RooFit workshop 2024

Jonas Rembser

April 10 and 11, CERN 2024

ROOT

Data Analysis Framework

https://root.cern

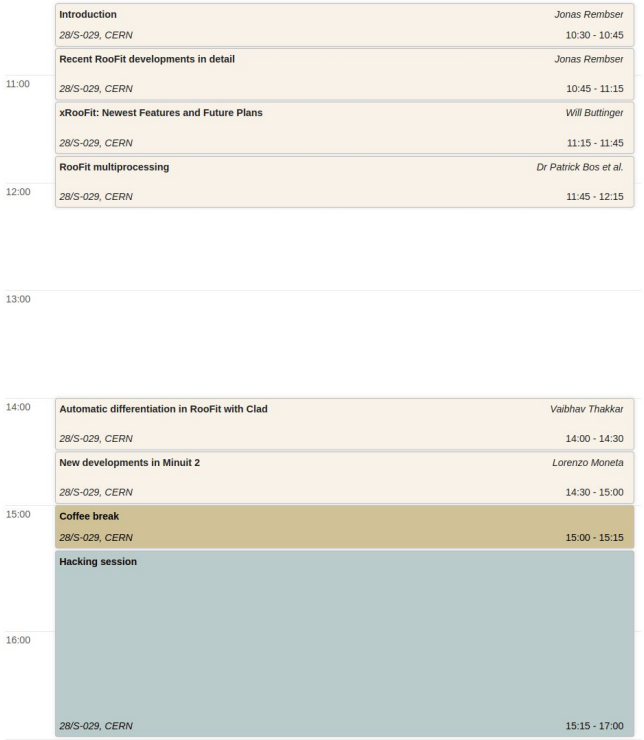
Workshop introduction

- **RooFit workshop 2024** with RooFit developers and key users and framework developers

- Goals:
  - **Showcasing** new RooFit development
  - **Support** experiments in making use of new features
  - Hear user **stories** and **feedback** on RooFit
  - **Plan** RooFit development for the years ahead

- Structure:
  - **interactive**: 50 % talks, 50 % questions and discussions
  - *Thursday*: **developer** presentations and **hackathon**
  - *Friday*: **user** presentations and **planning** discussions

Agenda today:



Recent RooFit developments in detail

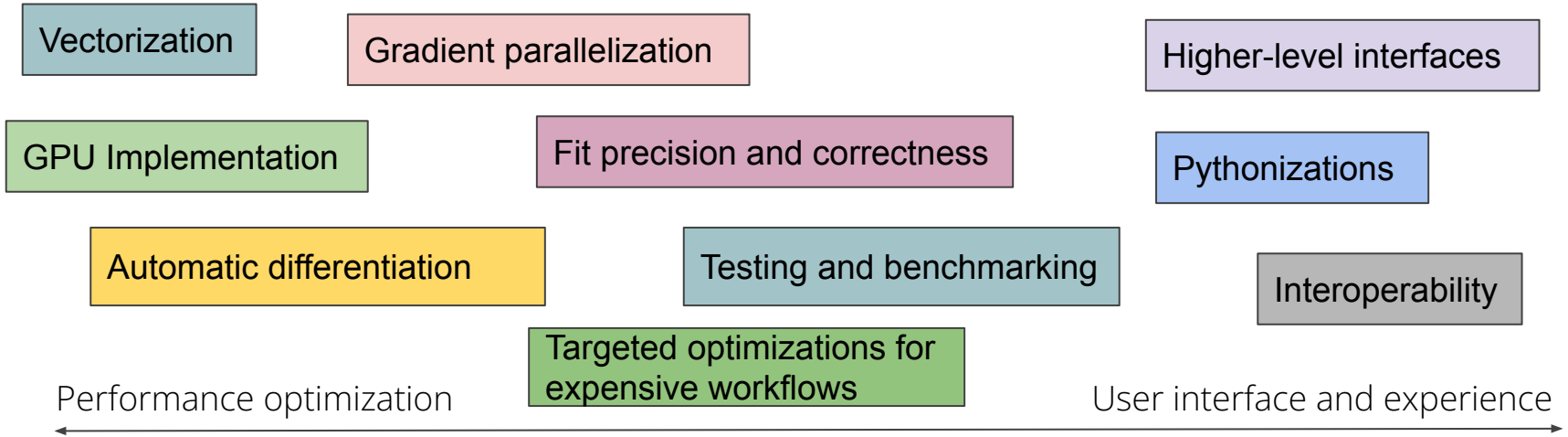
About RooFit

- **RooFit**: C++ library for statistical data analysis in ROOT
  - RooFit + RooStats + HistFactory
  - Supports **diverse usecases** with different performance requirements, e.g.:
    - Complicated *binned* models with many nuisance parameters but few data entries
    - *Unbinned* fits of analytic shapes to huge datasets

- Recent development focused on:
  - **Performance** boost (preparing for larger datasets of **HL-LHC**)
  - More **user friendly** interfaces and high-level tools

RooFit development areas

In which areas does RooFit evolve (besides bugfixes)?



- Not all areas are covered with the same level of activity
- Some areas started to be covered only recently (*automatic differentiation, interoperability*)

CC BY

Computation graphs in RooFit

RooFit evaluates expression trees many times for different parameter values to find NLL minima.

Why rewriting RooFit NLL evaluation backend:

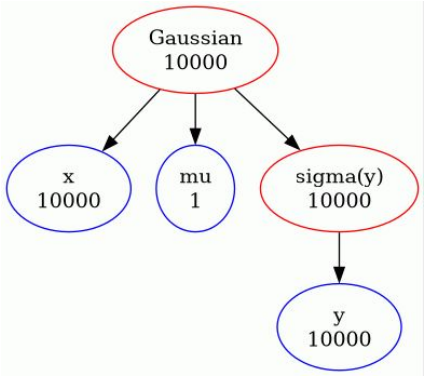
- Old RooFit computation: re-evaluate expression tree of *for each event*
- Lots of function calls, **no vectorization possible**

Should have been easy to improve and do on GPU?!

