Future Analysis
According to ROOT

Lorenzo Moneta
(CERN/EP-SFT)

12/7/2022
Outline

- Analysis landscape:
  - I/O developments
    - RNTuple
  - data exploration, declarative analysis
    - RDataFrame
  - machine learning, interoperability and analysis integration
    - TMVA/SOFIE
  - statistical analysis and modelling
    - RooFit
- What we have in ROOT now
- Planned future developments

A lot of material presented coming from recent ICHEP talks
Introduction

- ROOT is an open source, community driven project
- Despite existing for many years ROOT is still very active
  - high number of contributors/month
- ROOT is used by all HEP experiments
  - more than 1EB of data stored in ROOT format
ROOT Core Components

- I/O: reading and writing data efficiently
- Analysis interfaces: histograms and RDataFrame
- Interactivity: C++ interpreter and Python bindings
- Math libraries: PRNG, numerical algorithms (Minuit)
- Statistics and modeling: fitting
- Graphics: scientific visualization
- Machine learning: model evaluations and interoperability

Will show the major recent developments in some of these components and the future plans
RNTuple

- Successor of TTree
  - columnar storage of event data optimised for selective reads
- Schedule for production for HL-LHC
- Based on 25+ years of TTree experience,
- Redesigned I/O subsystem providing:
  - Less disk and CPU usage for the same data content
  - 25% smaller files, ×2–5 better single-core performance
  - 10 GB/s per box and 1 GB/s per core sustained end-to-end throughput
- And also:
  - Systematic use of exceptions to prevent silent I/O errors
  - Efficient support of modern hardware (built for multi-threading and async I/O)
  - Native support for object stores
RNTuple Performances: I/O Size

Size on disk, CMS Higgs4Leptons (84 branches)
RNTuple Performances: I/O Speed

CMS Higgs4Leptons (10/84 branches)

LHCb B2HHH (10/26 branches)
RNTuple Plans

- Smaller files and significantly faster reads compared to TTree
- Modern and robust API
- Capable of making efficient use of modern devices and storage systems (such as SSD, object stores, many cores)
- RNTuple is work in progress in ROOT::Experimental
- The on-disk format is still subject to small changes!
- We are happy to get your feedback!
HEP Analysis Landscape

Analysis life cycle
- skimming, ntuple production
- quick exploration, first implementation
- systematics, scale out
- statistical analysis

Platforms
- laptop or PC
- many-core machine
- computing cluster + job submission

Analysis languages
- ~50% C++
- ~50% Python

Storage
- local disk
- fast-access network storage
- EOS or other not-so-fast backend
**RDataFrame**

A Swiss army knife for data analysis

**Analysis life cycle**
- skimming, ntuple production
- quick exploration, first implementation
- systematics, scale out

**Platforms**
- laptop or PC
- many-core machine
- computing cluster + job submission

**Storage**
- local disk
- fast-access network storage
- EOS or other not-so-fast backend

**ROOT.RDataFrame** is a modern analysis interface that addresses all these use cases with **one high-level programming** model that performs well, scales well and enables **HEP-specific ergonomics**, in C++ and Python.
RDataFrame Code

Example of RDF code in Python

```python
df = ROOT.RDataFrame(dataset)  # on this (ROOT, CSV, ...) dataset
df = df.Filter("x > 0")  # only accept events for which x > 0
    .Define("r2", "x*x + y*y")  # define r2 = x^2 + y^2
rHist = df.Histo1D("r2")  # plot r2 for events that pass the cut
df.Snapshot("newtree", "out.root")  # write the skimmed data and r2 to a new ROOT file
```

Physicists describe analysis ingredients, ROOT handles technicalities.

Users can inject **arbitrary code** at all steps, which makes this relatively simple API extremely versatile.

