Final IRIS HEP Presentation: Implementing Automatic Differentiation Support in RooFit

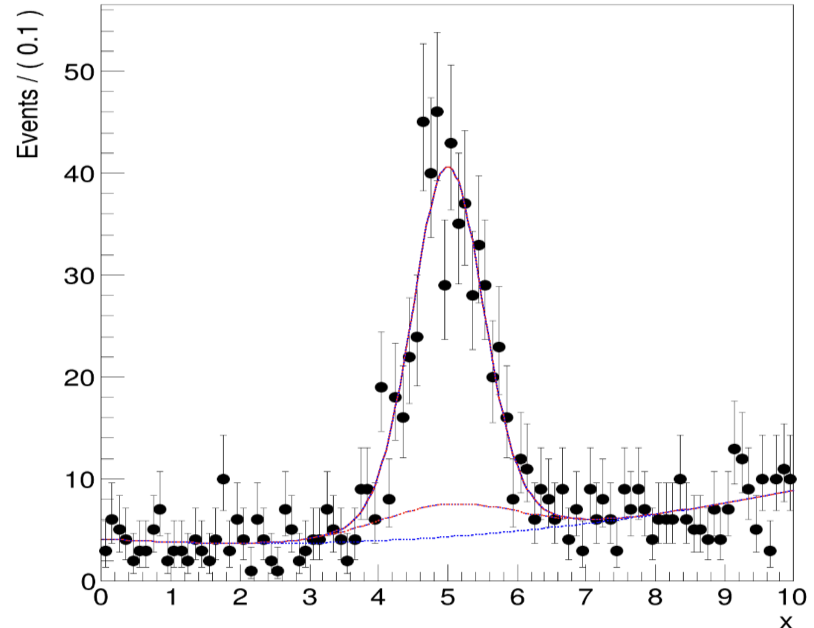
Abhigyan Acherjee
Mentors:

Overview

- Overview of Root and RooFit.
- Automatic vs Numerical Differentiation.
- Identifying classes.
- Adding Automatic Differentiation support to RooFit classes.
- Benchmarking.
- Implementing support for Benchmarks.
- Amending Binned Benchmark script to accept command line inputs.
- Created script to visualize the compare benchmarks
- Wrote bash script to integrate process to create benchmarks and also generate visualizations.

RooFit and Root Overview:

- ROOT's RooFit library facilitates modeling event data distributions for physics analyses.
- RooFit enables unbinned maximum likelihood fits, plot creation, and toy Monte Carlo sample generation.
- Event data distributions are discrete occurrences with measured observables, often following Poisson or binomial statistics.
- RooFit's core function is modeling event data distributions for physics experiments.



What is Automatic Differentiation?

- Simply a method to compute derivatives for a given function.

- It relies on the chain rule which uses the individually computed derivatives:

$$: \frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} \frac{\partial w_{i+1}}{\partial w_i} \text{ with } w_0 = x.$$

- The above expression which is representative of backward mode AD is computationally of low expense.
- Forward and Backward mode differ in chronology of intermediate operations, and as such memory and runtime varies

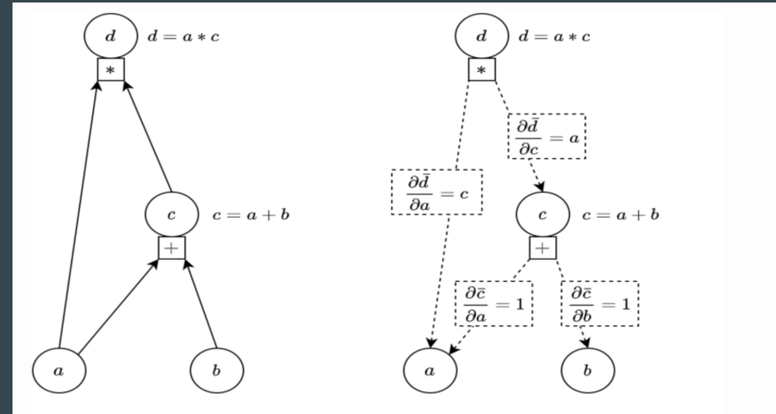


Fig: Demonstrating the evaluation of 2 expressions d=a*c and c=a+b from an AD point of view.

Numerical Differentiation

Numerical Differentiation

$$\frac{f(x+h)-f(x)}{h}.$$

- It uses traditional method of differentiation encompassed by Newton's differentiation quotient.

- Numerical Differentiation is computationally much more expensive than Automatic Differentiation.

Benefits of AD in Clad

- Clad, which functions as a plug in for the Clang compiler, uses AD.
- Given the C++ source code of a mathematical function, it can traverse the computation graph and generate code to compute the derivative of the function using automatic differentiation in reverse mode-backpropagation.
- Efficient and more precise derivatives.
- This leads to significant computational speed-ups.
- Performance comparision for the computation of pdf of a multi variate normal function is shown.

