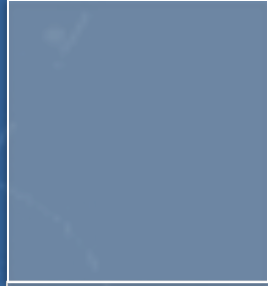
diana hep

NEW YORK UNIVERSITY



a pyROOT Basic Course

Vincent Croft - with help from the ROOT team

A Modified Fork of

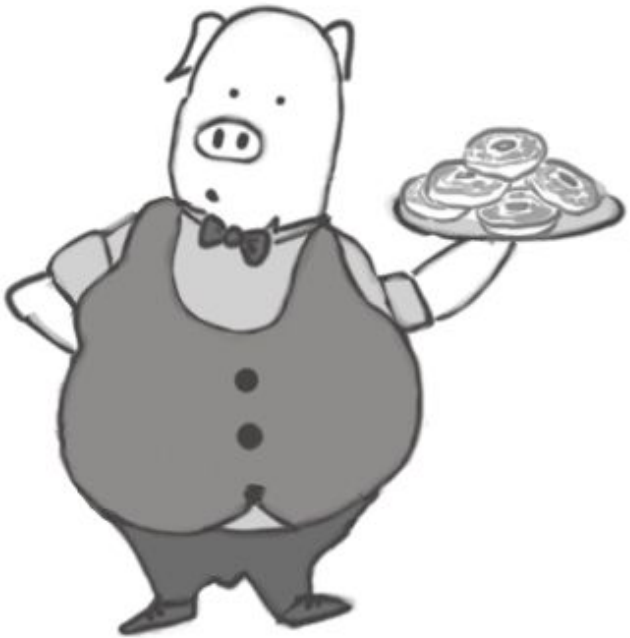
https://github.com/root-project/training/tree/master/BasicCourse

ROOT

Data Analysis Framework

https://root.cern

About Me



- NIKHEF PhD
- ATLAS Higgs
- Multivariate Analysis
- Statistics Combination
- MC-Net Fellow
- ROOT Contributor

This Course

Divided in "Learning modules" over the course of the day

- Morning:
  - Introduction
  - Python interface to C++ Interpreter
  - IO - TFiles, TTrees
  - Histograms, Graphs and Graphics
- Afternoon
  - Machine Learning with TMVA
  - RooFit

(cc) BY

The “Learning Modules”

- Each module treats a well defined topic
- Each module relies on concepts treated in previous modules
- We will declare before each module the “Learning Objectives”

Resources

- ROOT Website: https://root.cern
- Material online: https://github.com/root-project/training
- More material: https://root.cern/getting-started
  - Includes a booklet for beginners: **the “ROOT Primer”**
- Reference Guide: https://root.cern/doc/master/index.html
- Forum: https://root-forum.cern.ch

Where to get ROOT

- Install yourself
- Use SWAN
- Tunnel…

1. First ssh in to the remote server (for CERN users I'm just going to call this lxplus for now).
   - `ssh -L 7000:localhost:6000 username@lxplus.cern.ch`
   - this tells ssh to form the tunnel between lxplus and localhost port 7000
   - the port numbers can be any 4 digit numbers. Have fun.
2. Set up password for the notebook.app
   - `bash-4.1$ ipython`
   - `In[1]: from Ipython.lib import passwd`
   - `In[2]: passwd()`
   - `Enter password:`
   - `Verify password:`
   - `Out[2]: 'some:numbersandletters:morenumbersandletters'`
3. Create Profile
   - `bash-4.1$ ipython profile create`
4. edit ~/.config/ipython/profile_default/ipython_notebook_config.py
   - `# Password to use for web authentication`
   - `c = get_config()`
   - `c.NotebookApp.password=u'some:numbersandletters:morenumbersandletters'`
5. Start notebook on port listed above
   - `bash-4.1$ ipython notebook --no-browser --port=6000`
6. Now in your web browser of choice open `http://localhost:7000`
7. Enjoy

How to Use ROOT

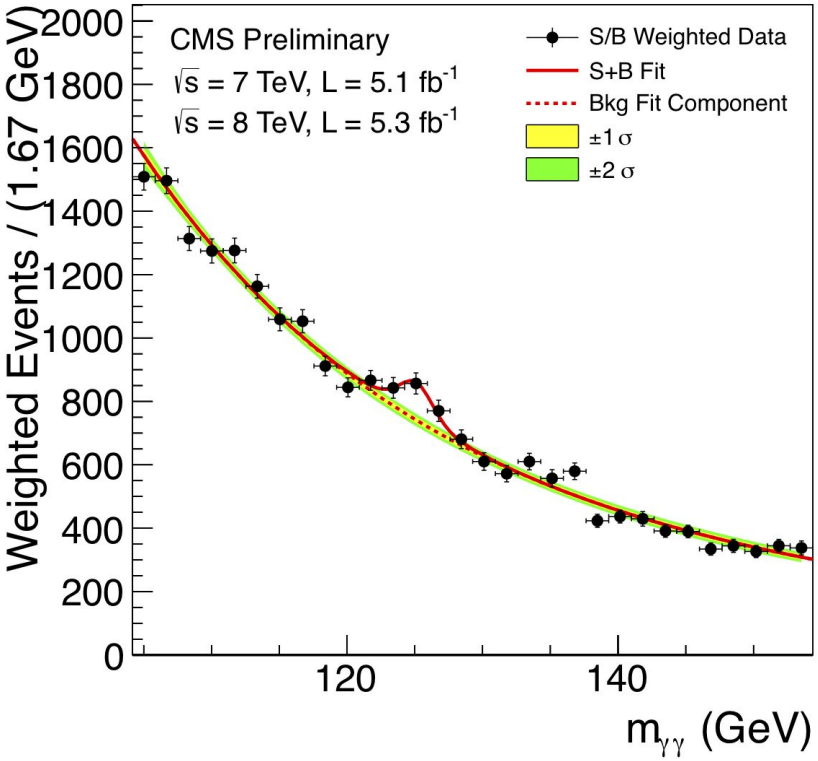
- In progress!

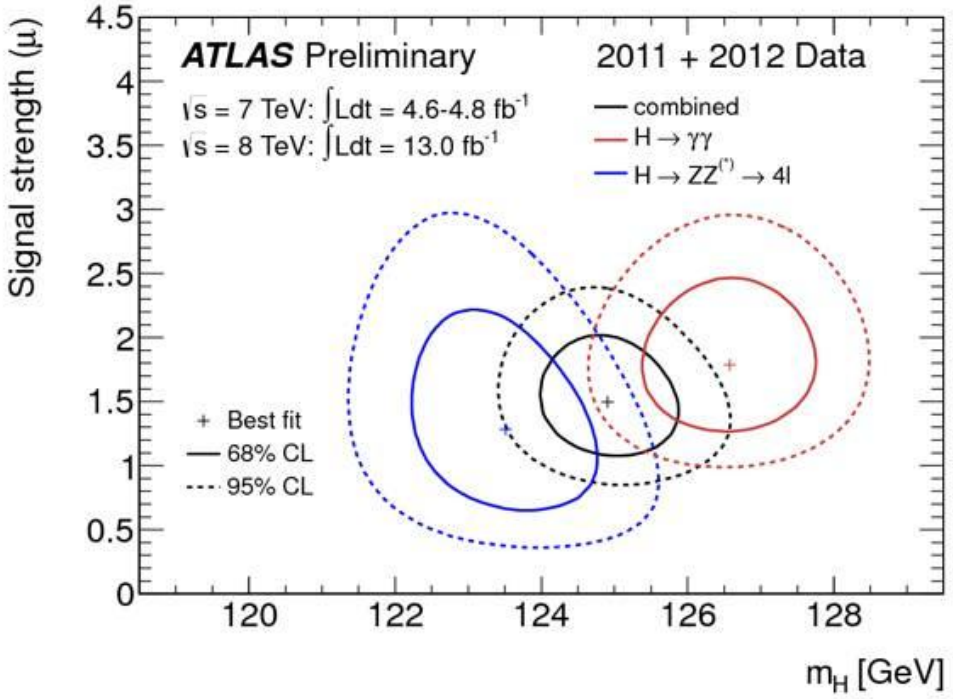
Morning

Introduction

A Quick Tour of ROOT

What can you do with ROOT?





ROOT in a Nutshell

- ROOT is a software framework with building blocks for:
  - Data processing
  - Data analysis
  - Data visualisation
  - Data storage

An Open Source Project
We are on github
github.com/root-project
All contributions are warmly welcome!

- ROOT is written mainly in C++ (C++11/17 standard)
  - Bindings for Python the focus here!


- Adopted in High Energy Physics and other sciences (but also industry)
  - 1 EB of data in ROOT format
  - Fits and parameters' estimations for discoveries (e.g. the Higgs)
  - Thousands of ROOT plots in scientific publications

Interpreter

- ROOT has a built-in interpreter : CLING
  - C++ interpretation: highly non trivial and not foreseen by the language!
  - One of its kind: Just In Time (JIT) compilation
  - A C++ interactive shell

```
$ root
root[0] 3 * 3
(const int) 9
```

- Can interpret “macros” (non compiled programs)
  - Rapid prototyping possible
- ROOT provides also Python bindings
  - Will use Python interpreter directly after a simple *import ROOT*

Persistency or Input/Output (I/O)

- ROOT offers the possibility to write C++ objects into files
  - This is impossible with C++ alone
  - Used the LHC detectors to write several petabytes per year
- Achieved with serialization of the objects using the reflection capabilities, ultimately provided by the interpreter
  - Raw and column-wise streaming
- As simple as this for ROOT objects: one method - *TObject::Write*

Cornerstone for storage of experimental data

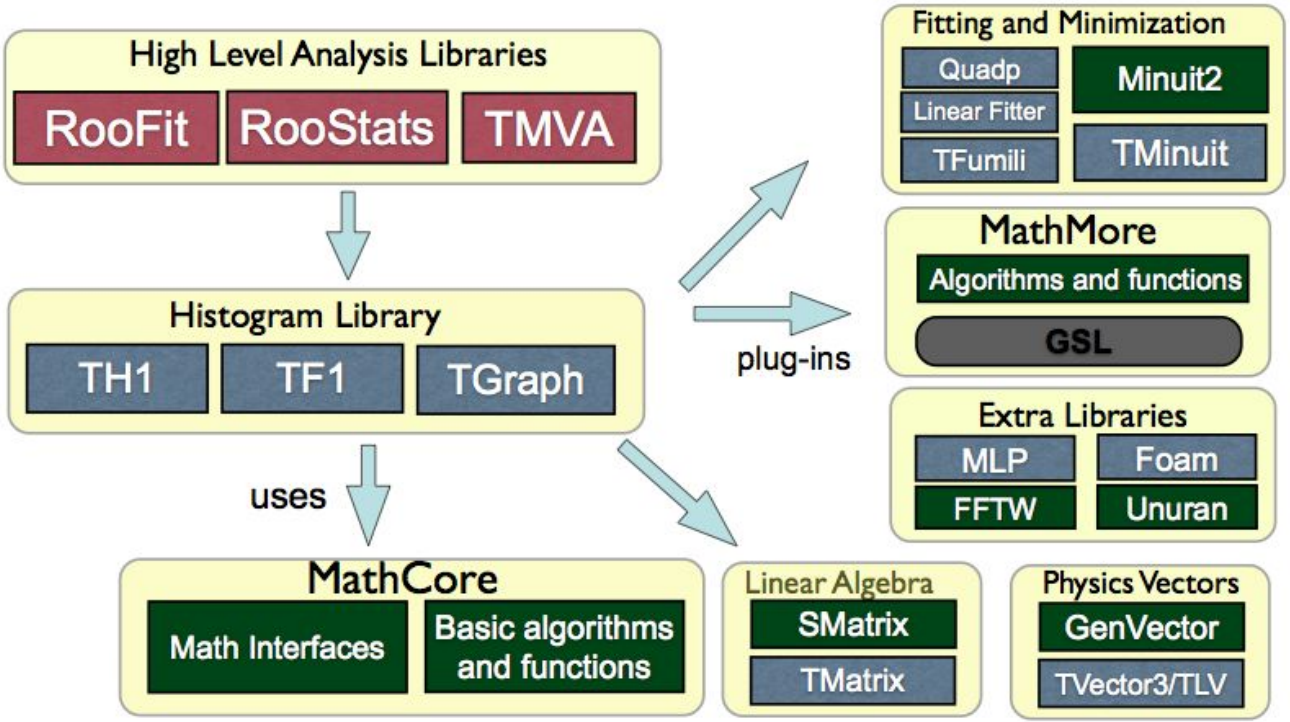
LHC Data in ROOT Format

1 EB

as of 2017

Mathematics and Statistics

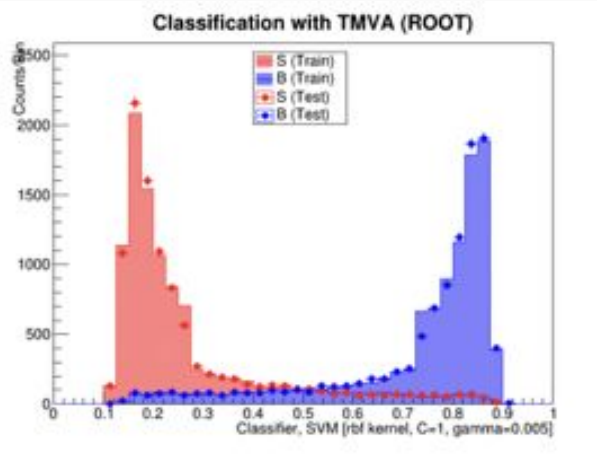
- ROOT provides a rich set of mathematical libraries and tools for sophisticated statistical data analysis