---

from E. Guiraud: RDF@ICHEP 2022
Swtich to multi-thread execution (Python):

```python
ROOT.EnableImplicitMT()  # Run a multi-thread event loop on this (ROOT, CSV, ...) dataset
df = ROOT.RDataFrame(dataset).Filter("x > 0")  # only accept events for which x > 0
define r2 = x^2 + y^2
```
```
rHist = df.Histo1D("r2")  # plot r2 for events that pass the cut
```
```
df.Snapshot("newtree", "out.root")  # write the skimmed data and r2 to a new ROOT file
```

from E. Guiraud: RDF@ICHEP 2022
Switch to distributed execution (Python):

```python
cluster = dask_jobqueue.HTCondorCluster(n_workers=64)
df = RDataFrame(dataset, daskclient=Client(cluster))

df = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")

rHist = df.Histo1D("r2")
df.Snapshot("newtree", "out.root")
```

Available since v6.26 (experimental)  Also see this tutorial, the docs, the recent ATTF talk
Distributed RDF

- Enables **interactive large-scale distributed** data analysis
- Python RDF API, C++ event loop
- Full access to ROOT I/O
- Let **Spark/Dask/HTCondor/Slurm/SSH**…. take care of scheduling and resource management
- Transiently merges results coming from different computing nodes

Distributed RDF benchmark (dimuon, 4000x data)

See [this talk](#) from E. Guiraud: RDF@ICHEP 2022
RDF Performances

- RDataFrame enables fast turnaround for complex analysis use cases
- RDataFrame scales well to many cores, many nodes, many histograms
- Independent study shows RDataFrame analysis to be significantly faster
- Performance is always ongoing work: we are constantly looking for feedback/use cases

Study by ETH Zuerich + IRIS-HEP
https://arxiv.org/abs/2104.12615

from E. Guiraud: RDF@ICHEP 2022
## RDF Benchmarks

### Fully compiled C++ RDataFrame

<table>
<thead>
<tr>
<th>query, 1x data (s), 10x data (s)</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.37, 1.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.46, 3.70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.73, 6.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.65, 5.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.84, 7.45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.08, 27.99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.56, 22.27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.17, 10.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Coffea 0.7.12 (using chunksize=2**19)

<table>
<thead>
<tr>
<th>query, 1x data (s), 10x data (s)</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.40, 4.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.51, 5.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.81, 7.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.65, 6.58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.41, 12.43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.89, 124.59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.19, 29.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.27, 17.70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- note that these benchmarks are not representative of large analysis workloads
- see also [this ACAT talk](https://example.com/acat-talk) by Nick Smith

Benchmark from [github.com/nsmith-/coffea-benchmarks](https://github.com/nsmith-/coffea-benchmarks)

Setup: AMD EPYC 7702P, using 48 physical cores, data read from filesystem cache
RDF: Working with Collections

Select and fill: quick one-liner

```cpp
RVecD selectPt(RVecD &pt, RVecD &eta) {
    return pt[abs(eta) < 2];
}

auto h = df.Define("pt", selectPt,
    {"muon_pt", "muon_eta"}).Histo1D("pt");
```

Compiled C++

Python+Numba

```python
def select_pt(muon_pt, muon_eta):
    return muon_pt[np.abs(muon_eta) < 2]

h = df.Define("pt", select_pt).Histo1D("pt")
```

See docs about injecting Python into RDF in v6.26.
Systematic Variations

Variations automatically propagate to selections, derived quantities and results. **Multi-thread** and **distributed** execution **just works**. Only needed quantities are re-computed, all in **one event loop**.

```python
nominal_hx = df.Vary("pt", "RVecD\{pt*0.9, pt*1.1\}", ["down", "up"])
  .Filter("pt > k")
  .Define("x", someFunc, ["pt"])
  .Histo1D("x")

hx["nominal"].Draw()  # obtain all variations
hx["pt:down"].Draw("SAME")
```
NumPy Interoperability

- **TTree → NumPy** via RDataFrame

  ```python
  cols = df.Filter("x > 10").AsNumpy(["x", "y"])
  ```

- **NumPy → RDataFrame**

  ```python
  data = {"x": np.array(...) , "y": np.array(...) , ...}
  df = ROOT.RDF.MakeNumpyDataFrame(data)
  ```

- Work in progress: RDF ↔ Awkward arrays, see [here](#)

- Working on developing direct batch generator of NumPy arrays for training of ML models (see later)
Summary RDF

