



Differentiable Programming in C++ and ROOT with Clad

Jonas Rembser[*], Vassil Vassilev[+]
{ [*]CERN, [+]Princeton, compiler-research.org }



This work is partially supported by National Science Foundation under Grant OAC-2311471



Brief Introduction to Automatic Differentiation (AD)

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

Symbolic via Wolfram Alpha

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

Figure out the analytical fn

Handcode

```
// f(x)=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

```
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;
}
```

Reverse mode AD: evaluating the chain rule top-down

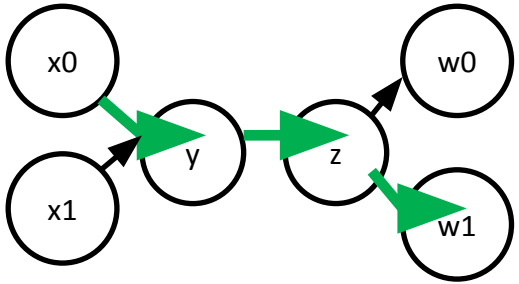
One can get the gradient of one output wrt. all inputs in **two passes** through the computation graph:

- **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

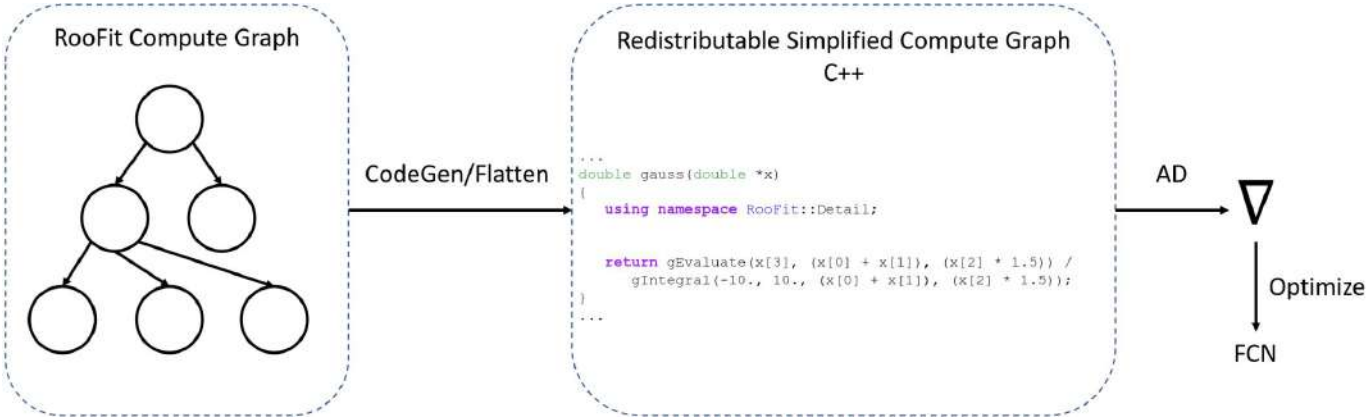
- **Source transformation based AD tool for C++**
  - Runs at compile time - clad generates readable (and debuggable) code for derivatives.
  - **Optimization capabilities** of the Clang/LLVM Infrastructure enabled by default.

- **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.

- **A standalone tool, integrated with ROOT infrastructure.**
  - Clad's compiler research team has integration in High Energy Physics (HEP), and making significant improvements for RooFit and CMS Combine use cases.

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.

Differentiable HEP: Joint Instrument & Physics Optimization

Fix hardware → Calibrate reconstruction → Run physics

Errors propagate; you can't optimize the hardware for the physics goal easily.

Conceptually, moving from black-box grid scans to "Glass-Box" gradient descent:

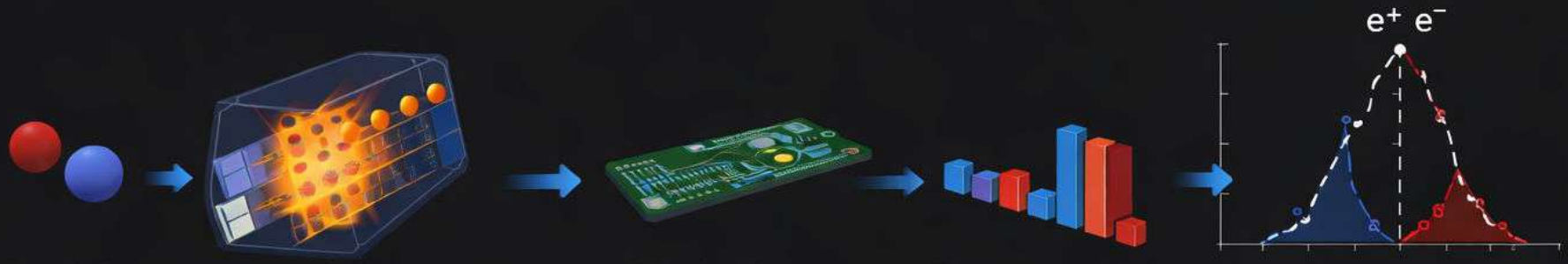
- Every step from particle hit to invariant mass is a function with a derivative.
- By chaining these, we can ask: "How does my physics mass change if I tweak a single ADC gain?”

Setup

We aim to build a pedagogical, tool-agnostic pipeline that demystifies end-to-end differentiability by treating every LHC-scale system, from shower to mass reconstruction, as a simple, chainable function.

We prioritize scientific clarity over tool-specific advocacy, focusing on the fundamental trajectory of the gradient rather than the syntax of a framework. The system should be a conceptual proof where we assemble and outline limitations with the current approach rather than a vehicle for advertising our favorite tool, framework or language