1. Allocate memory for results
2. Call vectorized function/CUDA kernel for each node[1] in topological order if values of children have changed

[1] `RooAbsArg` in RooFit

Expression tree with observables x and y for 10000 data points:
Gaussian(x|mu,sigma(y))




```
RooRealVar x{"x", "x", 0.0, -20.0, 20.0};
RooRealVar y{"y", "y", 0.0, 0.0, 1.0};

RooRealVar mu{"mu", "mu", 0.0, -20.0, 20.0};
RooFormulaVar sigma{"sigma", "1.0 + 2.0 * y", {y}};

RooGaussian gauss{"gauss", "gauss", x, mu, sigma};
```

Computation graphs in RooFit

RooFit model evaluation is not straight forward:

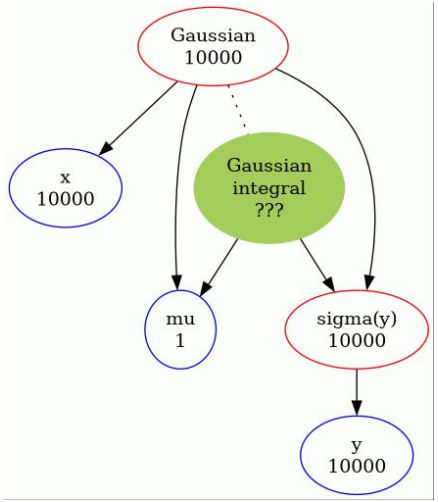
- Nodes often own other nodes that they evaluate
- These *internal nodes* are not registered in the graph
- Sometimes these nodes are even clones of entire subgraphs

Typical example: **normalization integrals**
(still harmless compared to other cases, but good for illustration)

Dynamic nature of computation graphs in RooFit makes organizing data flow and computations in a heterogeneous computing environment a challenge.

In other words: data structure for model *building* not completely suitable for *evaluation*.

Evaluating model for given normalization observables dynamically extends computation graph, adding new disconnected nodes

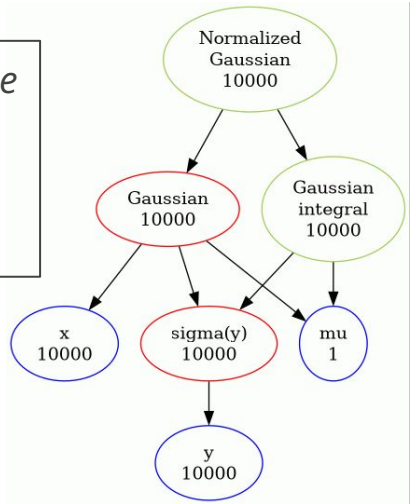



```
gauss.getVal(/*normSet=*/x);
```

Computation graphs with fixed normalization

New mechanism to "compile" the graph for a given normalization set to fulfill condition ⟶

Each RooAbsArg involved in the evaluation must be connected to the top node via RooFits client-server relations.



If your RooFit classes don't fulfill this yet, you should consider overriding:
`RooAbsArg::compileForNormSet()`

- Function called recursively in NLL creation when using `BatchMode()`
- Result is ready for heterogeneous eval.
- Mechanism also used for the C++ code generation from RooFit models that enables automatic differentiation

This function can also be used to hook in graph **optimizations**.

```
auto nll = gauss.createNLL(
    *data,
    ConditionalObservables(y),
    EvalBackend("cpu")
); // create NLL object
```

```
nll->Print("v"); // get some info on the graph evaluation order
```

```
--------------------------------------------------------------------------------
| Idx | Name                    | Class            | Size    | From Data |
--------------------------------------------------------------------------------
| 1   | y                       | RooRealVar       | 10000   | 1         |
| 2   | sigma                   | RooFormulaVar    | 10000   | 0         |
| 3   | mu                      | RooRealVar       | 1       | 0         |
| 4   | x                       | RooRealVar       | 10000   | 1         |
| 5   | gauss                   | RooGaussian      | 10000   | 0         |
| 6   | gauss_Int[x]            | RooRealIntegral  | 10000   | 0         |
| 7   | gauss_over_gauss_Int[x] | RooNormalizedPdf | 10000   | 0         |
| 8   | nll                     | RooNLLVar        | 1       | 0         |
--------------------------------------------------------------------------------
```

The vectorized evaluation functions

- The BatchMode backend uses **new functions** in RooAbsReal that you can **override** to add support for CPU and GPU of your class:
  - `RooAbsReal::canComputeWithCuda()`
  - `RooAbsReal::doEval()`

- Implementation of RooFit classes in ROOT uses `RooBatchCompute` library to implement `computeBatch()`:
  - **Architecture-specific accelerator libraries** for key functions
  - **Optimal one loaded at runtime**, given current architecture
  - More details in the ACAT 2021 talk