Machine Learning: TMVA

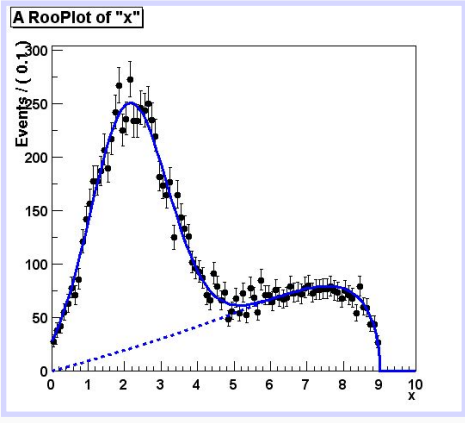
TMVA : Toolkit for Multi-Variate data Analysis in ROOT

- provides several built-in ML methods including:
  - Boosted Decision Trees
  - Deep Neural Networks
  - Support Vector Machines
- and interfaces to external ML tools
  - scikit-learn, Keras (Theano/Tensorflow), R



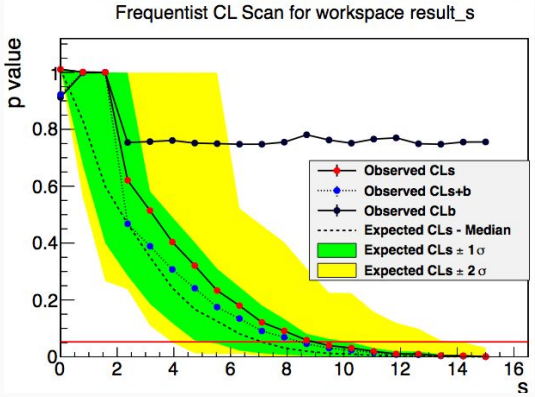
RooFit: Toolkit for Data Modeling and Fitting

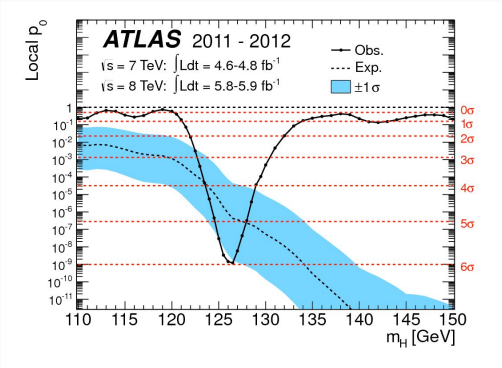
- functionality for building models: probability density functions (p.d.f.)
  - distribution of observables in terms of parameters $P(x;p)$
- complex model building from standard components
  - e.g. composition, addition, convolution,...
- RooFit models have functionality for
  - maximum likelihood fitting for parameter estimation
  - toy MC generation
  - visualization
  - sharing and storing (workspace)



RooStats

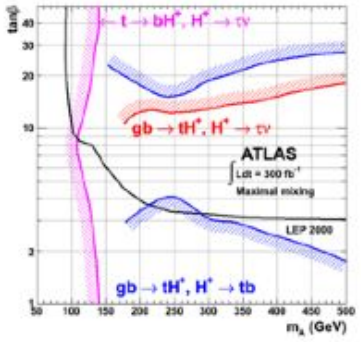
- Advanced Statistical Tools for HEP analysis. Used for :
  - estimation of Confidence/Credible intervals
  - hypotheses Tests
    - e.g. Estimation of Discovery significance
- Provides both Frequentist and Bayesian tools
- Facilitate combination of results

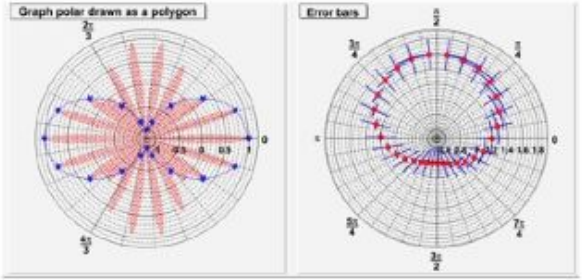


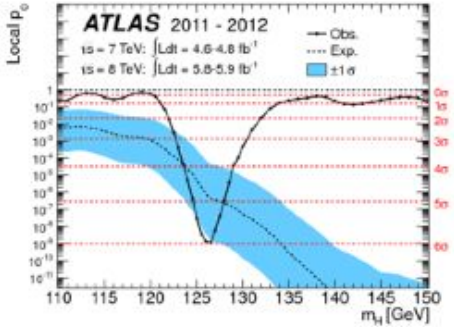


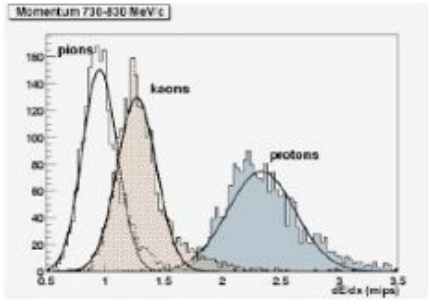
Graphics in ROOT

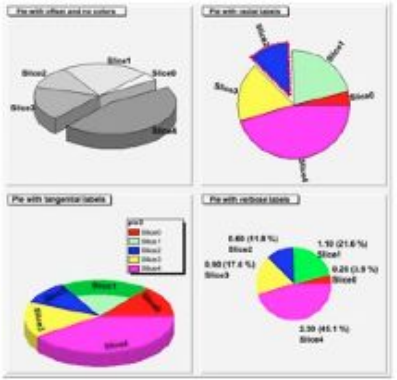
- Many formats for data analysis, and not only, plots

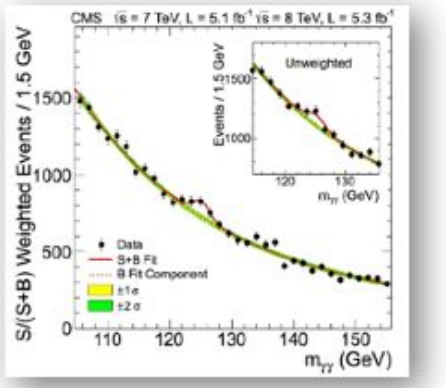




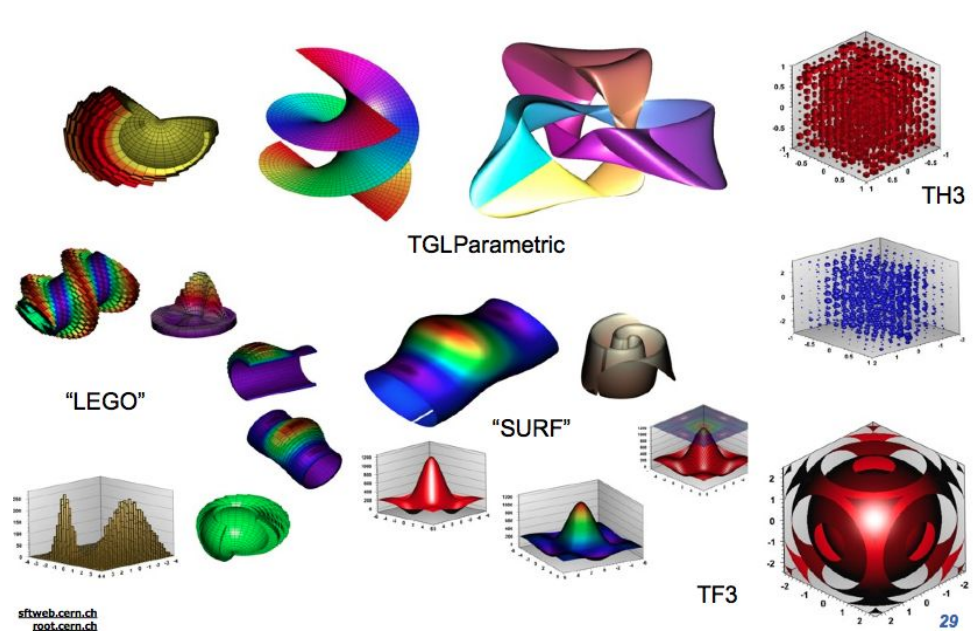


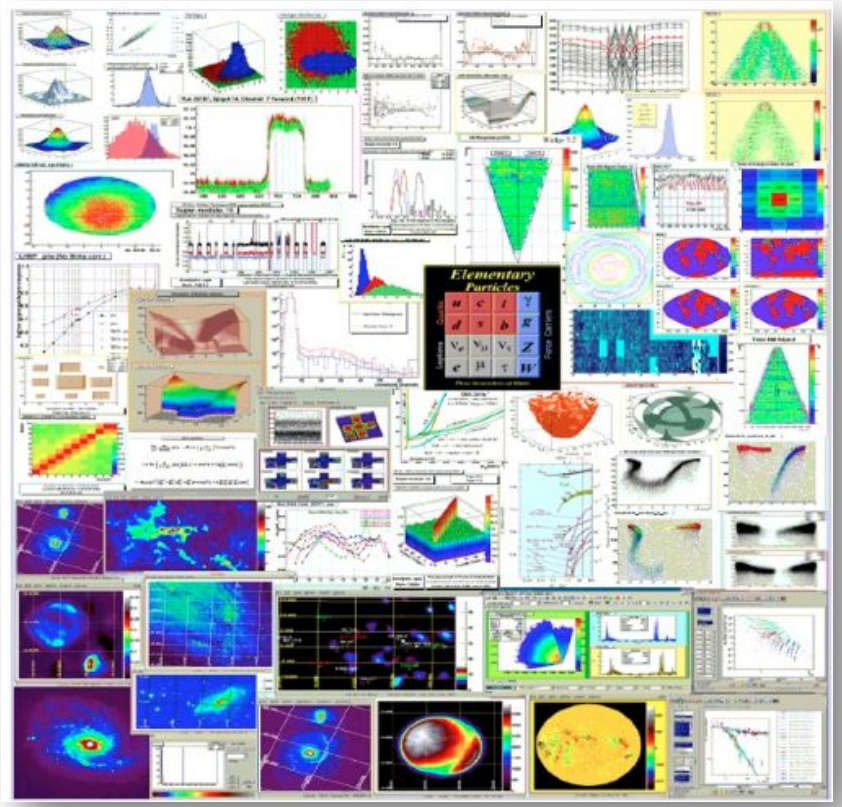






2D and 3D Graphics



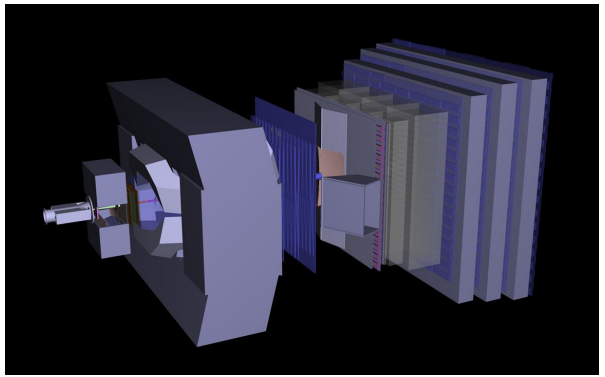


Can save graphics in many formats:
ps, pdf, svg, jpeg, LaTex, png, c, root ...