- and many more features, e.g.
- transparent support for RNTuple, with no code changes
- machine learning inference as part of the event loop
- SOFIE, RBDT
- RDF keeps evolving
- cooperation with HEP community
- Critical to focus on the right features
- need your feedback and help

from E. Guiraud: RDF@ICHEP 2022
**SOFIE: Fast Inference in ROOT**

- New ROOT tool for inference code generation for DL models
- Input, a trained ML model:
  - ONNX (new standard for ML)
  - TF/Keras
  - PyTorch
- Output:
  - C++ code
  - Minimal dependency (BLAS)
  - Can be compiled on the fly (e.g. with Cling)

```cpp
using namespace TMVA::Experimental::SOFIE;
RModelParser_ONNX parser;
RModel model = parser.Parse("Model.onnx");
// generate text code internally (with some options)
model.Generate();
// write output header file and data weight file
model.OutputGenerated();
```
SOFIE with RDataFrame

- Model evaluation with SOFIE can be integrated in RDF event loop
- SofieFunctor: adapter for using SOFIE within RDF
- support for multi-threaded evaluation using RDF slots
- see full Example tutorial code in C++ or Python

```cpp
auto h1 = df.DefineSlot("DNN_Value", 
  SofieFunctor<7,TMVASOFIE_higgs_model_dense::Session>(nslots), 
  {"m_jj", "m_jjj", "m_lv", "m_jlv", "m_bb", "m_wbb", "m_wwbb"}). 
  Histo1D("DNN_Value");
```
# Supported ONNX Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptron: Gemm</td>
<td>Implemented and integrated (ROOT 6.26)</td>
</tr>
<tr>
<td>Activations: Relu, Seul, Sigmoid, Softmax, LeakyRelu</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td>Convolution (1D, 2D and 3D)</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td>Recurrent: RNN, GRU, LSTM</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td>BatchNormalization</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td>Pooling: MaxPool, AveragePool, GlobalAverage</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td><strong>Layer operations:</strong> Add, Sum, Mul, Div, Reshape, Flatten, Transpose, Squeeze, Unsqueeze, Slice, Concat, Identity</td>
<td>Implemented and integrated</td>
</tr>
<tr>
<td>InstanceNorm</td>
<td>Implemented but to be integrated (PR #8885)</td>
</tr>
<tr>
<td>Deconvolution, Reduce operators (for generic layer normalisation), Gather (for embedding) and more…</td>
<td>Planned for next release</td>
</tr>
<tr>
<td>???</td>
<td>Depending on user needs</td>
</tr>
</tbody>
</table>
SOFIE Benchmarks

Comparison with ONNXRuntime and LWTNN
- 2-3 faster than ONNXRuntime for DNN with batch size=1
- 20% faster for RNN operators
- slower for CNN (20% for 2D, x2 for 3D)

Time per event for different batch size, cache flushed

- Lightweight Trained Neural Network (lwt_nn)
- ONNX Runtime
- SOFIE (netlib blas)
- SOFIE (OpenBLAS)

10 Dense layers (50x50)

Using RDF
5 Dense layers
(200x200)

Batch size=1

Ubuntu 20.04 Intel 5000MHz

Larger = Better

Processed Events/sec

DNN Model(5 layers of 200)

90
80
70
60
50
40
30
20
10
0

Time per event/μs

Batchsize

Smaller = Better

Batch size=1

using different models (DNN, CNN, RNN)

Larger = Better
Fast Decision Tree Inference

- Inference engine taking model parameters from externally trained models
- Features:
  - Simple to use from Python and C++
  - Thread-safe
  - Zero-copy
  - Fast for single event and batch inference
- External training and model conversion

Python application

```python
xgb = xgboost.BDTClassifier(options)
xgb.fit(x, y)
ROOT.TMVA.SaveXGBoost(xgb, "myBDT", "model.root")
```

C++ application

```cpp
TMVA::RBDT bdt("myBDT", "model.root");
auto y1 = bdt.Compute({1.0, ...});
auto x = TMVA::RTensor<float>(data, shape);
auto y2 = bdt.Compute(x);
```
Interoperability of TMVA

- Working on Pythonization of TMVA interfaces
  - moving from string API to python keyword args
- Possible training of Python ML within TMVA workflow
  - interfaces to scikit-learn, Keras and PyTorch
- Working on generic data-loader for ML workflows
  - Generator doing batching and shuffling from ROOT files on the fly
  - Allows for training on huge datasets
  - Direct feeding of data from disk to GPU

Example of a possible ML workflow loading batches

```python
df = ROOT.RDataFrame("Events", "http://file.root")
generator = TMVA.BatchGenerator(df, cols, batchSize)
for step in gradientSteps:
    x = generator()
    model.fit(x)
```
Physics analyses continuously generate increasingly complex likelihood models to describe their data