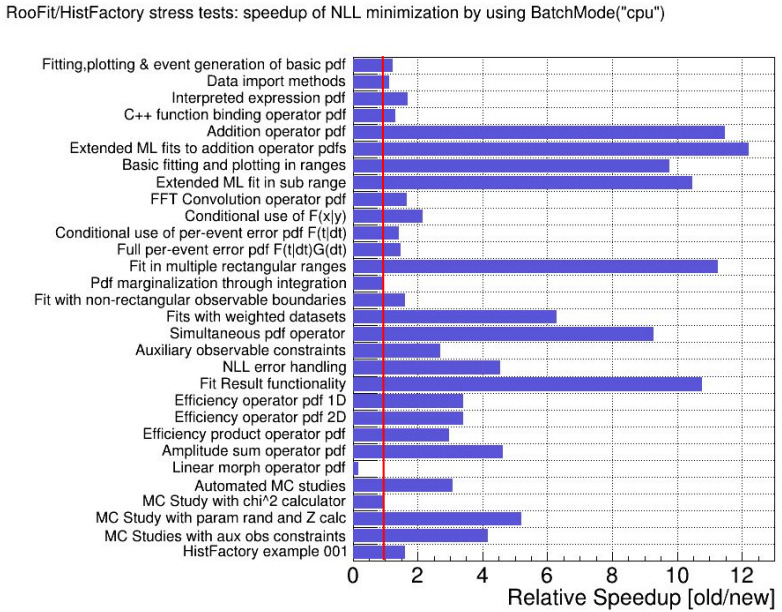
- Add the FastEvaluations stream to the RooMsgService the get **info printouts when** your RooAbsArgs **don't support** the new `RooAbsReal::doEval()`:

```
RooMsgService::instance().addStream(
        RooFit::Info, Topic(RooFit::FastEvaluations)
);
```

Benchmarking the RooFit test suite

- Plot shows relative time spent for minimizations in stressRooFit tests for `EvalBackend("cpu")` and `"legacy"`
- Significant speedup for almost all tests from a combination of:
  a. **Vectorized** evaluation
  b. Optimized computation graphs
  c. Less function calls
- Average speedup of **4.4x**

The new CPU backend will be the default in ROOT 6.32!



Results obtained with ROOT 6.28.04 from CHEP 2023.

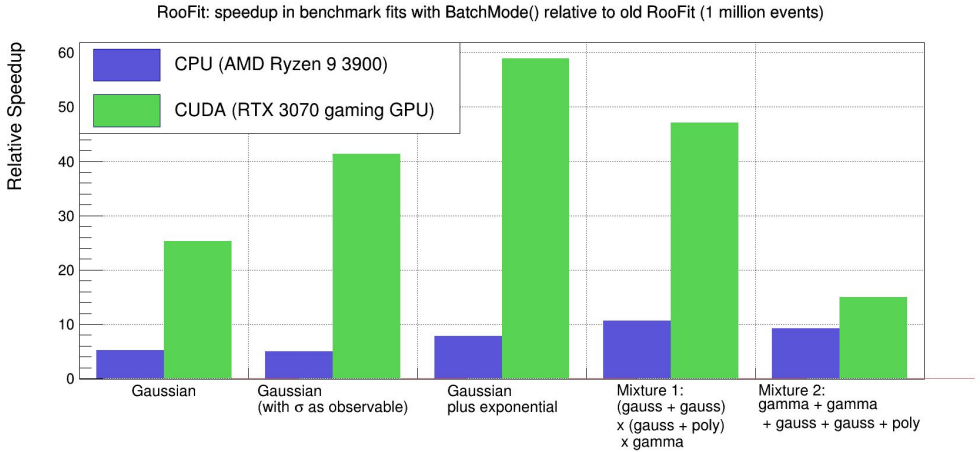
Overview on NLL evaluation backends

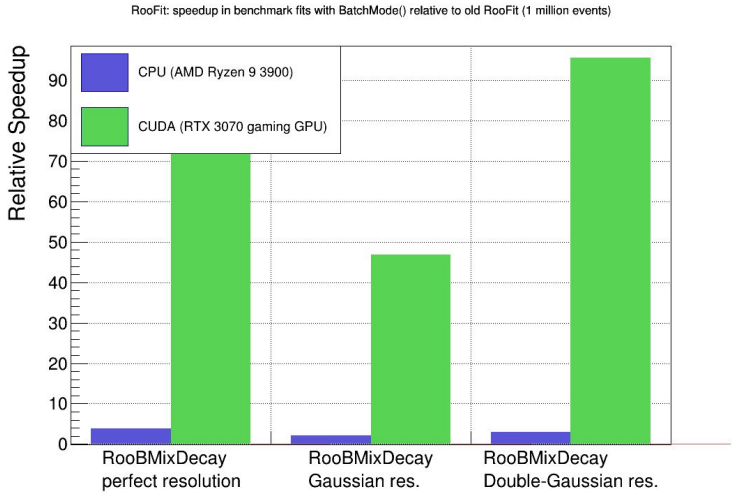
From the RooAbsPdf documentation:

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.

The CUDA evaluation backend

Great speedup of unbinned fits with many events with the CUDA backend:





- *Caveat*: comparisons vs. one CPU thread, but RooFit on CPU is notoriously difficult to parallelize
- *Still* potential for more speedup: no caching in GPU backend yet
- Plan to also **do numeric integrals on GPU in the future** to support more B-physics usecases, i.e. amplitude fits.

benchRooFitBackends and RooFitUnBinnedBenchmarks in rootbench repo, plotting script is in same directory. Try it yourself! Remember to use a ROOT build with -Dcuda=ON

Example: RooAbsPdf with support for new evaluation

- **Fastests way to get started** implementing the interfaces: generate implementation code with the RooClassFactory and use it as a template:

free function:

```
inline double MyPdf_evaluate(double x, double alpha) {
  return (1+0.1*fabs(x)+sin(sqrt(fabs(x*alpha+0.1))));
}
```

old interface:

```
double MyPdf::evaluate() const { return MyPdf_evaluate(x, alpha); }
```

vectorized:

```
void MyPdf::doEval(RooFit::EvalContext &ctx) const {
  std::span<const double> xSpan = ctx.at(x);
  std::span<const double> alphaSpan = ctx.at(alpha);

  std::size_t n = ctx.output().size();
  for (std::size_t i = 0; i < n; ++i) {
     ctx.output()[i] = MyPdf_evaluate(xSpan.size() > 1 ? xSpan[i] : xSpan[0],
                                      alphaSpan.size() > 1 ? alphaSpan[i] : alphaSpan[0]);
  }
}
```

for CUDA, replace this with kernel call

broadcasting of scalar values

to support "codegen":
(see later slides on AD)

```
void MyPdf::translate(RooFit::Detail::CodeSquashContext &ctx) const {
  ctx.addResult(this, ctx.buildCall("MyPdf_evaluate", x, alpha));
}
```