JSROOT

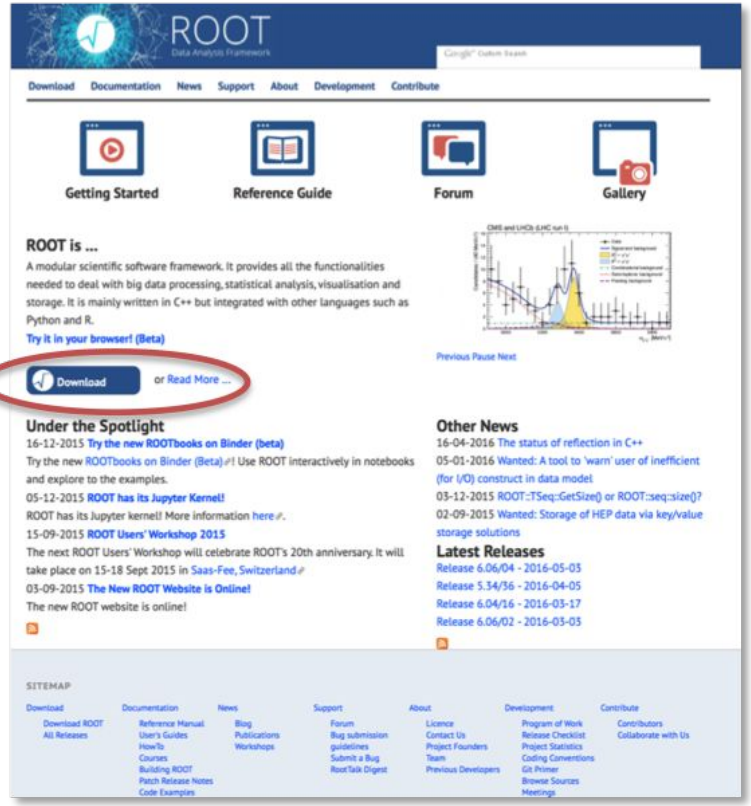
- JSROOT: a JavaScript version of ROOT graphics and I/O
- Complements traditional graphics
- Visualisation on the web or embedded in notebooks
- Basic functionality for exploring data in ROOT format

More details to come!



https://root.cern

- ROOT web site: **the** source of information and help for ROOT users
  - For beginners and experts
  - Downloads, installation instructions
  - Documentation of all ROOT classes
  - Manuals, tutorials, presentations
  - Forum
  - …



Exercise

- ROOT provides mathematical functions, for example the widely known and adopted Gaussian
- For x values of 0,1,10 and 20 check the difference of the value of a hand-made non-normalised Gaussian and the TMath::Gaus routine

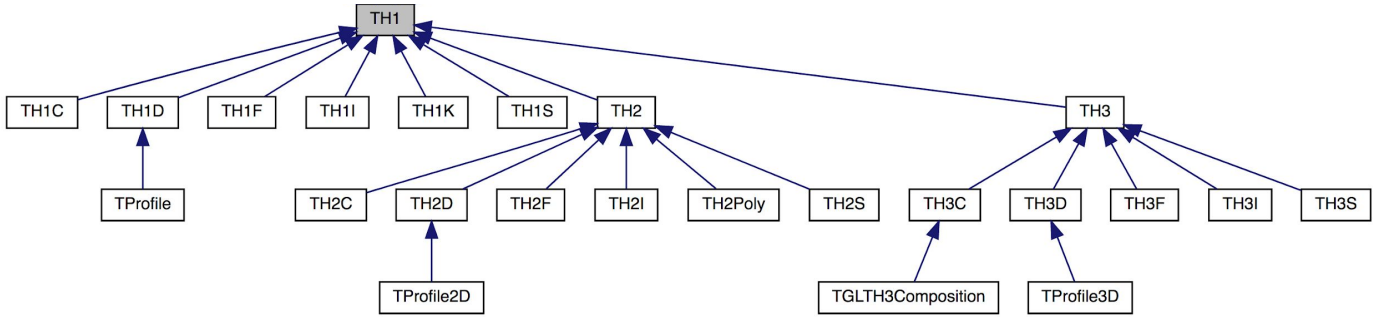

```
import ROOT
x = 0
ROOT.TMath.Exp(-x*x*.5) - ROOT.TMath.Gaus(x)
```

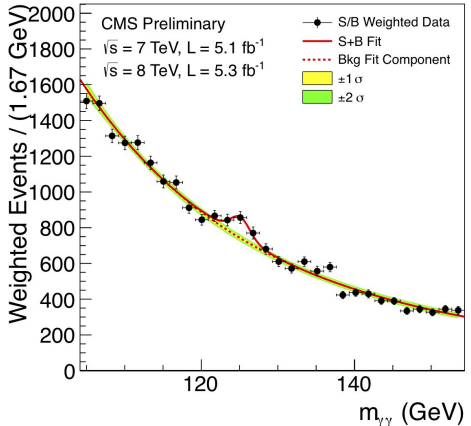
- Can you get this with scipy.stats.norm ?

Histograms, Graphs and Functions

Histograms

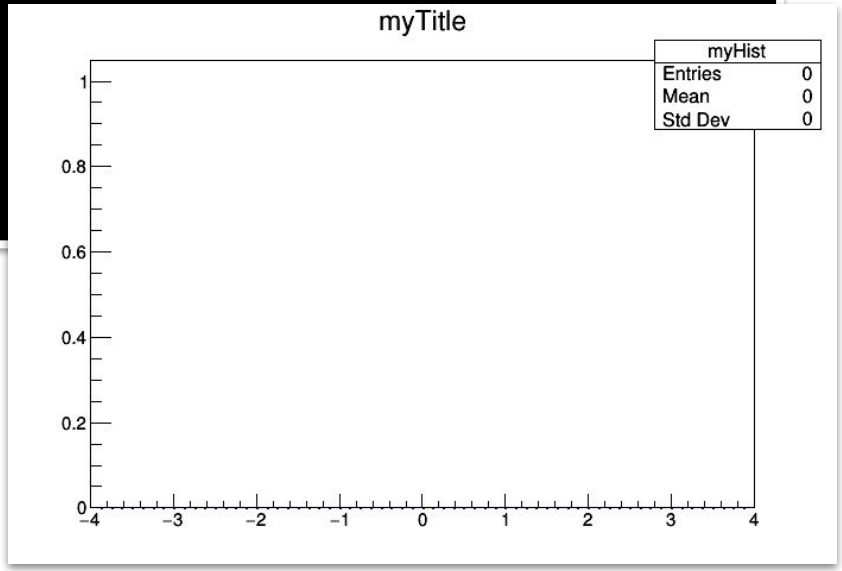
- Simplest form of data reduction
  - Can have billions of collisions, the Physics displayed in a few histograms
  - Possible to calculate momenta: mean, rms, skewness, kurtosis …
- Collect quantities in discrete categories, the bins
- ROOT Provides a rich set of histogram types
  - We'll focus on histogram holding a *float* per bin





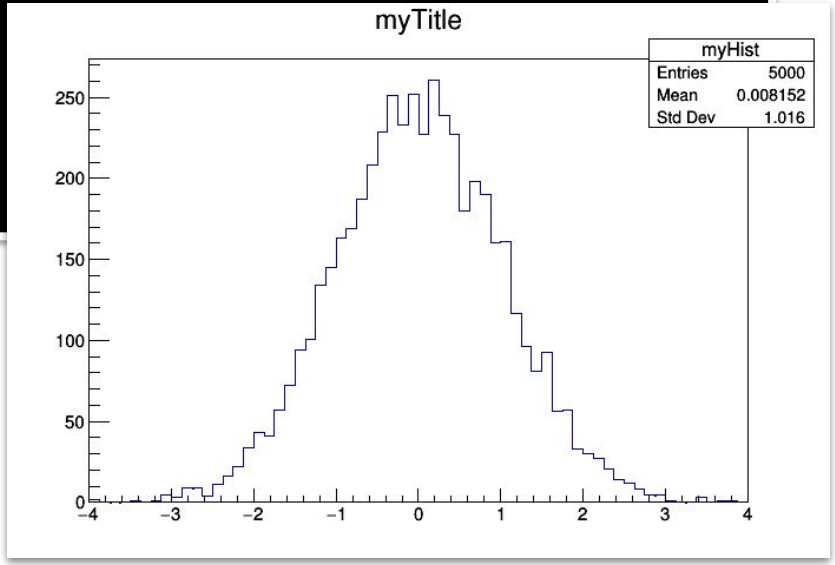
My First Histogram

```
h = ROOT.TH1F("myHist", "myTitle", 64, -4, 4)
h.Draw()
```



My First Histogram

```
h.FillRandom("gaus")
h.Draw()
```



Interlude: Scope

Bad for graphics:

```
// makeHist.C:
void makeHist() {
  TH1F hist("hist", "My Histogram");
  hist.Draw(); // shows histogram
}
```

ROOT doesn't show my histogram!

Interlude: Scope

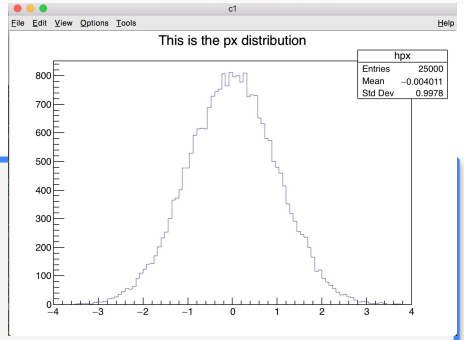
Bad for graphics:

```
// makeHist.C:
void makeHist() {
  TH1F hist("hist", "My Histogram");
  hist.Draw(); // shows histogram
} // destroys hist
```

ROOT doesn't show my histogram!

Interlude: Heap

Need a way to control lifetime

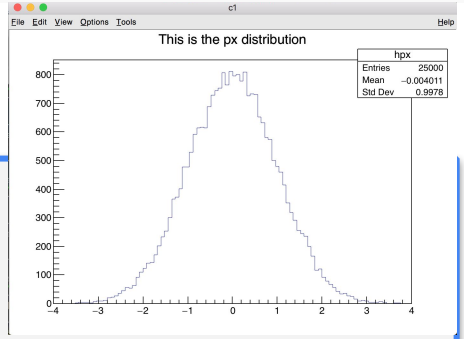



```
// makeHist.C:
void makeHist() {
  TH1F *hist = new TH1F("hist","My Histogram");
  hist->Draw(); // shows histogram
} // does not destroy hist!
```

`new` puts object on “heap”, escapes scope

Interlude: Heap

Need a way to control lifetime




```
// makeHist.C:
void makeHist() {
  TH1F *hist = new TH1F("hist","My Histogram");
  hist->Draw(); // shows histogram
} // does not destroy hist!
```

Not necessary in

Statistics and Fit parameters

- ROOT histograms have additional information called "statistics"
- ROOT adds them automatically to the plot when a histogram is drawn
- They can be turned on or off with the histogram method `SetStats()`
- `gStyle->SetOptStat()` defines which statistics parameters must be shown.

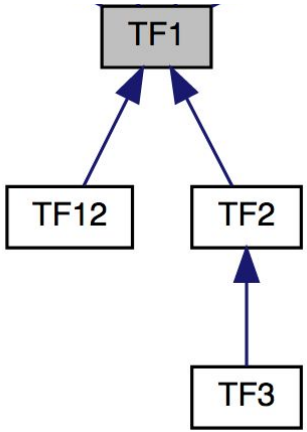
Also, after a fit, the fit result can be drawn on the plot and customized with `gStyle->SetOptFit()`

hpx	
Entries	25000
Mean	−0.004011
Std Dev	0.9978

Functions

- 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 the time being

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 smooth curve
"FC"	draw a fill area below a smooth curve



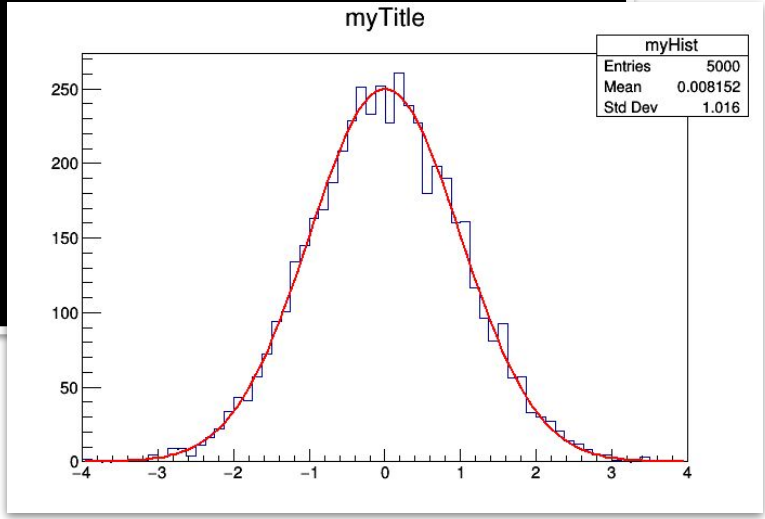
Functions

Can describe functions as:

- Formulas (strings)
- C++ functions/functors/lambdas
  - Implement your highly performant custom function
- With and without parameters
  - Crucial for fits and parameter estimation

My First Function

```
root [0] TH1F 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")
```



ROOT as a Function Plotter

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

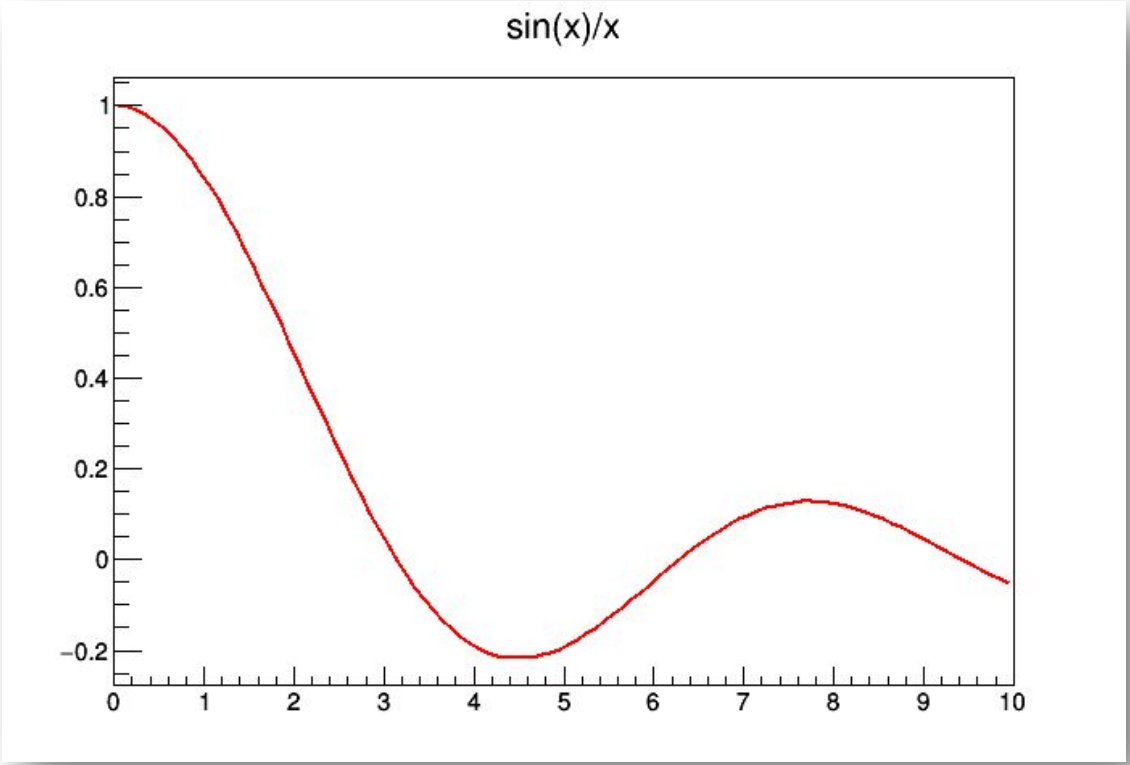
```
root [0] TF1 f1("f1","sin(x)/x",0.,10.);//name,formula, min, max
root [1] f1.Draw();
```

