ROOT Lecture

Jonas Rembser

CERN (EP-SFT)

July 20, 2023

Acknowledgments

This lecture and slides was inspired by the following lectures in the past:

- The lecture and exercises from the HASCO 2018 by Steven Schramm
- The lecture and exercises from the HASCO 2021 by Antonio Sidoti
- The ROOT Summer Student Course at CERN from 2023

This year, we will also cover an additional new ROOT feature: the **RDataFrame**.

About me:

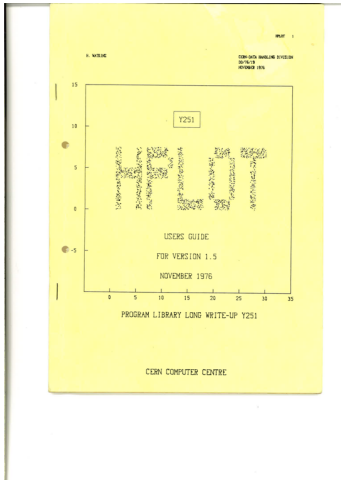
- I did a PhD working with the CMS collaboration, graduated in 2020
- Right after, I joined the ROOT team at CERN to work on support and development of the **RooFit** package

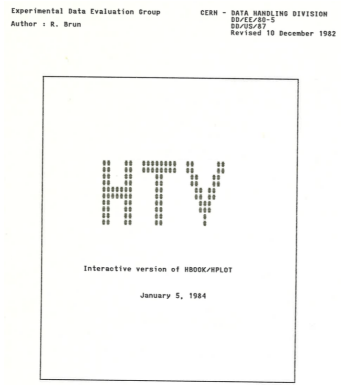
Introduction

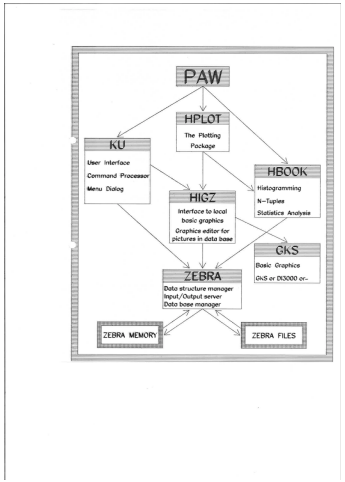
Introduction

- **Analysis** and **visualization** are fundamental in particle physics (experimental and not only)
- ROOT is the primary tool for data analysis in **high energy physics** (not only collider physics)
- Maybe not the reference tool if you are doing detector R&D, theory/phenomenology, machine learning $\Rightarrow$ Useful to have an idea on how to use it since at some point you will use it

History







HPLOT: ≈mid 70s HTV: ≈early 80s PAW: ≈1985

For a nice presentation on ROOT history and development take a look at [this CERN Data Science Seminar talk] by **Rene Brun** (includes also a recording).

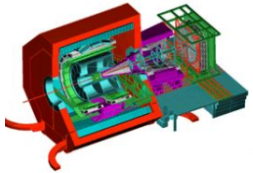
ROOT in a nutshell

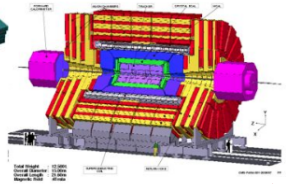
ROOT can be seen as a collection of building blocks, like:

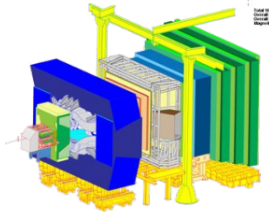
- **Data analysis: histograms, graphs, functions**
- **I/O: row-wise, column-wise** storage of any C++ object
- **Statistical tools** (RooFit/RooStats): rich modeling and statistical inference
- Math: **non-trivial functions** (e.g. Erf, Bessel), optimized math functions
- **C++ interpretation**: full language compliance
- **Multivariate Analysis** (TMVA): e.g. Boosted decision trees, neural networks
- **Advanced graphics** (2D, 3D, event display)
- **Declarative Analysis**: RDataFrame
- And more: HTTP server, JavaScript visualization

ROOT Application Domains

A selection of the experiments adopting ROOT







Data

Event Filtering

Analysis

Offline Processing

Event Selection, statistical treatment …

Further processing, skimming

Reconstruction

Raw

Reco

…

Analysis Formats

Plots

Data Storage: Local, Network

Resources

- ROOT website: https://root.cern/
- Training: https://github.com/root-project/training
- More material: https://root.cern/get_started/
  - Includes a booklet for beginners: the **"ROOT primer"**
- Reference guide: https://root.cern/doc/master/index.html
- Forum: https://root-forum.cern.ch/
- Tutorials: on the ROOT website

I encourage you to install ROOT yourself, just follow the install instructions on the ROOT website!

- It is easiest if you have an un-to-date **macOS** or **Linux OS**
- On Windows, we recommend the **Windows subsystem for Linux** (WSL)

If you find any bugs please help us by opening a GitHub issue!

The different ways to use ROOT

Ways to use ROOT

- The ROOT C++ command line interpreter
- C++ ROOT macros
- Compiled C++ using ROOT as a library
- As a library in Python code (**PyROOT**)

The ROOT C++ interpreter

By typing `root` in a terminal, you can fire up the ROOT C++ interpreter and execute arbitrary C++ statements

```
> root
root [0] int x = 4
(int) 4
root [1] int y = 5
(int) 5
root [2] x + y
(int) 9
root [3]
```

Special commands in the ROOT C++ interpreter

Special commands that are not C++ can be typed into the prompt, they start with a ".":

```
root [0] .<command>
```

For example:

- To quit ROOT, use **.q**
- To issue a shell command, use **.! <OS command>**
- **.help** or **.?** gives the full list

ROOT macros

Put a function with the same name as file in a **.C** file, for example **myMacro.C**:

```
void myMacro() {
   std::cout << "Hello World!" << std::endl;
}
```

You can now run the code when starting the ROOT session:

```
> root myMacro.C
```

Or you can run in from within a session:

```
> root
root [0] .x myMacro.C
```

Or you can load it and run the function later:

```
root [0] .L myMacro.C
root [1] myMacro();
```

Compiled C++ using ROOT as a library

If you have a **main** function in the file, you can also compile the code with g++ for example:

```
void myMacro() {
   std::cout << "Hello World!" << std::endl;
}

int main() {
    myMacro();
}
```

In the compilation command, you need to add the ROOT compiler flags:

```
> g++ -o myMacro myMacro.C `root-config --cflags --libs`
```

You can now run your code like any other executable in the shell:

```
> ./myMacro
```

As a library in Python code (also in Jupyter notebooks)

If you import ROOT in a Python session or Python script, you can access ROOT functionality via its Python bindings (more on this later)

```
import ROOT
```

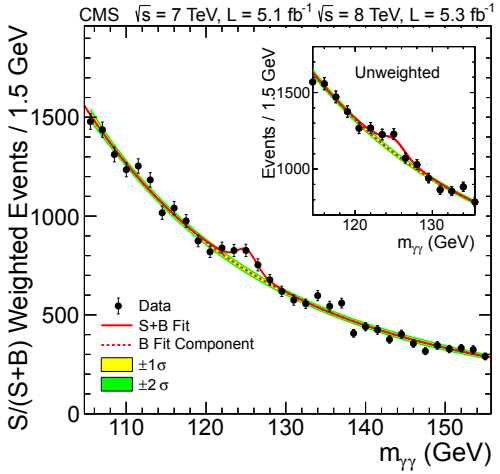
- You can also use ROOT in Jupyter notebooks like this
- If you want to run a notebook in the cloud, take a look at the SWAN analysis interface
- There is also a C++ kernel for the notebooks that comes with ROOT
- Notebooks are a great way to share and explain code to colleagues!
- However, for a large analysis project, it is better to **organize your code in Python scrips and modules!**
  - This makes your analysis more reproducible and more independent of the backend