The new RooFit::Evaluator and CudaInterface

- Flags to fitTo() and createNLL() are only scratching the surface. For frameworks, we need **low-level interfaces**!
  - The aforementioned `compileForNormSet()` to build safe computation graphs
  - An experimental CudaInterface abstraction to manage data on the GPU without CUDA
  - The new `RooFit::Evaluator` (formerly known as *RooFitDriver*) to manage vectorized evaluation on CPU or GPU *(or mixed)*

Example on the right: evaluate RooFit model at many points **on the GPU**.

```
namespace CudaInterface = RooFit::Detail::CudaInterface;

// Define model
RooRealVar x{"x", "x", 0, -10, 10};
RooRealVar mu{"mu", "mu", 0, 0, 10};
RooRealVar sigma{"sigma", "sigma", 3, 0.1, 10};

RooGaussian gauss{"gauss", "gauss", x, mu, sigma};

std::size_t numBins = 20;
x.setBins(numBins);

RooArgSet normSet{x};

// Create array of observable values
std::vector<double> xValsHost;
for (std::size_t i = 0; i < numBins; ++i) {
  x.setBin(i);
  xValsHost.push_back(x.getVal());
}

// Copy observable values to the array
CudaInterface::DeviceArray<double> xValsDevice{numBins};
CudaInterface::copyHostToDevice(xValsHost.data(), xValsDevice.data(), numBins);

// Compile pdf for evaluation with RooFit::Evaluator
std::unique_ptr<RooAbsReal> gaussCompiled{RooFit::Detail::compileForNormSet(gauss, normSet)};

// Instantiate Evaluator and set device input array
RooFit::Evaluator evaluator{*gaussCompiled, /*useGPU*/true};
evaluator.setInput("x", {xValsDevice.data(), numBins}, /*isOnDevice=*/ true);

// Evaluate
std::span<const double> deviceOut = evaluator.run();

// Copy back to host and print results
std::vector<double> hostOut(deviceOut.size());
CudaInterface::copyDeviceToHost(deviceOut.data(), hostOut.data(), numBins);
for (int i = 0; i < numBins; ++i) {
  std::cout << xValsHost[i] << "    " << hostOut[i] << std::endl;
}
```

```
root [0]
Processing cuda_example.C...
[#1] INFO -- using CUDA computation
library
-9.5    0.000884418
-8.5    0.0024041
-7.5    0.00584778
-6.5    0.0127285
-5.5    0.0247917
-4.5    0.0432096
-3.5    0.0673907
-2.5    0.0940513
-1.5    0.117456
-0.5    0.131259
0.5     0.131259
1.5     0.117456
2.5     0.0940513
3.5     0.0673907
4.5     0.0432096
5.5     0.0247917
6.5     0.0127285
7.5     0.00584778
8.5     0.0024041
9.5     0.000884418
root [1] .q
```

RooFit pythonizations

- PyROOT bindings **more pythonic** in *6.26*
- Now you can for example:
  - use **Python keyword arguments** instead of RooFit command arguments
  - pass around **Python sets or lists** instead of `RooArgSet` or `RooArgList`
  - pass **Python dictionaries** to functions that take `std::map<>`
  - implicitly convert floats to `RooConstVar` in `RooArgList/Set` constructors
- All pythonizations are documented
- Some Pythonizations to help with C++/Python lifetime issue
  - Still there are memory leaks when returning owning pointers
- *See also this ROOT meeting presentation*

Example code from the rf316_llratioplot.py tutorial showcasing the pythonizations:

```
# Create background pdf poly(x)*poly(y)*poly(z)
px = ROOT.RooPolynomial("px", "px", x, [-0.1, 0.004])
py = ROOT.RooPolynomial("py", "py", y, [0.1, -0.004])
pz = ROOT.RooPolynomial("pz", "pz", z)
bkg = ROOT.RooProdPdf("bkg", "bkg", [px, py, pz])

# Create composite pdf sig+bkg
fsig = ROOT.RooRealVar("fsig", "signal fraction",
                       0.1, 0., 1.)
model = ROOT.RooAddPdf("model", "model",
                       [sig, bkg], [fsig])

data = model.generate((x, y, z), 20000)

# Make plain projection of data and pdf on x observable
frame = x.frame(Title="Projection on X", Bins=40)
data.plotOn(frame)
```

RooFit with NumPy, Pandas, and RDataFrame

- ROOT *v6.26* **new converters** between NumPy arrays/Pandas dataframes and **RooDataSet/RooDataHist**
  - No translation from RooDataHist to dataframe because histograms are in general multi-dimensional
  - Tutorial in Python
- New `RooRealVar.bins()` function to get RooFit **bin boundaries** as NumPy array
- Creating **RooFit datasets** from **RDataFrame**
  - Works for both `RooDataSet` and `RooDataHist`
  - Weighted filling still needs to be implemented *(does LHCb need this?)*
  - Tutorial in C++ and Python

Example of exporting RooDataSet to Pandas:

```
from ROOT import RooRealVar, RooCategory, RooGaussian

x = RooRealVar("x", "x", 0, 10)
cat = RooCategory("cat", "cat",
                  {"minus": -1, "plus": +1})

mean = RooRealVar("mean", "mean",
                  5, 0, 10)
sigma = RooRealVar("sigma", "sigma",
                   2, 0.1, 10)

gauss = RooGaussian("gauss", "gauss",
                    x, mean, sigma)


data = gauss.generate((x, cat), 100)

df = data.to_pandas()
```

	x	cat
0	6.997865	-1
1	7.211196	-1
2	3.198248	1
3	5.015824	1
4	7.782388	1
...	...	...
95	6.878027	-1
96	0.475900	1
97	4.451101	-1
98	3.481015	-1
99	4.010105	-1

100 rows × 2 columns

RooFit ATLAS benchmarks

RooFit development mainly done by the ROOT team…

…but it works best in **collaboration with the experiments**!

