Automatic Differentiation of Binned Likelihoods With Roofit and Clad

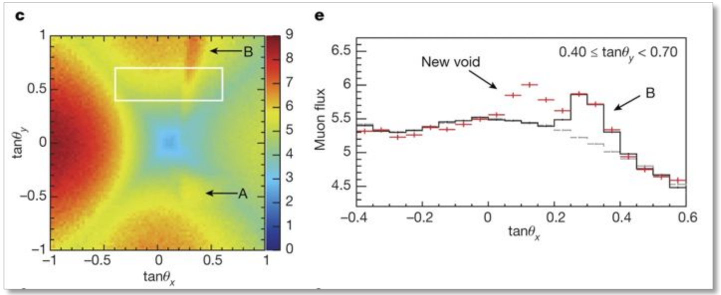
Garima Singh (Princeton University), Jonas Rembser (CERN), Lorenzo Moneta (CERN), Vassil Vassilev (Princeton University)

compiler-research.org

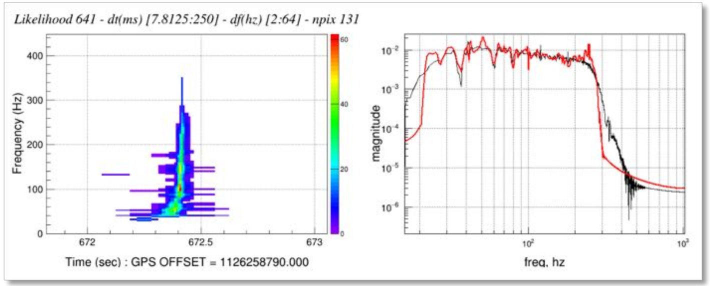
This project was supported in part by the NSF (USA) Grant OAC-1931408 and NSF (USA) Cooperative Agreement OAC-1836650.

ROOT

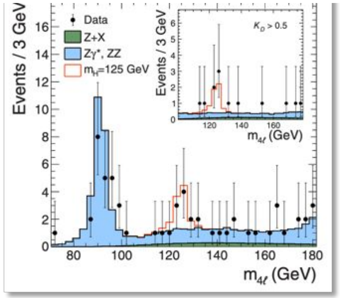
An Exabyte Data Analysis Framework



[1]



[2]



[3]

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.

RooFit is ROOT's high level statistical analysis library.

[1] Morishima, K., Kuno, M., Nishio, A. *et al.* Discovery of a big void in Khufu's Pyramid by observation of cosmic-ray muons. *Nature* 552, 386–390 (2017).
[2] B. P. Abbott, et al, Observation of Gravitational Waves from a Binary Black Hole Merger (2016)
[3] CMS Collaboration, Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC (2012)

HistFactory

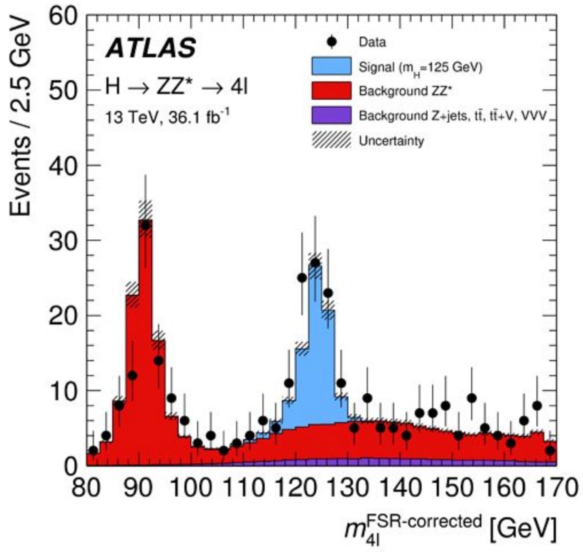
A gateway to binned likelihood fits using RooFIt

Many binned likelihoods follow a similar pattern:

$$L(\vec{n}, \vec{a} \mid \vec{\eta}, \vec{\chi}) = \underbrace{\boxed{\prod_{c \in \text{ channels}} \prod_{b \in \text{ bins}} \text{Pois}(n_{cb} \mid \nu_{cb}(\vec{\eta}, \vec{\chi}))}}_{\text{product of Poisson terms}} \underbrace{\boxed{\prod_{\chi \in \vec{\chi}} c_{\chi}(a_{\chi} \mid \chi)}}_{\text{constraints}}$$

$\vec{n}$: data, $\vec{a}$: auxiliary data
$\vec{\eta}$: unconstrained parameters
$\vec{\chi}$: constrained parameters

