Automatic Differentiation in ROOT

Garima Singh (Princeton University), Jonas Rembser (CERN), Lorenzo Moneta (CERN), Vassil Vassilev (Princeton University)

compiler-research.org
Scientific breakthrough such as the discovery of the big void in the Khufu’s Pyramid, the gravitational waves and the Higgs boson heavily rely on the ROOT software package.


Methods of Automatic Differentiation

The Two Techniques

**Source Code Transformation AD**

- Synthesize derivative code from the input program *automatically*.
- **Faster** - allows for easier compiler optimization.
- Eg. Tapenade, Enzyme, **Clad**

**Operator Overloading AD**

- Use a new data type and operator overloading to keep track of derivatives as the original program executes.
- **Slower** and requires hand writing annotations and changing data types.
- Eg. PyTorch/TensorFlow, CoDiPack, etc.

---

[1]: https://en.wikipedia.org/wiki/Automatic_differentiation
An Efficient Method of Differentiation

*Compiler-Based Source Transformation AD: Clad*

Clad\(^2\), a source code transformation AD tool, implemented as a plugin to the clang compiler. Clad inspects the internal compiler representation of the target function to generate its derivative.

```cpp
double absFunc(double x) {
    if (x < 0) return -x;
    else return x;
}
```

```cpp
double absFunc_darg0(double x) {
    double _d_x = 1;
    if (x < 0) return -_d_x;
    else return _d_x;
}
```

- Proximity to compiler allows for more control over code generation.
- Support for a good subset of modern C++ constructs.

\(^2\) : https://github.com/vgvassilev/clad

Automatic Differentiation in ROOT - Garima Singh | 2\textsuperscript{nd} MODE AD Workshop 14 Sept. 2022
An Efficient Method of Differentiation

Compiler-Based Source Transformation AD: Clad

Clad also can be used within Cling\[3\], the C++ interpreter used with ROOT.

```
[2]: double fn(double x, double y) {
    return x*x*y + y*y;
}

[3]: auto fn_dx = clad::differentiate(fn, "x");

[4]: fn_dx.execute(5, 3)

[4]: 30.000000

Binder Tutorial

[3]: https://github.com/root-project/cling
```
AD in ROOT
ROOT’s Math and Statistical Libraries

Then we demonstrate AD on one of the high-level analysis libraries - *RooFit*.

As a first, we demonstrate AD on one of the simpler *TFormula/TF1* class.

https://laconga.redclara.net/courses/modulo-datos/claseMD06/materialesMD06/ROOT_Introduction_CEVALEVE_class_04_16_2016.pdf
Automatic Differentiation for TFormula

What is TFormula?

TFormula models formulae in ROOT by connecting compiled and interpreted code offering both performance and flexibility. It allows users to define their formulae as strings which are then JIT compiled to functions that are used to fit and model data distributions.

TFormula f("f", "x*std::sin([0]) - y*std::cos([1])");

double p[2] = {TMath::Pi()/6, TMath::Pi()/3};
f.SetParameters(p);
f.Eval(1, 0);

The following function is JIT compiled by Cling:

```cpp
double f(double *x, double *p) {
    return x[0]*std::sin(p[0]) - x[1]*std::cos(p[1]);
}
```

This code can easily be differentiated by clad!

This call internally calls: `f({1, 0}, p)`
Automatic Differentiation for TFormula

How to use AD in TFormula?

```c
root [0] TFormula f("f", "x*std::sin([0]) - y*std::cos([1])");
root [2] f.SetParameters(p);
root [3] f.Eval(1, 0)
(double) 0.50000000
root [4] 1*std::sin(p[0])
(doub) 0.50000000
root [5] TFormula::CladStorage result(2);
root [7] f.GradientPar(in, result);
root [8] result
(TFormula::CladStorage &) { 0.86602540, 0.00000000 }
root [9] std::cos(p[0])
(double) 0.86602540
```

This will generate the gradient of the function with respect to the parameters `p`.

It is also possible to compute AD Hessians for TFormula through `HessianPar`.
Automatic Differentiation for TFormula

Benchmarks

TF1 based benchmarks. TF1 is the TFormula fitting interface for fitting histograms.

