RooFit: a probabilistic programming language

Jonas Rembser

CERN (EP-SFT)

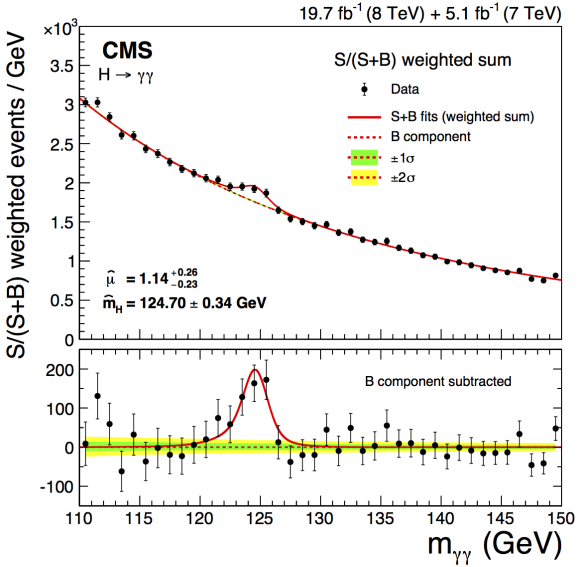
June 23, 2023

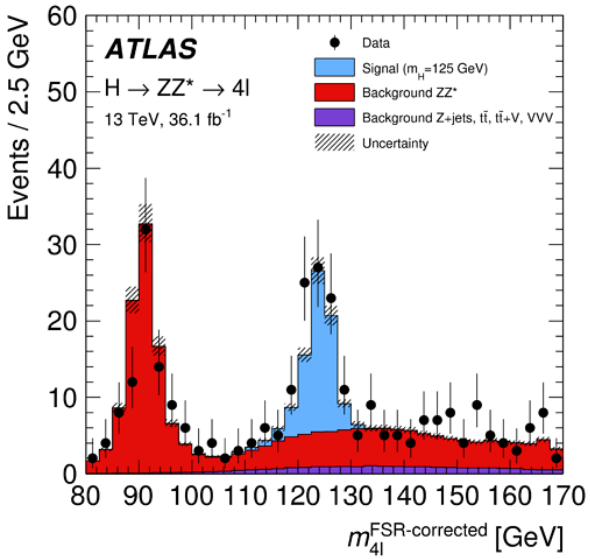
Introduction

- **ROOT** is a C++ data analysis framework widely used in HEP
  - It also provides automatically-generated Python bindings
- **RooFit** is a data modelling library that is part of ROOT
- Two other components of ROOT build on top of RooFit:
  - **RooStats**: statistical procedures with RooFit functions
  - **HistFactory**: higher-level interface to implement binned likelihood models in RooFit

- RooFit is most often used for parameter estimation, e.g. find $\hat{\vec{\theta}}$ that minimizes $NLL = -\sum_i \log p(\vec{x}, \vec{\theta})$.
- But also hypothesis testing, unfolding, etc.

Likelihood fits in HEP





Unbinned likelihood fits

- often many events
- sums of PDFs of different types

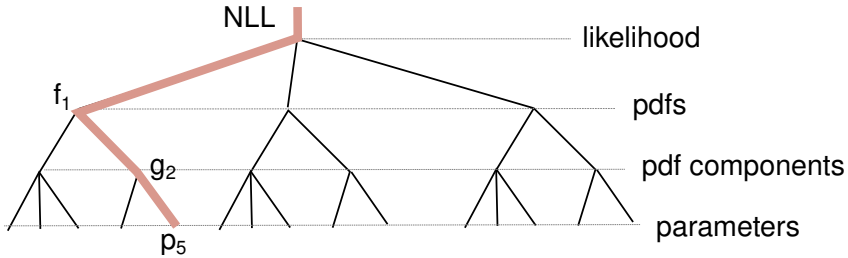
Binned likelihood fits

- often more params. than data points
- many per-bin nuisance parameters

There are also often **combinations** of many binned and unbinned **channels**.

