Comparative benchmarks for statistical analysis frameworks in Python

Daniel Werner

ROOT-Team, CERN

supervised by: Jonas Rembser, Lorenzo Moneta

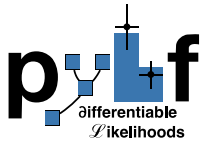
2023-10-11

PyHEP 2023 - "Python in HEP" Users Workshop

Overview

- Statistical analysis frameworks
- Likelihood evaluation
- Benchmark:
  - HistFactory benchmark
  - Unbinned benchmark
- Comparing user experiences
- Summary

Statistical Analysis frameworks



- Python implementation of HistFactory
- Multiple available backends
- High-level interface

- Fitting library mostly focused on unbinned fits
- Based on tensorflow

RooFit

- C++ library for statistical analyses in ROOT
- Both unbinned and binned fits supported
- Different backends for likelihood evaluation

RooFit Likelihood evaluation

```
EvalBackend('legacy')
```

- Outer loop over events (No vectorization)
- Caching of intermediate results for faster numerical gradients

For more information see [1] (CPU), [2] (CUDA), [3] (Clad), [4] (Codegen).

RooFit Likelihood evaluation

`EvalBackend('legacy')`

- Outer loop over events (No vectorization)
- Caching of intermediate results for faster numerical gradients

`EvalBackend('cpu')`

- Each mathematical operation performed in a vectorizable loop
- Hardware dependent auto-vectorization

For more information see [1] (CPU), [2] (CUDA), [3] (Clad), [4] (Codegen).

RooFit Likelihood evaluation

`EvalBackend('legacy')`

- Outer loop over events (No vectorization)
- Caching of intermediate results for faster numerical gradients

`EvalBackend('cpu')`

- Each mathematical operation performed in a vectorizable loop
- Hardware dependent auto-vectorization

`EvalBackend('cuda')`

- Launch cuda kernels to compute likelihoods
- CPU is used in parallel

For more information see [1] (CPU), [2] (CUDA), [3] (Clad), [4] (Codegen).

RooFit Likelihood evaluation

`EvalBackend('legacy')`

- Outer loop over events (No vectorization)
- Caching of intermediate results for faster numerical gradients

`EvalBackend('cuda')`

- Launch cuda kernels to compute likelihoods
- CPU is used in parallel

`EvalBackend('cpu')`

- Each mathematical operation performed in a vectorizable loop
- Hardware dependent auto-vectorization

`EvalBackend('codegen')`

- Minimal C++ code for likelihood generated and compiled on-the-fly
- Automatic differentiation with CLAD
- no caching of intermediate results

For more information see [1] (CPU), [2] (CUDA), [3] (Clad), [4] (Codegen).

pyhf Likelihood evaluation

- Values saved in arrays
- Efficient element-wise operations

pyhf Likelihood evaluation

- Values saved in arrays
- Efficient element-wise operations

(2015)

- Auto-diff with GradientTape
- Accelerated Linear Algebra (XLA)

pyhf Likelihood evaluation

- Values saved in arrays
- Efficient element-wise operations

(2016)

- NumPy-like arrays
- Automatic differentiation with Autograd

(2015)

- Auto-diff with GradientTape
- Accelerated Linear Algebra (XLA)

pyhf Likelihood evaluation

- Values saved in arrays
- Efficient element-wise operations

(2016)

- NumPy-like arrays
- Automatic differentiation with Autograd

(2015)

- Auto-diff with GradientTape
- Accelerated Linear Algebra (XLA)

(2022)

- Updated Autograd version
- XLA optimises code at compilation
- Auto vectorization through vmap