Example binned likelihood with one channel: Higgs to 4 leptons



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

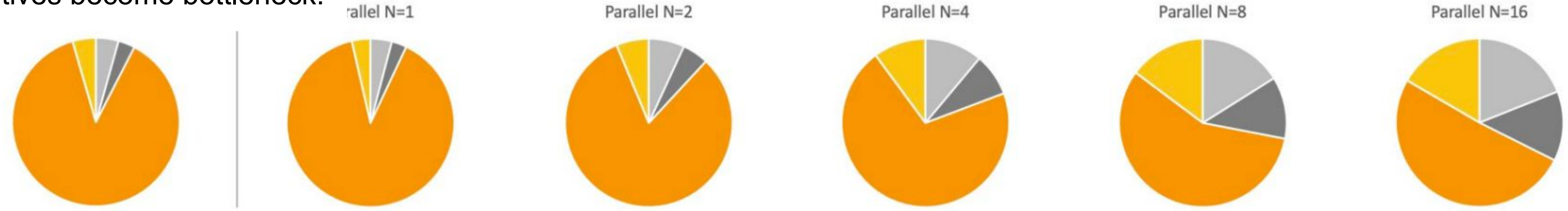
Why Add Automatic Differentiation to RooFit?

Derivative Bottleneck

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

	serial old	parallel N=1	parallel N=2	parallel N=4	parallel N=8	parallel N=16
setup_roofit	313	327	315	315	312	327
minuit_init	230	231	231	231	231	231
gradient_calc	6289	7102	3734	1997	1107	879
line_search	323	287	287	287	287	287

Derivatives become bottleneck!



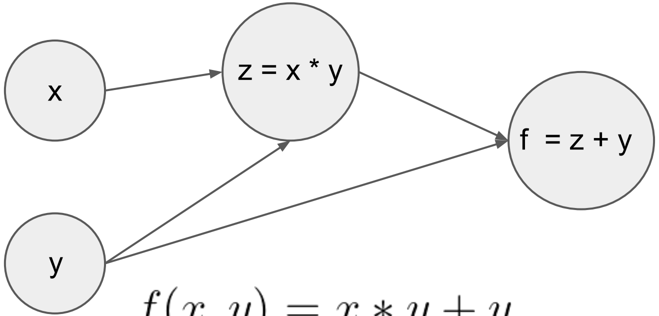
ICHEP 2022 - Zeff Wolffs - https://agenda.infn.it/event/28874/contributions/169205/attachments/93887/129094/ICHEP_RooFit_ZefWolffs.pdf

- Good results, but still use numerical differentiation.
- Potential next step – use Automatic Differentiation to compute the gradients.

Automatic Differentiation Crash Course

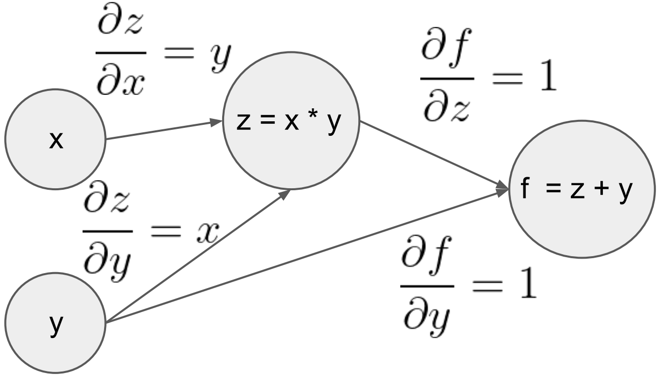
What is Automatic Differentiation?

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.



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

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



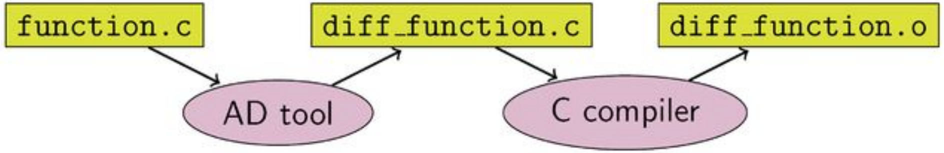
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} * \frac{\partial z}{\partial x} = y \quad \frac{\partial f}{\partial y} = (\frac{\partial f}{\partial z} * \frac{\partial z}{\partial y}) + \frac{\partial f}{\partial y} = x + 1$$

