Automatic Differentiation in RooFit for fast and accurate likelihood fits

Jonas Rembser [1]
David Lange [2+]
Vaibhav Thakkar [2+]
Petro Zarytsky [2+]
Vassil Vassilev [2+]
Lorenzo Moneta [1]

1 - *CERN*
2+ - *Princeton University (US) and supported by the National Science Foundation under Grant OAC-2311471.*

ICHEP 2024, Prague, July 20th

Introduction

Re-engineering RooFit with **Automatic Differentiation** for faster likelihood fits.

What happened before:

- Work on AD in RooFit **stated two years ago**
- Presentation at ACAT 2022 with first proof of concept outside ROOT
- Presentation at CHEP 2023 with benchmarks of our approach integrated in ROOT

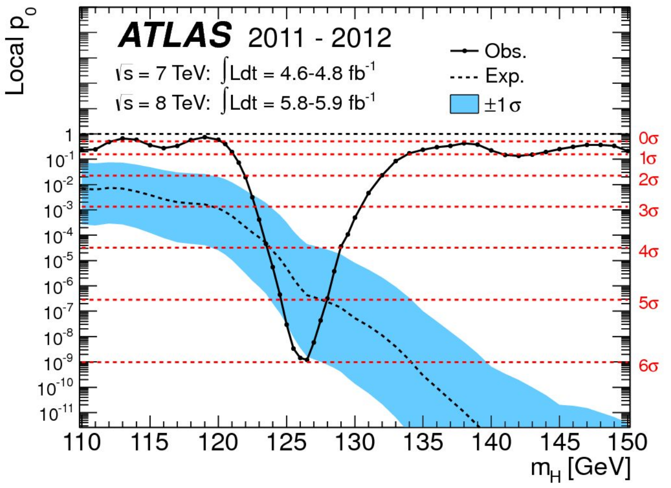
Today:

Update on recent improvements in RooFit AD, cumulating in the support of **CMS** and **ATLAS** Higgs combination models.

RooFit

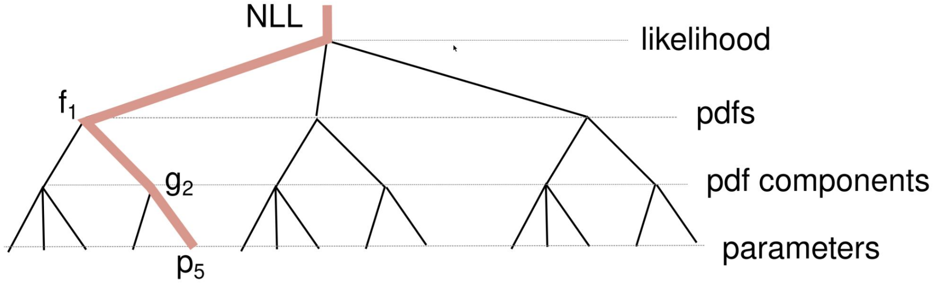
RooFit: C++ library for statistical data analysis in ROOT.

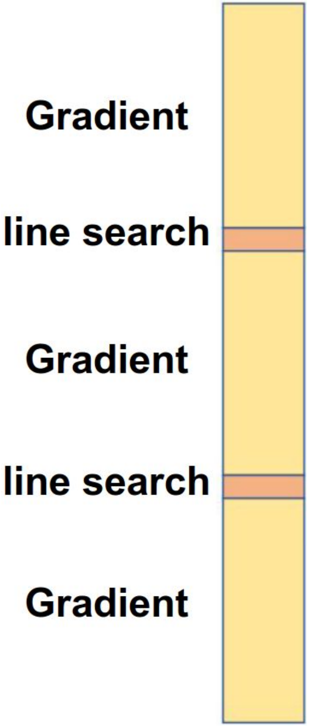
- Used for modelling and normalization of probability density functions (p.d.f)
- Fitting likelihood models to the event data set.
  - Minimizing both binned and unbinned likelihoods
- Used most prominently by the LHC experiments, also for discovering the Higgs boson in 2012
  - Example of profile likelihood scan on the right