Histograms, Graphs and Functions

Histograms

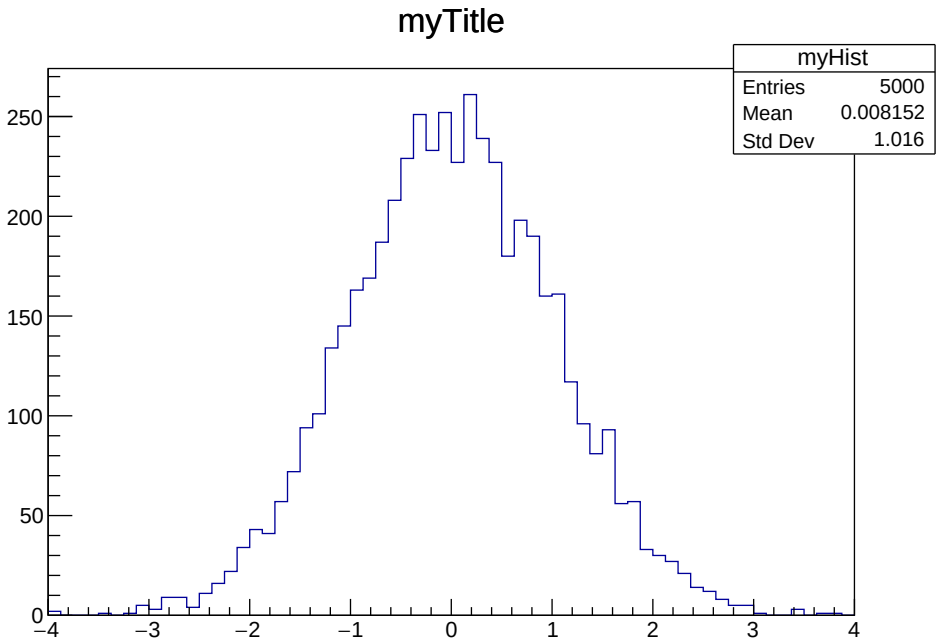
- A simple form of data reduction
  - Can have billions of collisions, the Physics displayed in a few histograms
  - It is like an empirical density estimator
  - Possible to calculate momenta: mean, rms, skewness, kurtosis
- Collect quantities in discrete categories, the bins

- ROOT Provides a rich set of histograms
  - In multiple dimensions: `TH{1,2,3}` classes + `THN`
  - Holding different precision types
    - `TH1D` is a one-dim histogram holding doubles
    - Have also:
      `TH{1,2,3}F` (float), `I` (int32), `S` (int16), `C` (int8)



My First Histogram

```
root [0] TH1D h("myHist", "myTitle", 64, -4.0, 4.0)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
```



My First Histogram (in a notebook)

Note| that in *Jupyter notebooks* (also in *SWAN*), the figure is not shown directly. You have to:

1. Either call `gPad->Draw()` at the end:

```
TH1D h("myHist", "myTitle", 64, -4.0, 4.0)
h.Draw()
gPad->Draw()
```

2. Or you can create a TCanvas and draw it:

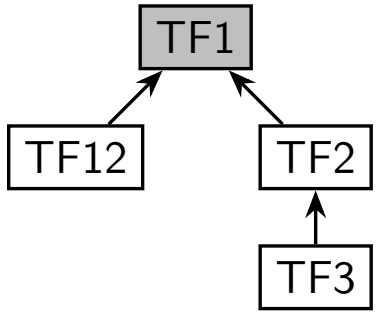
```
TCanvas c1;
TH1D h("myHist", "myTitle", 64, -4.0, 4.0);
h.Draw();
c1.Draw();
```

Functions

- Mathematical functions are represented by the **TF1** class
- They have names, formulas, line properties, can be evaluated as well as their integrals and derivatives
  - Numerical techniques for generic cases
  - Automatic differentiation can be used for derivatives

Option	Description
"SAME"	superimpose on top of existing picture
"L"	connect all computed points with a straight line
"C"	connect all computed points with a straight curve
"FC"	draw a fill area below a smooth curve

From the TGraphPainter documentation.



Functions

Can describe functions as:

- Mathematical formulas (written as strings)
- C++ functions/functors/lambdas
  - Implement your highly performant custom function
- Python functions
  - Fast prototyping on the Python side
- With and without parameters
  - Crucial for fits and parameter estimation

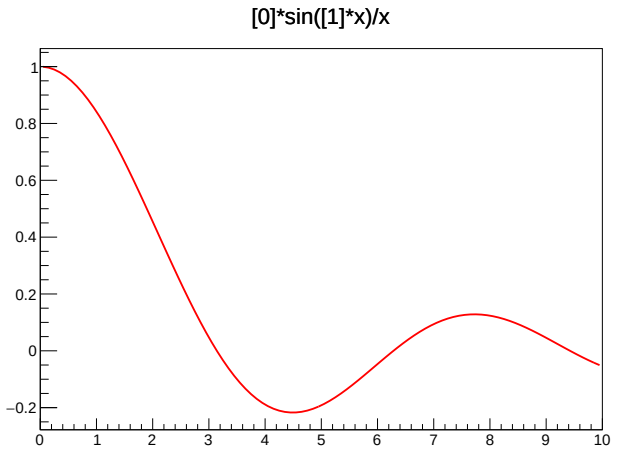
ROOT as a Function Plotter

The class TF1 represents one-dimensional functions (e.g., $f(x)$):

```
root [0] TF1 f1("f1","sin(x)/x",0.,10.); // name, formula, min, max
root [1] f1.Draw();
```

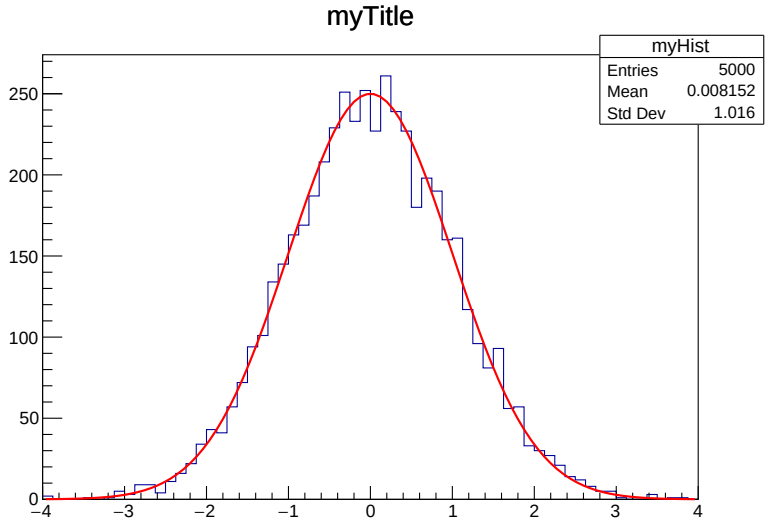
An extended version of this example is the definition of a function with parameters:

```
root [2] TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
root [3] f2.SetParameters(1,1);
root [4] f2.Draw();
```



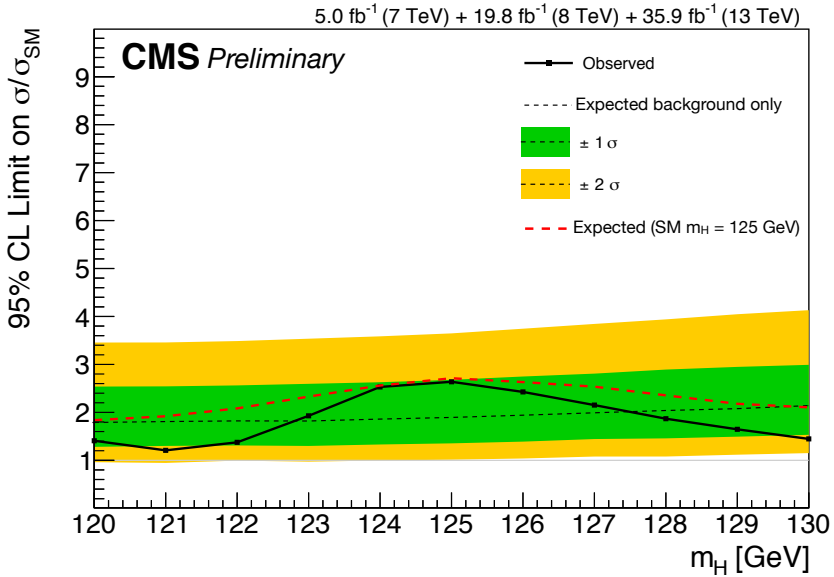
Another example

```
root [0] TH1D h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
root [3] TF1 f("g", "gaus", -8, 8)
root [4] f.SetParameters(250, 0, 1)
root [5] f.Draw("Same")
```



