EPJ Web of Conferences **245**, 02015 (2020) https://doi.org/10.1051/epjconf/202024502015
CHEP 2019

Automatic Differentiation in ROOT

Vassil Vassilev[1,*] *Aleksandr* Efremov[1,**] *Oksana* Shadura[2,***]

[1]Princeton University, Princeton, New Jersey 08544, United States
[2]University of Nebraska Lincoln, 1400 R St, Lincoln, NE 68588, United States

Abstract. In mathematics and computer algebra, automatic differentiation (AD) is a set of techniques to evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.), elementary functions (exp, log, sin, cos, etc.) and control flow statements. AD takes source code of a function as input and produces source code of the derived function. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

This paper presents AD techniques available in ROOT, supported by Cling, to produce derivatives of arbitrary C/C++ functions through implementing source code transformation and employing the chain rule of differential calculus in both forward mode and reverse mode. We explain its current integration for gradient computation in TFormula. We demonstrate the correctness and performance improvements in ROOT's fitting algorithms.

1 Introduction

Accurate and efficient computation of derivatives is vital for a wide variety of computing applications, including numerical optimization, solution of nonlinear equations, sensitivity analysis, and nonlinear inverse problems. Virtually every process could be described with a mathematical function, which can be thought of as an association between elements from different sets. Derivatives track how a varying quantity depends on another quantity, for example how the position of a planet varies as time varies.

Derivatives and gradients (vectors of partial derivatives of multivariable functions) allow us to explore the properties of a function and thus the described process as a whole. Gradients are an essential component in gradient-based optimization methods, which have become more and more important in recent years, in particular with its application training of (deep) neural networks [1].

Several different techniques are commonly used to compute the derivatives of a given function, either exactly or approximately [1–3]. The most prevalent techniques are:

*e-mail: vvasilev@cern.ch
**e-mail: xefremale@gmail.com
***e-mail: oksana.shadura@cern.ch

- *Numerical differentiation*, based on the *finite difference* method, provides a way to evaluate derivatives approximately. While simple, numerical differentiation can be slow (the runtime complexity grows linearly with the number of input variables) and may have problems with accuracy due to round-off and truncation errors.
- *Symbolic differentiation*, based on transformations of symbolic expressions of functions, provides exact closed-form expressions for the derivatives. It faces difficulties when the function to be differentiated is not available in a closed form, which is often the case for computer programs which may contain control flow. Symbolic differentiation can produce derivative expressions that are computationally expensive to evaluate due to difficulties in exploiting common subexpressions.
- *Automatic differentiation* (AD) computes derivatives accurately to the precision of the original function, supports control flow and uses at most a small constant factor more time and space than it takes to evaluate the original function, at the expense of increased implementation complexity and introducing more software dependencies.

Numerical and symbolic differentiation methods are slow at computing gradients of functions with many input variables, as is often needed for gradient-based optimization algorithms. Both methods have problems calculating higher-order derivatives, where the complexity and errors due to numerical precision increase. Automatic differentiation largely avoids the problems of numerical and symbolic differentiation.

In this paper, we describe the implementation of automatic differentiation techniques in ROOT, which is the data analysis framework broadly used High-Energy Physics [4]. This implementation is based on Clad [5, 6], which is an automatic differentiation plugin for computation expressed in C/C++.

2 Background

Here, we briefly discuss main algorithmic and implementation principles behind AD. An in-depth overview and more formal description can be found in [2] and [3], respectively.

2.1 AD and its Modes

AD is based on the decomposition of the procedure (e.g. a source code that computes the original function) into a sequence of simple mathematical operations (e.g. $+, -, *, /, \sin, \cos, \exp$) that can be expressed using a series of intermediate results. Subsequently, derivatives of every intermediate result are evaluated and combined via the chain rule of calculus to obtain the derivatives of the whole sequence. The control flow (e.g. branches, loops) can be incorporated by differentiating the control flow of the original function during the derivative evaluation. Two main modes of AD, which differ in the order of application of the chain rule, are used:

- *Forward mode* operates in a top-down approach and computes the derivative of every intermediate result with respect to a single selected input variable of the function. As soon as a final result of the function is reached, the partial derivative with respect to the selected input is available. A single evaluation of the forward mode can only compute partial derivatives with respect to a single input variable. Thus, when the whole gradient is required, forward mode must be invoked once per every input variable, leading to $m \cdot c_F \cdot n$ runtime complexity, where m is the number of input variables, n is the algorithmic complexity of the original function and $c_F < 3$ is a small constant factor overhead of a single invocation of the forward mode [3].