The need for an optimized data modeling library

- Minimizing $NLL(\vec{x}, \vec{\theta})$ with respect to $\vec{\theta}$ is done with the `Minuit` package, which uses numerical differentiation
- In num. diff, parameters are **one at a time** before re-evaluating the function
- $\Rightarrow$ idea: caching all intermediate results and re-evaluate only what is needed

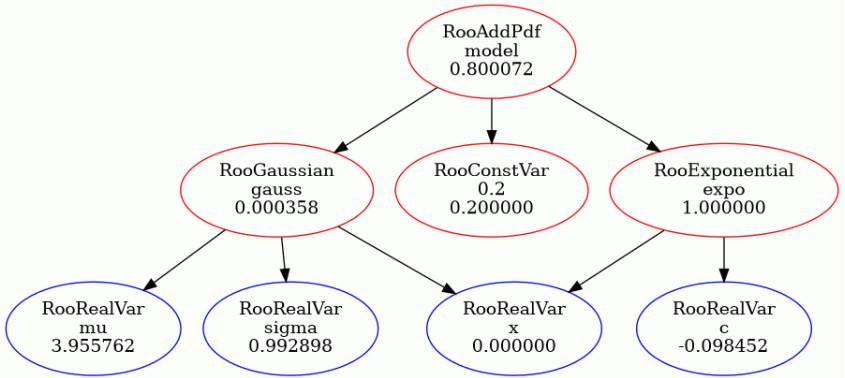


- Hopefully reducing order n^2 cost of gradient closer to $O(n)$

This is one of the central concepts in **RooFit**, enabling fits with 100s of pdfs and 1000s of parameters.

How RooFit models are implemented

- Computation graph represented by C++ objects
- Objects are instances of classes that inherit from `RooAbsArg` class
- Top-level node is evaluated via chain of virtual function calls
- Has also some overhead from caching mentioned before



The computation graph for a simple RooFit model.

Model definition by user is done usually at the level of declaring these C++ classes, although there are higher-level frameworks on top of RooFit.

RooFit example

Define variables (observables and params.):

```
RooRealVar x{"x", "x", 0, 0, 10}; // observable

RooRealVar mu{"mu", "mu", 4, 0, 10};
RooRealVar sigma{"sigma", "sigma", 1, 0.01, 10};
RooRealVar c{"c", "c", -0.1, -10, -0.001};
```

Define pdf (here, Gaussian plus expo.):

```
RooGaussian gauss{"gauss", "gauss", x, mu, sigma};
RooExponential expo{"expo", "expo", x, c};
RooAddPdf model{"model", "0.2 * gauss + 0.8 * expo",
                {gauss, expo}, {RooFit::RooConst(0.2)}};
```

Sample toy dataset:

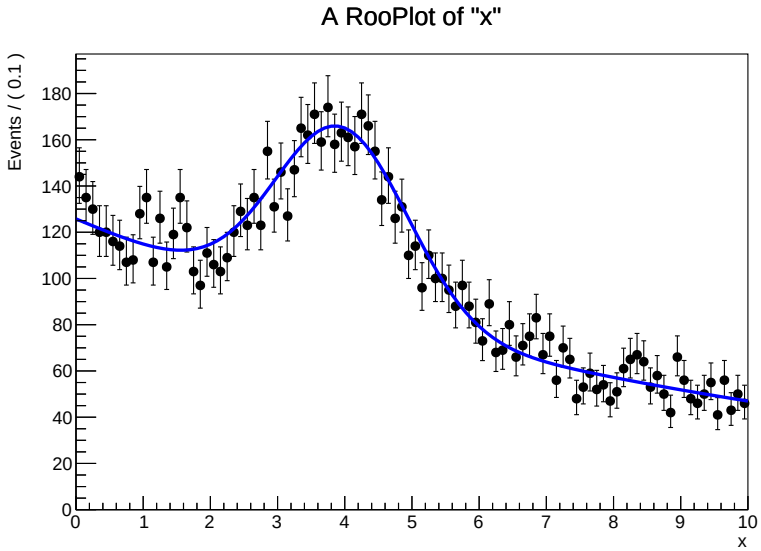
```
std::unique_ptr<RooDataSet> data{model.generate(x, 10000)};
```

Fit model to data with likelihood minim.:

```
std::unique_ptr<RooFitResult> res{model.fitTo(*data)};
res->Print();
```

Plotting

```
RooPlot* frame = x.frame();
data->plotOn(frame);
model.plotOn(frame);
```



