Adding Automatic Differentiation Support to RooFit

Garima Singh | CERN Internship 2022

Introduction

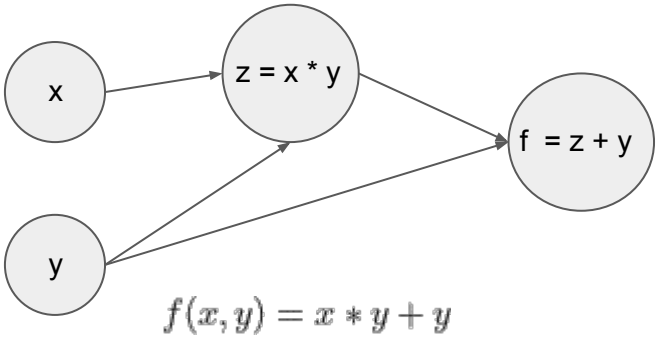
Problem Statement

Add automatic differentiation (AD) to RooFit, a statistical modelling library packed in ROOT. Specifically, utilize Clad's AD abilities in likelihood fits with RooFit.

Introduction

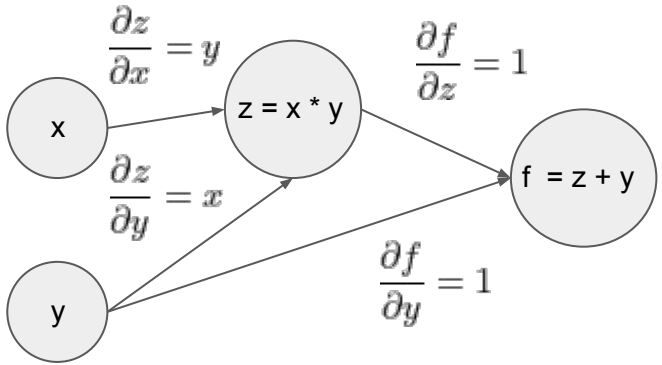
What is Automatic Differentiation?

Simply put, it's a way for computers to differentiate computer programs. Automatic Differentiation (AD) applies the chain rule of differential calculus throughout the semantics of the original program.



$$f(x,y) = x * y + y$$

$$f'(x,y)_x = y \qquad f'(x,y)_y = x + 1$$



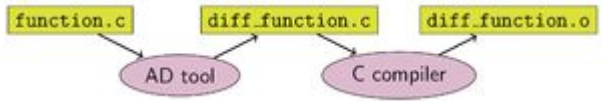
$$\frac{\partial f}{\partial x} = \frac{\partial z}{\partial x} * \frac{\partial f}{\partial z} = y \qquad \frac{\partial f}{\partial y} = (\frac{\partial z}{\partial y} * \frac{\partial f}{\partial z}) + \frac{\partial f}{\partial y} = x + 1$$

Introduction

Why Automatic Differentiation?

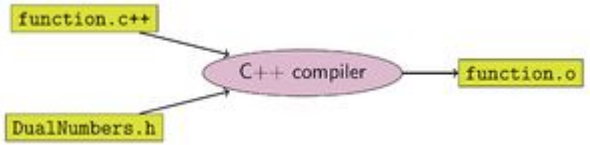
- Produces exact derivatives for complex functions.
- Highly scalable in the number of input parameters.
- Fairly immune to numerical or round-off errors.

Source Code Transformation AD



- Synthesize derivative code from the input program.
- Faster - allows for easier compiler optimization

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.

Introduction

Clad - A source code transformation AD tool

Clad is implemented as a plugin to the clang compiler. It inspects the internal compiler representation of the target function to generates its derivative.

```
double sqr(double x){
      return x * x;
}
```


```
double sqr_darg0(double x){
      double _d_x = 1;
      return _d_x * x + x * _d_x;
}
```

- Clad supports a wide variety of modern C++ constructs.
- Clad's proximity to the compiler allows more control over code generation.