HistFactory benchmark

- Exclusion limit setting through asymptotic formulae [5]
- Likelihood generated by different backends
- Fits performed with `MINUIT2` [6], [7]
  - Strategy is `0 - fast`
  - Tolerance is `0.1`
  - Hesse matrix for errors is not needed
- Same number of minimizer calls

$$L\left(n \,|\, \mu, \theta\right) = \prod_{j=1}^{N} \underbrace{\frac{\left(\mu s_j + b_j(\theta)\right)^{n_j}}{n_j!} e^{-\left(\mu s_j + b_j(\theta)\right)}}_{\text{bkg sample and signal with strength } \mu} \cdot \underbrace{\frac{b_j^{n_j}(\theta)}{n_j!} e^{-b_j(\theta)}}_{\substack{\text{stat. uncertainty} \\ \text{of bkg in each bin}}}$$

$$\mathrm{CL}_s(\mu) = \frac{\int_{\tilde{q}_{\mu,\mathrm{obs}}}^{\infty} f\left(\tilde{q}_\mu | \mu\right) \mathrm{d}\tilde{q}_\mu}{\int_{\tilde{q}_{\mu,\mathrm{obs}}}^{\infty} f\left(\tilde{q}_\mu | 0\right) \mathrm{d}\tilde{q}_\mu}$$

HistFactory benchmark

Using CPU Intel Core i7-6700 (4-Core)

- Multithreading with `jax` (2 cores) and `pytorch` (4 cores)
- Larger overhead when automatic differentiation is used
- `'cpu'` faster than `'legacy'` for many bins
- `jax` is fastest for many bins

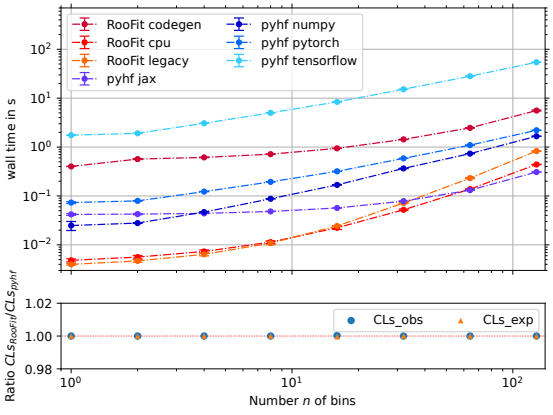


Figure: Benchmarking results for RooFit and pyhf with n bins for one channel.

HistFactory benchmark

- pyhf very slow for many channels
  - Python loop over channels
- behaviour between `'cpu'` and `'legacy'` unaffected by number of channels

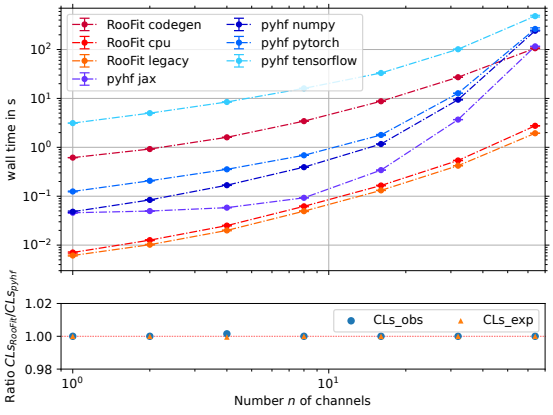


Figure: Benchmark of RooFit and pyhf with n channels for four bins per channel.

Unbinned benchmark

- Analytical function describes data and model
- Fits performed with `MINUIT2` [6], [7]
  - Strategy is `0 - fast`
  - Tolerance is `1`
  - Hesse matrix is calculated for better error estimation
- Same number of minimizer calls
- Computation graph can be optimized by `tensorflow`
- Gradient can be provided by `tensorflow` (`'grad'`) or `MINUIT2` (`'no_grad'`)

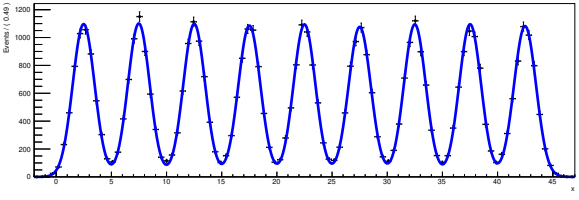


Figure: Fit of 9 shifted Gaussian distributions to data.