Graphs

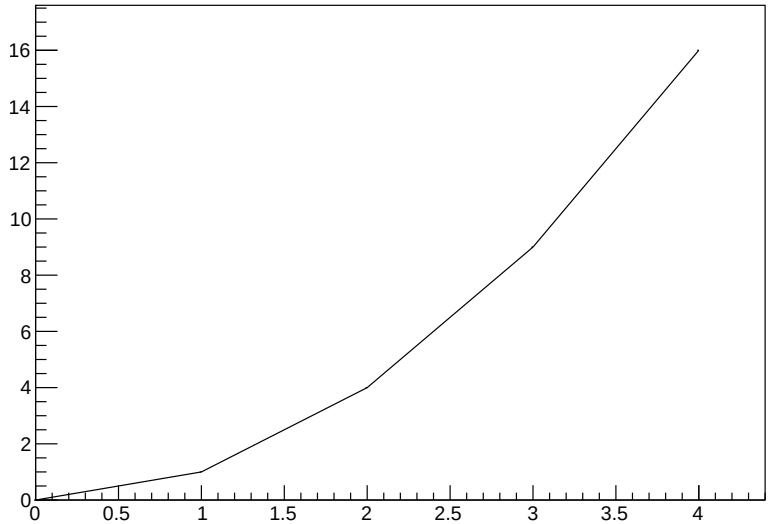
1. Display points and errors
2. Not possible to calculate momenta
3. Not a data reduction mechanism
4. **Fundamental to display trends**
5. Focus on TGraph and TGraphErrors classes in this course



From [CMS-PAS-HIG-17-019](CMS-PAS-HIG-17-019).

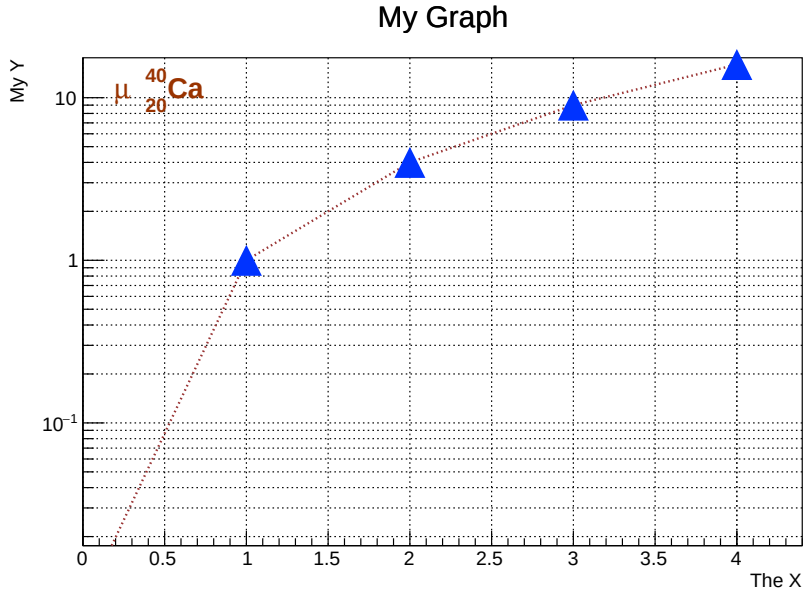
My first graph

```
root [0] TGraph g;
root [1] for (auto i : {0,1,2,3,4}) g.SetPoint(i,i,i*i)
root [2] g.Draw("APL")
```



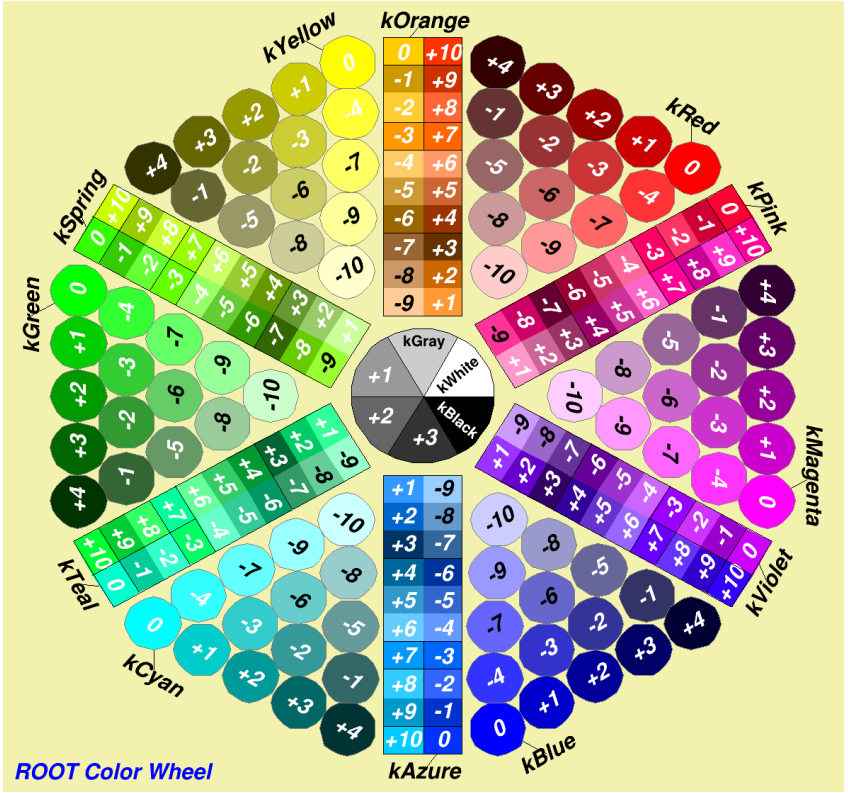
Styling your graph

```
g.SetMarkerStyle(kFullTriangleUp);
g.SetMarkerSize(3);
g.SetMarkerColor(kAzure);
g.SetLineColor(kRed - 2);
g.SetLineWidth(2);
g.SetLineStyle(3);
g.SetTitle("My Graph;The X;My Y");
gPad->SetGrid();
auto txt = "#color[804]{#mu {}^{40}_{20}Ca}";
TLatex l(.2, 10, txt);
l.Draw();
gPad->SetLogy();
```



See also the TAttMarker documentation for details on the marker styles like `kFullTriangleUp` used in this example.

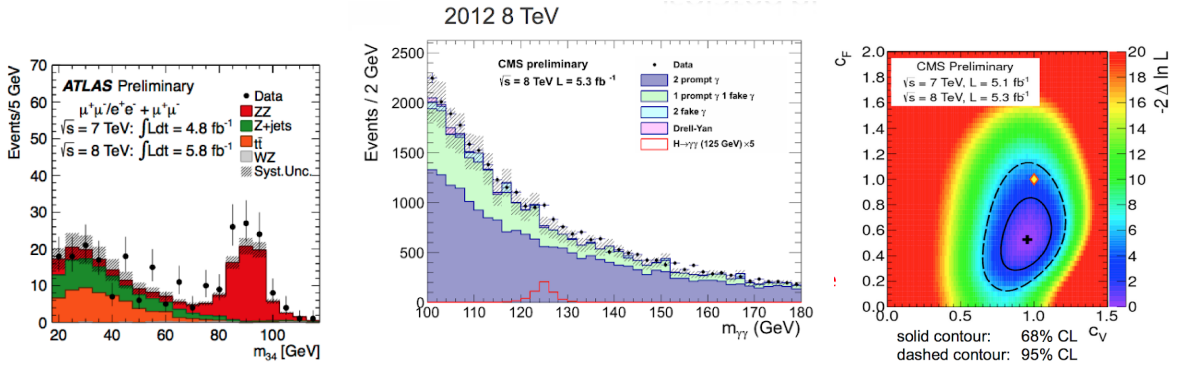
The Color Wheel



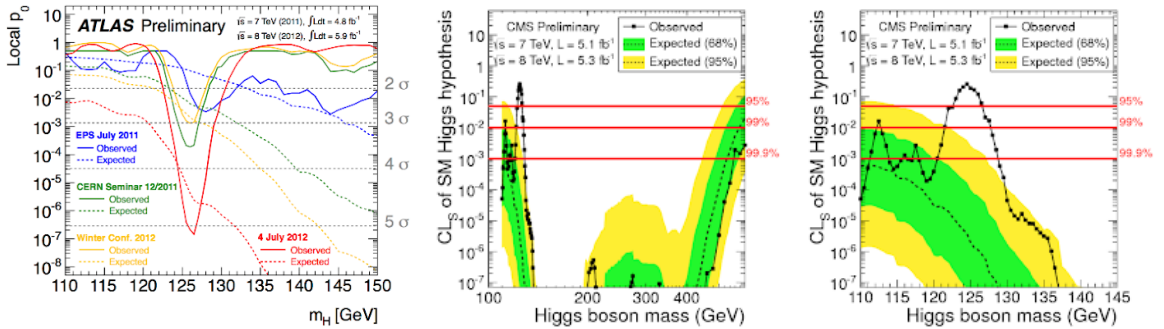
From the TColor documentation.