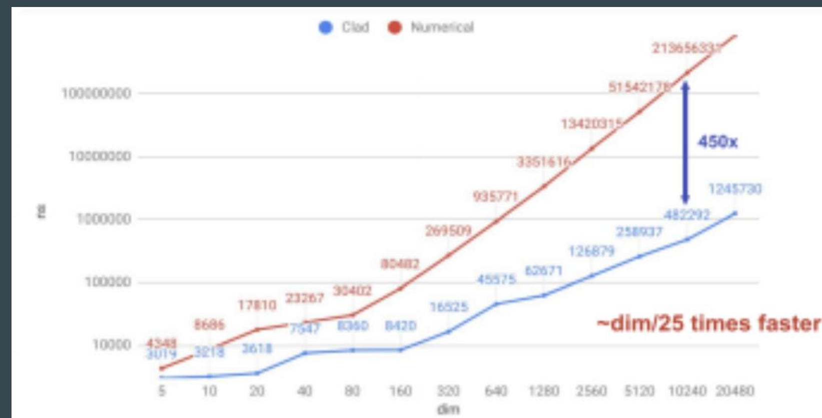


Fig: Referenced from a paper on Automatic Differentiation in Root by Vassilev et al

Identifying Classes

Tutorial #	Tutorial Name	Message	Categpry
1		OK	
2		OK	
3		ERROR:Minimization -- Translate function for class "RooGenericPdf" has not yet been implemented.	3
4		ERROR:Minimization -- Translate function for class "RooCFunction3PdfBinding<double,double,double,dou	3
5		OK	
6		OK	
7		OK	

Fig: Snapshot of dataset documenting classes

- Identified classes for which translate function i.e code generation as an evaluation backend is not implemented yet.
- This was done by running the test suite (link) stressRooFit with –b codegen command.
- Also helped identify problematic pdfs.

Ref: https://github.com/root-project/root/blob/master/roofit/roofitcore/test/stressRooFit_tests.h

Implemented Support :

In context of this work, the classes for which support has been implemented (not an exhaustive list)_(ref to App A)

- RooFormulaVar
- RooGenericPdf
- RooEffProd
- RooEfficiency
- RooRatio

Reference on how to implement this support: https://root.cern.ch/doc/master/group__roofit__dev__docs.html

```
316 + void RooFormulaVar::translate(RooFit::Detail::CodeSquashContext
      &ctx) const
317 + {
318 +    // If the number of elements to sum is less than 3, just
      build a sum expression.
319 +    // Otherwise build a loop to sum over the values.
320 +    unsigned int eleSize = _actualVars.size();
321 +    std::string className = GetName();
322 +    std::string varName = "elements" + className;
323 +    std::string sumName = "sum" + className;
324 +    std::string code;
325 +    std::string decl = "double " + varName + "[" +
      std::to_string(eleSize) + "]{";
326 +    int idx = 0;
327 +    for (RooAbsArg *it : _actualVars) {
328 +       decl += ctx.getResult(*it) + ",";
329 +       ctx.addResult(it, varName + "[" + std::to_string(idx) +
      "]");
330 +       idx++;
331 +    }
332 +    decl.back() = '}';
333 +    code += decl + ";\n";
334 +
335 +    ctx.addResult(this, (_formula->getTFormula()-
      >GetUniqueFuncName() + "(" + varName + ")").Data());
336 +
337 +    ctx.addToCodeBody(this, code);
338 + }
```

Fig: Example of implemented support: RooFormulaVar

Benchmarking

`EvalBackend(std::string const&)`	Choose a likelihood evaluation backend:

Backend	Description
cpu - *default*	New vectorized evaluation mode, using faster math functions and auto-vectorisation. Since ROOT 6.23, this is the default if `EvalBackend()` is not passed, succeeding the **legacy** backend. If all RooAbsArg objects in the model support vectorized evaluation, likelihood computations are 2 to 10 times faster than with the **legacy** backend • unless your dataset is so small that the vectorization is not worth it. The relative difference of the single log-likelihoods with respect to the legacy mode is usually better than 10^{-12}, and for fit parameters it's usually better than 10^{-6}. In past ROOT releases, this backend could be activated with the now deprecated `BatchMode()` option.
cuda	Evaluate the likelihood on a GPU that supports CUDA. This backend re-uses code from the **cpu** backend, but compiled in CUDA kernels. Hence, the results are expected to be identical, modulo some numerical differences that can arise from the different order in which the GPU is summing the log probabilities. This backend can drastically speed up the fit if all RooAbsArg object in the model support it.
legacy	The original likelihood evaluation method. Evaluates the PDF for each single data entry at a time before summing the negative log probabilities.
codegen	**Experimental** - Generates and compiles minimal C++ code for the NLL on-the-fly and wraps it in the returned RooAbsReal. Also generates and compiles the code for the gradient using Automatic Differentiation (AD) with Clad. This analytic gradient is passed to the minimizer, which can result in significant speedups for many-parameter fits, even compared to the **cpu** backend. However, if one of the RooAbsArg objects in the model does not support the code generation, this backend can't be used.
codegen_no_grad	**Experimental** - Same as **codegen**, but doesn't generate and compile the gradient code and use the regular numerical differentiation instead. This is expected to be slower, but useful for debugging problems with the analytic gradient.