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

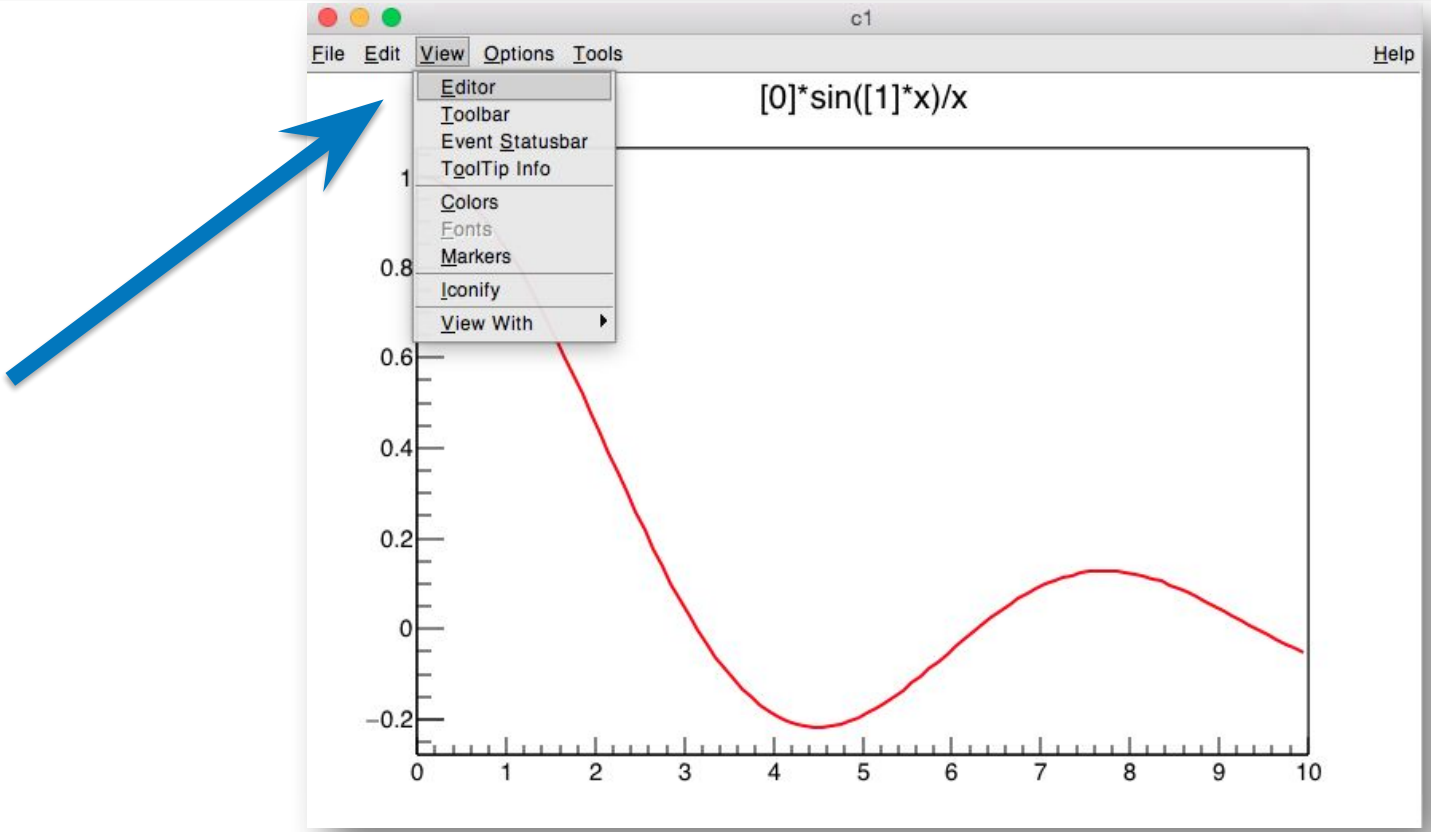
```
root [2] TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
root [3] f2.SetParameters(1,1);
root [4] f2.Draw();
```

Try it!