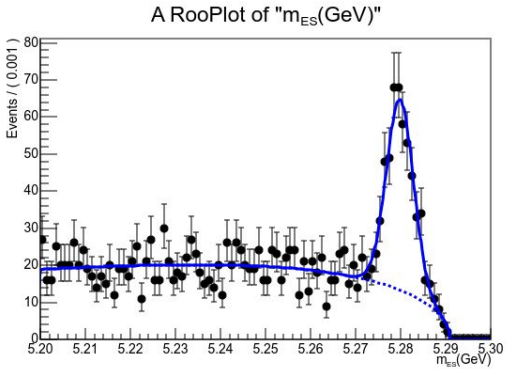
Introduction

What is RooFit?

RooFit is a library that comes packaged in ROOT - a data analysis framework for high energy physics. It is used by scientists all over the world in different type of data analysis fields. it enables the modelling of event data distributions and perform further mathematical analysis on it such as likelihood fits, generating plots etcetera.

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

Math Notations		RooFit Object
variable	x	RooRealVar
function	$f(x)$	RooAbsReal
PDF	$f(x)$	RooAbsPdf



A Gaussian Probability Density Function (PDF) plot from RooFit.

Introduction

Motivation

For all its differentiability needs, RooFit relies solely on numerical differentiation capabilities of the MathCore library in ROOT. Not only is numerical differentiation less accurate and more sensitive to errors, it also does not scale well with larger number of parameters.

This means, fits with large number of parameters may take upto hours to complete. This greatly hinders the development process and makes collaborations slower.

This is where Automatic Differentiation comes in. However, adding AD is not straight forward.

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

Gaussian Probability Distribution Function (pdf)

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

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?

Design

Code Generation : making RooFit classes differentiable

A way of having some context for AD is to introduce a ‘translate’ function for each of the RooFit nodes. This function would represent the underlying mathematical notation as code.

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

Gaussian Probability Distribution Function (pdf)

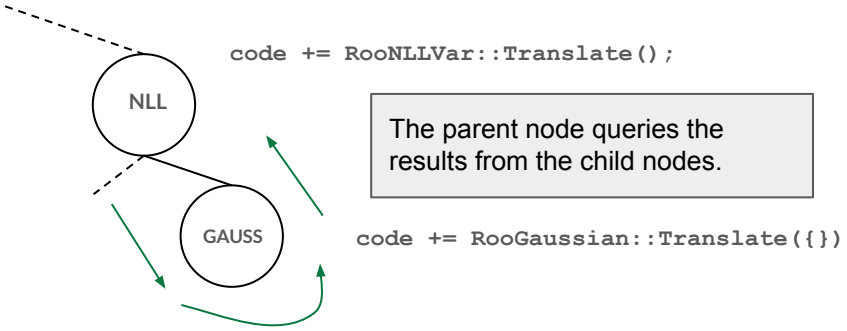
```
double RooGaussian::Translate() const
{
 const double arg = x - mean;
 const double sig = sigma;
 return std::exp(-0.5*arg*arg/(sig*sig));
}
```

This would allow us to calculate the derivatives of a RooGaussian just by differentiating its ‘translate’ function. However, how do we chain these individual translates to create code that represents a given RooFit model? And then how do we subsequently differentiate such a code?

Design

Code Squashing : translating RooFit models

Now that we have a way to translate each class, we only have to link these classes together as per the compute graph being analysed. We do this by making each translate function return an std::string that can then be concatenated together to form a function.




```
// 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.
```

Design

Code Squashing : translating RooFit models

```
std::string RooGaussian::translate(std::string
&globalScope, std::vector<std::string> &preFuncDecls)
override {
   result = "RooGaussian::gauss(" + _x->getResult() +
" ," + _mu->getResult() + " ," + _sigma->getResult() +
")";
   return "";
 }
```

```
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()) *
sigma) * out;
 }
```