Normalization of pdfs

Functions in RooFit can be evaluated with `getVal()`:

```
RooAbsReal &func = ...;
double val1 = func.getVal();
param.setVal(4.0);
double val2 = func.getVal(); // value updated automatically
```

But the value of a pdf is `not well defined` without specifying which variables to normalize over!

```
RooAbsPdf &pdf = ...;
double val3 = pdf.getVal(); // not okay!
```

You have to pass a "normalization set" to evaluate a pdf.

```
RooArgSet normSet{x1, x2};
double val4 = pdf.getVal(normSet);
```

RooFit takes care of the integrals

- Normalization is done **automatically**
- Functions can be queried for analytical integral capabilities
  - Similar interface for **sampling** as well
- RooFit evaluates your likelihood with as little numerical integrals as possible

Conditional pdf example:

$$p(x|y) = \frac{p(x,y)}{p(y)} = \frac{p(x,y)}{\int p(x,y)dx}$$

Observable subdomain example:

$$p(x|\text{subrange}) = p(x)\frac{\int_{\text{full}} p(x)dx}{\int_{\text{subrange}} p(x)dx}$$

Define "model" as a pdf depending on x and y without caring about integrals:

```
model = ...
```

NLL using p(x,y):

```
nll1 = model.createNLL(dataXY);
```

NLL using p(x,y), restricted to defined **subrange**:

```
nll2 = model.createNLL(dataXY, Range("subrange"));
```

Conditional NLL using p(x|y):

```
nll3 = model.createNLL(dataXY, ConditionalObservables(y));
```

More on this in the RooFit conditional fit tutorial.

Automatic/algorithmic differentiation in RooFit

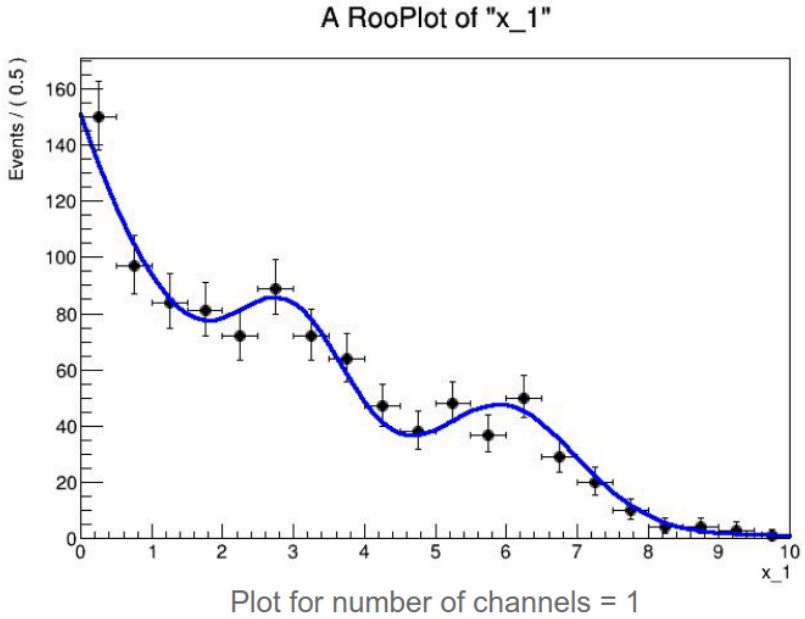
AD is manly important for minimizing the likelihood funcitons with Minuit.

Our approach to get there:

1. Generate no-overhead C++ code for the model based of the RooFit model
   - everything inlined
   - no virtual function calls
   - no caching
2. Use the **clad** source-code-transformation AD tool to generate the gradient for that function (and eventually also the Hessian for parameter uncertainty estimation)
3. Pass generated sources to Minuit for minimization