Unbinned benchmark CPU

Using CPU AMD Ryzen 9 3900
(no multithreading)

- Backends with AD have larger overhead
- `tensorflow` gradient slightly faster
- "Eager mode" without graph optimization much slower
- `'codegen'` beats `'legacy'`
- `'cpu'` much better than `'legacy'`
- Identical linear behaviour for many events

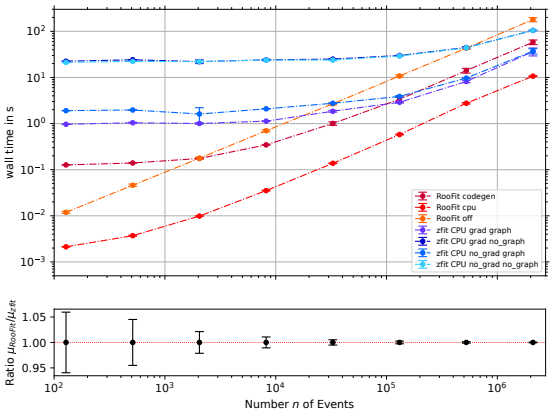


Figure: Benchmarking results for RooFit and zfit with n events and 18 model parameters performed on a CPU.

Unbinned benchmark GPU

Using GPU NVIDIA RTX A4500

- Little overhead for `'cuda'` backend
- `'cuda'` backend can handle $\approx 30\,000$ events at once on this device

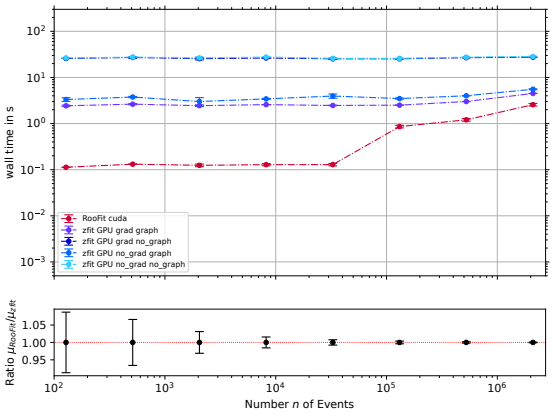


Figure: Benchmarking results for RooFit and zfit with n events and 18 model parameters performed on a GPU.

Summary

See the code in https://gitlab.cern.ch/dawerner/statanabenchmark

Outlook

- More models (e.g. custom pdfs with unknown analytical integrals)
- More frameworks (e.g. GooFit)
- Automize regular benchmarks through GitHub Actions

Results

- First comparative benchmarks between these frameworks
- Detailed validation of minimization for fair comparison

Thank you for your attention

References I

[1] S. Hageboeck and L. Moneta. "Making RooFit Ready for Run 3". In: *J. Phys. Conf. Ser.* 1525 (2020), p. 012114. DOI: 10.1088/1742-6596/1525/1/012114.

[2] Emmanouil Michalainas, Jonas Rembser, Stephan Hageboeck, et al. "Acceleration with GPUs and other RooFit news". In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023), p. 012066. DOI: 10.1088/1742-6596/2438/1/012066. URL: https://dx.doi.org/10.1088/1742-6596/2438/1/012066.

[3] V Vassilev, M Vassilev, A Penev, et al. "Clad — Automatic Differentiation Using Clang and LLVM". In: *Journal of Physics: Conference Series* 608.1 (Apr. 2015), p. 012055. DOI: 10.1088/1742-6596/608/1/012055. URL: https://dx.doi.org/10.1088/1742-6596/608/1/012055.