Drawing Options Documentation

- See the documentation of the THistPainterClass for **histogram** drawing options



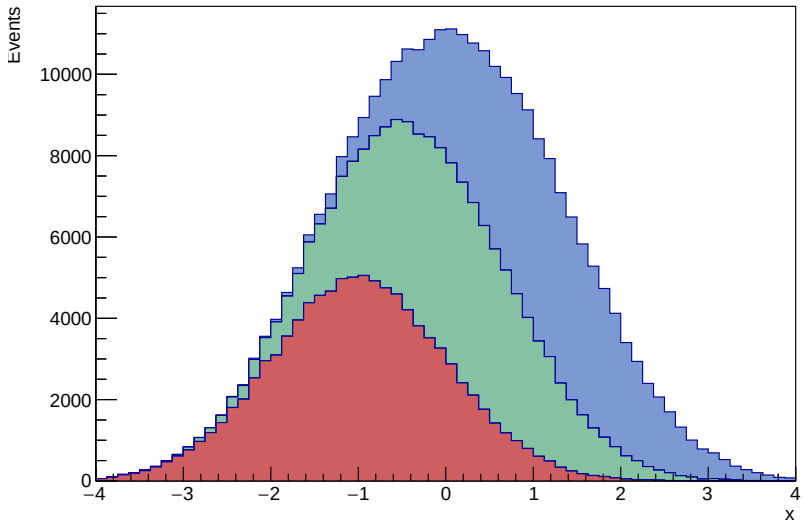
- See the documentation of TGraphPainter for the **graph** drawing options



Example: stacked histograms

In high energy physics, we often plot stacked histograms, for example to show the contributions of different processes. This can be doe with the THStack.

```
TF1 f1{"f1", "gaus", -4.0, 4.0};

TH1D h1("h1", "x", 64, -4.0, 4.0);
TH1D h2("h2", "x", 64, -4.0, 4.0);
TH1D h3("h3", "x", 64, -4.0, 4.0);

THStack hs("hs","");
hs.SetTitle(";x;Events");

std::vector<TH1D*> histos{&h1, &h2, &h3};
std::vector<int> colors{46, 30, 38};

for(int i = 0; i < histos.size(); ++i) {
    TH1D & h = *histos[i];
    f1.SetParameters(1.0, i - 1, 1.0);
    h.FillRandom("f1", 100000);
    h.SetFillColor(colors[i]);
    hs.Add(&h);
}

hs.Draw();
```



PyROOT: The ROOT Python Bindings

PyROOT

- Python bindings for ROOT
- Access all the ROOT C++ functionality from Python
  - Benefit from C++ performance
- Dynamic, automatic
- "Pythonizations" for specific cases

In Python, you have also other great scientific and HEP-specific libraries that you **should also consider when they are the right tool for the job**!

- For example NumPy, Pandas, PyTorch, and Jax
- PyROOT is increasingly compatible with standard Python data structures like NumPy arrays

PyROOT spinoff: cppy

- ROOTs technology to generate Python bindings automatically from C++ became an acclaimed standalong project: **cppyy**
- Prime example for how ROOT does cutting-edge R & D also on the compiler and interpreter level

Video on cppyy in the C++ weekly channel.

Using PyROOT

Entry point to use ROOT from Python:

```
import ROOT
```

All the ROOT classes you have learned so far can be accessed from Python:

```
ROOT.TH1F
ROOT.TGraph
...
```

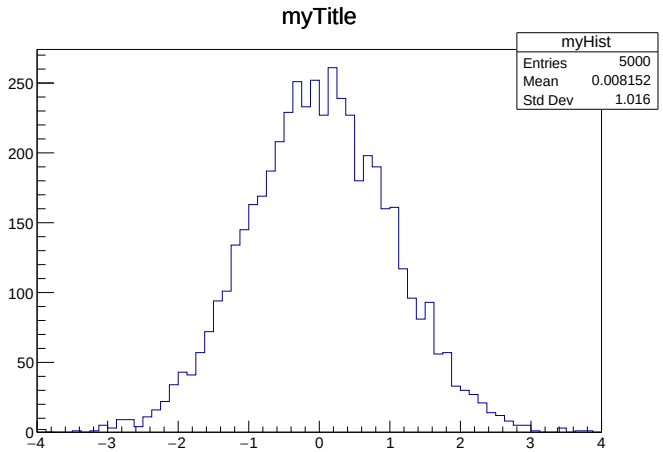
Example: C++ to Python

C++

```
> root
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
```

Python

```
> python
>>> import ROOT
>>> h = ROOT.TH1F("myHist", "myTitle", 64, -4, 4)
>>> h.FillRandom("gaus")
>>> h.Draw()
```



Note: you can also use individual imports:

```
>>> from ROOT import TH1F
```

Writing new C++ functions in PyROOT

Using `ROOT.gInterpreter.Declare()`, you can also define **new C++ functions**!

```
ROOT.gInterpreter.Declare("""
void myAdd(double a, double b) {
    return a + b;
}
""")
```

They are now in the ROOT module:

```
print(ROOT.myAdd(1.0, 2.0))
```

In a **Jupyter notebook**, you can also use the %%cpp magic command:

```
%%cpp
void myAdd(double a, double b) {
    return a + b;
}
```

```
print(ROOT.myAdd(1.0, 2.0))
```

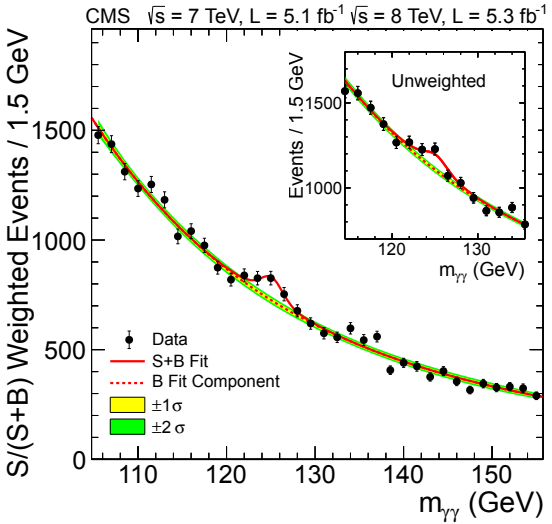
Now you can define performant C++ functions for example to:

- define fit functions
- transform event data (more on this later)

Parameter Estimation and Fitting

What is Fitting?

- Estimate parameters of a hypothetical distribution from the observed data distribution
  - $y = f(x|\theta)$ is the fit model function
- Find the best estimate of the parameters θ assuming $f(x|\theta)$
- Both Likelihood and Chi2 fitting are supported in ROOT



Example: Higgs $\rightarrow \gamma\gamma$ spectrum

We can fit for:

- the expected number of Higgs events
- the Higgs mass

Fitting in ROOT

- **Create first a parametric function object**, `TF1`, which represents our model
  - need to set the initial values of the function parameters.
- **Fit the data object** (Histogram or Graph):
  - Call the `Fit` method passing the function object
  - various options are possible (see the `TH1::Fit` documentation)
- **Examine result**:
  - get parameter values, uncertainties, correlation
  - get fit quality estimation
- The resulting fit function is also drawn automatically on top of the Histogram or the Graph when calling `TH1::Fit` or `TGraph::Fit`