Automatic Differentiation Crash Course

The Two Techniques

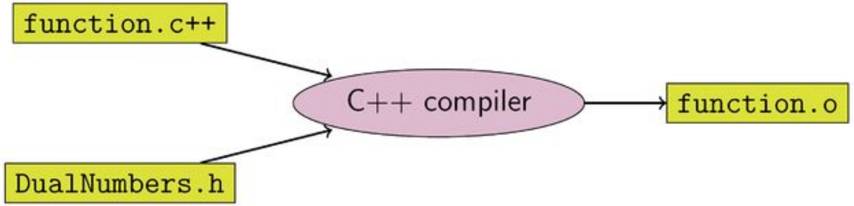
- Broadly two ways to implement AD:

Source Code Transformation AD



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

Operator Overloading AD



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

Automatic Differentiation Crash Course

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 generates its derivative.

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

`clad::differentiate(absFunc)` →

```
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 Crash Course

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

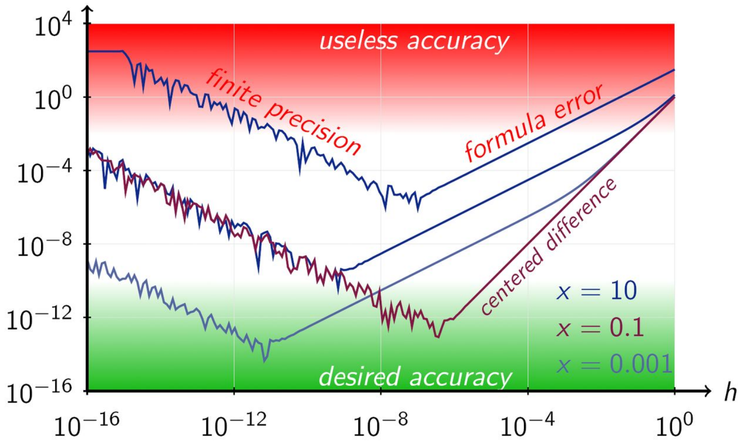
[3] :https://github.com/root-project/cling

Automatic Differentiation Crash Course

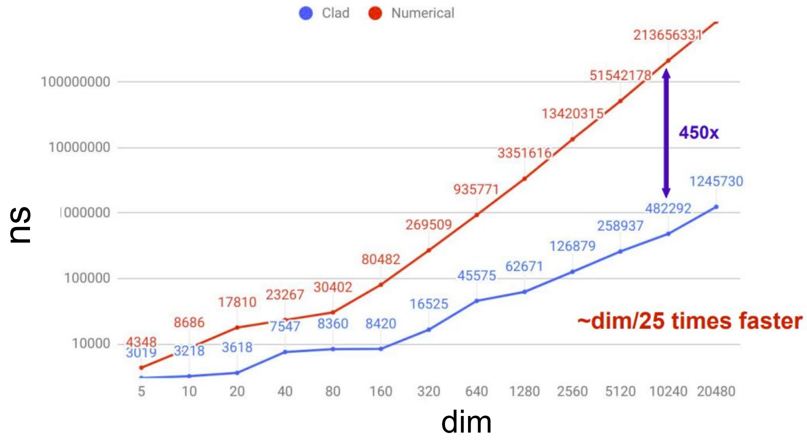
Why AD over numerical differentiation?

- Calculates exact derivatives of programs, free from numerical errors.
- More performant for functions with high number of parameters.

Difficulty in choosing step size due to numerical error



Comparison between Clad's AD and numerical diff



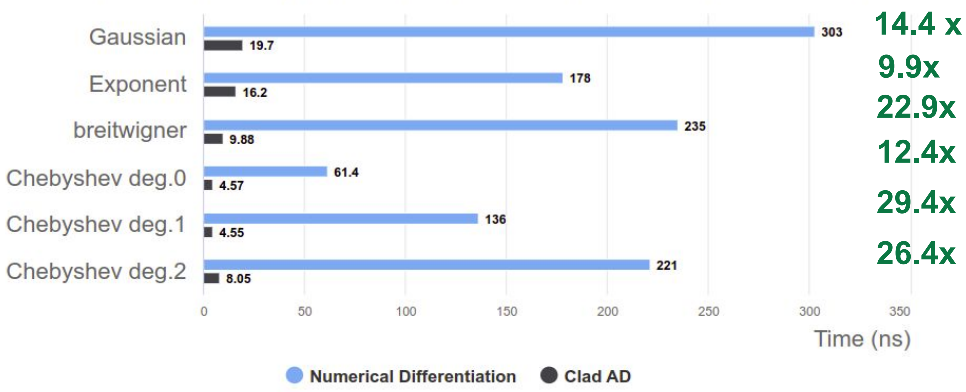
https://www.researchgate.net/publication/346917467_Automatic_Differentiation_in_ROOT

Automatic Differentiation Crash Course

But, still, why AD???

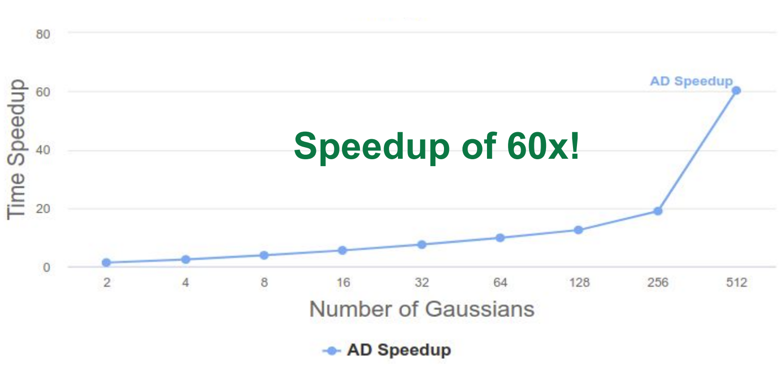
- We have seen some promising results (in ROOT) already!

Performance Comparison of Generation in TFormula



TFormula benchmarks of gradient generation time from numerical differentiation and clad AD.

Performance Speedup of a Multi-Gaussian Fit (10000 bins)

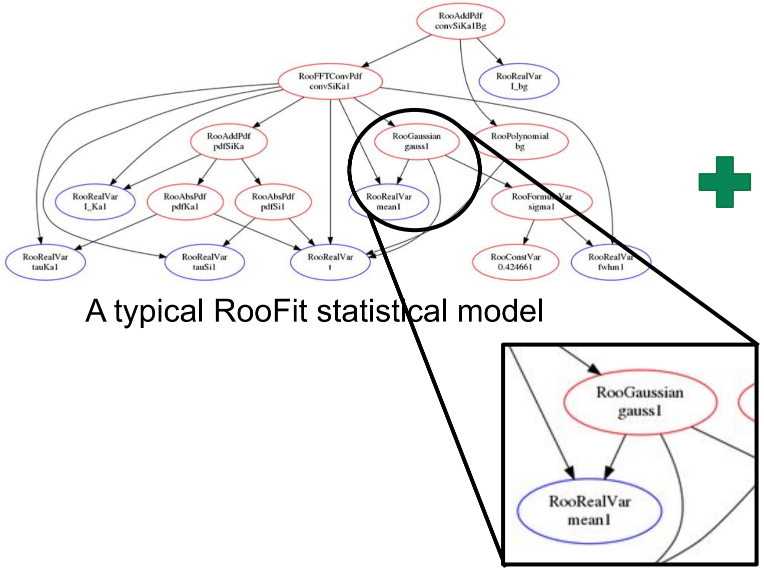


TF1 based benchmarks. TF1 is the TFormula fitting interface for fitting histograms.