Numeric minimization of RooFit Likelihoods

- By default, RooFit uses **numerical differentiation**: Minuit 2 changes parameters **on-at-a-time** to get the full gradient
- One key concept of RooFit: **caching of intermediate results** to minimize redundant computations in gradient evaluation
- Still, gradient dominates minimization time *(see also the ICHEP 2022 RooFit presentation)*





- Our goal: make evaluating gradients cheap with **Automatic differentiation (AD)**

An automatic differentiation engine for RooFit

- RooFit is a framework to build **computation graphs for function minimization**, similar to the ML frameworks **TensorFlow** or **PyTorch**
- Different from other frameworks, RooFit didn't have an **automatic differentiation engine**
- However, the other frameworks are generally not optimized for HEP usecases and workflows

Therefore, we have added a differentiation engine based on Clad and C++ code generation to RooFit: so you can get **analytic likelihood gradients without compromising**.

Brief Introduction to Automatic Differentiation (AD)

$$f(x) = e^{e^{e^{e^{e^x}}}}$$

Symbolic via Wolfram Alpha

$$\frac{d}{dx}\left(e^{e^{e^{e^{e^x}}}}\right) = e^{x+e^{e^{e^{e^x}}}+e^{e^{e^x}}+e^{e^x}+e^x}$$

Figure out the analytical fn

Handcode

```cpp
// f(x)=e^(e^(e^(e^(e^x))))
#include <cmath>
double f (double x, int N=5) {
  double result = x;
  for (unsigned i = 0; i < N; i++)
    result = std::exp(result);
  return result;
}
```

AD

```cpp
double f_dx(double x, int N=5) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < N; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

Reference: *V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation*

Reverse mode AD: evaluating the chain rule top-down

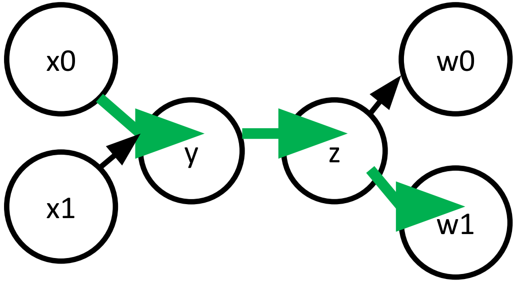
One can get the gradient of one output wrt. all inputs in **two passes** through the computation graph:

- **Forward pass**: evaluate computation graph and cache intermediate results (*aka. "store them on tape"*)
- **Reverse pass**: evaluate and accumulate the partial derivatives

Most prominent application: backpropagation in deep learning.

Why it's great: runtime **scales with the size of the computation**, *not the number of parameters.*