Example: hackathon with ATLAS in December 2021

- ATLAS physics coordination agreed to openly provide Higgs combination with toy data (cutting edge analyses!)
- Hackathon: ATLAS physicists and ROOT team together work on making fits faster for three full days

Result: great speedups, avoiding unnecessary overhead, and benchmark suite that *inspires RooFit development to this day*!

Output of the ATLAS Higgs combination benchmark:

```
Benchmarking workspace WS-Comb-STXS_toy.root...
-----------------------------------------------------------------------
Benchmark                                       Time           CPU
-----------------------------------------------------------------------
createNLL/0/iterations:1                        56.8 s         56.8 s
createNLL_BatchMode/1/iterations:1              75.7 s         75.5 s
evaluateNLL/1/0                                 680 ms         679 ms
evaluateNLL_BatchMode/1/1                       592 ms         592 ms
evaluateNLL_SingleKick/0/0                      1.01 ms        1.01 ms
evaluateNLL_BatchMode_SingleKick/0/1            3.04 ms        3.03 ms

Initial NLL values
  - BatchMode("off"): 3719009.08412513602524996
  - BatchMode("cpu"): 3719009.08412513323128223
```

- Benchmark helped to bring `createNLL` time **from 30 to 1 min** (ROOT 6.28 vs 6.24)
- Also evaluation time was reduced
- Result: workflows can be moved **from grid to laptop**

BatchMode("cpu") is not faster here, the complicated model with custom pdf classes doesn't support it well. But note that the NLL values agrees to 10^{-14}!

Automatic differentiation in RooFit

Likelihood minimization with AD in RooFit

RooFit has prototype support of for **minimization with AD gradients** generated by Clad.

- Activate by passing `"codegen"` or `"codegen_no_grad"` to the `EvalBacked()` argument of `RooAbsPdf::fitTo()` / `RooAbsPdf::createNLL()`:
  - `pdf.fitTo(data, RooFit::EvalBackend("codegen"))`

It's a **one-line change** again!

- Under the hood, this generates, compiles and **uses gradient code in the minimization**
- It doesn't work for all types of pdfs yet, but support is growing. In particular, standard **HistFactory** models are fully supported.
- See also the RooAbsPdf documentation.

RooFit AD in detail

- Helper class to manage jitted code: **`RooFuncWrapper`** *(experimental)*
- Takes the *compiled* computation graph as input (similar to `RooFit::Evaluator`)
- Generates the code that calls free functions for functions and integrals
- You can add support for the code generation by implementing `RooAbsArg::translate()`
- More details in the RooFit developer documentation
- The RooFuncWrapper provides the analytic gradient via `RooAbsReal::gradient(double *)`

```
gauss_compiled = ROOT.RooFit.Detail.compileForNormSet(gauss, ROOT.RooArgSet(x))

wrapper = ROOT.RooFuncWrapper("wrap", "wrap", gauss_compiled, ROOT.nullptr, ROOT.nullptr, False)
wrapper.createGradient()  # calls Clad under the hood with the interpreter

wrapper.dumpCode() # inspect the generated code
wrapper.dumpGradient() # inspect the gradient code

grad_out = np.zeros(3)
wrapper.gradient(grad_out) # gradient for floating parameters in alphabetical order
```

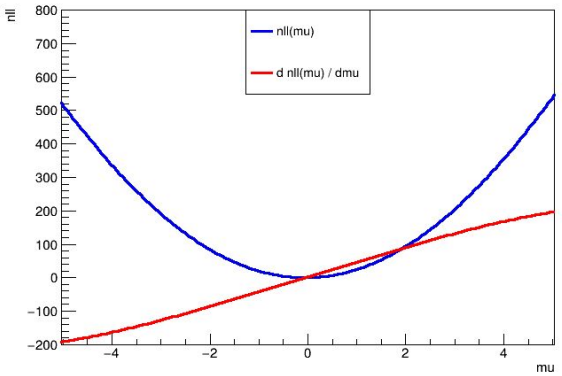
The generated code dump:

```
double roo_func_wrapper_0(double *params, double const *obs, double const *xlArr)
{
  const double t3 = gaussianIntegral(-6.000000, 6.000000, params[0], params[1]);
  const double t4 = gaussianEvaluate(params[2], params[0], params[1]);
  const double t5 = t4 / t3;

  return t5;
}
```

Manually evaluating gradients

- For advanced use, you can **access the gradient values** manually
- With the "`codegen`" backed, RooAbsPdf::`createNLL()` returns a `RooFuncWrapper`
- You can evaluate the gradient with `RooAbsReal::gradient()`

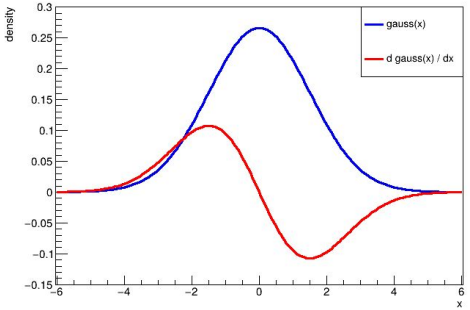
```
data = gauss.generate(x, 100)
nll = gauss.createNLL(data, EvalBackend="codegen")  # this is a RooFuncWrapper

n_vals = 200
mu_vals = np.linspace(mu.getMin(), mu.getMax(), n_vals)
f_vals = np.zeros(n_vals)
d_f_vals = np.zeros(n_vals)

# Evaluate function and gradient
grad_out = np.zeros(2) # two parameters
for i in range(n_vals):
    mu.setVal(mu_vals[i])
    f_vals[i] = nll.getVal()
    grad_out[0] = 0.0
    nll.gradient(grad_out)
    d_f_vals[i] = grad_out[0] # mu is fist (parameters sorted alphabetically)
```