[4] Garima Singh, Jonas Rembser, Lorenzo Moneta, et al. "Making Likelihood Calculations Fast: Automatic Differentiation Applied to RooFit". In: *26th International Conference on Computing in High Energy & Nuclear Physics*. **unpublished**.

[5] Glen Cowan, Kyle Cranmer, Eilam Gross, et al. "Asymptotic formulae for likelihood-based tests of new physics". In: *Eur. Phys. J. C* 71 (2011). [Erratum: Eur.Phys.J.C 73, 2501 (2013)], p. 1554. DOI: 10.1140/epjc/s10052-011-1554-0.

References II

[6] F. James and M. Roos. "Minuit: A System for Function Minimization and Analysis of the Parameter Errors and Correlations". In: *Comput. Phys. Commun.* 10 (1975), pp. 343–367. DOI: 10.1016/0010-4655(75)90039-9.

[7] M. Hatlo, F. James, P. Mato, et al. "Developments of mathematical software libraries for the LHC experiments". In: *IEEE Trans. Nucl. Sci.* 52 (2005), pp. 2818–2822. DOI: 10.1109/TNS.2005.860152.

[8] ATLAS Collaboration. *Measurements of the Higgs boson inclusive, differential and production cross sections in the 4ℓ decay channel at $\sqrt{s} = 13$ TeV with the ATLAS detector*. ATLAS-CONF-2019-025. 2019. URL: https://cds.cern.ch/record/2682107.

Backup

Test statistics

Likelihood function: $L(\mu, \theta)$ $\longrightarrow$ Profile Likelihood: $\lambda(\mu) = \frac{L\left(\mu, \hat{\hat{\theta}}\right)}{L\left(\hat{\mu}, \hat{\theta}\right)}$

Test statistic: $q_\mu = \begin{cases} -2\log\left(\lambda(\mu)\right) & \hat{\mu} \leq \mu \\ 0 & \hat{\mu} > \mu \end{cases}$

Improved test statistic: $\tilde{q}_\mu = \begin{cases} -2\log\left(\frac{L\left(\mu, \hat{\hat{\theta}}\right)}{L\left(0, \hat{\theta}\right)}\right) & \hat{\mu} < 0 \\ -2\log\left(\lambda(\mu)\right) & 0 \leq \hat{\mu} \leq \mu \\ 0 & \hat{\mu} > \mu \end{cases}$

$p_\mu = \int_{q_{\mu,\text{obs}}}^{\infty} f\left(q_\mu | \mu\right) \mathrm{d}q_\mu$

Exclusion limits

$$\mathrm{CL}_s(\mu) = \frac{\int_{\tilde{q}_{\mu,\mathrm{obs}}}^{\infty} f\left(\tilde{q}_\mu|\mu\right) \mathrm{d}\tilde{q}_\mu}{\int_{\tilde{q}_{\mu,\mathrm{obs}}}^{\infty} f\left(\tilde{q}_\mu|0\right) \mathrm{d}\tilde{q}_\mu} = \frac{\alpha}{1-\beta}$$

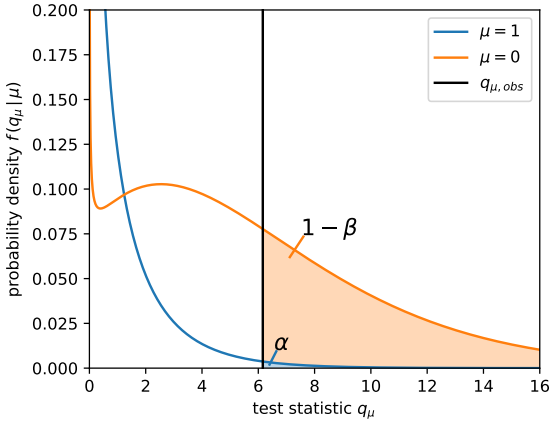


Figure: Visualization of the distributions used in setting exclusion limits.