Differentiable Programming in Particle Physics



$\partial L/g$ → $\partial L/\theta$ → $\partial L/\theta$ → $\partial L/\Sigma s$

g = electronics gain

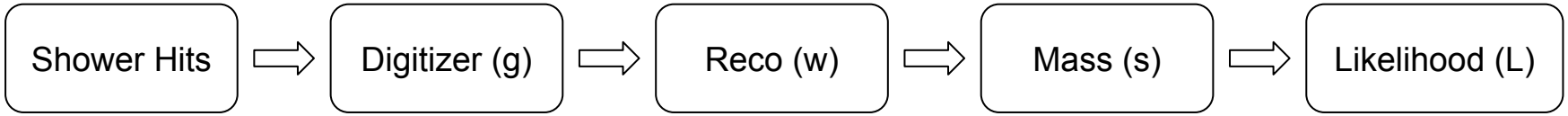
θ = model parameters

s = energy scale

s = energy scale

A Tiny Pipeline

Calibration parameters in red



Toy simulation of two electrons from Z-decay at rest into one-layer calorimeter

*ADC = noise + energy * gain*

Sum ADC counts over threshold

Invariant mass formula, correct with linear scale factor

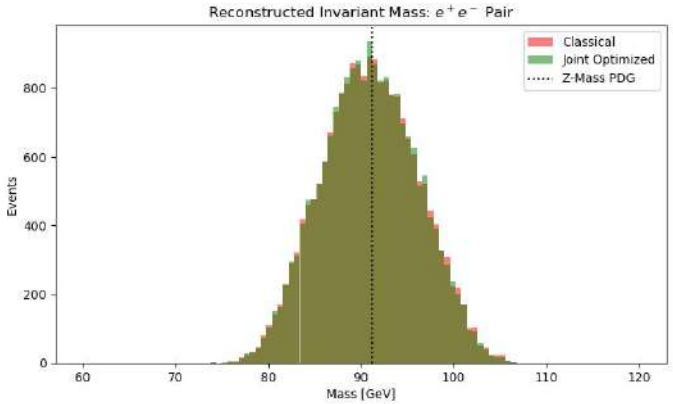
Assuming "delta" Z-peak and Gaussian detector that matches the simulation chain

- Initial **Z-Peak demo pipeline** focuses on differentiation wrt. calibration parameters
- **Alignment and calibration are massive differentiable systems** where demonstrated improvements with differentiable programming can have huge impact

A Tiny Differentiable Pipeline

Our idea: “top-down” instead of “bottom-up”

- Set up an initial toy pipeline with the help of an **LLM**
  - Note: gradients can also be created by LLM, so no AD tool needed at this point. Pure C++!
- **Validate** workflow and parameter optimization by comparing:
  - Parameter optimization by multidimensional grid scan
  - Parameter optimization by gradient descent with ADAM
- Differentiation of **specialized algos** can be **plugged in** and tested in this framework without losing the context of the end-to-end workflow that want to optimize
  - Helps to keep the **big picture** and focus on what matters

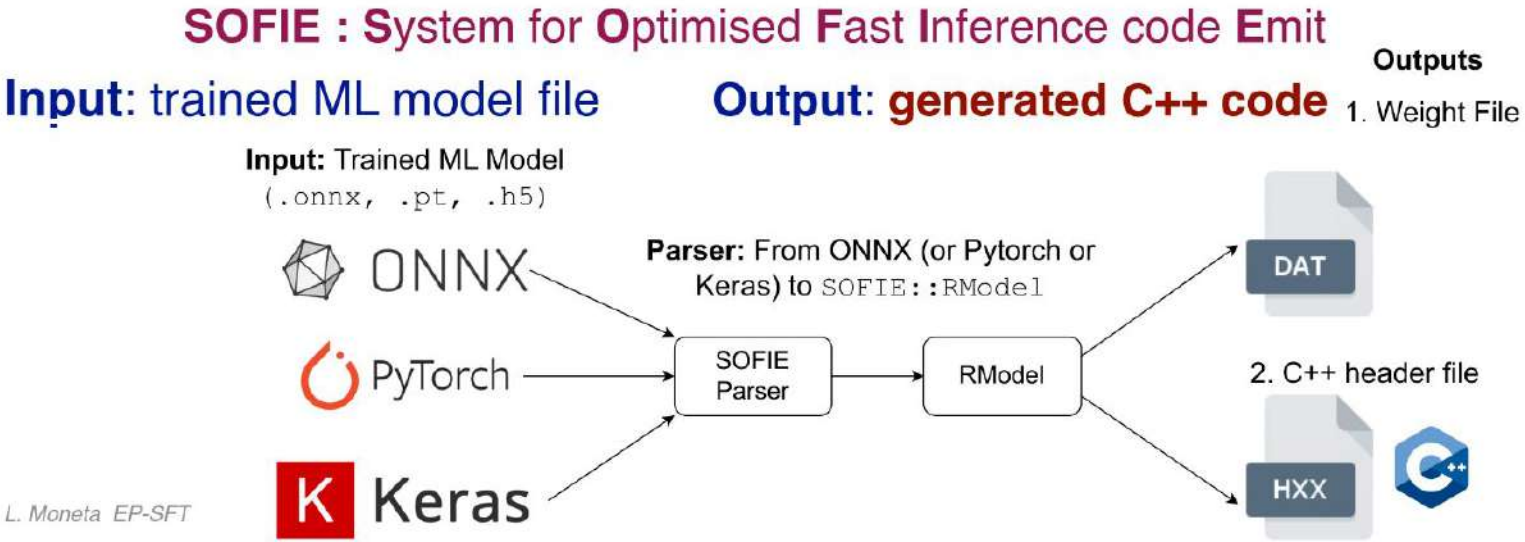


***Validation cross check**:jointly optimizing the calibration parameters gives the correct result that we also get from the “classical” grid search: the calibration parameters are tweaked to center the Z-peak at 91 GeV.*

RooFit and TMVA-SOFIE

- If the ML model could be **translated to simple C++ code**, it would naturally work with **RooFit AD** thanks to **Clad**

- ROOT already includes a package to do that: **TMVA-SOFIE**!



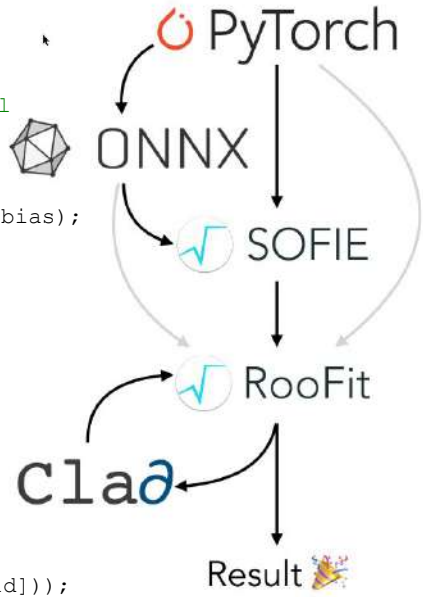
Fully differentiable SBI with RooFit, SOFIE and Clad

- Code emitted by **SOFIE** can be integrated in RooFit codegen and **differentiated** with **Clad**
- Implemented by ROOT PR #18332 (to be merged soon)
- Using SOFIE is not required: simple neural nets can be **written directly in C++**

Side effect: *shows the applicability of **Clad** to **deep learning**!*