Advanced example - gradients of PDFs

- Example of **manually** evaluating a pdf and its gradient
- `RooFit::Evaluator` wraps vectorized eval., `RooFuncWrapper` takes care of gradient generated with Clad
- Using **RooGenericPdf** to showcase how the gradient code includes the *numeric normalization integral!*
- In future: generalize `RooFit::Evaluator` to reduce boilerplate code

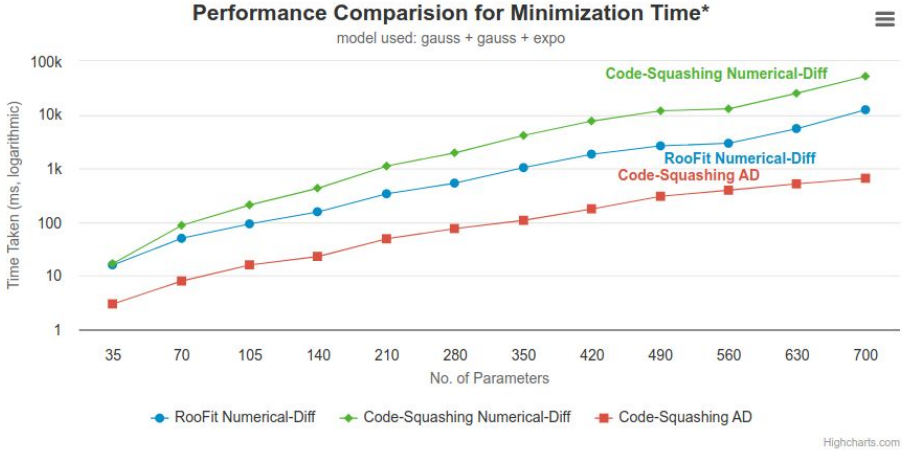
```
x = ROOT.RooRealVar("x", "x", -6, 6)
mu = ROOT.RooRealVar("mu", "mu", 0, -6, 6)
sigma = ROOT.RooRealVar("sigma", "sigma", 1.5, 0.1, 10)
gauss = ROOT.RooGenericPdf("gauss", "std::exp(-0.5*(x[0]-x[1])*(x[0]-x[1])/(x[2]*x[2]))",
                           [x, mu, sigma])

model = MyRooFitEvaluator(gauss, observables=[x]) # Object that manages

model.variables.Print() # Get variables information

x_vals = np.linspace(x.getMin(), x.getMax(), 200) # 1-d vector
X_vals = np.expand_dims(nx_vals, axis=1) # column vector

f_vals = model.eval(X_vals) # gauss(x)
d_f_vals = model.eval_gradient(X_vals)["x"] # d gauss(x) / dx
```



`MyRooFitEvaluator` (wraps `RooFit::Evaluator` and `RooFuncWrapper` for AD):

RooFit AD - benchmarking

- In simple benchmarks scaled to **many parameters**, minimization time with MIGRAD can be significantly sped up
  - Plot on the *top right* (binned Gauss + exp. in 100 channels, from CHEP 2023)
- In the real-world ATLAS HistFactory example, speedup for BFGS steps in Minuit 2 is sped up by **factor 4**
  - Table on the *bottom right*



- However, some fits spend most time in the **initial search for a convex starting point** *(aka. seeding in Minuit)*
- **JIT time** for big models can become very long: the current bottleneck of the approach
  - Plan to split NLL up in smaller functions that compile faster

ATLAS HistFactory (5 channels)

	Num. diff [ms]	AD [ms]
Function JIT time:	-	365
Clad JIT time:	-	26270
IR to machine code time:	-	21079
Time for Seeding:	4497	4639
Time for Minimization:	797	211

Conclusions so far

Conclusions so far

- RooFit development focused on **performance** in the last years:
  - new **vectorizing** evaluation backed that is the **default in ROOT 6.32**
  - **CUDA** implementation of new evaluation backend
  - initial support for **automatically-generated gradients**
  - With the **new interfaces** like `RooFit::Evaluator` and `compileForNormSet()`, we make these features available to *power users and framework developers* at a lower level

- On the **interfaces side**, the focus was Python:
  - **interoperability** with scientific Python ecosystem
  - **pythonizations** for standard RooFit functions

- Several benchmarking projects were initiated:
  - Benchmark suite with real-life **ATLAS models**
  - Comparative benchmark suite with other Python tools

Likelihood serialization with HS3

RooFit HS^3 example

Example to showcase the layout of the HS3 standard two-channel model with measurement data

Some concepts map directly to `RooWorkspaces`:

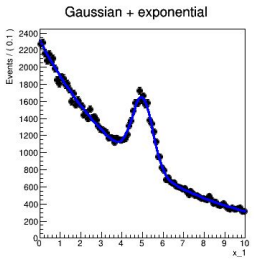
- `RooAbsPdfs`:
  distributions
- `RooAbsReals`:
- **functions**
- `RooRealVars`:
  initial value in "**parameter_points**", range in "**domains**"
- `RooDataHist` and `RooDataSet`:
  data

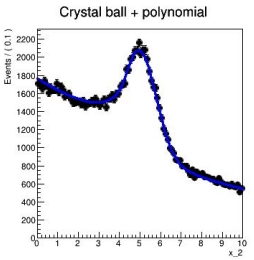
Some new ideas to aid likelihood definition:

- **likelihoods**: define combined likelihood terms
  closest RooFit concept: `RooSimultaneous`+ combined data
- **analyses**:
  define likelihood terms, plus POIs, nuisances, etc.
  closest RooFit concept: `RooStats::ModelConfig`