- *Reverse mode* operates in a bottom-up approach and computes the derivative of a function's output with respect to every intermediate result. Once every input variable of the function is reached, the whole gradient of an output is available. Note that, independently on the number of input variables N, a single evaluation of the reverse mode is sufficient to get the whole gradient of a function's output, leading to $c_R \cdot n$ runtime complexity, where n is the complexity of the original function and $c_R \leq 4$ is a small constant factor overhead [3]. This is a huge advantage in settings with a single scalar output and many inputs, which is often the case in machine-learning problems where $N >> 10^6$ that makes the forward mode infeasible. As a disadvantage, reverse mode implementations are more complicated, and dynamic memory allocations may be required when dynamic control flow is involved. Depending on the original function, this may cause a single evaluation of the reverse mode to be somewhat slower compared to a single evaluation of the forward mode.

2.2 AD Implementations

AD techniques have been implemented in a variety of programming languages and paradigms, ranging from classical tools for Fortran [7] and C [8], to recent active work on tools specific to machine-learning applications [9, 10], and modern general-purpose programming languages [11, 12]. We refer the reader to `www.autodiff.org` for a comprehensive list of available AD implementations for various languages.

In particular, several implementations exist for C++, e.g. [13–15]. Majority of implementations of AD fall into one of the two categories of implementation techniques:

- Tools based on *operator overloading* utilize features of programming languages like C++ and Python to define custom types and overload mathematical operators (e.g. +, -, *, /) and functions (e.g. $\exp, \sin, \cos$) on them. Such implementations are often based on custom AD-enabled types that wrap values of both the original and derivative functions and redefine operators to simultaneously act on original and derivative values. In C++, such tools are often implemented as a library that introduces templated differentiable types and corresponding mathematical operations. Then, functions called on the custom type return both original and derivative values. This is a powerful technique but has two primary limitations: legacy code and performance. Functions must be either polymorphic (templated) or explicitly defined on AD-enabled type to be differentiated. Differentiation of pre-existing source code using builtin types such as `double` and `float` is not possible. Users are required to use additional level of abstraction in the form of library-specific types instead of first-class language features. Moreover, the performance of the derivative generation can be suboptimal due to the C++ metaprogramming system which usually constructs deep template instantiation chains. Performance can be even more problematic when creating a higher order derivatives.
- Tools based on *source transformation* analyze the source code of the original function and build another source code for the derivative function. Such techniques typically accept and generate any code using built-in features of the original language and do not require custom libraries. On the other hand, they require an additional pass over the source file to analyze and generate derivative code. Source transformation can fully utilize source-level optimizations and has reasonably good performance. Implementation is more complicated and it is problematic to achieve full coverage of C++ language features. While full integration with a compiler can make AD a first-class language feature that is transparent for the user, most current implementations for C++ are based on custom parsers that do not have full coverage of the vast variety of C++ language constructs and require a separate step before compilation.

3 Architecture and Implementation

Automatic differentiation in ROOT is based on Clad [5, 6]. Clad is a source transformation AD tool for C++. It is based on LLVM compiler infrastructure [16] and is implemented as a plugin for C++ compiler *Clang*, which allows Clad to be transparently integrated into the compilation phase and to utilize large parts of the compiler. Clad relies on Clang's parsing and code generation functionality and can differentiate complicated C++ constructs. Clad supports both forward and reverse mode. It is available as a standalone Clang plugin that, when attached to the compiler, produces derivatives in the compilation phase.

On top of that, Clad is integrated directly into ROOT to provide AD functionality as an integral part of the framework. ROOT has a C++ interpreter *Cling* [17] which is built on the top of LLVM and Clang. This allows Clad to be attached to Cling as a plugin in a similar way as it can be attached to Clang. In this section, we discuss 1) architecture of Clad and its interaction with Cling; and 2) details of its integration into ROOT.

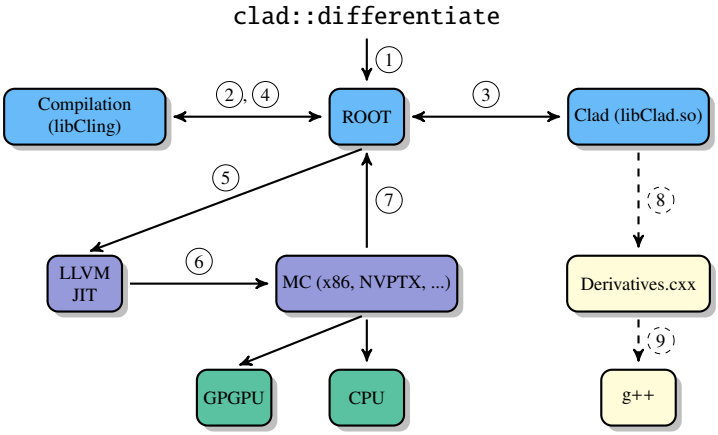


Figure 1: Information flow of Clad in ROOT

Clad operates on *Clang AST* (abstract syntax tree) by analyzing the AST of the original function and generating the AST of the derivative. Clad provides two API functions: `clad::differentiate` for forward mode and `clad::gradient` for reverse mode, which can be used directly in the source code to mark a function for differentiation (see [6] for more details on usage and code examples).

