (the initial parameter scale estimation step) as this step largely uses numerical-diff in a way that is not compatible with the AD pipeline. This usually results in very inflated seeding times for the code-squashing AD configuration. One way to fix these inflated times could be by using an AD-compatible seeding mechanism in this step. Since the seeding step and its implementation are largely dependent on MINUIT2 [9], the minimizer we used in our benchmarks, it is also possible to mitigate this issue by using a different minimizer altogether.
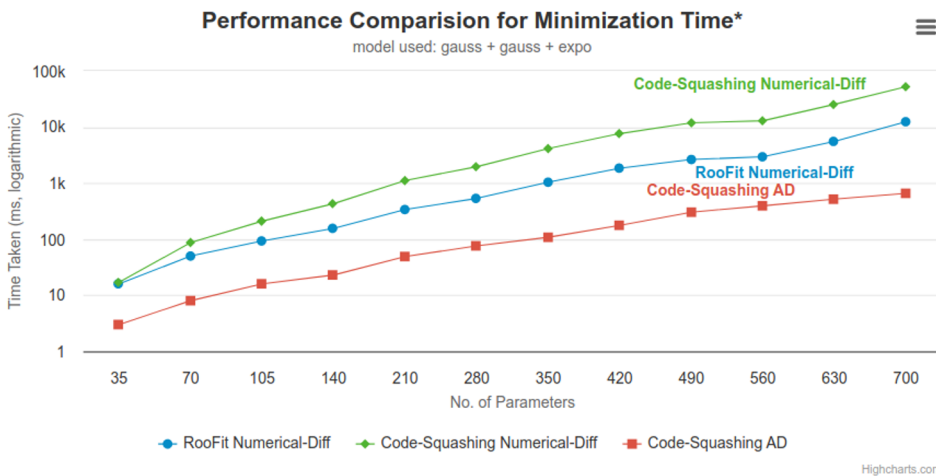
Figure 5. Comparison of the logarithmic time taken for the minimization as the number of parameters is varied. These times exclude the seed generation time. Tested using Google Benchmark on the ROOT development branch as of May 2023.

Table 1. Performance comparison breakdown of the 3 evaluated configurations in Fig. 5 for 700 parameters.

Configuration	Gradient Time (in ms)	Iterations	Minimum Value
RooFit Numerical-Diff	86	136	659552.2917
Code-Squashing Numerical-Diff	380	136	659552.29**18**
Code-Squashing AD	11	58	65955**1.9860**

To get a deeper understanding of the results and what that means for the total minimization process, we compare all 3 of the configurations on 3 different metrics - single gradient calculation time, total iterations taken to converge, and the final minimized model/function value. The results of these comparisons are defined in Tab. 1.

Here, while there is not much difference in the number of iterations to converge and the final minimum value for both the numerical-diff configurations, we notice a significant difference between the gradient calculation times. This is majorly because the Code-Squashing numerical-diff does not employ the many sophisticated caching mechanisms used in RooFit's numerical-diff. These caching mechanisms were put in place solely to optimize the numerical-diff calculation time, and this is reflected well in our analysis.

On the other hand, when comparing RooFit numerical-diff and Code-Squahing AD, we notice a significant difference in the time taken for gradient calculation. AD is up to **7 times** faster than numerical-diff for a single gradient evaluation. There is also a significant difference in the number of iterations to converge for both configurations, here we notice that AD converges much faster and part of the reason why can be that AD is much more numerically robust and does not suffer from the same numerical errors as in numerical-diff, leading to a more stable minimization path and hence faster convergence. This is especially true if the target model has a large number of parameters or contains exponent terms (as in our case) which are significantly more sensitive to even small errors. However, it is worth mentioning that the initial minimization setup might also largely affect the number of iterations to converge for all 3 of the configurations, i.e. for different minimizer starting configurations, it is possible to get vastly different results for the total iterations to converge. Lastly, minimization done with AD produces a marginally *better* minimum value as opposed to numerical-diff, which produces a similar minimum. This can also be attributed to AD's overall numerical stability when compared to numerical-diff and/or the starting minimizer configuration.

5 Conclusion and Future Work

Our work presents an efficient way to translate complex models such that they can be differentiated using AD. We demonstrate that AD can be used to effectively lower the fitting time for non-trivial models. In our future endeavors, we aim to improve our approach by eliminating the requirement for numerical derivatives in the minimization process. Currently, RooFit relies on numerical differentiation during the initial seeding phase of minimization, which leads to numerous compatibility challenges with AD. Consequently, substituting numerical derivatives with AD-based derivatives at this stage would facilitate a more seamless integration of AD. Additionally, we intend to investigate various methods to optimize the gradients generated by Clad, such as employing CPU parallelization and harnessing the computational power of GPUs to improve gradient (and possibly model) evaluation times. Moreover, we want to work towards producing more optimal Sqaushed-Code and reducing the JIT times for larger models.

Finally, we aim to collaborate with more experiments to help them onboard and integrate our work into their existing workflows. This will not only allow us to improve our work but also benefit others in the community.

6 Acknowledgements

References

[1] W. Verkerke, D.P. Kirkby, eConf **C0303241**, MOLT007 (2003), `physics/0306116`

[2] R. Brun, F. Rademakers, P. Canal, A. Naumann, O. Couet, L. Moneta, V. Vassilev, S. Linev, D. Piparo, G. GANIS et al., *root-project/root: v6.18/02* (2019), `https://doi.org/10.5281/zenodo.848818`

[3] G. Singh, J. Rembser, L. Moneta, D. Lange, V. Vassilev, *Automatic Differentiation of Binned Likelihoods With Roofit and Clad* (2023), `arXiv:2304.02650`

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., in *Advances in Neural Information Processing Systems 32* (Curran Associates, Inc., 2019), pp. 8024–8035, `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`

[5] N.G. M. Sagebaum, T. Albring, ACM Transactions on Mathematical Software (TOMS) **45** (2019)

[6] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, V. Ilieva, *Clad – Automatic Differentiation Using Clang and LLVM* (IOP Publishing, 2015), Vol. 608, p. 012055, `https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012055/pdf`

[7] W. Moses, V. Churavy, *Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients*, in *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, H. Lin (Curran Associates, Inc., 2020), Vol. 33, pp. 12472–12485, `https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf`

[8] V. Vassilev, P. Canal, A. Naumann, L. Moneta, P. Russo, *Cling – The New Interactive Interpreter for ROOT 6* (IOP Publishing, 2012), Vol. 396, p. 052071, `https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071/pdf`

[9] F. James, M. Roos, Comput. Phys. Commun. **10**, 343 (1975)