This example is the abbreviated output of the [testHS3SimultaneousFit] unit test.

```
"distributions": [
    {    "name" : "sig_1",
         "type" : "gaussian_dist",
         "x"    : "x_1",
         "mean" : "mean",
         "sigma": "sigma_1"
    },
    {    "name" : "sig_2",
         "type" : "crystalball_dist",
         "m"    : "x_2",
         "m0"   : "mean",
         "sigma": "sigma_2",
         "alpha": "alpha",
         "n"    : "ncb"
    },
    {   "name": "model_1",
        "type": "mixture_dist",
        "coefficients": ["n_sig_1", "n_bkg_2"],
        "summands": ["sig_1", "bkg_1"]
    },
    {    "name": "bkg_1",
         "type": "exponential_dist",
         "x"    : "x_1",
         "c"    : "c_1"
    },
    {    "name"         : "bkg_2",
         "type"         : "polynomial_dist",
         "x"            : "x_x",
         "coefficients": ["3", "a_1", "a_2"]
    },
    {    "name": "model_2",
         "type": "mixture_dist",
         "coefficients": ["n_sig_2", "n_bkg_2"],
         "summands": ["sig_2", "bkg_2"]
    }
],
"functions": [],
"likelihoods": [
    {    "data": ["data_channel_1", "data_channel_2"],
         "distributions": ["model_channel_1", "model_channel_2"],
         "name": "my_likelihood"
    }
],
"analyses": [
    {    "name": "my_analysis",
         "likelihood": "my_likelihood",
         ... etc. ...
    }
],
"data": [
    {   "name": "data_channel_1",
        ... content …
    },
    {   "name": "data_channel_2",
        ... content ...
    }
],
"domains": [ ... ],
"parameter_points": [ ... ]
```





RooFit HS^3: new HistFactory model-building

- ROOT ships with the **HistFactory** package:
  - higher-level interface to build binned RooFit pdfs from template histograms with systematic uncerts.

- In HS^3, this high-level pdf is called `histfactory_dist`
- Code to read `histfactory_dist` from JSON is a complete rewrite of the HistFactory model building
  - Could replace existing way to specify HistFactory models via XML files or C++ code

- The structure of the `histfactory_dist` is inspired by the **pyhf** format for easier interoperability
  - See this pyhf example for comparison

HistFactory models can **easily be combined with other pdfs** in the RooFit HS^3 framework!

```
"distributions": [
        {
        "name": "model_channel1",
        "axes": [
        {
                "name": "obs_x_channel1",
                "max": 2.0,
                "min": 1.0,
                "nbins": 2
        }
        ],
        "samples": [
        {
                "data": {
                "contents": [20, 10]
                },
                "modifiers": [
                {
                        "data": {
                        "hi": 1.05,
                        "lo": 0.95
                        },
                        "name": "syst1",
                        "type": "normsys"
                },
                {
                        "name": "mu",
                        "type": "normfactor"
                }
                ],
                "name": "signal"
        }
        ... background templates ...
        ],
        "type": "histfactory_dist"
        }
]
```

Abbreviated model from the rf515 tutorial

HS3 references and news

- Most recent presentation by Carsten
- take a look at the overleaf or GitHub to view the current draft or follow the discussions!

- Have received a substantial set of comments from CMS, working to implement them and formulate replies
- Have received suggestions from Mikhail on amplitude models for hadron physics, available online for discussion
- ATLAS close to publishing first results based on HS3 v0.2
- Looking forward to prepare draft v0.3 before summer

Making HS3 in RooFit a priority

- Should we make the JSON format the default serialization of RooFit at some point?

  Which problems would it solve or introduce?
- Suggestion: improve test coverage by roundtripping every model in every ROOT unit test

Possible future developments

RooFit comparative benchmark suite

The RooFit part of the ROOT 2024 plan of work:

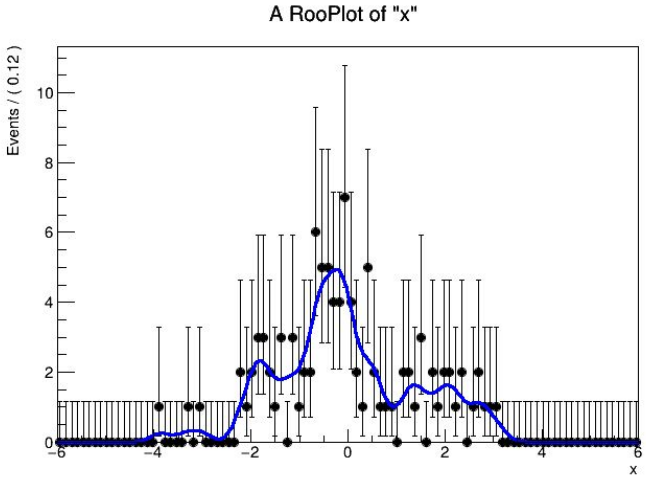
		priority
RooFit	Workshop with Experiments: promote features, gather input, speedup integration of RooFit in the existing sw setups	1
	Numeric integrals in n-dim with CUDA	1
	Evaluation of custom user functions in CUDA	1
	Group similar PDFs to speed up evaluation	1
	Make the new vectorized CPU likelihood evaluation interface the default	1
	Reduce JITting time for AD in RooFit	1
	PyROOT: express RooStats configuration with C++-oriented Set* as kwargs	2
	Integration of Fumili in RooFit	2

- The CUDA points can be worked on by other ROOT core team members too
- Reduce JITing time for AD in RooFit is important for ICHEP 2024 presentation on RooFit AD
- “Group similar PDFs to speed up evaluation” means here to compactify HistFactory models

Python functions in RooFit

- It would be very nice if we can use Python functions in the RooFit world, like this:

```
data = gauss.generate(x, 100)

samples = data.to_numpy()["x"]

from sklearn.neighbors import KernelDensity

kde = KernelDensity(kernel="gaussian",
                    bandwidth=0.2).fit(samples.reshape(-1, 1))

def func(x):
   return np.exp(kde.score_samples([[x]]))

kde_pdf = make_pdf_instance("kde_pdf", "kde_pdf", func, [x])

c1 = ROOT.TCanvas()
frame = x.frame()
data.plotOn(frame)
kde_pdf.plotOn(frame)
```



How was this possible?