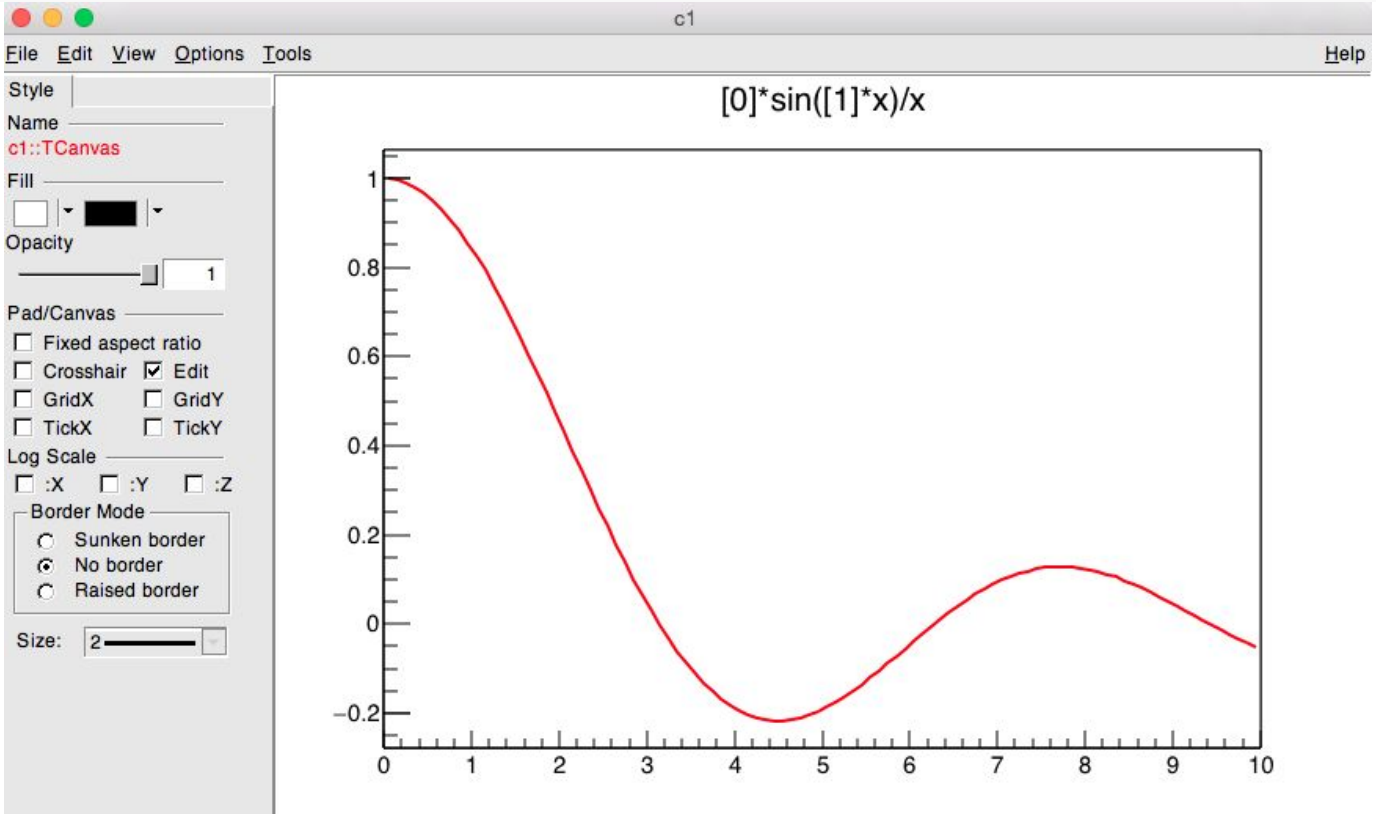
ROOT as a Function Plotter



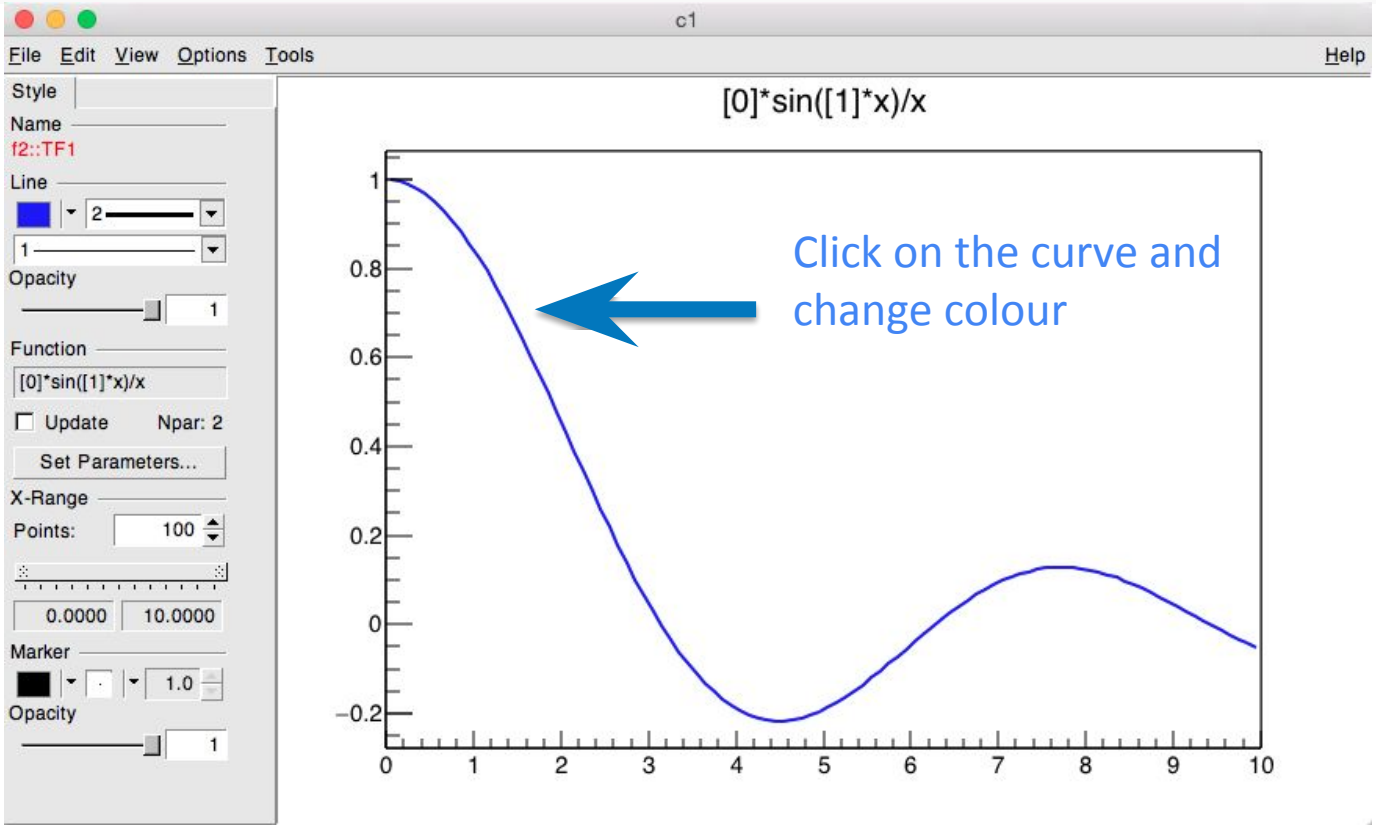
ROOT as a Function Plotter



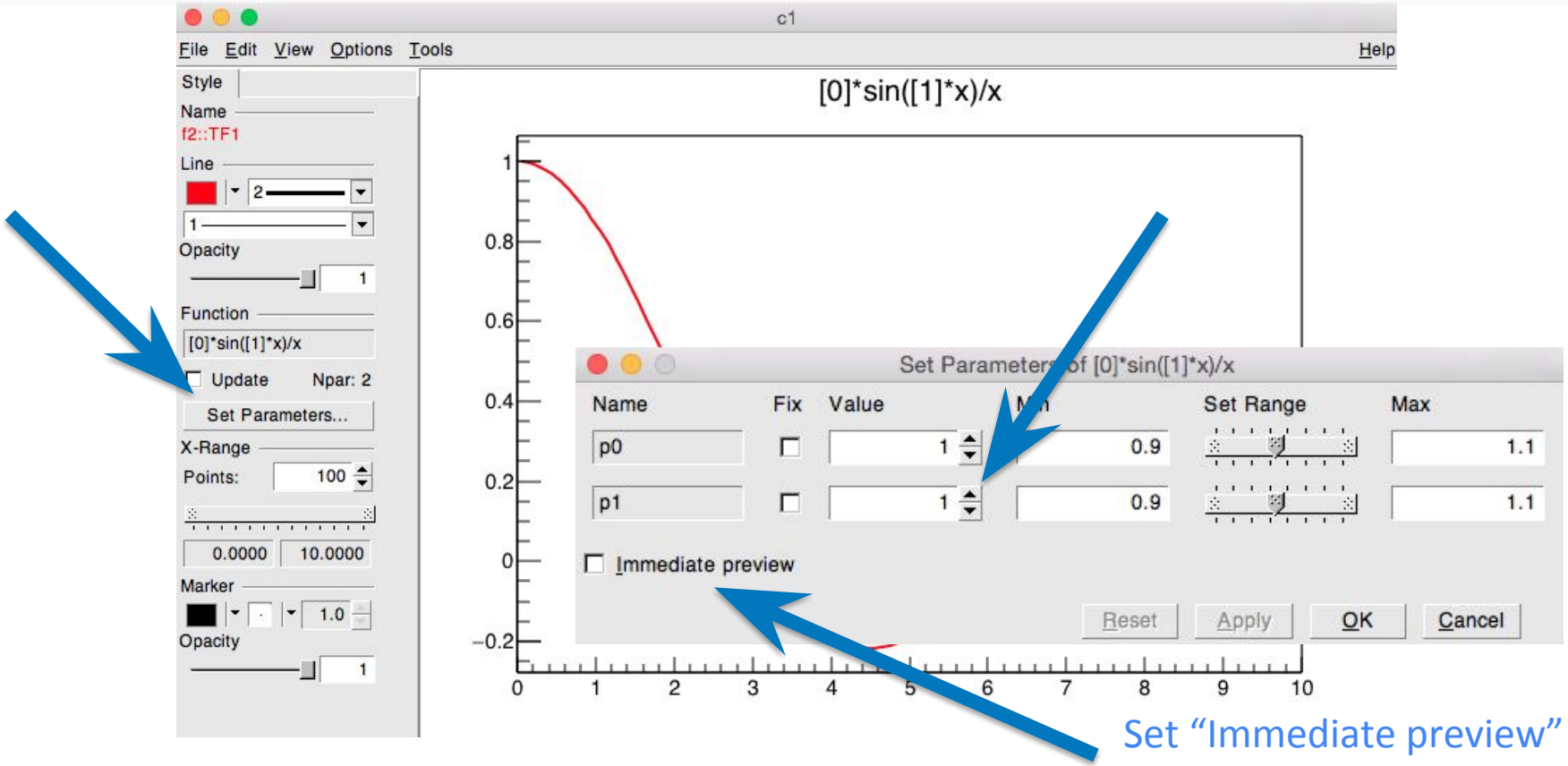
ROOT as a Function Plotter



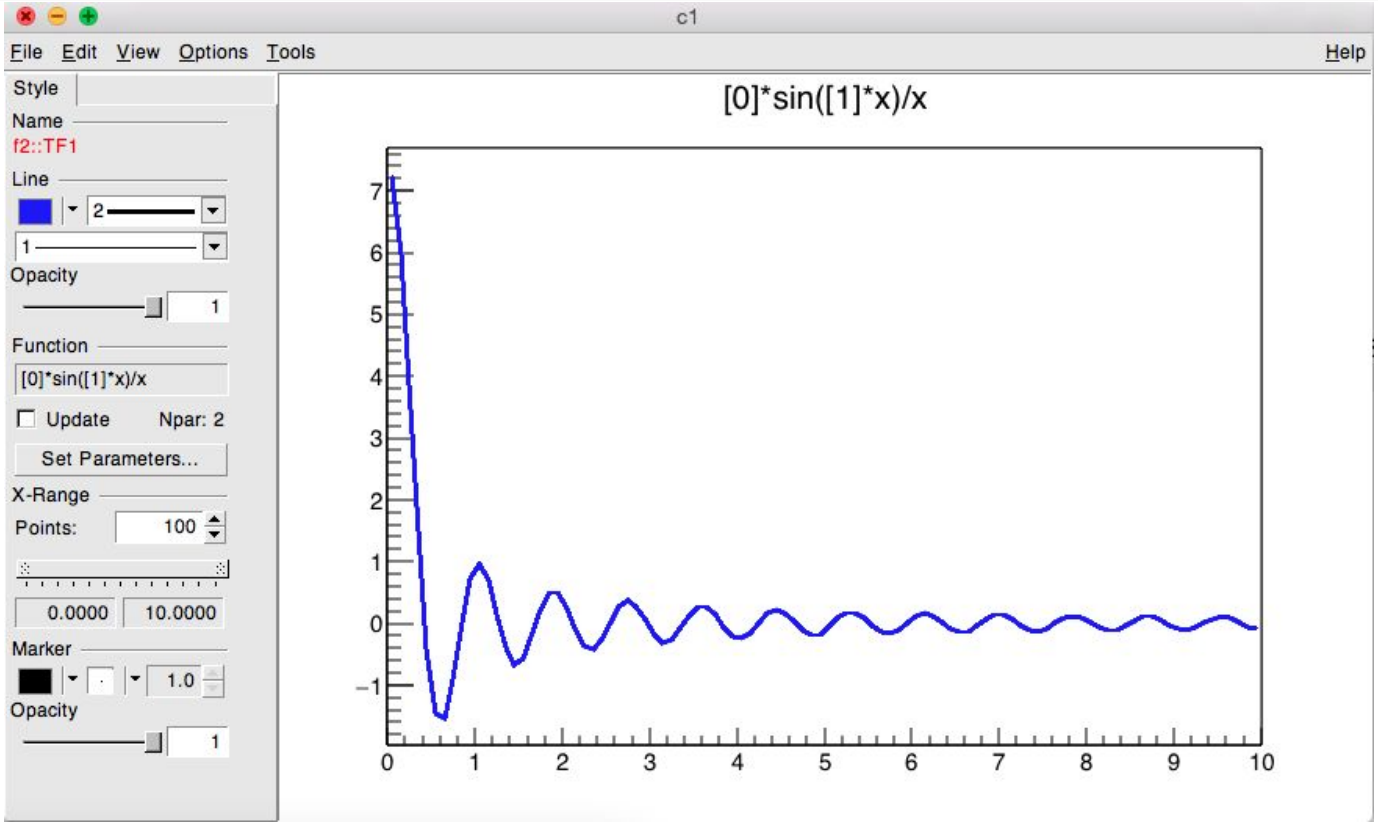
ROOT as a Function Plotter



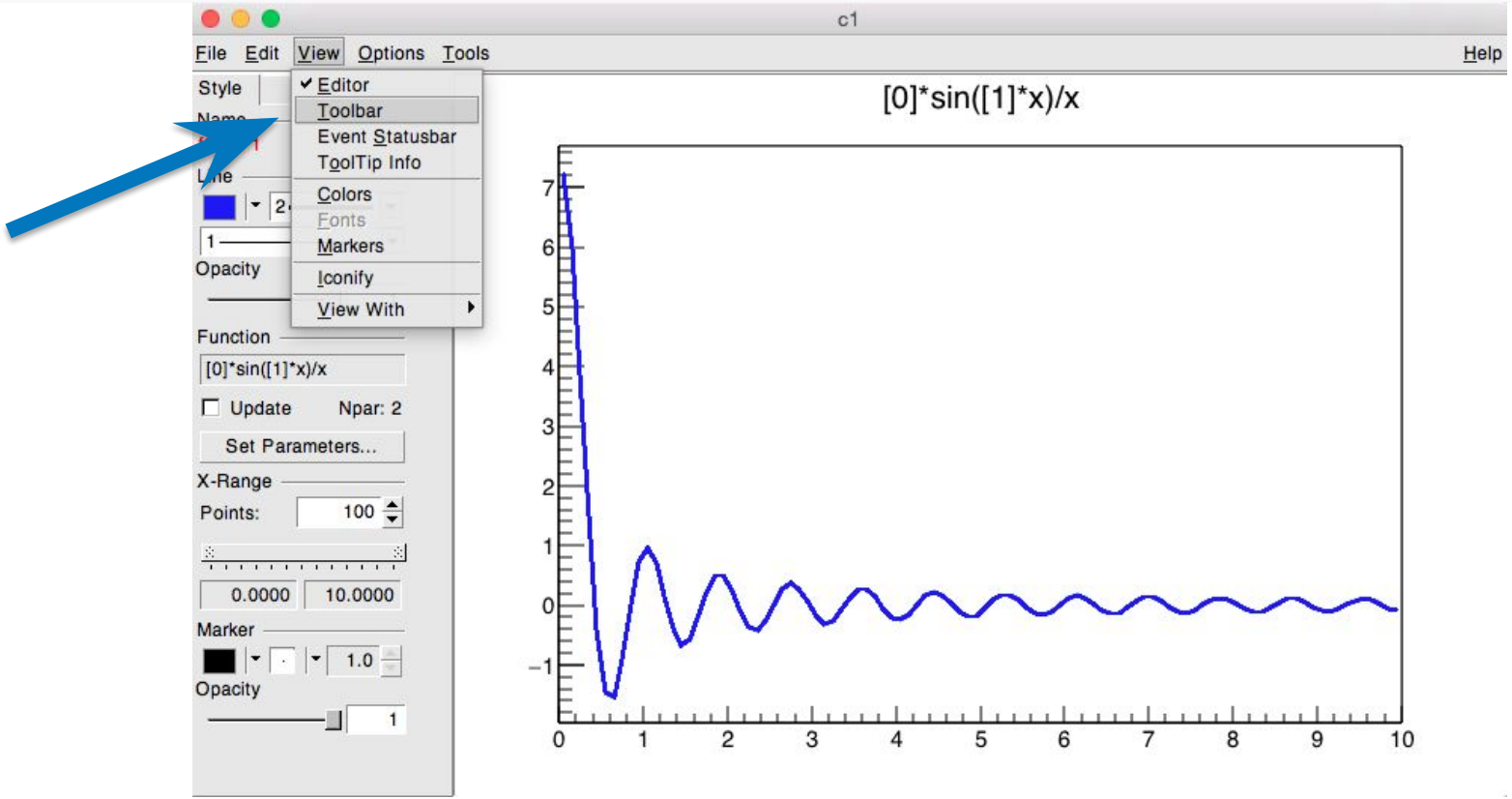
ROOT as a Function Plotter



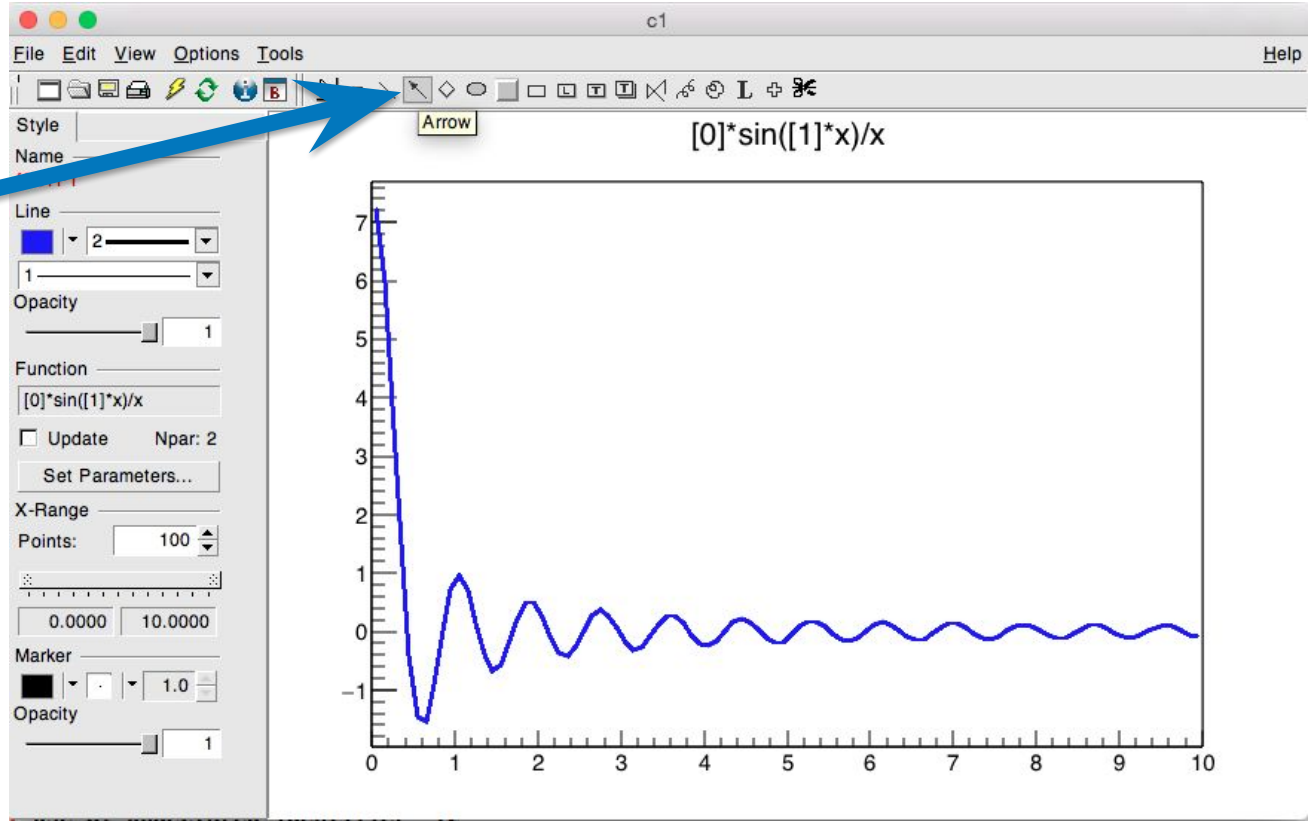
ROOT as a Function Plotter



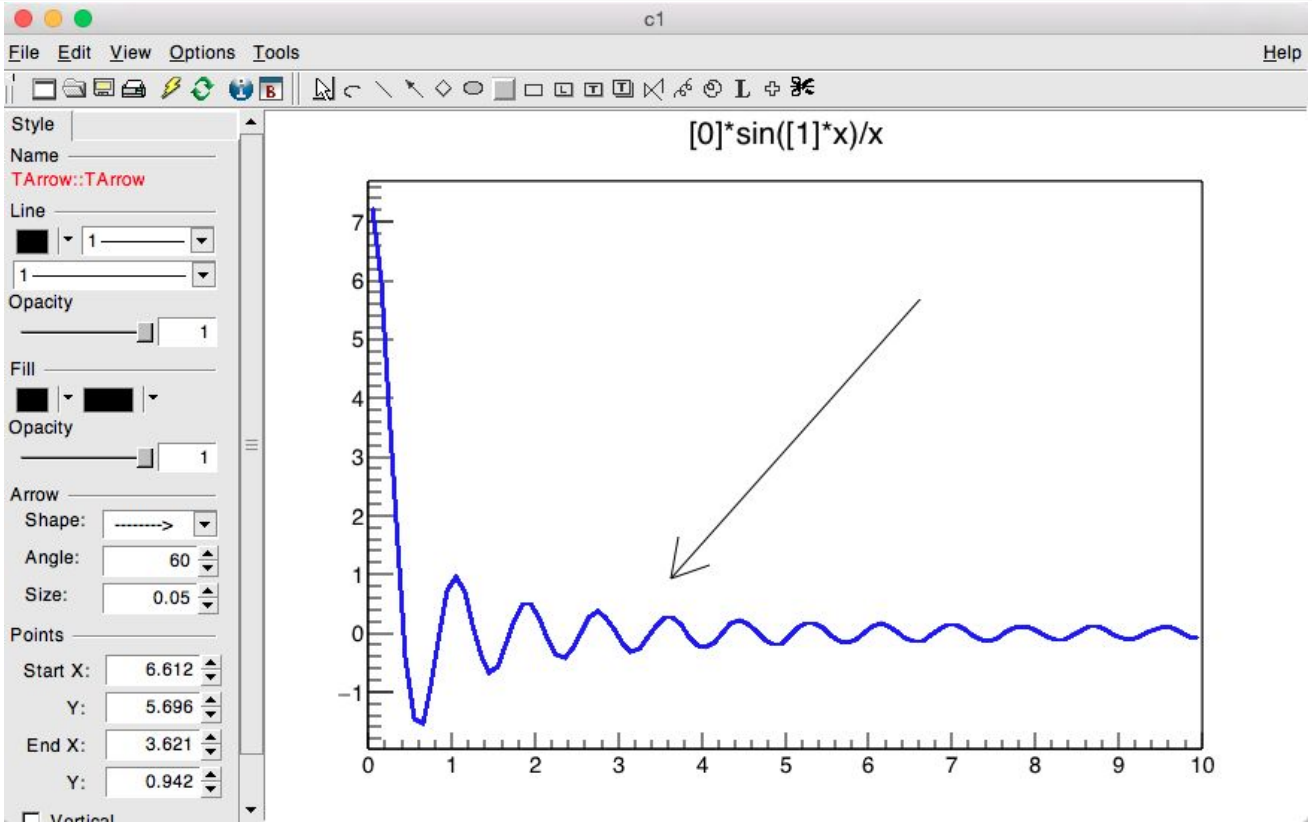
ROOT as a Function Plotter



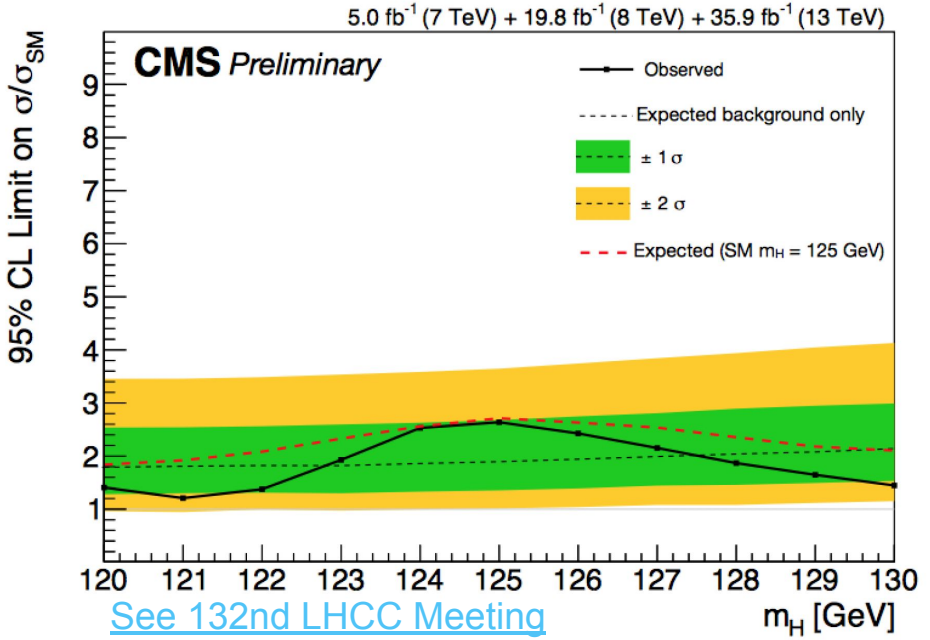
ROOT as a Function Plotter



ROOT as a Function Plotter



Graphs

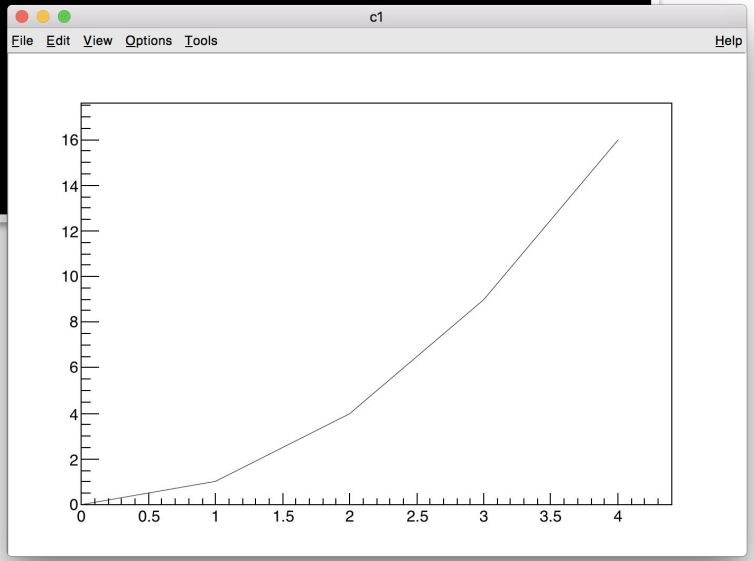


See 132nd LHCC Meeting

- Display points and errors
- Not possible to calculate momenta
- Not a data reduction mechanism
- **Fundamental to display trends**