```
y = f(x0, x1)
   z = g(y)
w0, w1 = l(z)
```



$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$

Clad

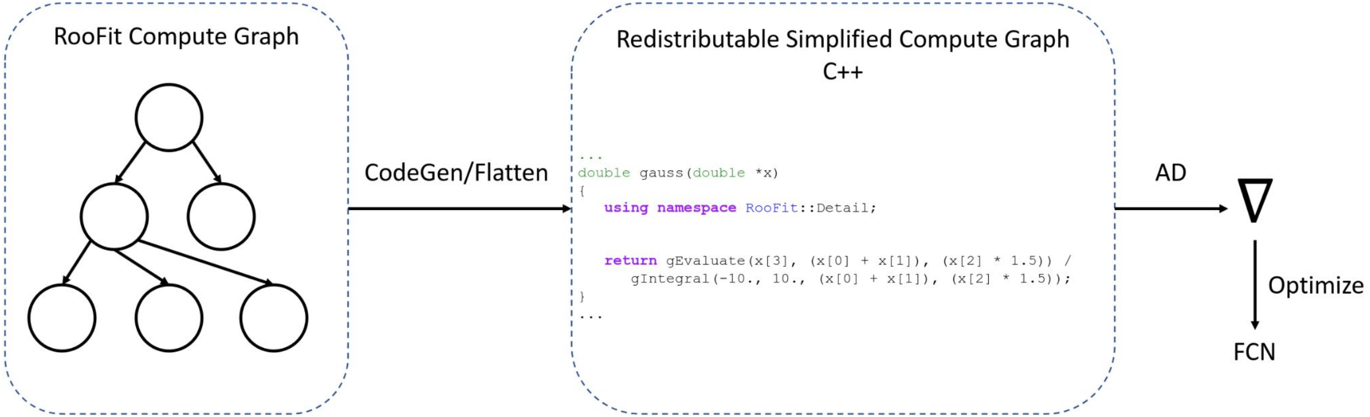
- **Source transformation based AD tool for C++**
  - Runs at compile time - clad generates readable (and debuggable) code for derivatives.
  - Optimization capabilities of the Clang/LLVM Infrastructure enabled by default.

- **Support for control flow expression - difficult with operator overloading approaches.**
  - Better handling of complex control flow logic handling compared to machine-learning frameworks like *Tensorflow* and *Pytorch*, hence more suitable for scientific computing.

- **Integrated with ROOT infrastructure.**
  - Clad's compiler research team has integration in High Energy Physics (HEP), and making significant improvements for RooFit use case.

https://github.com/vgvassilev/clad/

How RooFit uses Clad to get analytic gradients: Code generation (aka. "codegen")



1. **Mathematical** concept
2. **RooFit** user code
3. **Automatic translation** of RooFit model to simple C++ code
4. **Gradient** of C++ code **automatically generated** with **Clad**
5. Gradient code **wrapped** back into RooFit object

Note: for the **nominal NLL** function, we **still use RooFits CPU backend** to benefit from vectorization and caching outside the gradients.

Status of RooFit “codegen” backend

- You can enable it in your fit or likelihood creation with one additional argument:

    - `pdf.fitTo(data, RooFit::EvalBackend(“codegen”))`
    - `pdf.createNLL(data, RooFit::EvalBackend(“codegen”))`

- Many RooFit classes already support it, most notably all of **HistFactory** and also complicated ones like **numeric integrals**