Clad can be used in TF1 through the “G” parameter to `Fit`.

```cpp
h1->Fit(f1, "S G Q N");
```
Automatic Differentiation in RooFit

Overview

- A more complex application of AD, different from TF because it is hard to extract the code containing differentiable properties.

What that we want to differentiate

How do we make RooFit more malleable to AD?
Automatic Differentiation in RooFit

Challenges

RooFit represents all mathematical formulae as RooFit objects which are then brought, together into a compute graph. This compute graph makes up a model on which further data analysis is run.

<table>
<thead>
<tr>
<th>Math Notations</th>
<th>RooFit Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>$x$</td>
</tr>
<tr>
<td>function</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>PDF</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>space point</td>
<td>$\hat{x}$</td>
</tr>
<tr>
<td>integral</td>
<td>$\int_{a}^{b} f(x)$</td>
</tr>
<tr>
<td>list of space points</td>
<td>$\hat{x}_1, \hat{x}_1, \hat{x}_1, \ldots$</td>
</tr>
</tbody>
</table>

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

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

Gaussian Probability Distribution Function (pdf)
Equivalent Code in C++ with RooFit

Programmers/users know this relationship. But how do we connect these two together when a connection is not obvious programmatically?
Automatic Differentiation in RooFit

Why AD?

- One goal - Make RooFit Faster. Results from a Higgs-combination fit:

<table>
<thead>
<tr>
<th>serial old</th>
<th>parallel N=1</th>
<th>parallel N=2</th>
<th>parallel N=4</th>
<th>parallel N=8</th>
<th>parallel N=16</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_roofit</td>
<td>setup_roofit</td>
<td>setup_roofit</td>
<td>setup_roofit</td>
<td>setup_roofit</td>
<td>setup_roofit</td>
</tr>
<tr>
<td>minut_init</td>
<td>minut_init</td>
<td>minut_init</td>
<td>minut_init</td>
<td>minut_init</td>
<td>minut_init</td>
</tr>
<tr>
<td>gradient_calc</td>
<td>6289</td>
<td>7102</td>
<td>3734</td>
<td>1997</td>
<td>1107</td>
</tr>
<tr>
<td>line_search</td>
<td>323</td>
<td>287</td>
<td>287</td>
<td>287</td>
<td>287</td>
</tr>
</tbody>
</table>

Derivatives become bottleneck!

- Good results, but still use numerical differentiation.
- Potential next step – use AD to compute the gradients.
A way of having some context for AD is to introduce a function for each of the RooFit nodes that would represent the underlying mathematical notation as code.

```cpp
double RooGaussian::gauss(double x, double mu, double sig)
{
    const double arg = x - mu;
    double out = std::exp(-0.5 * arg * arg / (sig * sig));
    return 1. / (std::sqrt(TMath::TwoPi()) * sigma) * out;
}
```

This would allow us to calculate the derivatives of a RooGaussian just by differentiating just this function. However, how do we chain these individual functions to create code that represents a given RooFit model?
One way to do this is by defining a ‘translate’ function that returns an `std::string` representing the underlying mathematical notation of the class as code. This string can then be connected together to form a function.

```cpp
NLL

code +=
RooNLLVar::Translate();

// Declare the code
gInterpreter->Declare(code.c_str());

// Get the derivatives of 'code'
gInterpreter->ProcessLine("clad::gradient(code);");

// Use code_grad in wrappers that interface with
// the minimizer.

The parent node queries the results from the child nodes.
```

```
GAUSS

code +=
RooGaussian::Translate({});
```
Automatic Differentiation in RooFit

**Code Squashing : translating RooFit models**

The “glue” function enabling code squashing.

```cpp
std::string
RooGaussian::translate(...) override {
  result = "RooGaussian::gauss("
         + _x->getResult() + " ,"
         + _mu->getResult() + " ,"
         + _sigma->getResult() + ")";
  return "";
}
```

Stateless function enabling differentiation of each class.

```cpp
static double RooGaussian::gauss(double x, double mean, double sigma) {
  const double arg = x - mean;
  const double sig = sigma;
  double out = std::exp(-0.5 * arg * arg / (sig * sig));
  return 1. / (std::sqrt(TMath::TwoPi()) * sig) * out;
}
```

RooGaussian::evaluate()