- Focus on TGraph and TGraphErrors classes in this course

My First Graph

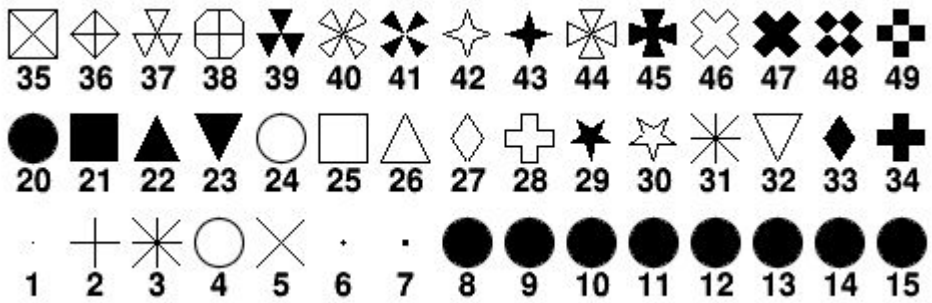
```
root [0] TGraph g;
root [1] for (auto i : {0,1,2,3,4}) g.SetPoint(i,i,i*i)
root [2] g.Draw("APL")
```



Creating a Nice Plot: Survival Kit

Credit: https://www.swissknifeshop.com

The Markers

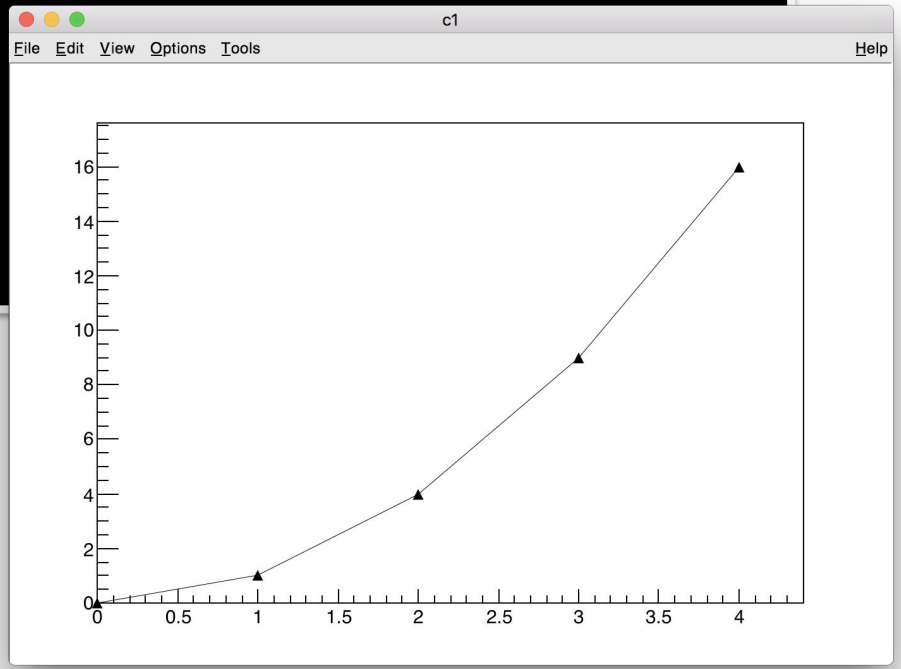



```
kDot=1, kPlus, kStar, kCircle=4, kMultiply=5,
kFullDotSmall=6, kFullDotMedium=7, kFullDotLarge=8,
kFullCircle=20, kFullSquare=21, kFullTriangleUp=22,
kFullTriangleDown=23, kOpenCircle=24, kOpenSquare=25,
kOpenTriangleUp=26, kOpenDiamond=27, kOpenCross=28,
kFullStar=29, kOpenStar=30, kOpenTriangleDown=32,
kFullDiamond=33, kFullCross=34 etc…
```

Also available through more friendly names ☺

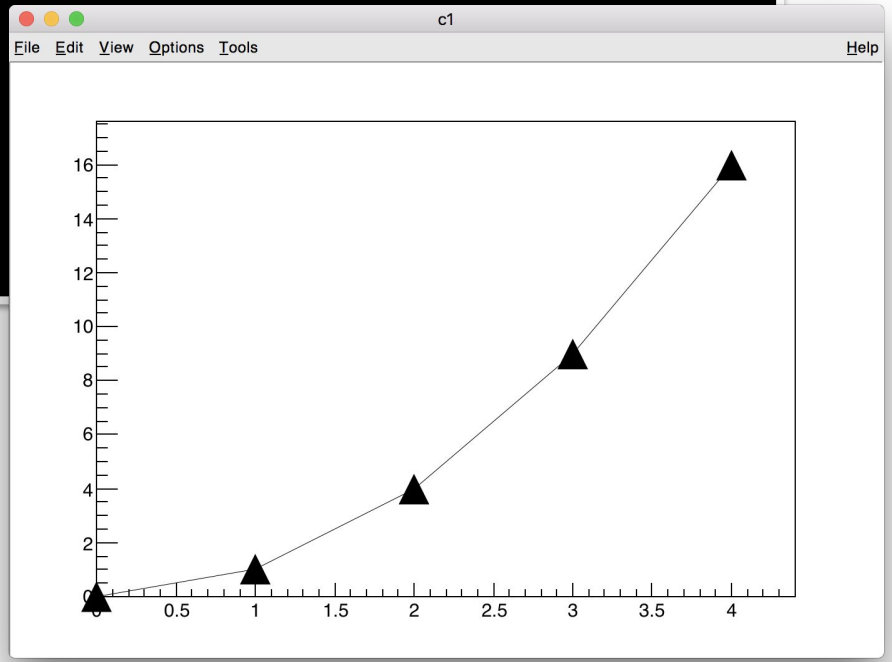
My First Graph

```
root [3] g.SetMarkerStyle(kFullTriangleUp)
```