```
void doInfer(float *tensor_x,
             float *tensor_theory_params,
             float *tensor_linear_3)
{
  using namespace TMVA::Experimental::SOFIE;// for Gemm_Call
  //--------- Matrix multiplication
  Gemm_Call(tensor_linear,5, 1, 1, 1, tensor_val_0,
            tensor_theory_params,1, tensor_theory_projectorbias);
  //------ Add
  for (size_t id = 0; id < 5; id++) {
     tensor_add[id] = tensor_x[id] + tensor_linear[id];
  }
  //--------- Matrix multiplication
  Gemm_Call(tensor_linear_1,128, 1, 5, 1, tensor_val_2,
            tensor_add,1, tensor_base_model0bias);
  //------ Sigmoid
  for (int id = 0; id < 128; id++) {
     tensor_val_4[id] =1 / (1 + std::exp(-tensor_linear_1[id]));
  }
  ...
}
```

Extract of SOFIE-emitted code for fully-connected neural net



Interoperability with the Python AD tools

Last week we wrote a little RooFit/Jax interoperability demo, doing a combined fit of a RooFit and a Jax likelihood with shared parameters and shared gradients.

Next steps:

- Add easy-to-use interfaces for **gradients** to RooFit-likelihoods, to be used in the context of **scipy minimizers** *(part of ROOT Plan of Work 2026)*
- Improve interfaces to communicate gradients into the ROOT world, possibly building on top of existing **TMath** functor interfaces *(see example on the right)*