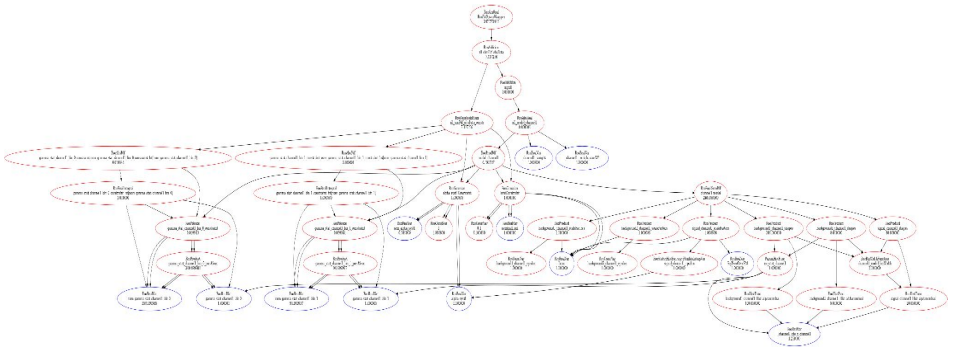
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Gaussian Probability Distribution Function (pdf)

⟹

```
RooGaussian::gauss(x, mu, sig)
```

The code generated



```
double nll(double lumi, double SigXsecOverSM, double gammaB1,
double gammaB2) {
 double nomGammaB1 = 100;
 double nomGammaB2 = 400;
 double nominalLumi = 1;
 double constraint[3]{
     ExRooPoisson::poisson(nomGammaB1, (nomGammaB1 * gammaB1)),
     ExRooPoisson::poisson(nomGammaB2, (nomGammaB2 * gammaB2)),
     ExRooGaussian::gauss(lumi, nominalLumi, 0.100000)};
 double cnstSum = 0;
 double x[2]{1.25, 1.75};
 double sig[2]{20, 10};
// cont. ->
```

```
 // .. cont.
 double binBoundaries1[3]{1, 1.5, 2};
 double bgk1[2]{100, 0};
 double binBoundaries2[3]{1, 1.5, 2};
 double histVals[2]{gammaB1, gammaB2};
 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 nllSum = 0;
 for (int iB = 0; iB < 2; iB++) {
   unsigned int b1 = ExRooHistFunc::getBin(binBoundaries1, x[iB]);
   unsigned int b2 = ExRooHistFunc::getBin(binBoundaries2, x[iB]);
   unsigned int b3 = ExRooHistFunc::getBin(binBoundaries3, x[iB]);
   double mu = 0;
   mu += sig[b1] * (SigXsecOverSM * lumi);
   mu += (bgk1[b2] * histVals[iB]) * (lumi * 1.000000);
   mu += (bgk2[b3] * histVals[iB]) * (lumi * 1.000000);
   double temp;
   temp = std::log((mu));
   nllSum -= -(mu) + weights[iB] * temp;
 }
 return cnstSum + nllSum;
}
```

Results

The current implementation is tested on the hf_01 HistFactory example.

Test	Squashed Code	Squashed Code w/ AD	Classic RooFit
hf_01 w/o constraints	1300 us	1570 us	2588 us
hf_01 w/ constraints	656 us	727 us	1534 us

The AD implementation, while still faster, is slower when compared to the squashed code because of lesser parameters, minimizer strategy, and the fact that we eventually end up using numerical differentiation for the hessians.

This work also inspired improvements in RooFit (example this and this).

Future & Ongoing Work

RooFit AD:

- Support the ShapeConvolutions class.
- Test the models with higher number of parameters and different minimization strategies.

Error Estimation/ Clad:

- Work on the content for the floating point error estimation paper.
- Put in a PR for static array failures in reverse mode - supporting every case is complex, how do you track variable sized arrays? Ex.
  - `double arr1[n]; n++; double arr2[n];`
- Enable const optimizations for Clad, i.e, don't push/clone const values/ const pointers.
- Fix error estimates not being created for inline declarations.
- Add support for inlining updates to a map that maintains the maximum/average error for each function.
- Work on the HPCCG/Luluesh benchmarks.

The End!

Questions?