The information flow of interactions with Cling during differentiation (Figure 1) is:

- A function is marked for differentiation with the C++ construct `clad::differentiate` or `clad::gradient` (step 1).
- Cling in ROOT performs incremental compilation and receives an abstract syntax tree (AST) representation of the code (step 2).
- Cling detects the differentiation marker and sends the AST of the original function to Clad, which transforms the AST to produce the AST of the derivative (step 3).
- Clad returns the derivative AST to Cling for code generation and execution by the low level LLVM primitives (steps 4, 5, 6, 7). *Alternatively, if Clad was configured for non-interactive use, the generated AST can be converted to a C++ source code and written to a text file. The generated code then can be compiled with any C++ compiler (steps 8, 9).*

Inside of ROOT, interface functions `clad::differentiate` and `clad::gradient` are accessible via `#include <Math/CladDerivator.h>`. Clad is also directly integrated into the `TFormula` class that encapsulates the concept of multidimensional mathematical functions in ROOT. `TFormula` is a primitive in ROOT's math package which is connected to the Cling interpreter. In the context of `TFormula`, Clad can differentiate functions available in the interpreter. The `TFormula::GenerateGradientPar` method uses Clad to differentiate the underlying code of the formula with respect to its parameters and generate the code for the gradient. `TFormula::GradientPar` method then evaluates the gradient at a specified point.

4 Results

In this section, we empirically compare automatic differentiation (AD, our implementation based on Clad) and numerical differentiation (ND, based on finite difference method) in ROOT. We show that AD can drastically improve accuracy and performance of derivative evaluation, compared to ND.

4.1 Accuracy

As stated in Section 1, numerical differentiation may give imprecise results while AD computes the derivatives exactly. We show an example of a function where this difference is apparent: AD provides exact result while ND suffers from the loss of accuracy.

$$p(x) = \frac{1}{\pi}\frac{\frac{1}{2}\Gamma}{x^2 + (\frac{1}{2}\Gamma)^2} \quad (1) \qquad \frac{\partial p(x)}{\partial \Gamma} = -\frac{2}{\pi}\frac{\Gamma^2 - 4x^2}{(\Gamma^2 + 4x^2)^2} \quad (2)$$

The function is the PDF of *Breit-Wigner* distribution (Eq. 1), whose derivative with respect to Γ (Eq. 2) has critical points at $\Gamma = \pm 2x$. In ROOT, the function is implemented as in (Listing 1).

```
inline double breitwigner_pdf(double x, double gamma, double x0 = 0) {
  double gammahalf = gamma/2.0;
  return gammahalf/(M_PI * ((x-x0)*(x-x0) + gammahalf*gammahalf));
}
```

Listing 1: Breit-Wigner PDF implementation in ROOT

When evaluating the derivative of `breitwigner_pdf` with respect to `gamma` at `x`=1, `gamma`=2, ND in ROOT the yields a result close to 0 with an absolute error of 10^{-13} despite the fact that the function is smooth and well-conditioned at this point. The approximation error becomes larger when the derivative is evaluated further from the critical point. In contrast, the automatic differentiation (in both modes) yields the *exact* result of 0.

4.2 Performance

Section 2 showed that reverse mode AD computes gradients in a single pass with a runtime complexity of at most $4 \cdot n$, which depends only on the complexity n and not the dimensionality dim of the original function. On the other hand, numerical differentiation requires a separate evaluation of the original function for every dimension to compute the entire gradient, making the overall the run-time complexity of gradient evaluation via central finite difference method $2 \cdot dim \cdot n$. Hence, in theory, reverse mode achieves an asymptotic speedup of $O(dim)$ over the numerical differentiation and can be up to $dim/2$ times faster.