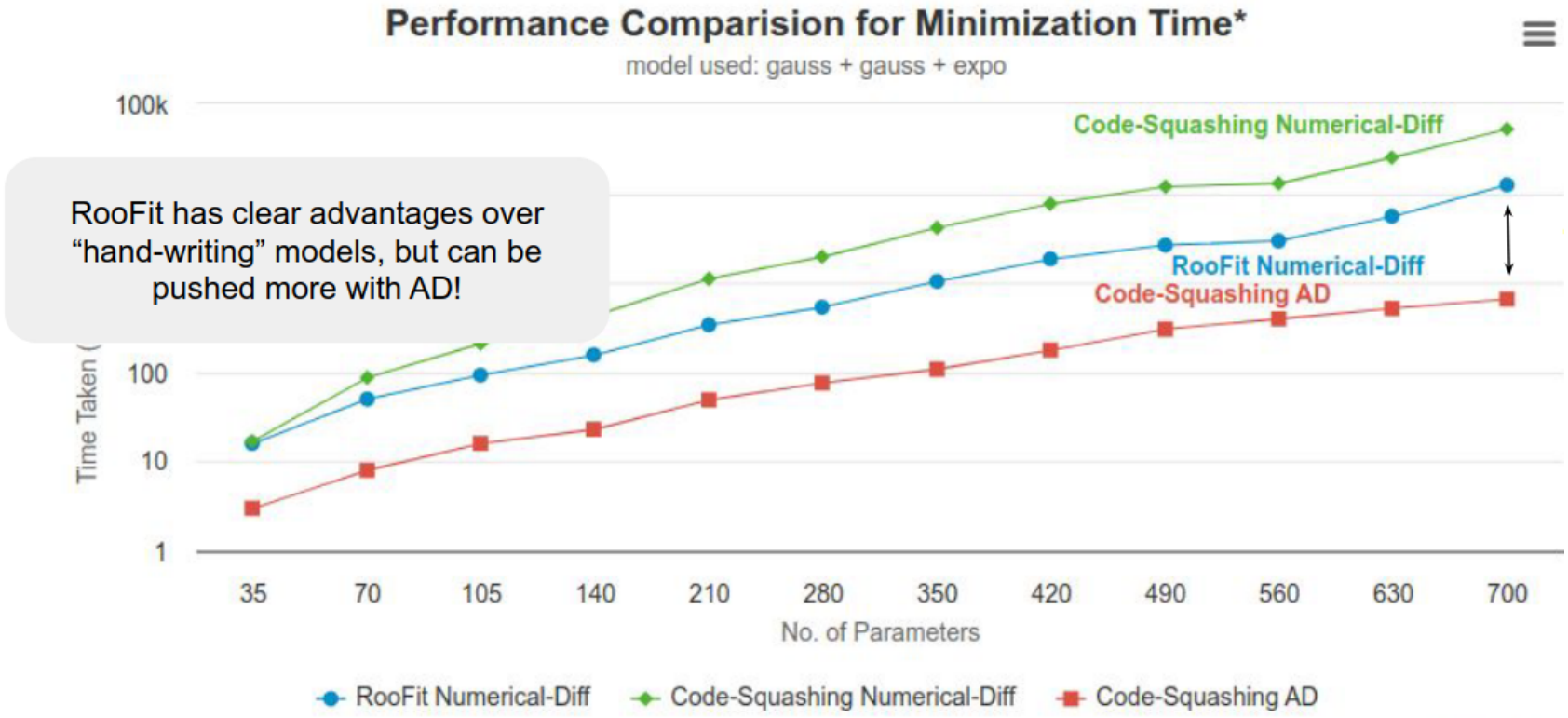
Still work in progress, in particular the user interface, will be part of the next ROOT release (v6.30).

Automatic/algorithmic differentiation in RooFit



For the next slide, we benchmarked a binned likelihood fit of model with two Gaussians plus exponential (7 parameters).