Fitting Histograms

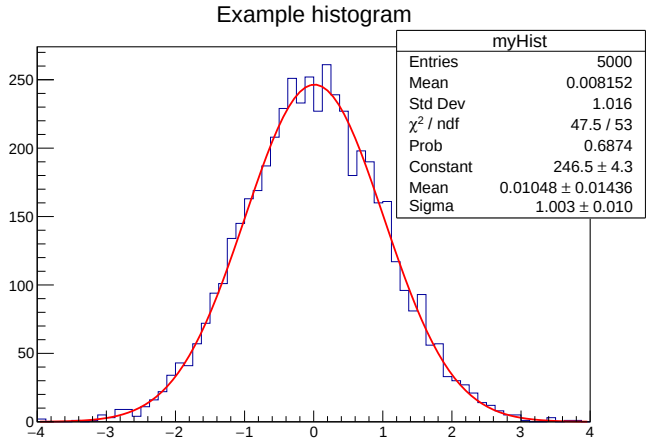
Create a histogram, h1, and we want to fit it:

```
root [0] TH1D h1("myHist", "myTitle", 64, -4.0, 4.0);
root [1] h1.FillRandom("gaus");
root [2] TF1 f1("f1","gaus");
root [3] h1.Fit(&f1);
```

```
FCN=27.2252 FROM MIGRAD    STATUS=CONVERGED      60 CALLS          61 TOTAL
                    EDM=1.12393e-07    STRATEGY= 1
ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  Constant     7.98760e+01   3.22882e+00   6.64363e-03  -1.55477e-05
   2  Mean        -1.12183e-02   3.16223e-02   8.18642e-05  -1.49026e-02
   3  Sigma        9.73840e-01   2.44738e-02   1.69250e-05  -5.41154e-03
```

For displaying the fit parameters:

```
root [4] gStyle->SetOptFit(1111);
```



Creating the Fit Function

How to create the parametric function object (**TF1**):

- We can write formula expressions using functions:

```
TF1 f1("f1","[0]*TMath::Gaus(x,[1],[2])");
```

  - we can use the available functions in ROOT library and stl
  - **[0],[1],[2] indicate the parameters.**
  - We could also use meaningful names, like [a],[mean],[sigma]

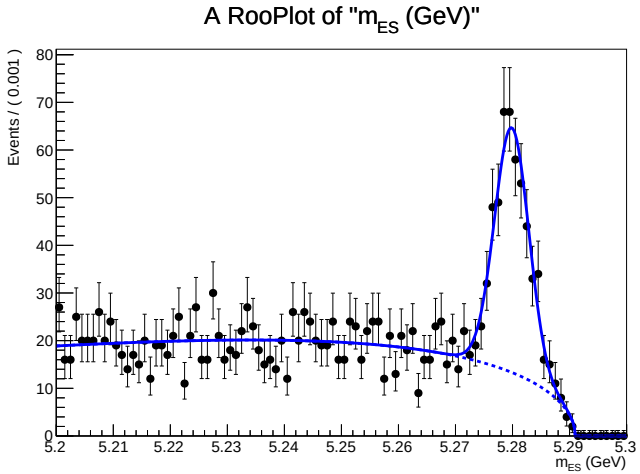
- There are pre-defined functions

```
TF1("f1","gaus");
```

- pre-defined functions available: *gaus*, *expo*, *landau*, *breitwigner*,*crystal_ball*,*pol{0,1..,N}*, *cheb{0,1,...10}*,*xygaus*,,*bigaus*
  - see full list in the documentation of TH1::Fit(), and also in the TFormula documentation

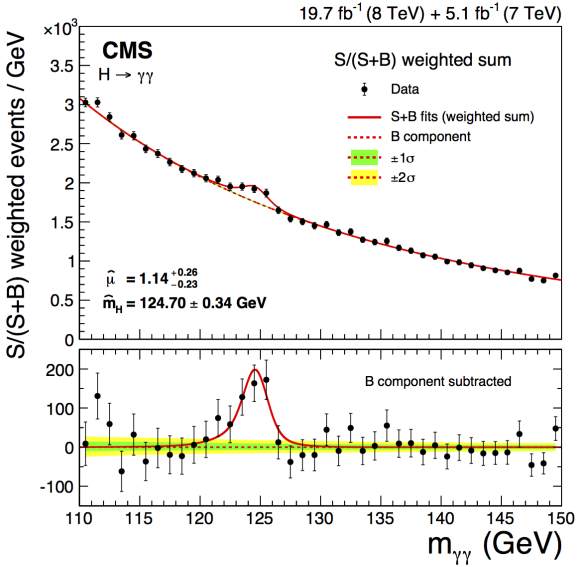
RooFit: ROOT toolkit for complex fitting

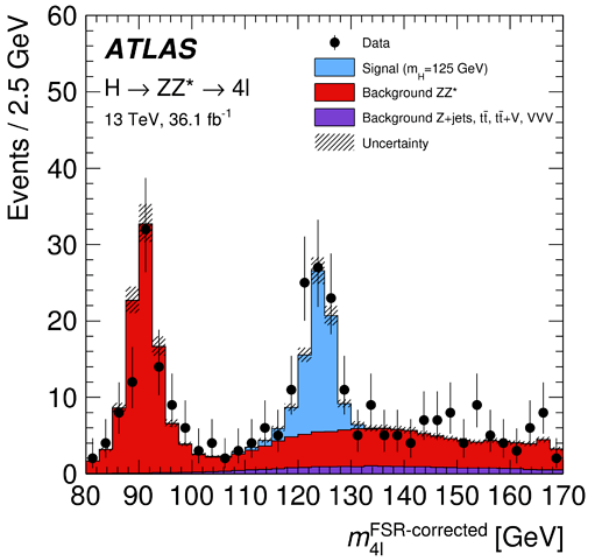
- ROOT fitting can handle complicated functions...
  - ...but requires much code when fitting complex models
- RooFit provides functionality for building fitting models
  - complex model building from standard components
    - composition with addition product and convolution
- Fitting often requires **normalization** of PDFs
  - not always trivial and RooFit does it automatically
- RooFit provides also
  - **MC data generation** from model
  - advanced **visualization** of fitting results
  - **simultaneous fit** to different data samples
  - full model description for **reusability**
  - **built-in optimization** for optimal computational performances
    - necessary for acceptable performance in complex fits



For more info, see the manual or the RooFit courses.

Likelihood fits in HEP





Unbinned likelihood fits

- often many events
- sums of PDFs of different types

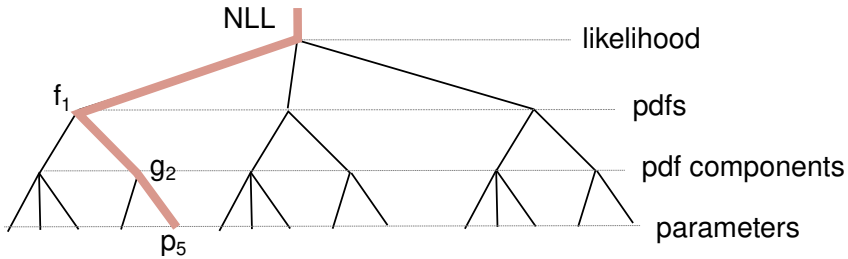
Binned likelihood fits

- 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

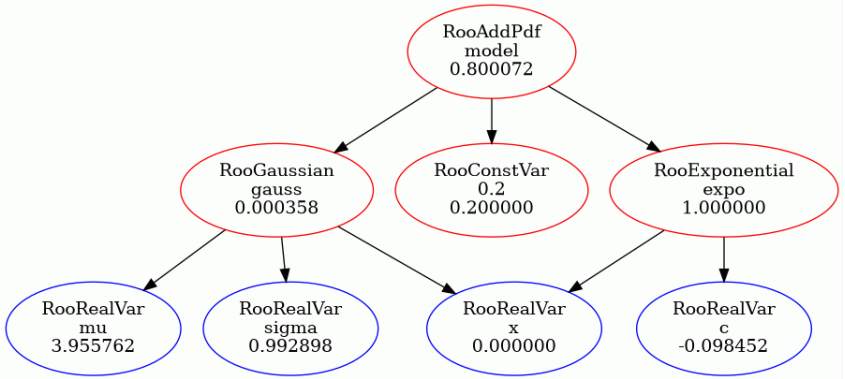


- Drastically decreases the cost of gradient evaluation

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:

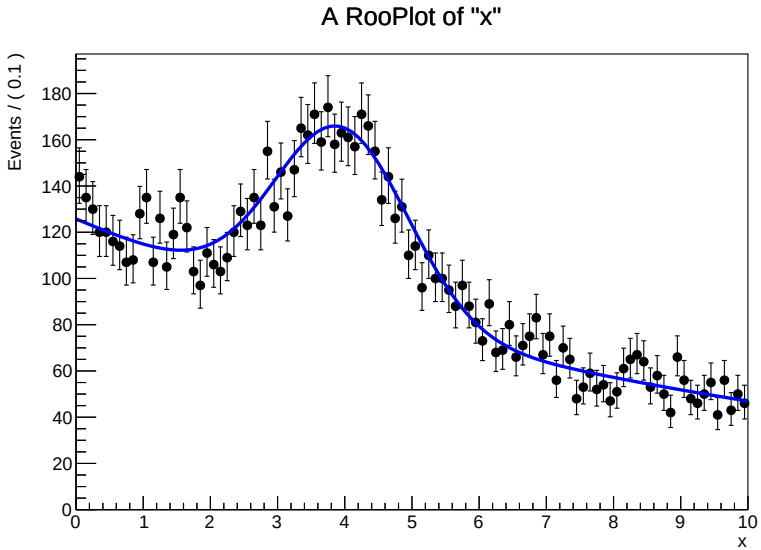
```
std::unique_ptr<RooDataSet> data{model.generate(x, 10000)};
```

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.