Benchmarking RooFit BackEnds

1. Tested whether the following benchmarks support Codegen() and CodegenNoGrad() as evaluation backends:

- benchRooFitBackEnds.cxx
- RooFitUnbinnedBenchmarks.cxx
- RooFitBinnedBenchMarks
- benchCodeSquashAd

2. Identified classes to add support to and issues to resolve to support all benchmarks.
3. Identified issue with RooNLLVar (link to PR)

Benchmark Script Amendment

```
C++
"./benchRooFitBinned -b codegen --benchmark_out=out_codegen.csv" "out_codegen.csv"
```

- Amended script to accept user defined evaluation backend.
- Provided functionality to add other flags (verbose) later in the future as required.

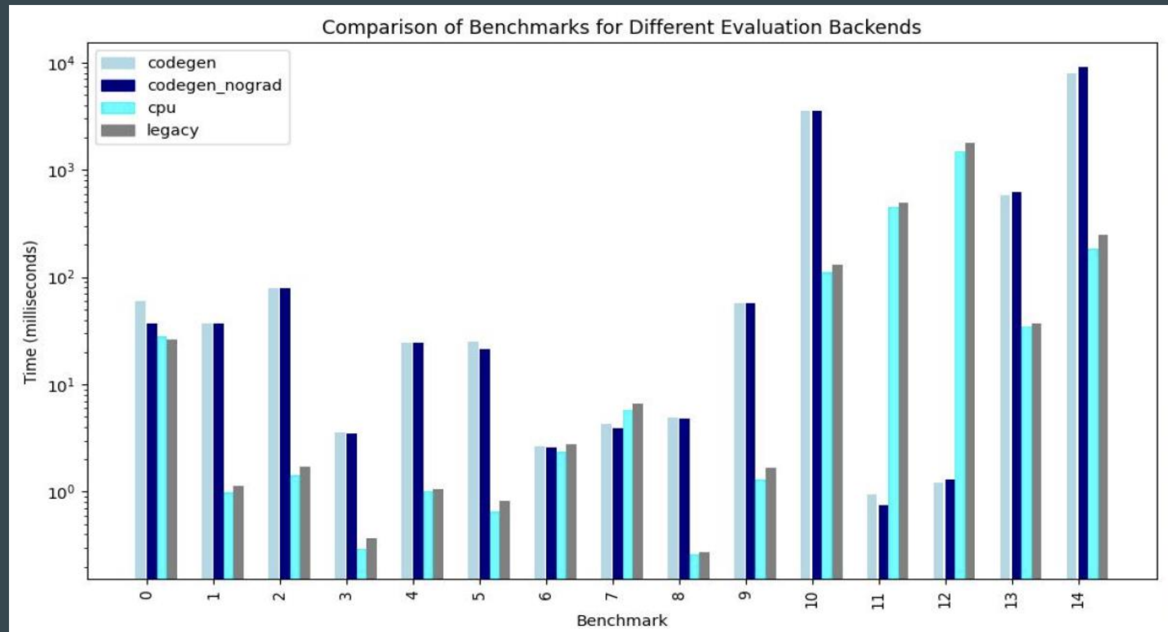
Link to PR

```
{
    benchmark::Initialize(&argc, argv);

    for (int i = 1; i < argc; ++i)
    {
        if (std::string(argv[i]) == "-b")
        {
            if (i + 1 < argc)
            {
                // Set the evalBackend value from the next command-line argument
                evalBackend = argv[i+1];//parseEvalBackend(argv[i + 1]);
            }
            else
            {
                std::cerr << "Missing value for --evalBackend argument" << std::endl;
                return 1;
            }
        }
```

Visualisations and Bash Script

- Created a script to compare the time taken by the different benchmarks.
- Created a bash script to integrate generating benchmarks, storing results of different evaluation backends and storing the visualizations.
- Appendix B: Dictionary explaining the different numbers on the x axis



Conclusions

- Identified classes for which AD support has not been implemented.
- Adding code generation support to those aforementioned classes.
- Implementing support for Benchmarks not currently supported.
- Amending Binned Benchmark script to accept command line inputs.
- Created script to visualize the compare benchmarks.
- Wrote bash script to integrate process to create benchmarks and also generate visualizations.
- Created functionality to measure the time required for the seeding step of migrad using a command line option.

Questions

Appendix A: List of Classes which have code-generation support enabled

Appendix B: dictionary.

Benchmark Name (BenchmarkStep_NChannels_NBins_NCPU)	Number on x-axis
BinnedMigrad1/10/1	1
BinnedMigrad2/10/1	2
BinnedMigrad3/10/1	3
BinnedMigrad1/5/1	4
BinnedMigrad1/15/1	5
BinnedHesse1/10/1	6
BinnedHesse2/10/1	7
BinnedHesse3/10/1	8
BinnedHesse1/5/1	9
BinnedHesse1/15/1	10
BinnedMinos1/10/1	11
BinnedMinos2/10/1	12
BinnedMinos3/10/1	13
BinnedMinos1/5/1	14
BinnedMinos1/15/1	15