*The RooFit call to evaluate a gaussian*

RooGaussian::gauss(x, mu, sig)

*The equivalent code generated*
AD for binned likelihoods from HistFactory

A first application of AD for RooFit models

Many binned likelihoods follow a similar pattern:

\[ L(\bar{n}, \bar{a} \mid \bar{n}, \bar{\chi}) = \prod_{c \in \text{channels}} \prod_{b \in \text{bins}} \text{Pois}(n_{cb} \mid \nu_{cb}(\bar{n}, \bar{\chi})) \prod_{\chi \in \bar{\chi}} c_{\chi}(a_{\chi} \mid \chi) \]

\(\bar{n}\) : data, \(\bar{a}\) : auxiliary data
\(\bar{n}\) : unconstrained parameters
\(\bar{\chi}\) : constrained parameters

HistFactory is a higher-level tool to build such likelihoods in RooFit.

Good model class for showing AD in RooFit:

- many parameters
- rich computation graph
- few normalization integrals
Preliminary Results

Explicit Computation Graphs: An Example HistFactory Model

An example histogram fitting model with 2 bins and 2 channels, with 3 samples per channel. Based on the hf_001 example.
Preliminary Results

Explicit Computation Graphs: An Example HistFactory Model

```c++
double nll(double *in)
{
    double nomGammaB1 = 400;
    double nomGammaB2 = 100;
    double nominalLumi = 1;

    double constraint[3]{ExRooPoisson::poisson(nomGammaB1, (nomGammaB1 * in[0])),
                        ExRooPoisson::poisson(nomGammaB2, (nomGammaB2 * in[1])),
                        ExRooGaussian::gauss(in[2], nominalLumi, 0.100000)};
    double cnstSum = 0;
    double x[2]{1.25, 1.75};
    double sig[2]{20, 10};
    double binBoundaries1[3]{1, 1.5, 2};
    double bgk1[2]{100, 0};
    double binBoundaries2[3]{1, 1.5, 2};
    double histVals[2]{in[0], in[1]};
    double bgk2[2]{0, 100};
    double binBoundaries3[3]{1, 1.5, 2};
    double weights[2]{122.000000, 112.000000};

    for (int i = 0; i < 3; i++) {
        cnstSum -= std::log(constraint[i]);
    }

    double mu = 0;
    double temp;
    double nllSum = 0;
    unsigned int b1, b2, b3;

    for (int iB = 0; iB < 2; iB++) {
        b1 = ExRooHistFunc::getBin(binBoundaries1, x[iB]);
        b2 = ExRooHistFunc::getBin(binBoundaries2, x[iB]);
        b3 = ExRooHistFunc::getBin(binBoundaries3, x[iB]);
        mu += (bgk1[b2] * histVals[iB]) * (in[2] * 1.000000);
        mu += (bgk2[b3] * histVals[iB]) * (in[2] * 1.000000);
        temp = std::log((mu));
        nllSum -= -(mu) + weights[iB] * temp;
    }
    return cnstSum + nllSum;
}
```

Constraints defined as calls to their respective ‘evaluate’s.

Translated RooProducts.

Constraint sum.

NLL
Automatic Differentiation in RooFit

Preliminary Results: HistFactory Minimization

Performance comparison AD vs numerical differentiation on hf_001 inspired example

~5.5x speedup

Tested on ROOT v6.26.
Automatic Differentiation in RooFit

Next Steps

● Adding externally provided Hessian support to MINUIT.

● Investigating applicability of AD to the rest of the HistFactory workflow - such as integrating AD based derivatives in profile likelihood calculations etc.

● Improving the external gradient interface in the RooFit minimizer wrappers

● Explore differentiating numerically computed integrals with AD.
Summary

- We present a compiler based AD tool - Clad, that is available as a plugin to the C++ compiler Clang.

- We showcase the addition of AD to ROOT's TFormula class and present relevant results from the same.

- We demonstrate our current progress with adding AD to RooFit, more specifically HistFactory. We present promising results for incorporating AD to a complex math library such as RooFit.

- We also discuss future plans towards making RooFit more AD aware.
The End!

Questions?

https://www.linkedin.com/in/garimasingh28/

https://github.com/grimmyshini

garima.singh@cern.ch