- RooFit has many classes with varying importance: we need your feedback on which RooFit primitives should be supported
- Adding codegen support for custom classes is not difficult *(see CMS combine example later)*

Experiments with ATLAS Benchmark models

- **ATLAS HistFactory model** (49 HistFactory channels, 739 parameter in total, in rootbench).

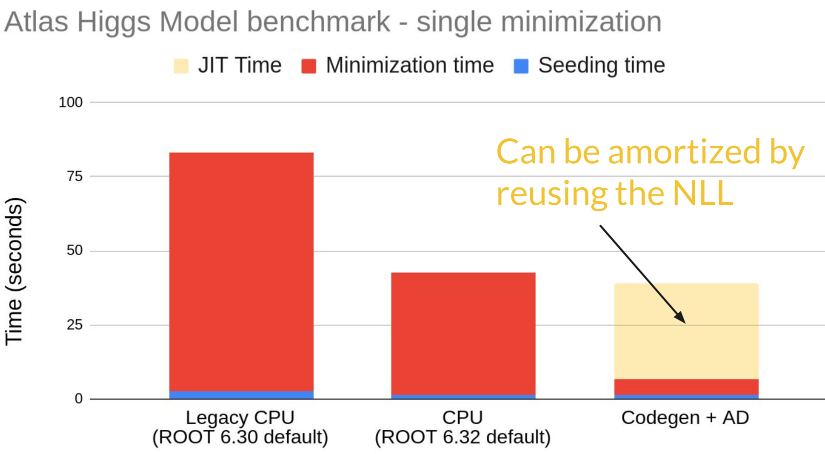
How to read this plot:

- **Seeding time**: initial Hessian estimate (num. second derivatives)
- **Minimization time**: finding the minimum
- JIT time: time to generate and compile the gradient code
  - The gradient can be be reused across different minimizations, amortizing the JIT time
  - For example, possible reuse in **profile likelihood scans**

Using AD drastically reduces minimization time on top of the new CPU backend in ROOT 6.32.

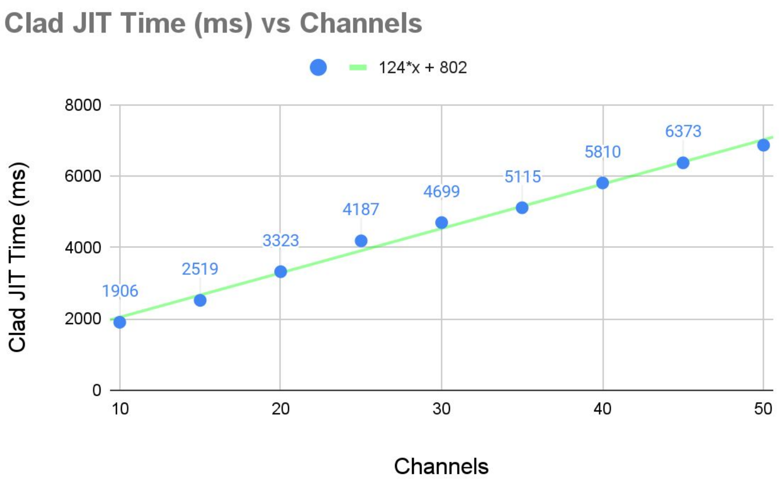
Bottom line: **10x faster minimization** compared to ROOT 6.30.

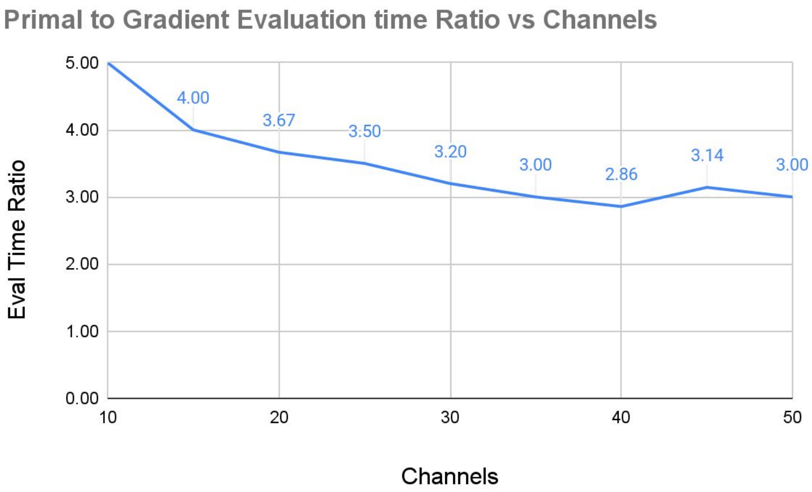
Instructions to reproduce in backup



Final Min Val = -368.36 for all evaluations

Experiments with ATLAS Benchmark models





- Memory consumption of gradient evaluation is very low compared to the python/ML based frameworks.
  - Constant factor of the consumption by primal function.

Benchmarks with the CMS Higgs Observation Model

- **Breaking news in April 2024**: CMS published RooFit-based Higgs observation likelihood!
- Very heterogeneous likelihood: **672 parameters** in **102 channels** with
  - Template histogram fits
  - Analytical shape fits, **numerical integration** necessary in some cases
- **Perfect example** to test the new RooFit developments

See also the presentation on CMS analysis tools at this conference.

The output will be:

```
 <<< Combine >>>
 <<< v9.2.1 >>>
>>> Random number generator seed is 123456
>>> Method used is Significance

 -- Significance --
Significance: 4.87557
Done in 1.76 min (cpu), 1.76 min (real)
```

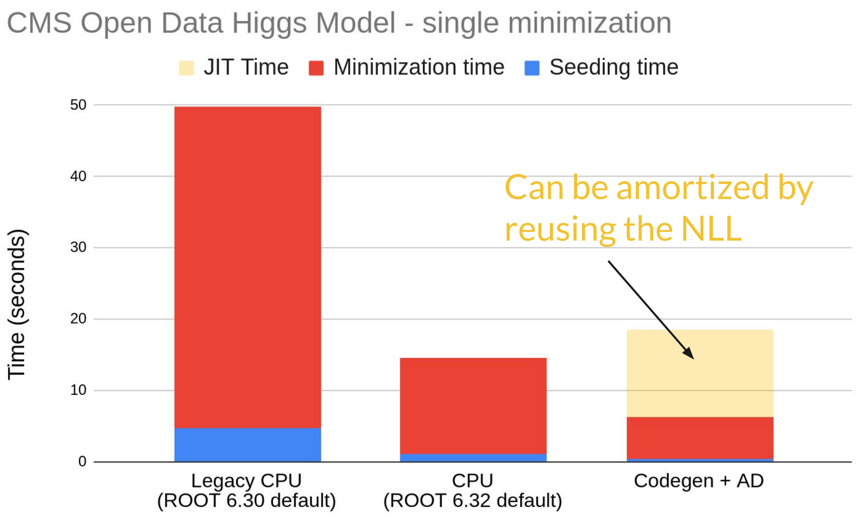
Screenshot from the README of the open likelihood: You can re-discover the Higgs.

Benchmarks with the CMS Higgs Observation Model

- We implemented necessary free functions for the generated code in a [custom **CMS combine branch**]
- Benchmarked one **minimization pass**

Observations:

- The new CPU code path default in **ROOT 6.32** is a big improvement to the old RooFit, possibly making many custom improvements in combine *obsolete*
- The AD backend further reduces minimization time
- Printing out the generated NLL code helps a lot to **understand** what's actually fitted



Instructions to reproduce in backup

Benchmarks with the CMS Higgs Observation Model

One more observation on numerical stability:

- For these kinds of fits, the derivatives are small compared to the NLL value
- Numerical differentiation often fails because the finite differences are smaller than numerical precision on the NLL
- Solution so far - offsetting the NLL but initial value:

```
pdf.createNLL(data, RooFit::Offset(true))
```

Problems with this:

- Offsetting might fail if initial value is far from the minimum
- Bookkeeping of offsets is error-prone

With AD, the offsetting is not necessary anymore!

```
36 - FCN = -9801946.549  Edm =   0.01129396511
37 - FCN = -9801946.566  Edm =   0.01497173883
38 - FCN = -9801946.574  Edm =  0.007242353199
39 - FCN = -9801946.583  Edm =  0.004954953322
40 - FCN = -9801946.589  Edm =  0.005774308843
41 - FCN = -9801946.596  Edm =  0.004695329674
42 - FCN = -9801946.602  Edm =  0.004558156748
43 - FCN = -9801946.615  Edm =  0.008141300763
44 - FCN = -9801946.625  Edm =  0.004861879849
45 - FCN = -9801946.628  Edm =  0.003472778648
46 - FCN =  -9801946.63  Edm =  0.001782083931
47 - FCN = -9801946.631  Edm = 0.0007515760698
```

Minimizer output, showing the small changes wrt. large NLL value

Possible next steps and perspectives

- Enabling analytic gradients by **default** if possible
- Work together with experiments to **support your usecases** and help out in **integration RooFit AD in experiment frameworks**
- Improve support for **non-trivial functions,** like numeric integrals
- **Extend RooFits interfaces** so it will be easy to get out the generated code and gradients to use them outside the RooFit minimization routines
- R & D on **analytic higher-order derivatives** that are used in Minuit

Your input and support is **highly welcome**!

Conclusions

Source-code transformation AD with Clad fits naturally into the ROOT ecosystem and RooFit benefits from it in many ways:

- **Faster** likelihood **gradients**
- No need for tricks to get **numerically stable** gradients
- Likelihoods can be expressed in **plain C++** without need for aggressive **caching** by the user or in frameworks like RooFit
  - **Good for understanding** the math: optimization gets decoupled from logic - simple code
  - **Good for collaboration**: simple C++ can easily be shared and used in other contexts

Backup

About Clad - usage example in ROOT interpreter

```
// example.C
#include <Math/CladDerivator.h>

double f (double x, double y) {
 return x*y; // <— Function to be differentiated
}

void example() {
 // Call clad to generate the derivative of f wrt x.
 auto f_dx = clad::differentiate(f,"x");

 // Execute the generated derivative function.
 std::cout << f_dx.execute(/*x=*/3, /*y=*/4) << std::endl;
 std::cout << f_dx.execute(/*x=*/9, /*y=*/6) << std::endl;