Computation graph

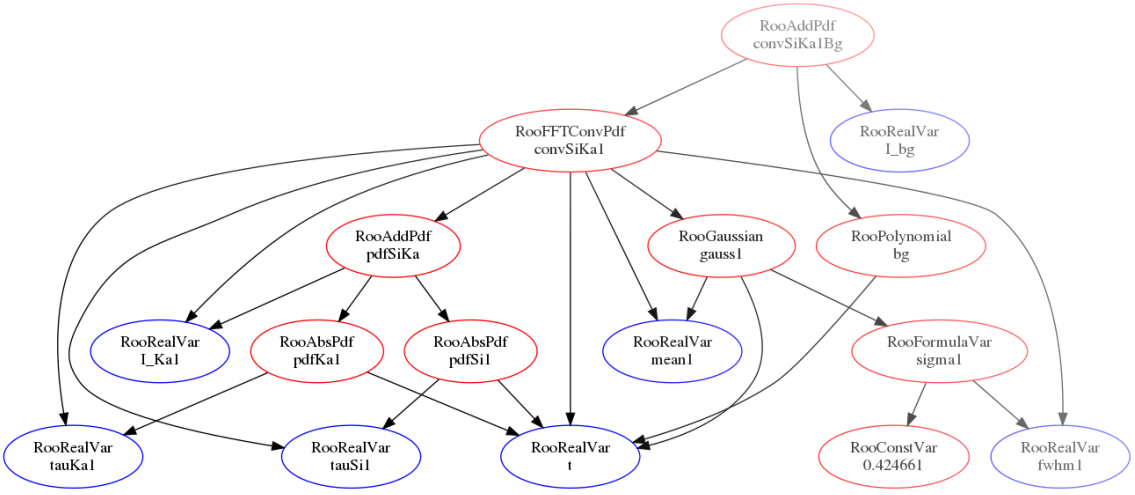


Figure: Example computation graph of a RooFit model.

Automatic differentiation

Chain rule:

$$f(l,r) = f(g(l,r)) \Rightarrow \frac{\partial f}{\partial r} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial r}$$

- Elementary functions in code
- Derivatives known

Forward mode:

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}}\frac{\partial w_{i-1}}{\partial x}$$

Backward mode:

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}}\frac{\partial w_{i+1}}{\partial w_i}$$

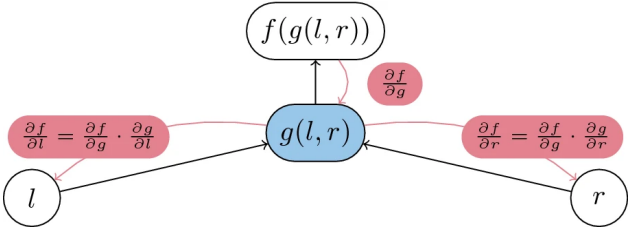


Figure: Example graph of automatic differentiation [MaxEmanuel, CC BY-SA 4.0, via Wikimedia Commons].

Template Histograms

$$L\left(n, m \,|\, \mu, \theta\right) = \prod_{j=1}^{N} \frac{\left(\mu s_j + b_j\right)^{n_j}}{n_j!} e^{-(\mu s_j + b_j)} \cdot \prod_{k=1}^{M} \frac{u_k^{m_k}}{m_k!} e^{-u_k}$$

- Poisson distributed bin contents
- No analytical function needed
- Signal strength allows for easy exclusion

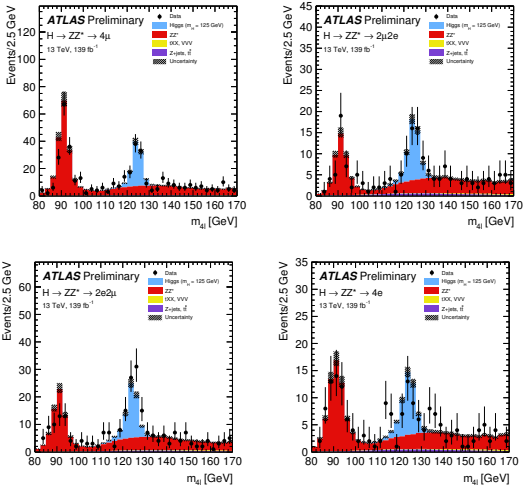


Figure: Template histograms from different Higgs discovery channels in an ATLAS analysis [8].