- Higgs combination fits, EFT interpretations
  - O(1000) parameters
  - O(100) likelihood components
  - O(100) datasets

- Only one tool capable of handling such models: RooFit
- Recent development give possibility to bring down fitting time from hours to minutes
  - from a work day to a coffee break!
RooFit Evolution

RooFit is evolving on many different areas:

- Vectorization
- Gradient parallelization
- Fit precision and correctness
- Higher-level interfaces
- Pythonizations
- Interoperability
- GPU Implementation
- Automatic differentiation
- Targeted optimizations for expensive workflows
- Performance optimization
- Testing and benchmarking
- User interface and experience

from Z. Wolfs: RooFit@ICHEP 2022
Gradient parallelization

- Parallelize at gradient calculation level
  - N parameters: $\sim O(2N)$ function evaluations for computing numerical derivatives
  - Line search: serial part $\sim O(3)$ function evaluation
- Dynamic load balancing over workers through random work stealing algorithm
- Complexity of derivative calculation varies
- Designed to have maximum speed impact of complex fits with many parameters

Wall time decrease in Higgs combination fit:
- from 2h12m26s → 28m52s
  (~4/5 times faster)
- Result validated: all parameters agreed with serial fit within 1% of their uncertainties!
Vectorisation and GPU Computation

- Batch evaluation of RooFit models:
  - allow code vectorisation
  - optionally allow multi-threading or GPU computation (with CUDA)
  - possible due to a re-structure of RooFit computational graph.
  - from event-by event to batch node evaluation

⇒ Additional speed-up obtained in Batch mode by better CPU caching and vectorisation!
RooFit Pythonizations

- PyROOT bindings more pythonic in latest ROOT (6.26)
- Now you can:
  - use Python keyword arguments instead of RooFit command arguments
  - pass around Python sets or lists instead of RooArgSet or RooArgList
  - pass Python dictionaries instead of std::map<> 
  - implicitly convert floats to RooConstVar in RooArgList/Set constructors
- All Pythonizations are documented in the reference guide

Example code from a RooFit tutorial

```python
# 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)
```

from J. Rembser@ACAT 2021
Interoperability with NumPy

- New **converters** between **NumPy** arrays/ **Pandas** dataframes and **RooDataSet**/ **RooDataHist**:  
  - RooDataSet.from_numpy()
  - RooDataSet.to_numpy()
  - RooDataSet.from_pandas()
  - RooDataSet.to_pandas()
  - RooDataHist.from_numpy()
  - RooDataHist.to_numpy()

- New **RooRealVar.bins()** function to get RooFit bin boundaries as NumPy array

```python
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()
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0978065</td>
<td>-1</td>
</tr>
<tr>
<td>7.211196</td>
<td>-1</td>
</tr>
<tr>
<td>3.198218</td>
<td>1</td>
</tr>
<tr>
<td>5.015824</td>
<td>1</td>
</tr>
<tr>
<td>7.782386</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>95</td>
<td>-1</td>
</tr>
<tr>
<td>0.475860</td>
<td>1</td>
</tr>
<tr>
<td>4.451101</td>
<td>-1</td>
</tr>
<tr>
<td>3.481015</td>
<td>-1</td>
</tr>
<tr>
<td>4.010105</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

from J. Rembser@ACAT 2021
Auto-differentiation (AD)

- AD available in ROOT `TFormula` for fitting
- Large speedup can be obtained with respect to using numerical differentiation in case of large number of parameters
  - ~10 speedup for 100 parameters fit
- Integrating also AD for Hessian
  - need to add support in Minuit

- Working on integrating AD in RooFit
  - starting with a prototype implementation for HistFactory models
Summary

- Future analyses in HEP require tools that are
  - efficient
  - with simple interfaces
  - reliable and supported
  - interoperability (e.g. with Python ecosystem)
- ROOT is evolving, large effort from the whole team
  - aiming to fulfil requirements of future HEP analysis
- Need your help for continuous feedback and support
References

- root.cern
- https://cern.ch/forum
- https://github.com/root-project

For more information of current developments see presentations at latest ROOT Users Workshop
see also the ICHEP 22 presentations in the Computing session