 // Dump the generated derivative code to stdout.
 f_dx.dump();
}
```

ROOT/Cling

```
root example.C
```

Produces

```
4 // df/dx for (x,y) = (3, 4)
6 // df/dx for (x,y) = (9, 6)

double f_darg0 (double x, double y)
{
  double _d_x = 1;
  double _d_y = 0;
  return _d_x * y + x * _d_y;
}
```

About Clad - usage example standalone

```
// Source.cpp
#include "clad/Differentiator/Differentiator.h"
#include <iostream>

double f (double x, double y) {
  return x*y; // <— Function to be differentiated
}

double main() {
  // Call clad to generate the derivative of f wrt x.
  auto f_dx = clad::differentiate(f, "x");

  // Execute the generated derivative function.
  std::cout << f_dx.execute(/*x=*/3, /*y=*/4) << std::endl;
  std::cout << f_dx.execute(/*x=*/9, /*y=*/6) << std::endl;

  // Dump the generated derivative code to stdout.
  f_dx.dump();
}
```

Compilation (+ execution)

```
clang++ -I clad/include/ -fplugin=clad.so Source.cpp
```

Produces

```
4 // df/dx for (x,y) = (3, 4)
6 // df/dx for (x,y) = (9, 6)

double f_darg0 (double x, double y)
{
  double _d_x = 1;
  double _d_y = 0;
  return _d_x * y + x * _d_y;
}
```

Providing custom derivatives to Clad

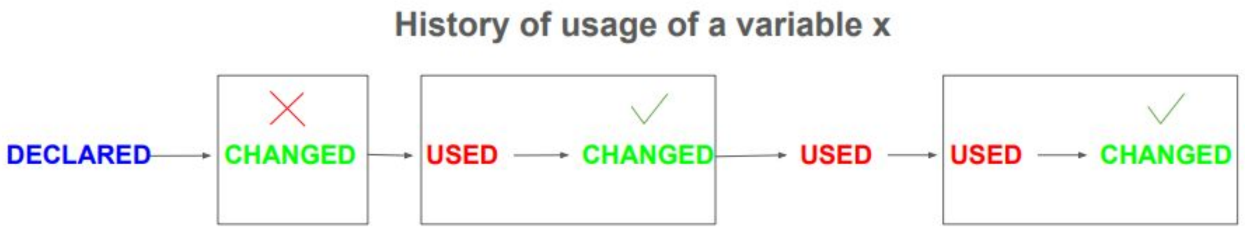
```
double my_pow (double x, double y) {
  // … custom code here for computing xʸ …
}

namespace clad {
namespace custom_derivatives {
// Providing custom code for derivative computation of my_pow.
double my_pow_darg0(double x, double y) {return y * my_pow(x, y - 1);} // ∂f/∂x.
double my_pow_darg1(double x, double y) {return my_pow(x, y) * std::log(x);} // ∂f/∂y.
}}
```

- Some use cases:
    - Calling a library function whose definition is not available.
    - Efficiency reasons - you have a better way.
    - Implicit function to be differentiated - for ex. requires solving some maximization problem

Some recent changes in Clad enabled these results

- Handling constant pointers for reverse mode AD : #919
- Reducing tape storage operations inside Clad for reverse mode AD : #655
- Dynamic capturing of differentiation plans - capturing and traversing call graph : #766, #873
- **To Be Recorded (TBR) analysis** in reverse mode:
  Reverse-mode AD requires storing intermediate values that have impact on derivatives to restore values in the backward pass. However, we don't actually have to store all of them:



Thanks a lot to the Clad team for these improvements!

To Be Recorded (TBR) analysis in reverse mode: example

Original function

```
double f_exp(double x, size_t N) {
    for (int i=0; i < N; ++i)
        x = 2 * x;
    return x;
}
```

In RooFit, more than 30% code size reduction.

3x speedup in jit time.

TBR analysis off

```
void f_exp_grad(...) {
    // forward pass
    …
    clad::tape<double> _t1 = {}; // used to store x
    _t0 = 0;
    for (i = 0; i < N; ++i) {
        _t0++;
        clad::push(_t1, x); // x is only transformed linearly so it's
        x = 2 * x;          // value is not needed in the reverse pass
    }
    …
    // reverse pass
    for (; _t0; _t0--) {
        --i;        // i is never used to compute the derivatives
        x = clad::pop(_t1); // no need to restore x
        …
    }
}
```

TBR analysis on