We experimentally verify this by comparing the performance of gradient evaluation produced by reverse mode AD against our an implementation of numerical differentiation via

the central finite difference method. We use the two functions in Listing 2: `sum`, which computes the sum of all values in a vector; and `mvn`, which implements the PDF of a multivariate normal distribution. Both functions have a parameter `dim` which defines the dimension, and gradients are taken with respect to `dim`-dimensional vector `p`. While closed-form expressions of these gradients are well-known, these functions make a good basis of a benchmark as they perform typical operations that are commonly found inside more complicated functions (e.g. `+`, `*`, `pow`, `exp` inside loop).

```
double sum(double* p, int dim) {
  double r = 0.0;
  for (int i = 0; i < dim; i++)
    r += p[i];
  return r;
}
```

```
double mvn(double* x, double* p /*means*/,
           double sigma, int dim) {
  double t = 0;
  for (int i = 0; i < dim; i++)
    t += (x[i] - p[i])*(x[i] - p[i]);
  t = -t / (2*sigma*sigma);
  return std::pow(2*M_PI, -n/2.0) *
         std::pow(sigma, -0.5) * std::exp(t);
}
```

Listing 2: Implementations of `sum` and `mvn` functions

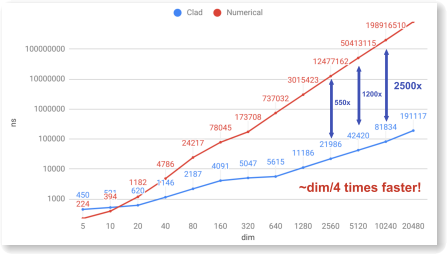
Gradients of `sum` produced by numerical differentiation and Clad are shown in Listing 3.

```
double* sum_num_grad(double* p, int dim,
                     double eps = 1e-8) {
  double result = new double[dim]{};
  for (int i = 0; i < dim; i++) {
    double pi = p[i];
    p[i] = pi + eps;
    double v1 = sum(p, dim);
    p[i] = pi - eps;
    double v2 = sum(p, dim);
    result[i] = (v1 - v2)/(2 * eps);
    p[i] = pi;
  }
  return result;
}
```

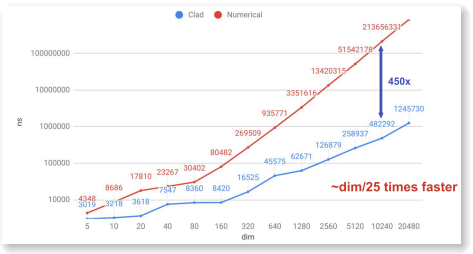
```
void sum_ad_grad(double *p, int dim,
                 double *_result) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    _d_r += 1;
    for (; _t0; _t0--) {
        double _r_d0 = _d_r;
        _d_r += _r_d0;
        _result[clad::pop(_t1)] += _r_d0;
        _d_r -= _r_d0;
    }
}
```

Listing 3: Gradient of `sum`: **(left)** using finite differences, **(right)** generated by Clad

We perform the evaluation for values of `dim` between 5 and 20480. Figure 2 shows the comparison for **(a)** `sum`; **(b)** `mvn` and confirms the expected theoretical speedup of $O(dim)$, with AD-generated gradient being $dim/4$ times faster for `sum` and $dim/25$ times faster for `mvn` (slowdown is due to more expensive operations like `pow`, `exp`).



(a) Performance of gradients of `sum`

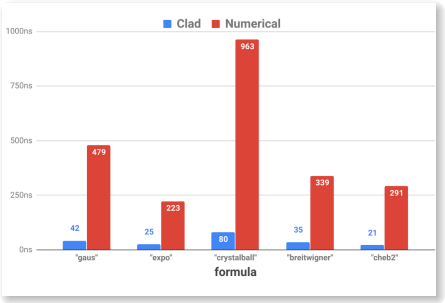


(b) Performance of gradients of `mvn`

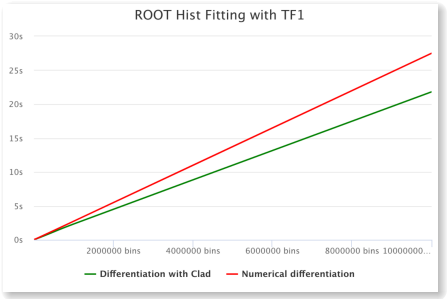
Figure 2: Comparison of reverse mode AD and ND with increasing dimension

4.3 Performance in *TFormula*

Figure 3a shows the performance comparisons of reverse-mode AD and ND for the task of evaluating gradients of `TFormula`'s builtin primitive probability density functions. The functions are `gaus` ($dim = 3$), `expo` ($dim = 2$), `crystalball` ($dim = 5$), `breitwigner` ($dim = 5$) and `cheb2` ($dim = 4$). Despite the low dimensionality ($dim \leq 5$), AD gives significant (approx. 10x) speedups. The speedups are even larger than expected factor of $dim/2$ that follows from theoretical results, apparently due to additional overhead of the implementation of numerical differentiation in ROOT, which tries to find the optimal step size for its finite difference method to improve accuracy.