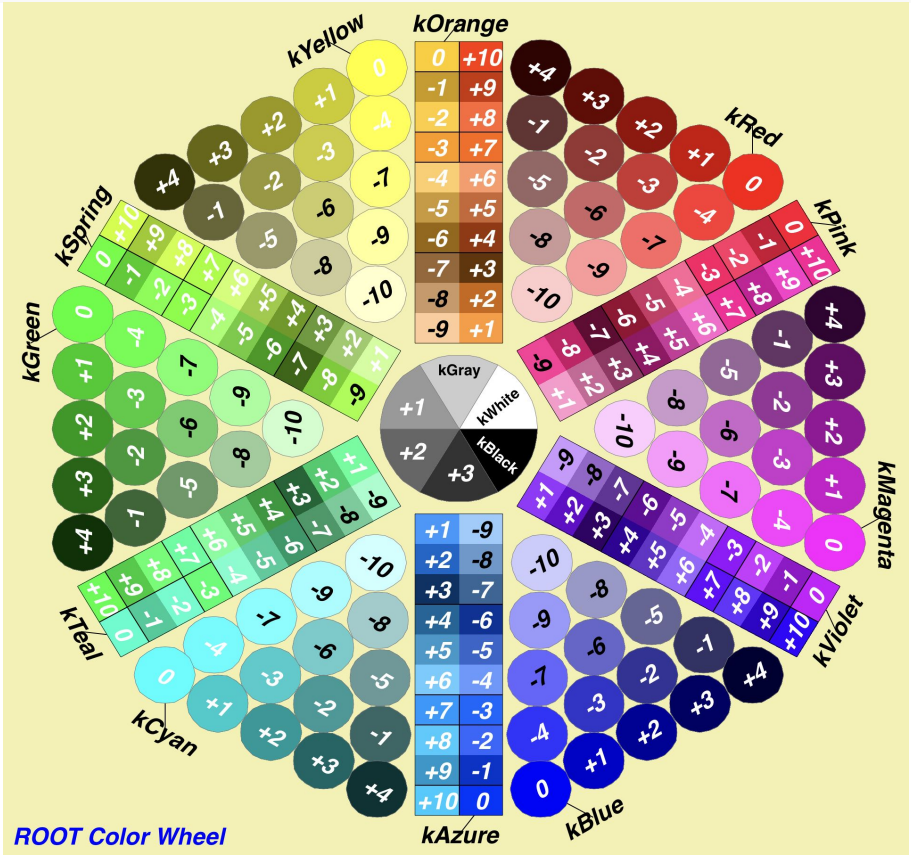


My First Graph

```
root [3] g.SetMarkerStyle(kTriangleUp)
root [4] g.SetMarkerSize(3)
```

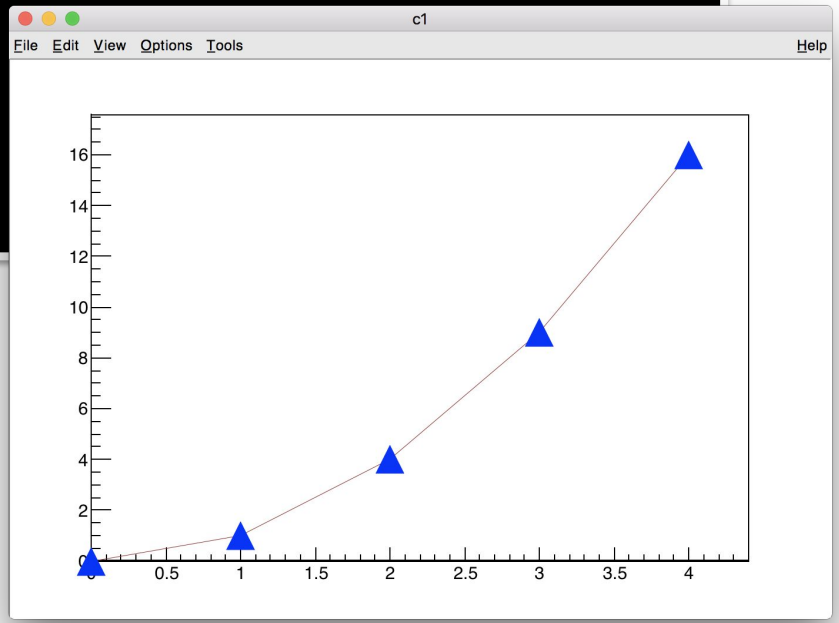


The Colors (TColorWheel)



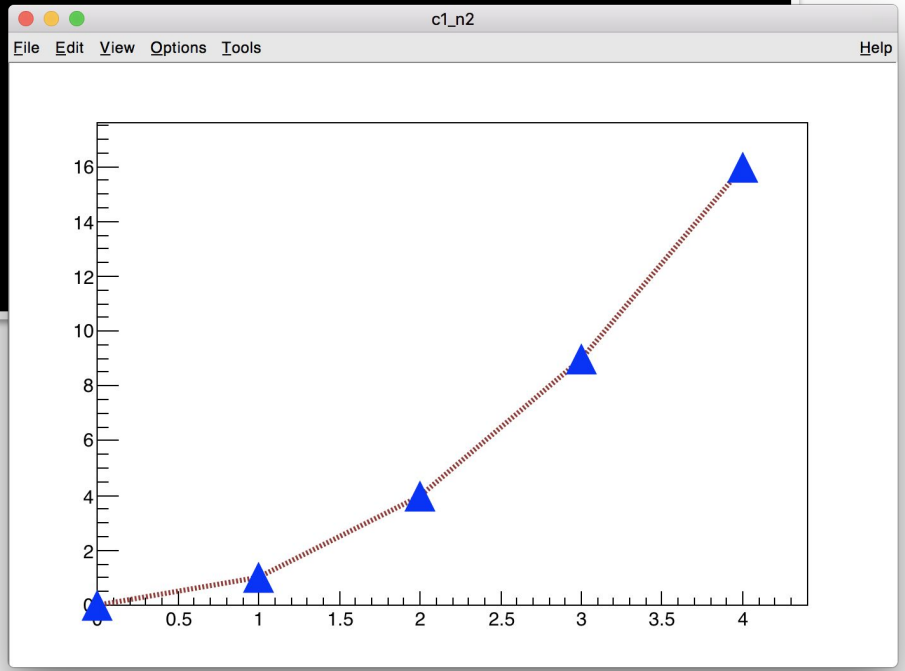
My First Graph

```
root [5] g.SetMarkerColor(kAzure)
root [6] g.SetLineColor(kRed - 2)
```



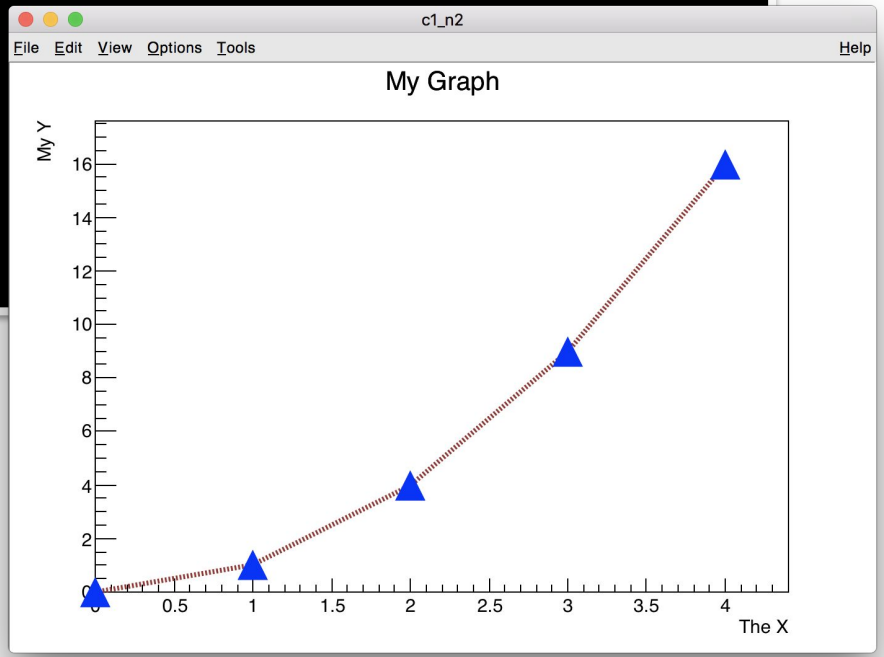
My First Graph

```
root [7] g.SetLineWidth(2)
root [8] g.SetLineStyle(3)
```



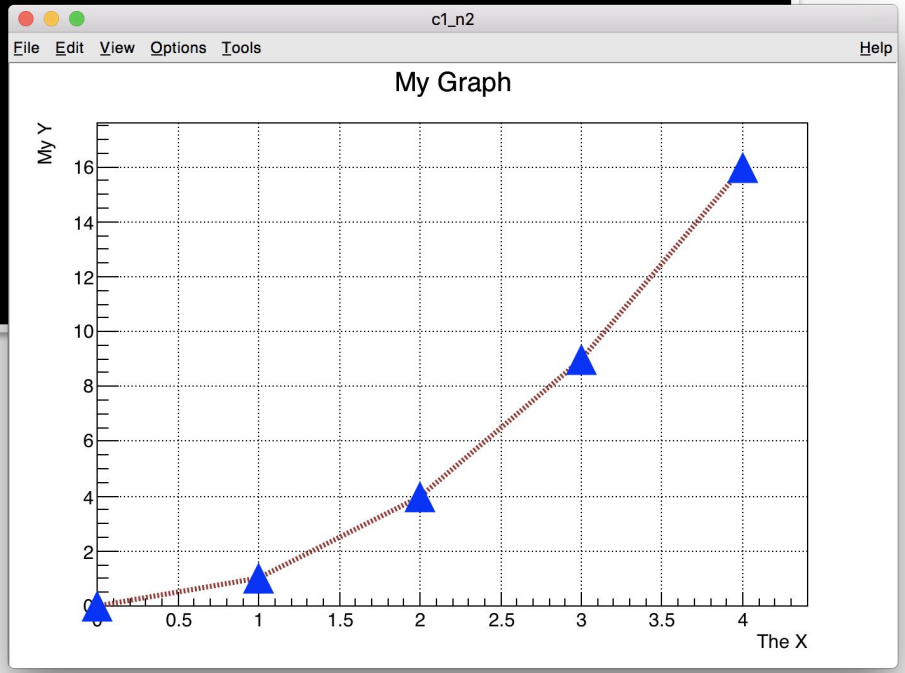
My First Graph

```
root [9] g.SetTitle("My Graph;The X;My Y")
```



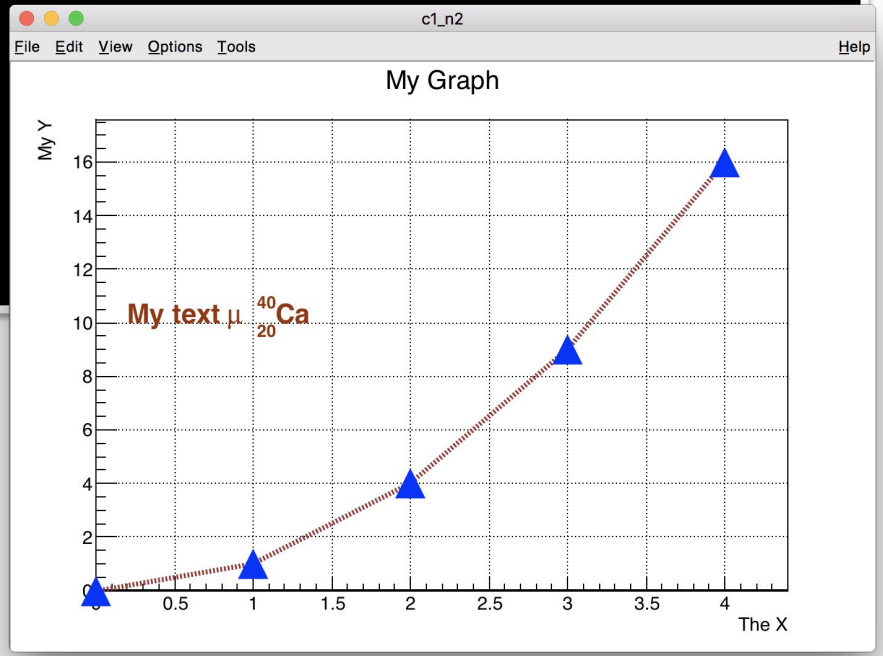
My First Graph

```
root [10] gPad->SetGrid()
```