RooFit Pythonizations

Analyzers prefer Python:

- they want RooFit to be more **"pythonic"**
- they want interoperability with **NumPy** and **Pandas**

We deliver now:

- **Pythonizations** of functions and classes, e.g., take builtin Python objects as arguments
- RooFit dataset classes interoperable with **NumPy** and **Pandas**

```python
import ROOT

x = ROOT.RooRealVar("x", "x", -5, 5)
y = ROOT.RooRealVar("y", "y", -5, 5)
z = ROOT.RooRealVar("z", "z", -5, 5)

# 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])

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

df = data.to_pandas()
```

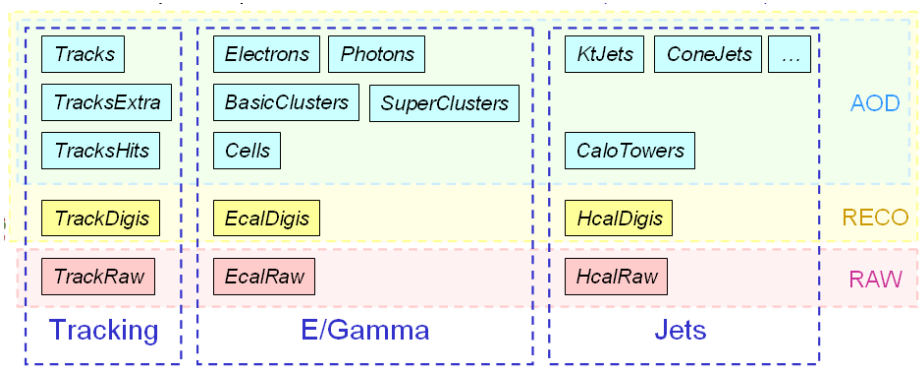
```
x          y         z
0     -2.365318  4.625480  3.836555
1     -3.884152 -0.374631 -0.421798
2     -4.859738  3.288175 -1.899461
3     -2.307831  2.966785  0.909296
4     -1.253818  3.671417  3.595242
...         ...       ...       ...
19995 -0.433255  2.272059 -4.670626
19996  4.141638  1.365243 -0.250328
19997 -1.394192  0.792205 -1.647825
19998 -4.075001  0.499360  0.730352
19999  4.977131  1.158074 -0.679951

[20000 rows x 3 columns]
```

Reading and Writing Data

The ROOT File

- With ROOT, you can write objects to files that are represented by **TFile** instances
- `.root` files are **binary** and can be compressed (transparently for the user)
- **TFiles are self-descriptive:**
  - The information how to retrieve objects from a file is stored with the objects

- ROOT files can contain simple tabluar data (aka. "n-tuples")
- It can also also contain **arbitrary custom C++ classes**



In case of future need: a tutorial on how to save your custom classes.

Example of the custom classes saved in the CMS reconstruction output ROOT files.

TFile in Action

```
TFile f("myfile.root", "RECREATE");
```

Option	Description
NEW or CREATE	Create a new file and open it for writing. If the file already exists, it is not opened.
RECREATE	Create a new file. If the file already exists, it will be overwritten.
UPDATE	Open an existing file for writing. If no file exists, it is created.
READ	Open an existing file for reading (default)

TFile in Action: Writing

```
TFile f("file.root", "RECREATE");
TH1F h("h", "h", 64, 0.0, 8.0);
h.Write("h");
f.Close();
```

- Write to a file
- Close the file and make sure the operation succeeded

```
> rootls -l file.root
TH1F  Jun 24 15:02 2022 h  "h"
```

TFile in Action: Reading

C++

```
TFile f("file.root");
TH1F* h = f.Get<TH1F>("h");
h->Draw();
```

Python

```
import ROOT
f = ROOT.TFile("file.root")
f.h.Draw();
```

Get the histogram as an **attribute** of the TFile instance! Possible only in Python.

Listing TFile Content

- **TBrowser**: interactive tool

```
root [0] TBrowser tb
```

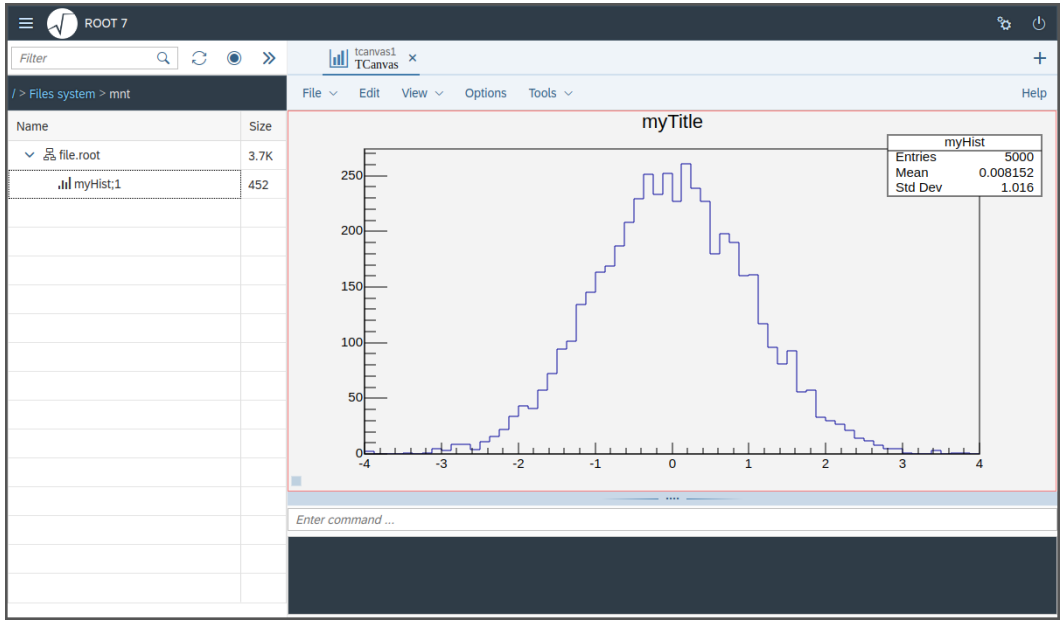
- **rootls** tool: list content

```
> rootls file.root
```

- **TFile::ls()**: prints content

```
> root file.root
root [0]
Attaching file /mnt/file.root as _file0...
(TFile *) 0x562cbf485d50
root [1] _file0->ls()
TFile**         /mnt/file.root
 TFile*         /mnt/file.root
  KEY: TH1D     myHist;1        myTitle
root [2] .q
```