Virtual dispatching from C++ to Python with cppyy technology!

Shoutout to **Vanya** who pioneered this idea in his ostap framework

```
class PyPdf : public RooAbsPdf {
public:
  PyPdf(const char *name, const char *title, RooArgList
&varlist)
     : RooAbsPdf(name, title), m_varlist("!varlist",
"All variables(list)" , this) {
     m_varlist.addTyped <RooAbsReal>(varlist);
  }
  // copy  constructor
  PyPdf(const PyPdf &right, const char *name = nullptr)
     : RooAbsPdf(right, name), m_varlist("!varlist",
this, right.m_varlist) {}

  PyPdf *clone(const char *name) const override { return
new PyPdf(*this, name); }

  const RooArgList &varlist() const { return m_varlist;
}

  // the actual evaluation of function ( will be
redefiend in python) !
  Double_t evaluate() const override { return 1; }

protected:
  RooListProxy m_varlist;
};
```

```
def make_pdf_instance(name, title, func, variables):

   class MyPdf(ROOT.PyPdf):

       def __init__(self, name, title, variables):
           super(MyPdf, self).__init__(name, title,
ROOT.RooArgList(variables))

       def evaluate(self):
           variables = self.varlist()
           return func(*(v.getVal() for v in self.varlist()))

       def clone(self, newname=False):
           cl = MyPdf(newname if newname else self.GetName(),
self.GetTitle(), self.varlist())
           ROOT.SetOwnership(cl, False)
           return cl

   return MyPdf(name, title, variables)
```

Using RooFit functions in Python

- In principle possible already, just not very pythonic
- Also here, pythonizations in the `RooFit::Evaluator` could help
- Benchmark to know if your interface is good:
  - RooFit NLL can be used in context of other Python libraries (e.g. scipy minimizers)
  - Minimizing RooFit likelihoods with iminuit would be a nice proof of concept
- I suggest to make this possible for ROOT 6.34

RooFit and SymPy

- Presentation at PyHEP 2023 on using SymPy expressions in RooFit
- This can help with amplitude analysis using AmpForm, which generates SymPy objects

```
import sympy as sp

x, mu = sp.symbols('x mu')

gauss = sp.exp(-0.5*(x - mu)**2)

c_code = sp.ccode(gauss)

ws = ROOT.RooWorkspace()
ws.factory(f"CEXPR::gauss('{c_code}', x[0., 10.], mu[5., 0., 10.])")
```

RooFit comparative benchmark suite

- Summer student project from 2023 (report, also in GitLab repository)
- Work didn't continue after that project
- One of the resulting plots (HistFactory benchmarks)

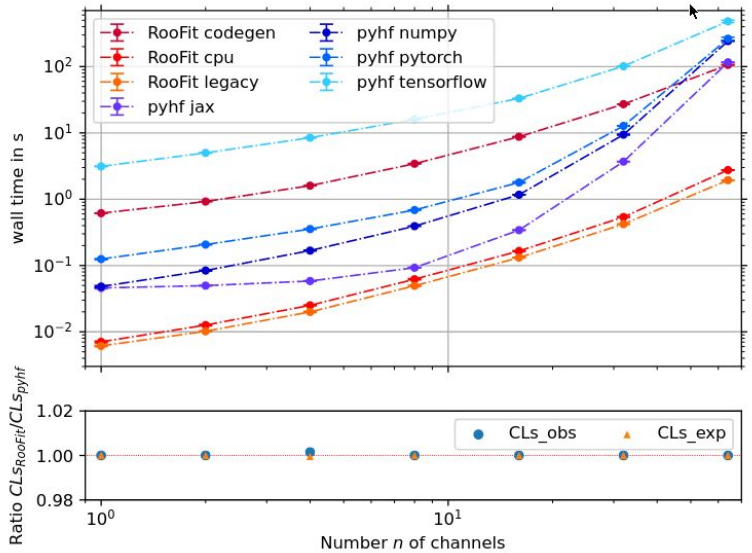


Figure 5: Benchmark of HistFactory implementations for different number n of channels with 4 bins per channel.

Memory safe interfaces

- Starting with 6.32, you can define the `ROOFIT_MEMORY_SAFE_INTERFACES` macro restrict RooFit to memory safe interfaces (see release notes)
- This coverage of this feature will gradually expand and hopefully make memory leaks impossible in the future
- Maybe make memory safe mode the default by ROOT 7?

New training material

- Existing RooFit tutorials are more “atomic”, they each show a feature but don’t necessarily give an overview
- I started to put together new Jupyter notebooks for giving an overview on RooFit
- We planned in ROOT to record ourselves presenting such tutorials and publish the videos on the internet
- Worth pursuing this further for RooFit with more specific tutorials?

Ideas already discussed on the workshop

- Speeding up Hessian evaluation
- Upstream discrete profiling and analytical BB-lite from HiggsCombine
- Gather examples of non-converging fits with explanations, also becoming tests in ROOT
- Better support for chi-square fitting, on equal footing with NLL
- Also add documentation for more niche features of RooFit
- Better support for toys
- Better plotting functionality
- Improve minimizer documentation
- etc.

Conclusions

- RooFit is constantly evolving, but can evolve in many different directions
- We need prioritizing
- Let's jump to the RooFit planning google doc to do that!