```python
def func(x):
    return x[0] * x[0]

def grad(x, grad_out):
    grad_out[0] = 2 * x[0]

# A wrapper class for functions to be used in ROOT Math
# algorithms, e.g. Minuit 2
functor = ROOT.Math.GradFunctor(func, grad, 1)

# Use e.g. in the RooFit world(right now the RooFunctorBinding
# ignores the gradient but we'll change this)
x = ROOT.RooRealVar("x", "x", 5, -10, 10)
roo_func = ROOT.RooFunctorBinding("func", "", functor, x)
```

ROOT already has some interfaces to make functions “gradient-aware”, and they can interoperate with Python!

Interfacing different gradient-aware components

A small comment on a question asked yesterday:

“What is the required interface if we want to connect gradient-aware tools?”

- In our experience: one function for the **forward pass**, and one for the **backward pass** *(aka. pullback)*, related by a clear interface contract
- The C**++ type system** helps us to reduce the margin for error in defining the pullbacks

```
// Example: neural network inference function

// Forward pass
//
// input   : input values
// output  : output values
// context : resource-heavy context object with constants and pre-allocated
//             working memory

void inference(Context  &context, float const *input, float *output);

// Backward pass ("pullback")
//
// The "pullback" also takes pointers to the data structures needed for the
// backwards pass, corresponding to each function argument.
// These pullbacks are used internally by Clad, but are useful also for
// manually building differentiable pipelines.
//
// The pullback can be generated by Clad, written by hand, or by an LLM.

void inference_pullback(Context &context, const float *input, float *output,
                        Context *_d_context, float *_d_input, float *_d_output);
```

Conclusions

We believe that if HEP picks up on differentiable programming differentiable **C++ becomes structurally unavoidable** to complement the advanced Python AD ecosystem.

- You can build **production-ready AD pipelines with C++**, automatically generating gradients with **Clad** or even some "*vibe coding*"
- In ROOT, we have successfully applied this to **likelihood evaluation with RooFit** and **neural network inference with TMVA-SOFIE**
  - Both can also be combined (e.g. for **Simulation Based Inference**)
  - See also talk on RooFit AD on last weeks blueprint workshop on statistical tools
- Differentiable **end-to-end toy pipelines** in C++ can help us to identify new opportunities and gradually replace toy substitutes with existing HEP codes