(a) Comparison of the performance of `TFormula` gradients between AD and ND



(b) ROOT Histogram fitting using numerical differentiation versus Clad differentiation

Figure 3: Performance benchmarks in ROOT

In Figure 3b, we perform fitting of a Gaussian distribution to a histogram of random samples via gradient-based optimization. In ROOT, this functionality is implemented in `TFormula`-based `TF1` class. We can therefore use AD due to the integration of Clad into `TFormula`. Figure 3b compares the performance of the AD-based `TF1` fitting with the numerical fitting in the `Hist` package. As in previous experiments, we show that AD scales better with problem dimensionality (number of histogram bins) on this task. The integration of Clad into `TFormula` makes it straightforward to use AD for fitting in ROOT.

5 Conclusion

We discussed our implementation of automatic differentiation in ROOT based on Clad. We demonstrated that Clad is integrated into ROOT and can be easily used in various contexts inside ROOT (e.g. histogram fitting). Furthermore, we showed that automatic differentiation in ROOT achieves significant improvements in accuracy and performance over numerical differentiation. The performance and accuracy are promising and encourage further work in the development of Clad and its integration in ROOT.

Possible further improvements for Clad include optimizations to code transformation and design of a consistent interface for derivatives and gradients computation. This functionality can be further extended, including the computation of Jacobians and higher-order derivatives. In order to achieve optimal performance, the evaluation of individual derivatives could be executed in parallel. Besides, the Clad API should enable a flexible execution method based on the needs of its user.

6 Acknowledgments

This work has been supported by U.S. NSF grants PHY-1450377 and 1450323.

References

[1] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, *Automatic differentiation in machine learning: a survey*, in *The Journal of Machine Learning Research* (JMLR.org, 2017), Vol. 18, pp. 5595–5637

[2] A.H. Gebremedhin, A. Walther, *An introduction to algorithmic differentiation*, in *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (Wiley Online Library, 2020), Vol. 10, p. e1334

[3] A. Griewank, A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Vol. 105 (SIAM, 2008)

[4] R. Brun, F. Rademakers, *ROOT — An object oriented data analysis framework*, in *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* (Elsevier BV, 1997), Vol. 389, pp. 81–86

[5] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, V. Ilieva, *Clad – automatic differentiation using Clang and LLVM*, in *Journal of Physics: Conference Series* (IOP Publishing, 2015), Vol. 608, p. 012055

[6] V. Vassilev, *Clad – automatic differentiation for C/C++* (2014), `https://github.com/vgvassilev/clad/`

[7] C. Bischof, P. Khademi, A. Mauer, A. Carle, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, in *IEEE Computational Science and Engineering* (IEEE, 1996), Vol. 3, pp. 18–32

[8] C.H. Bischof, L. Roh, A.J. Mauer-Oats, *ADIC: an extensible automatic differentiation tool for ANSI-C*, in *Software: Practice and Experience* (Wiley Online Library, 1997), Vol. 27, pp. 1427–1456

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, *Automatic differentiation in PyTorch* (2017)

[10] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, S. Wanderman-Milne, *JAX: composable transformations of Python+NumPy programs* (2018), `http://github.com/google/jax`

[11] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V.B. Shah, W. Tebbutt, *A differentiable programming system to bridge machine learning and scientific computing* (2019)

[12] *Swift for TensorFlow*, `https://www.tensorflow.org/swift`

[13] M. Pulver, *Autodiff – automatic differentiation c++ library* (2019), `https://github.com/pulver/autodiff`

[14] R.J. Hogan, *Fast reverse-mode automatic differentiation using expression templates in C++*, in *ACM Transactions on Mathematical Software (TOMS)* (ACM New York, NY, USA, 2014), Vol. 40, pp. 1–16

[15] A. Walther, A. Griewank, *Getting Started with ADOL-C.*, in *Combinatorial scientific computing* (2009), 09061, pp. 181–202

[16] C. Lattner, V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation* (San Jose, CA, USA, 2004), pp. 75–88

[17] V. Vasilev, P. Canal, A. Naumann, P. Russo, *Cling – The New Interactive Interpreter for ROOT 6*, in *Journal of Physics: Conference Series* (IOP Publishing, 2012), Vol. 396, p. 052071