Benchmark implementation

```python
import pytest

backend_list = ["legacy", "cpu", "numpy", "jax"]
backend_ids = ["RooFit_legacy", "RooFit_cpu",
               "pyhf_numpy", "pyhf_jax"]

@pytest.mark.parametrize("backend",
                         backend_list,
                         ids=backend_ids)
def test_hypo_test(benchmark, backend, n_events):
    set_backend(backend)
    model = get_model(n_events)
    data = get_data(n_events)
    result = benchmark(hypo_test, data, model)
```

Using `pytest-benchmark` for benchmark

- Arguments can be parameterised
- `benchmark` fixture measures performance
- Preparations are not benchmarked

HistFactory benchmark 2 channels

- almost identical to behaviour with 1 channel
- `jax` is fastest for ≈ 32 bins

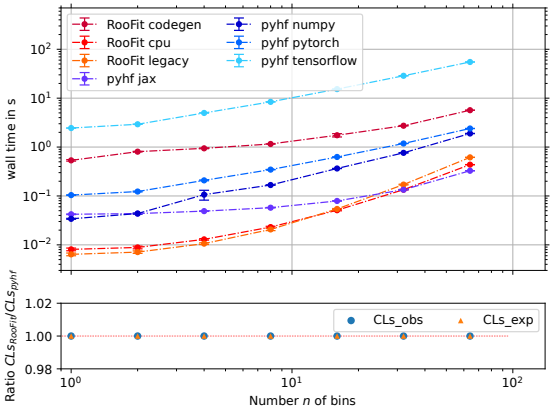


Figure: Benchmarking results for RooFit and pyhf with n bins for two channels.

Unbinned benchmark 2 parameter Gaussian

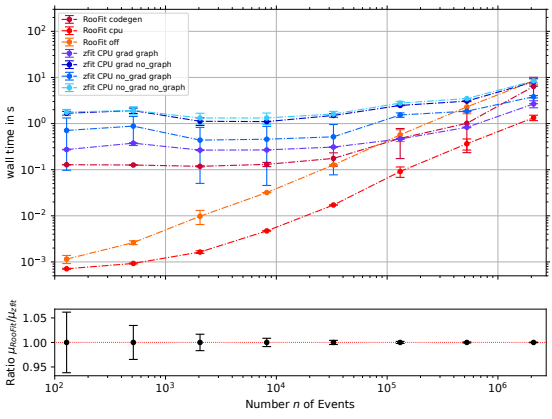


Figure: Benchmarking results for RooFit and zfit with n events and 2 model parameters performed on a CPU.

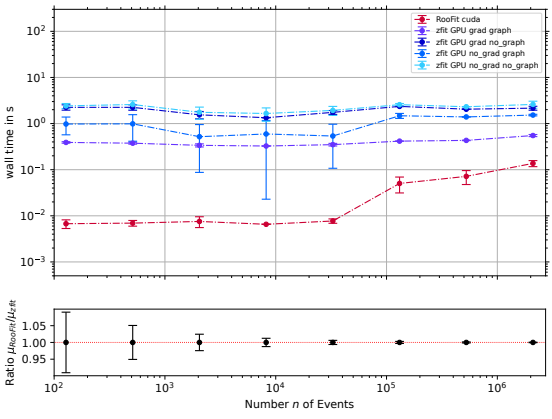


Figure: Benchmarking results for RooFit and zfit with n events and 2 model parameters performed on a GPU.