Automatic/algorithmic differentiation in RooFit



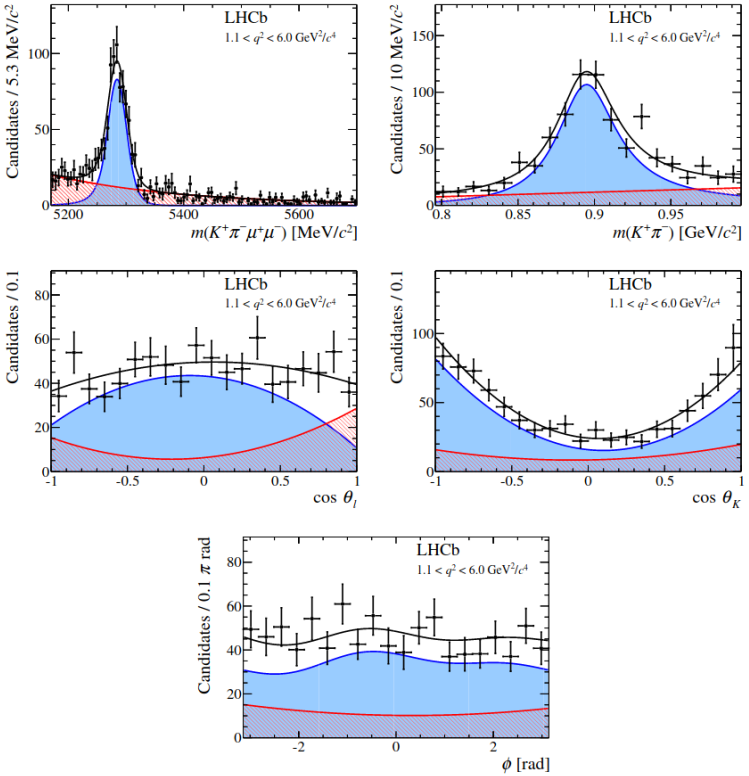
Measuring the minimization (Minuit2/migrad) time for n simultaneous liklihood fits as on the slide before. From [Garimas talk at CHEP 2023](#).

Numerical integrals

First example of where I worry that black-box AD is not optimal:

- E.g., LHCb is sometimes fitting 4 or 5 observables, analytic integral not known
- Numerical integral necessary (MC integration with importance sampling...)
- Is this example for infamous *nested fixed point operations* we should avoid in AD?
  1. Inside AD: numeric integration of observables until tol. is reached
  2. "Consumer" of AD grads.: likelihood (L) optimization with respect to parameters

$\Rightarrow$ We want to look into the numeric integrals and L optimization more holistically.



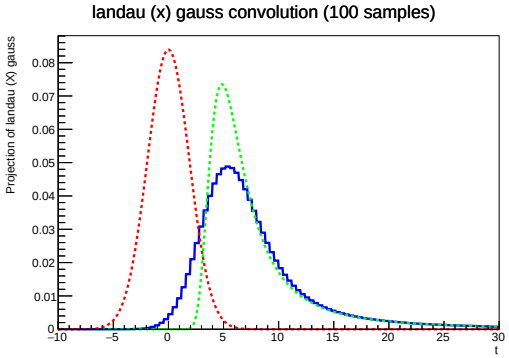
Example: angular analysis by LHCb on $B^0 \to K^{*0}\mu^+\mu^-$ *decays.*

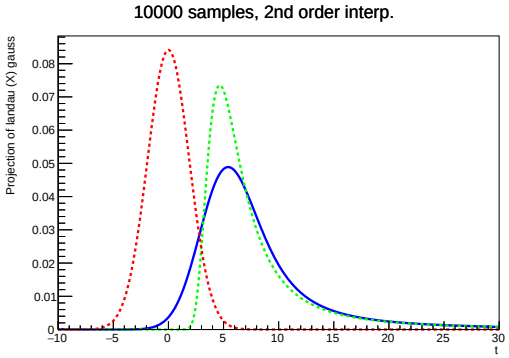
Modeling convolutions

Second example of where I worry that black-box AD is not optimal:

- Often, the final pdf is the **convolution** of two pdfs (e.g., physics model and detector effects)
- How convolutions are modeled in RooFit:
  1. Sample input pdfs at n points (default $n = 10000$)
  2. Compute discrete convolution with FFT
  3. Smooth result with 2nd order polynomial interpolatios

Is black-box AD on this procedure a good idea, or are there short cuts?





From the convolution tutorial.

Conclusions

RooFit is a data modeling package in C++ that is used by many HEP experiments.

- Provides many features you expect from a probabilistic programming language:
  - Arbitrary pdfs can be defined that are **automatically normalized** over observables
  - The pdfs can be used for efficient **sampling**
  - Easy **creation of likelihood functions** by combining pdf with data
  - Interface to minimizer (`Minuit`)
- Functions are represented by graph of C++ objects that re-evaluate each other
- Our approach to automatic/algorithmic differentiation in NLL minimization:
  - Generate overhead-free C++ code from RooFit function
  - JIT + source-code-transformation AD applied on that code with `Clad`
  - Black-box AD so far, but that might hit limits in e.g. numerical integration or interpolating FFT results