Automatic Differentiation in RooFit

Sounds easy…

What we want to differentiate



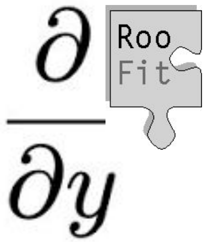
A typical RooFit statistical model

Made up of various RooFit objects

Our AD tool of choice

Clad

Differentiable RooFit Models!



Actually, not so simple…

RooFit has an object oriented model which deliberately hides the differential properties of the nodes in favor of ease of use.

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.

Math Notations		RooFit Object
variable	x	RooRealVar
function	$f(x)$	RooAbsReal
PDF	$f(x)$	RooAbsPdf
space point	$\hat{x}$	RooArgSet
integral	$\int_a^b f(x)$	RooRealIntegral
list of space points	$\hat{x}_1, \hat{x}_1, \hat{x}_1...$	RooAbsData

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

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

Gaussian Probability Distribution Function (pdf)

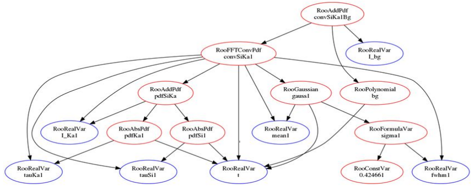
Equivalent Code in C++ with RooFit

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

Automatic Differentiation in RooFit

A different approach: Translating models to code

What that we want to differentiate



Some way to expose differentiable properties of the graph as code.

C++ code the AD tool can understand

C++ code the AD tool can understand

The AD tool

Cla∂

Derivative code of the model!

Decoding The BlackBox

Step one: Making RooFit classes differentiable

A way of exposing some context/code for AD is to introduce a function for each of the RooFit nodes that would represent the underlying mathematical notation as code.

`RooGaussian::evaluate()` — – *Caching & Bookkeeping* / + *Normalization* →