```
void f_exp_grad(...) {
    // forward pass
    …
    _t0 = 0;
    for (i = 0; i < N; ++i) {
        _t0++;
        x = 2 * x;
    }
    …
    // reverse pass
    for (; _t0; _t0--) {
        …
    }
}
```

RooFit code generation: Gaussian example

```
Gauss(x, mu + shift, sigma * scale)
```

1. **Mathematical** concept
2. **RooFit** user code
3. **Automatic translation** of RooFit model to simple C++ code
4. **Gradient** of C++ code **automatically generated** with **Clad**
5. Gradient code **wrapped** back into RooFit object

```
RooRealVar x("x", "", 0, -10, 10);
RooRealVar mu("mu", "", 0, -10, 10);
RooRealVar shift("shift", "", 1.0, -10, 10);
RooAddition muShifted("mu_shifted", "", {mu, shift});
RooRealVar sigma("sigma", "", 2.0, 0.01, 10);
RooConstVar scale("scale", "", 1.5);
RooProduct sigmaScaled("sigma_scaled", "", sigma, scale);
RooGaussian gauss{"gauss", "", x, muShifted, sigmaScaled};
```

```
double gauss(double *x)
{
   using namespace RooFit::Detail;

   return EvaluateFuncs::gaussEvaluate(x[3], (x[0] + x[1]), (x[2] * 1.5)) /
       AnalyticalIntegrals::gaussianIntegral(-10., 10., (x[0] + x[1]), (x[2] *
1.5));
}
```

```
void gauss_grad(double *x, double *out);
```

Planned improvements to further speedup RooFit

On the **Clad** side:

- Using Automatic Differentiation for computing Hessians
  - Computing only the diagonal entries of Hessians for the seeding step.
- Further improvements in Clad to remove redundant computations for Gradients.
  - Advanced analysis for improving the efficiency of Gradient computations.
- Experimenting with make the gradient computation parallelizable.
  - Trying vector forward mode for Hessians.

On the **ROOT** side:

- More efficient **Hessian evaluations**: minimization is fast now, evaluating the Hessian of the NLL for parameter estimation became the new bottleneck

Reproducing the benchmark fits: ATLAS likelihood

- **ATLAS HistFactory model** (49 HistFactory channels, 739 parameter in total).
- It is part of rootbench and can be built with a special configuration flag.
- You need at least ROOT 6.32.04 (recipe below uses ROOT master on lxplus).

```
# source LCG environment with ROOT nightlies
source /cvmfs/sft.cern.ch/lcg/views/dev3/latest/x86_64-el9-gcc13-opt/setup.sh

git clone git@github.com:root-project/rootbench.git
cd rootbench
mkdir build
cd build
cmake -DROOFIT_ATLAS_BENCHMARKS=ON ..
make -j8
cd root/roofit/atlas-benchmarks/
./download_workspaces.sh
./roofitAtlasHiggsBenchmark
```

Reproducing the benchmark fits: CMS likelihood

- CMS Higgs discovery model (102 channels, 672 floating parameter in total).
- Requires custom “CMS combine” branch.
- You need at least ROOT 6.32.04 (recipe below uses ROOT master on lxplus).

```
# source LCG environment with ROOT nightlies
source /cvmfs/sft.cern.ch/lcg/views/dev3/latest/x86_64-el9-gcc13-opt/setup.sh

git clone https://github.com/guitargeek/HiggsAnalysis-CombinedLimit.git  HiggsAnalysis/CombinedLimit
cd HiggsAnalysis/CombinedLimit
git checkout roofit_ad_ichep_2024
mkdir build install
cd build
cmake -DCMAKE_INSTALL_PREFIX=../install  ..
make install -j8
cd ..
chmod +x install/bin/*
wget "https://repository.cern/records/c2948-e8875/files/cms-h-observation-public-v1.0.tar.gz"
tar -xf cms-h-observation-public-v1.0.tar.gz
source env_standalone.sh
text2workspace.py cms-h-observation-public-v1.0/125.5/comb.txt  --mass 125.5
python scripts/fitRooFitAD.py
```