  - great for interactive usage



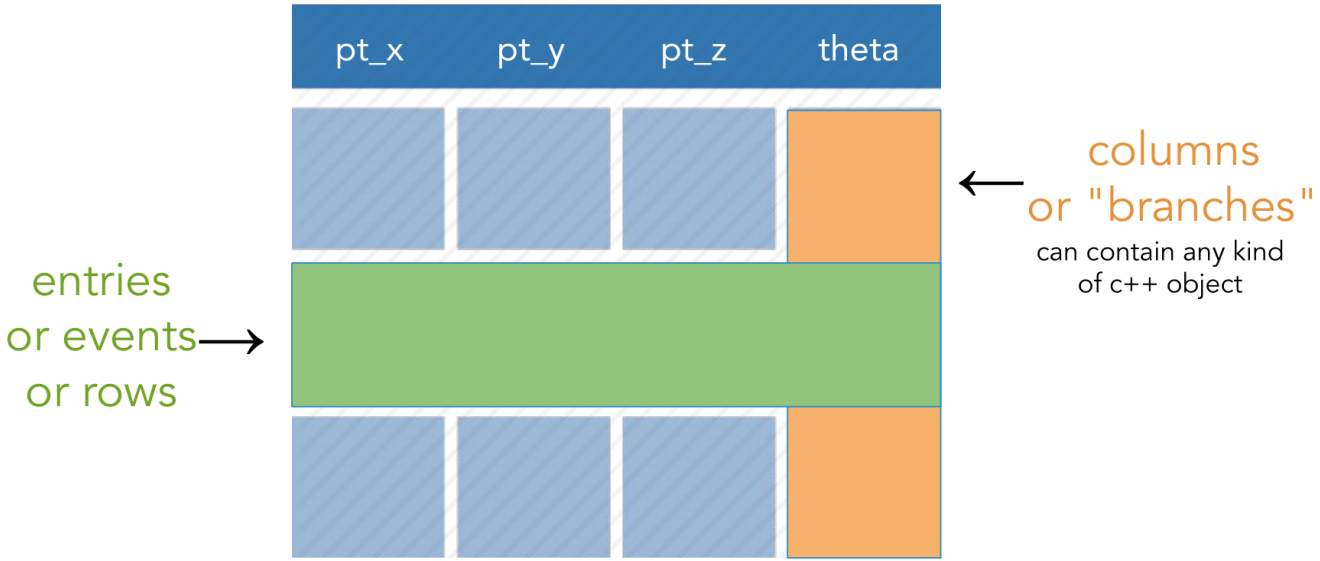
The new TBrowser in ROOT 6.26.

Note you can also open the file by passing it as a command line artument to `root`.

The ROOT Columnar Format and RDataFrame

Columns and Rows

- High Energy Physics: many statistically independent *collision events*
- Create an event class, serialize and write out N instances into a file?
  $\rightarrow$ No. Very inefficient!
- Organize the dataset in **columns**



The TTree

A columnar dataset in ROOT is represented by the TTree class:

- Also called *tree* columns also called *branches*
- Coloumns can contain different types
- **Supports any kind of object**
- One row per *entry* (or, in collider physics, *event*)

If just a **single number** per column is required, you can also use the simpler TNtuple class.

A modern and simple way to interact with ROOT datasets is to use RDataFrame.

- Low-level interfaces to deal with datasets do exists
- There are many older scripts around that directly interact with TTrees

RDataFrame: quick how-to

1. **Build a dataframe** object by specifying your dataset
2. Apply a series of **transformations** to your data
   - **filter** (e.g., apply some cuts)
   - define **new columns**
3. Apply **actions** to the transformed data to produce results (e.g., filling a histogram)

See the RDataFrame tutorials for a comprehensive feature demo.

Simple Code Example (Python)

```
rdf = ROOT.RDataFrame("tree", "file.root")
hist = rdf.Filter("theta > 0").Histo1D("pt")
hist.Draw()
```

1. Build RDataFrame
2. Cut on `theta`
3. Fill a histogram with p_T and draw it

Filling multiple histograms

```
h1 = rdf.Filter("theta > 0").Histo1D("pt")
h2 = rdf.Filter("theta < 0").Histo1D("pt")

h1.Draw()       // lazy evaluation: event loop is triggered here
h2.Draw("SAME") // no need to run event loop again!
```

- Book all your actions upfront.
- The first time a result is accessed, RDataFrame will fill all booked results
- This **lazy evaluation** means that the **iteration over events is only done once!**
- It is one of key ingredients for RDataFrames high performance
- Having a single event loop is *particularly beneficial if you have to iterate over many files that don't fit in memory*

More on histograms

```
h = rdf.Histo1D(("myName", "Title;x", 10, 0.0, 1.0), "x")
```

- You can specify a model histogram with:
  - a name and a title
  - a predefined axis range and binning
- Similar to the TH1 constructor you already know
- Here, the histogram is created with 10 bins ranging from 0 to 1, and the axis is labelled "x"

Define a new column

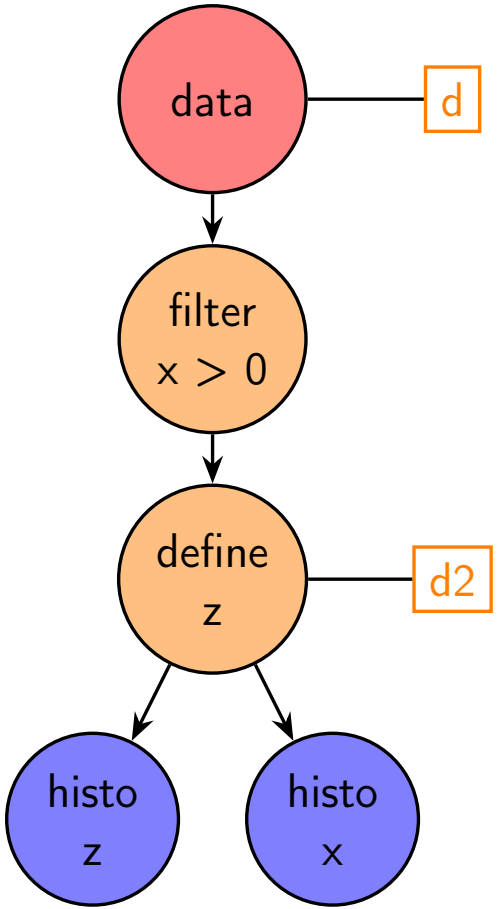
```
mean = d.Filter("x > y")
        .Define("z", "sqrt(x*x + y*y)")
        .Mean("z");
```

`Define()` takes the name of the new column and its expression. Later you can use the new column as if it was present in your data.

Think of your analysis as a data flow

```
d2 = d.Filter("x > 0")
      .Define("z", "x*x + y*y")
# d2 is a new data-frame,
# a transformed version of d

# make multiple histograms out of it
hz = d2.Histo1D("z")
hx = d2.Histo1D("x")
```

You can store transformed data-frames in variables, then use them as you would use an RDataFrame.



Cutflow reports

```
d.Filter("x > 0", "xcut")
 .Filter("y < 2", "ycut");

d.Report().Print();
```

When called on the main RDF object, `Report()` prints statistics for all filters *with a name*.

```
Out[0]:    xcut     : pass=49     all=100     --     49.000 %
           ycut     : pass=22     all=49      --     44.898 %
```

Saving data to a file

```
new_df = df.Filter("x > 0")
           .Define("z", "sqrt(x*x + y*y)")
           .Snapshot("tree", "newfile.root")
```

We filter the data, add a new column, and then save everything to file. No boilerplate code at all.

RDataFrame: declarative analysis

```
df = ROOT.RDataFrame("treename", "file.root")
histo = df.Filter(is_good_entry, ["x","y"])
          .Histo1D("x")
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented
- **parallelization is not trivial?**

RDataFrame: parallelism

```
ROOT.EnableImplicitMT()
d = ROOT.RDataFrame("treename", "file.root")
h = d.Filter(is_good_entry, ["x","y"]).Histo1D("x")
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented
- parallelization is **automatic**!

Defining new columns with C++

You can always use C++ functions to define your new columns:

```
ROOT.gInterpreter.Declare("""
auto calculateZ(float x, ROOT::VecOps::RVec<float>& y) {
    RVecF out;
    for(auto yi : y) {
        out.emplace_back(x * yi);
    }
    return out;
}
""")

rdf = rdf.Define("z", "calculateZ(x, y)")
```

Column `x` is of type `float`, `y` is a vector of floats, new column is vector of floats.
This was just to demonstrate the principle. If you want to add `x + y` you better do:

```
rdf = rdf.Define("z", "x + y")
```

ROOT's cutting-edge features: RDataFrame::Vary()

With the experimental RDataFrame::Vary(), you can efficiently declare variations of your analysis flow:

```
%%cpp
auto varyPt(double pt) {
    RVecD{pt*0.9, pt*1.1}; // returns a vector of variations
}
```

```
nominal_hx = df.Vary("pt", "varyPt", ("down", "up"))
      .Filter("pt > k")
      .Define("x", "someFunc", ("pt"))
      .Histo1D("x");

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

This streamlines greatly the treatment of **systematic variations**!

RDataFrame::Vary(): Example problem

You have some analysis where you select events with exactly two electrons above some p_T threshold:

```
rdf = rdf.Filter("nElectron == 2")
// some logic to compute electron pair mass...