```
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;
}
```

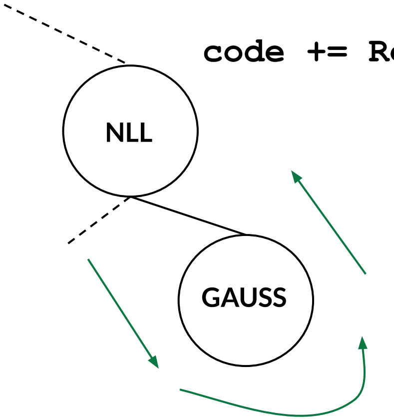
This would allow us to calculate the derivatives of a RooGaussian just by differentiating just this function. However, how do we chain these individual functions to create code that represents a given RooFit model?

Decoding The BlackBox

Step two: connecting the nodes

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.

```
code += RooNLLVar::Translate();
```



The parent node queries the results from the child nodes.

```
code += RooGaussian::Translate({});
```

Decoding The BlackBox

Step two: connecting the nodes

An example translate function looks like so:

'translate' builds a call to the simplified *'evaluate'* of the RooFit class.

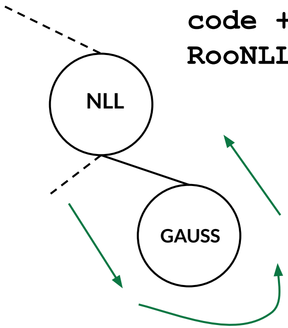
'getResult' queries the child nodes for information, allows for result propagation.

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

Decoding The BlackBox

Step three: Wrapping it up!

Once the translate functions are defined and the graph is traversed, we can use Cling to declare our string code.




```
code +=
RooNLLVar::Translate();
```

```
code +=
RooGaussian::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.
```

Decoding The BlackBox

Summary

The “glue” function enabling code squashing.

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

Stateless function enabling differentiation of each class.

```
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;
}
```

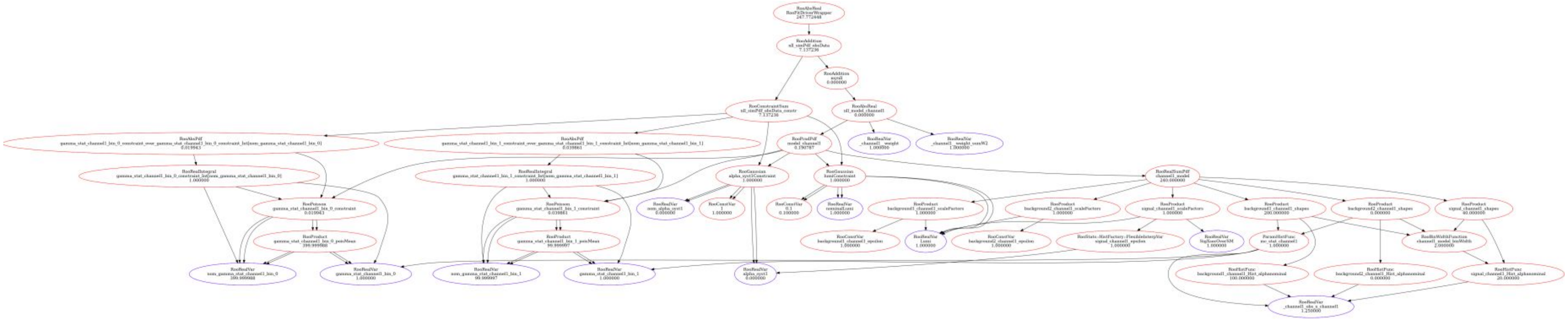
`RooGaussian::evaluate()` ⟹ `RooGaussian::gauss(x, mu, sig)`

The RooFit call to evaluate a gaussian

The equivalent code generated

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

```
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]);
  }
// cont…
```

Constraints defined as calls to their respective 'evaluate' functions.

Constraint sum.

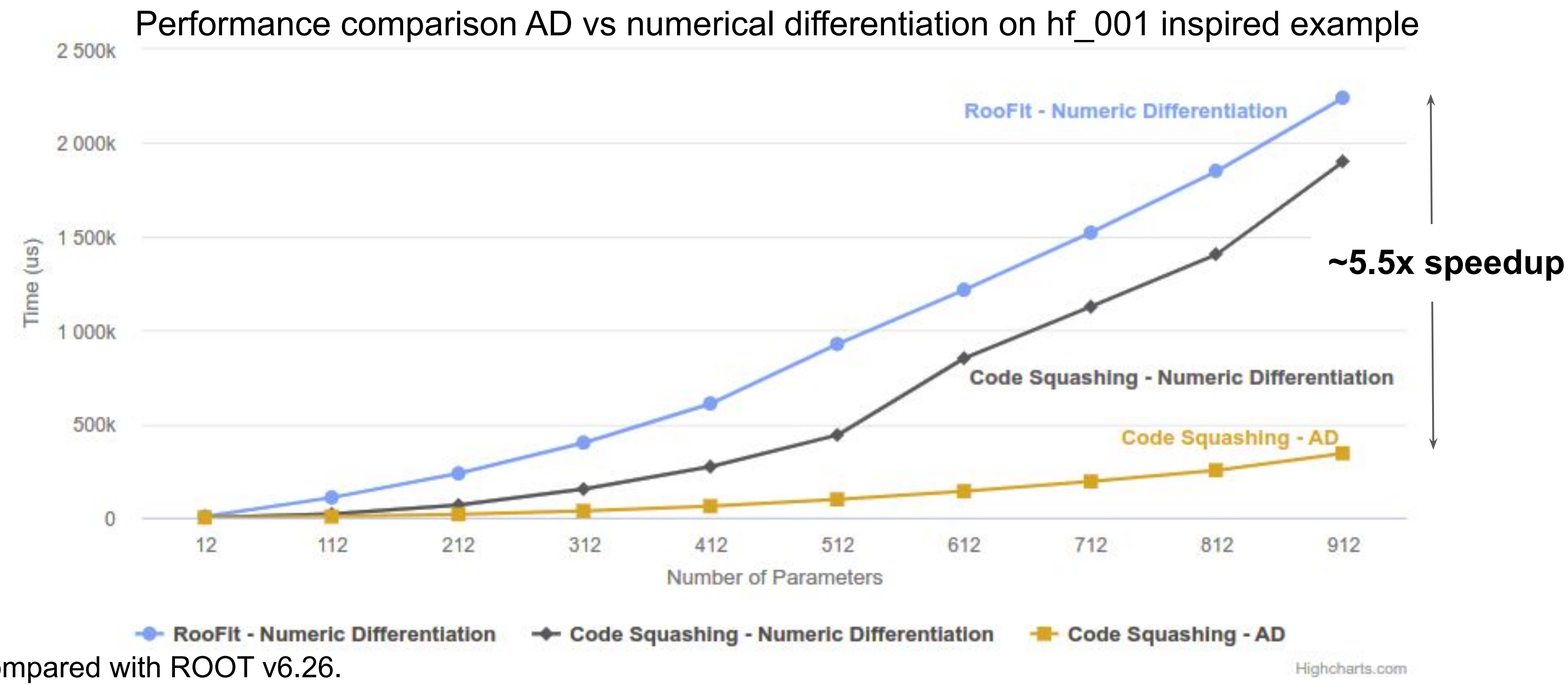
```
// cont..
  double mu = 0;
  double temp;
  double nllSum = 0;
  unsigned int b1, b2, b3;
  for (int iB = 0; iB < 2; iB++) {
    b1 = ExRooHistFunc::getBin(binBoundaries1, x[iB]);
    b2 = ExRooHistFunc::getBin(binBoundaries2, x[iB]);
    b3 = ExRooHistFunc::getBin(binBoundaries3, x[iB]);
    mu = 0;
    mu += sig[b1] * (in[3] * in[2]);
    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;
}
```

Translated RooProducts.

NLL

Preliminary Results

HistFactory Minimization



Compared with ROOT v6.26.

Automatic Differentiation in RooFit

Next Steps

- Adding externally provided Hessian support to MINUIT.
- Improving the external gradient interface in the RooFit minimizer wrappers.
- Getting an initial implementation of our work to ROOT master. This would cover a subset of the HistFactory classes.

Summary

- Our work presents an efficient way to translate complex models such that they can be differentiated using AD.
- 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/grimmmyshini

garima.singh@cern.ch

Backup