rdf = rdf.Define("ptCut", "10.0")
rdf = rdf.Filter("Electron_pt[0] > ptCut && Electron_pt[1] > ptCut")

h = rdf.Histo1D("Dielectron_mass")
h.Draw()
```

Your professor asks you:

> "What if you change the p_T cut? Produce all histos with different cut values."

$\Rightarrow$ **RDataFrame::Vary()** to the rescue!

RDataFrame::Vary(): A possible solution

```
%%cpp
auto varyPtCut(double ptCut) {
    return RVecD{ptCut - 5, ptCut + 5};
}
```

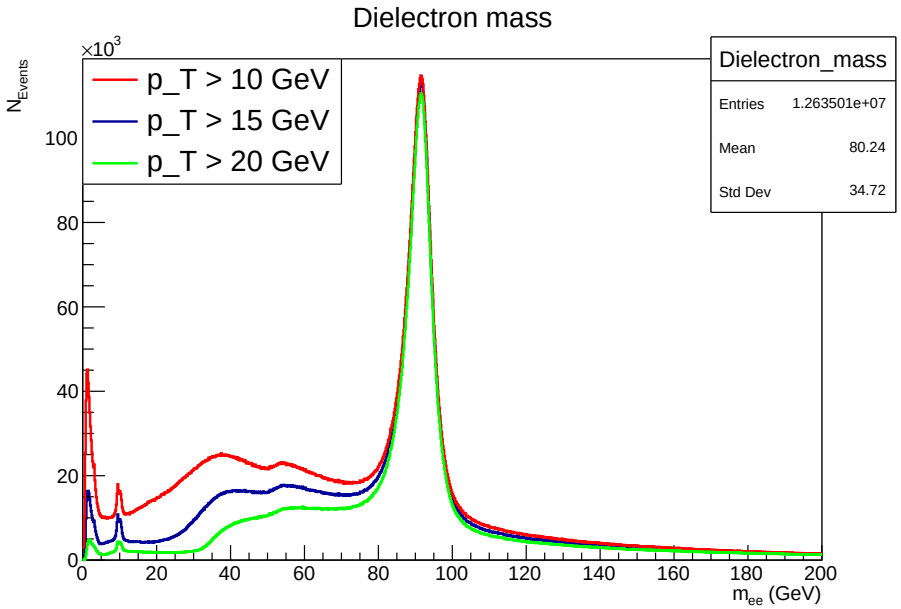
```
rdf = rdf.Filter("nElectron == 2")
# some logic to compute electron pair mass...

rdf = rdf.Define("ptCut", "10.0")
rdf = rdf.Vary("ptCut", "varyPtCut(ptCut)", ("down", "up"))
rdf = rdf.Filter("Electron_pt[0] > ptCut && Electron_pt[1] > ptCut")

h = rdf.Histo1D("Dielectron_mass")
hvars = ROOT.RDF.Experimental.VariationsFor(h)

hvars["nominal"].Draw()
hvars["ptCut:down"].Draw("SAME")
hvars["ptCut:up"].Draw("SAME")
```

RDataFrame::Vary(): Example

You are very happy with the solution:

- Your whole analysis still runs in a single event loop
- Your new code with the three p_T cut variations is **only 50 % slower** than the original code
- Can you explain why that is?

In particular for many variations, this sub-linear computation cost scaling is very important!



Wrapping up

Conclusions

- ROOT is a **powerful toolkit** for the workflows specific to particle physics
  - For analysis, **RooFit** and **RDataFrame** are useful in particular
- There is lots of documentation on the internet, and if you are still stuck you will quickly get help on the very active **ROOT forum**!
- ROOT actively developed by the community for the **community**
  - If you have ideas for improvements, don't hesitate to **engage** with the developers
- ROOT provides you performant algorithms written in **C++** that you can use in **Python** too, together with many other great Python libraries

Thank you for you attention!

Backup - ROOT fitting details (if you're not using RooFit)

Building more Complex Functions

- Any C++ object (functor) implementing

  `double operator() (double *x, double *p)`

```
struct Function {
    double operator() (double *x, double *p){
        return p[0]*TMath::Gaus(x[0],p[1],p[2]);
    }
};
Function f;
TF1 f1("f1",f,xmin,xmax,npar);
```

- Also a lambda function (with Cling and C++-11)

```
TF1 f1("f1",[](double *x, double *p){return p[0]*x[0];},0,10,1);
```

- a lambda can be used also as a string expression, which will be JIT'ed by CLING

```
TF1 f1("f1","[](double *x, double *p){return p[0]*x[0];}",0,10,1);
```

Functionality provided by TFormula

TFormula is based on Cling. Additional functionality provided:

- better parameter definition
  - `TF1("f1","gaus(x, [Constant],[Mean],[Sigma])");`
- function composition by concatenating expressions
  - `TF1 fs("sigma","[0]*x+[1]");`
  - `TF1 f1("f1","gaus(x,[C],[Mean],sigma(x,[A],[B])");`
- normalized sum for component fitting
  - `TF1 model("model","NSUM(expo, gaus)", ...);`
- convolutions
  - `TF1 voigt("voigt", "CONV(breitwiegner, gaus)", xmin, xmax);`
- can define vectorized functions for faster fitting and evaluation
  - see vectorizedFit tutorial
- support for auto-differentiation (automatic generation of gradient and Hessian)

Fitting options

- Likelihood fit for histograms
  - option "L" for count histograms; `h1->Fit("gaus","L");`
  - option "LW" in case of weighted counts. `h1->Fit("gaus","LW");`
- Default is chi-square with observed errors (and skipping empty bins)
  - option "P" for Pearson chi-square expected errors, and including empty bins `h1->Fit("gaus","P");`
- Use integral function of the function in bin `h1->Fit("gaus","L I");`
- Compute MINOS errors : option "E" `h1->Fit("gaus","L E");`

Some more Fitting Options

- Fitting in a Range

```
h1->Fit("gaus","","",-1.5,1.5);
```

- For doing several fits

```
h1->Fit("expo","+","",2.,4);
```

- Quiet / Verbose: option "Q"/"V"

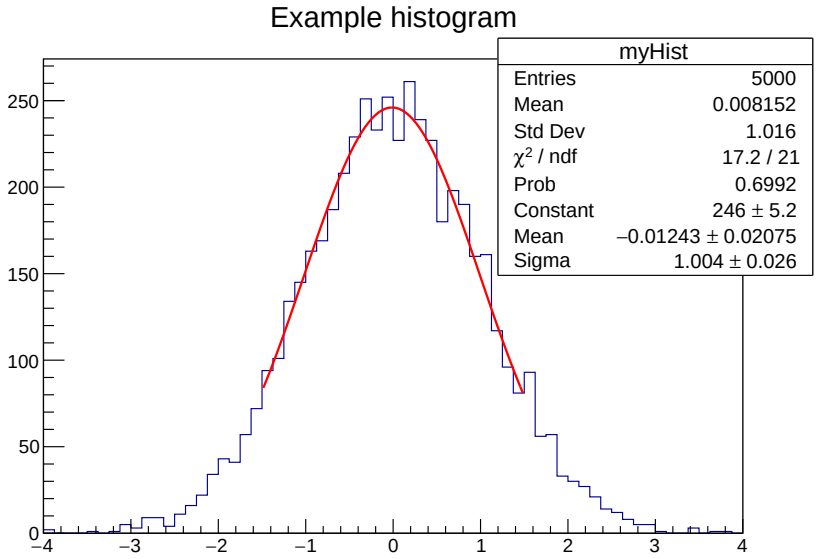
```
h1->Fit("gaus","V");
```

- Avoid storing and drawing fit function (useful when fitting many times)

```
h1->Fit("gaus","L N 0");
```

- Save result of the fit, option "S"

```
auto result = h1->Fit("gaus","L S");
result->Print("V");
```



Example of ranged fit.

All fitting options documented in reference guide or User Guide (Fitting Histogram chapter)

Parameter Errors

Errors returned by the fit are computed from the second derivatives of the log-likelihood function

- Assume the negative log-likelihood function is a parabola around minimum
- This is true asymptotically and in this case the parameter estimates are also normally distributed.
- The estimated correlation matrix is then:

$$\hat{V}(\hat{\theta}) = \left[\left(-\frac{\partial^2 \ln L(x;\theta)}{\partial^2 \theta} \right)_{\theta=\hat{\theta}} \right]^{-1} = H^{-1}$$