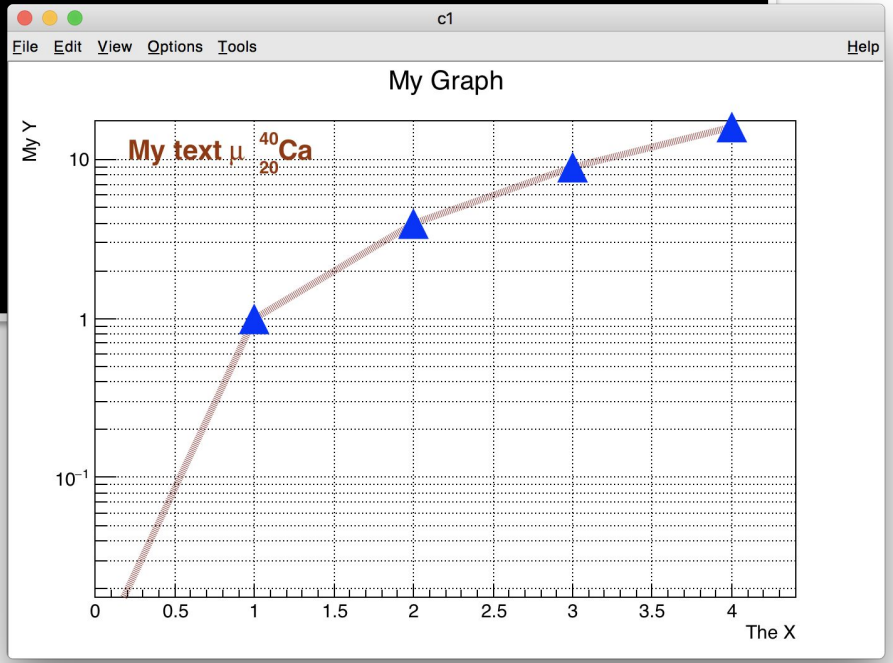
My First Graph

```
root [10] auto txt = "#color[804]{My text #mu {}^{40}_{20}Ca}"
root [11] TLatex l(.2, 10, txt)
root [12] l.Draw()
```



My First Graph

```
root [13] gPad->SetLogy();
```



myFirstGraph.C

https://github.com/root-project/training/tree/master/BasicCourse/Exercises/Graphics

Time for Exercises!

https://github.com/vincecr0ft/pyROOT_lectures/tree/master/ChapterOne_Syntax/1_TH1s_andTGraphs.ipynb

Graphics functionalities: a more structured approach

Features of a Good Plot

Every plot should be **self-contained** and deliver a **clear message**, even if extracted from the publication in which it's shown.

To achieve this goal the mandatory pieces of a plot are:

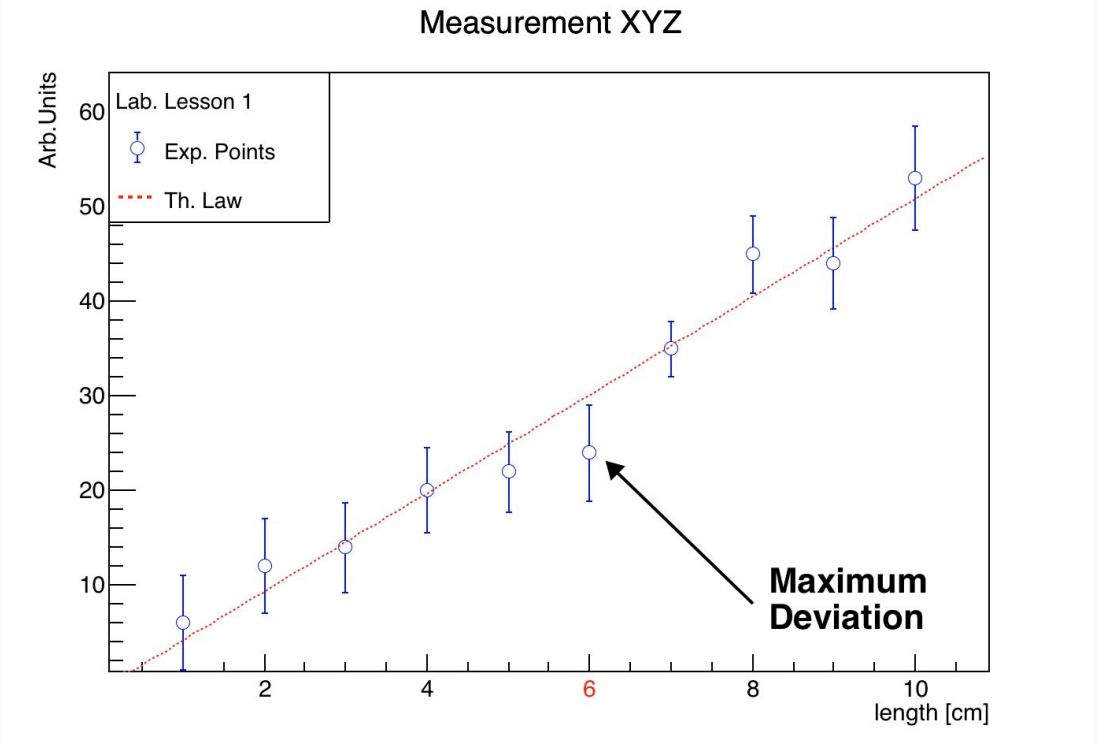
1. **The data**, of course. The best representation should be chosen to avoid any misinterpretation.
2. **The plot title** which should summarize clearly what the data are.
3. **The axis titles**. The X and Y titles should be properly titled with the variable name and its unit.
4. **The axis** themselves. They should be clearly labelled to avoid any ambiguity.
5. **The legend**. It should explain clearly the various curves on the plot.
6. **Annotations** highlighting specific details on the plot.

"Good Plot" example

The next slides will detailed how to build this plot step by step.

Each exercise will be marked as:

⇒ Something to do



The data set

```
n_points = 10
# The values along X and Y axis
x_vals = [1,2,3,4,5,6,7,8,9,10]
y_vals = [6,12,14,20,22,24,35,45,44,53]
# The errors on the Y axis
y_errs = [5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43]
```

This code creates the **data set**.

⇒ Create a macro called "macro1.py" and execute it.

The data drawing

As this graph has error bars along the Y axis an obvious choice will be to draw it as an **error bars plot**. The command to do it is:

```
graph.Draw("APE")
```

The Draw() method is invoked with three options:

- "A" the **axis** coordinates are automatically computed to fit the graph data
- "P" **points** are drawn as marker
- "E" the **error bars** are drawn

⇒ Add this command to the macro and execute it again.

⇒ Try to change the options (e.g. “APEL”, “APEC”, “APE4”).

Customizing the data drawing

Graphical attributes can be set to customize the visual aspect of the plot.
Here we change the **marker style** and **color**.

```
# Make the plot esthetically better
graph.SetMarkerStyle(ROOT.kOpenCircle)
graph.SetMarkerColor(ROOT.kBlue)
graph.SetLineColor(ROOT.kBlue)
```

⇒ Add this code to the macro. Notice the visual changes.

⇒ Play a bit with the possible values of the attributes.

Fitting and customizing the fit drawing

The data set we built up looks very linear. It could be interesting to **fit it with a line** to see what the linear law behind this data set could be.

```
# Define a linear function
f = ROOT.TF1("Linear law","[0]+x*[1]",.5,10.5)
# Let's make the function line nicer
f.SetLineColor(ROOT.kRed)
f.SetLineStyle(2)
# Fit it to the graph
graph.Fit(f)
```

The function "**f**" graphical aspect is customized the same way the graph was before.

⇒ Add this code to the macro. Execute it again.

⇒ Notice the output given by the **Fit** method.

⇒ Play with the graphical attributes for the "**f**" function.

Add a Function

The data set we built up looks very linear. It could be interesting to **compare it with a line** to see what the linear law behind this data set could be.

```
# Define a linear function
f = ROOT.F1("Linear law","[0]+x*[1]",.5,10.5)
# Let's make the function line nicer
f.SetLineColor(ROOT.kRed)
f.SetLineStyle(2)
# Set parameters
f.SetParameters(-1,5)
f.Draw("Same")
```

The function "**f**" graphical aspect is customized the same way the graph was before.

⇒ Add this code to the macro. Execute it again.

⇒ Play with the graphical attributes for the "**f**" function.

Plot Titles

The graph was created without any titles. It is time to define them. The following command **define the three titles** (separated by a ";") in one go. The format is:

"Main title ; x axis title ; y axis title"

```
graph.SetTitle("Measurement XYZ;length [cm];Arb.Units")
```

The axis titles can be also set individually:

```
graph.GetXaxis()->SetTitle("length [cm]")
graph.GetYaxis()->SetTitle("Arb.Units")
```

⇒ Add this code to the macro. You can choose one or the other way.

⇒ Play with the text. You can even try to add TLatex special characters.

Axis labelling (1)

The axis labels must be very **clear**, **unambiguous**, and **adapted to the data**.

- ROOT by default provides a powerful mechanism of **labelling optimisation.**
- Axis have three levels of divisions: Primary, Secondary, Tertiary.
- The axis labels are on the Primary divisions.
- The method `SetNdivisions` change the number of axis divisions

```
graph.GetXaxis().SetNdivisions(10, 5, 0)
```

⇒ Add this command. nothing changes because they are the default values :-)

⇒ Change the number of primary divisions to 20. Do you get 20 divisions ? Why ?

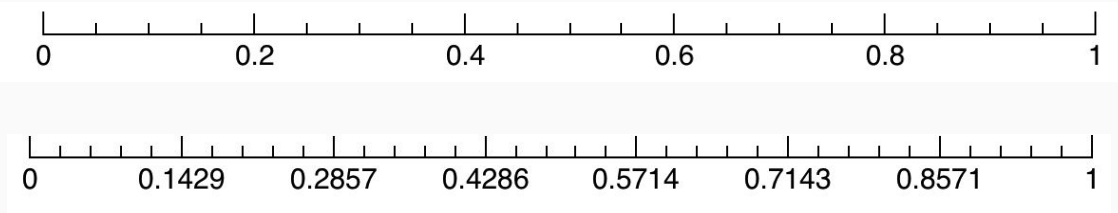
Axis labelling (2)

The number of divisions passed to the `SetNdivisions` method are **by default optimized** to get a comprehensive labelling. Which means that the value passed to `SetNDivisions` is a maximum number not the exact value we will get.

To turn off the optimisation the fourth parameter of `SetNDivisions` to `kFALSE`

⇒ try it

As an example, the two following axis both have been made with **seven** primary divisions. The first one is optimized and the second one is not. The second one is not really clear !!



Nevertheless, in some cases, it is useful to have non optimized labelling.

Other kinds of axis

In addition to the normal linear numeric axis axis ROOT provides also:

- **Alphanumeric labelled axis**. They are produced when a histogram with alphanumeric labels is plotted.
- **Logarithmic axis**. Set with `gPad.SetLogx()` ⇒ try it (`gPad.SetLogy()` for the Y axis)
- **Time axis**.

Fine tuning of axis labels

When a specific **fine tuning** of an individual label is required, for instance changing its color, its font, its angle or even changing the label itself the method **`ChangeLabel`** can be used on any axis.

⇒ For instance, in our example, imagine we would like to highlight more the X label with the maximum deviation by putting it in red. It is enough to do:

```
graph.GetXaxis().ChangeLabel(3,-1,-1,-1,ROOT.kRed)
```

Legend (1)

ROOT provides a powerful class to build a legend: **`TLegend`**. In our example the legend is build as follow:

```
legend = ROOT.TLegend(.1,.7,.3,.9,"Lab. Lesson 1")
legend.AddEntry(graph,"Exp. Points","PE")
legend.AddEntry(f,"Th. Law","L")
legend.Draw()
```

⇒ Add this code, try it and play with the options

- The `TLegend` constructor defines the **legend position and its title**.
- Each legend item is added with method **`AddEntry`** which specify
  - **an object to be part of the legend** (by name or by pointer),
  - the **text of the legend**
  - the **graphics attributes to be legended**. "L" for TAttLine, "P" for TAttMarker, "E" for error bars etc ...

Legend (2)

ROOT also provides an **automatic way to produce a legend** from the graphics objects present in the pad. The method **`TPad::BuildLegend()`**.

⇒ In our example try **`ROOT.gPad.BuildLegend()`**

This is a quick way to proceed but for a plot ready for publication it is recommended to use the method previously described.

⇒ Keep only the legend defined in the previous slide

Annotations (1)

Often the various parts of a plot we already saw are not enough to convey the complete message this plot should. Additional **annotations** elements are needed **to complete and reinforce the message** the plot should give.

ROOT provides a collection of basic graphics primitives allowing to draw such annotations. Here a non exhaustive list:

- `TText` and `TLatex` to draw text.
- `TArrow` to draw all kinds of arrows
- `TBox`, `TEllipse` to draw boxes and ellipses.
- Etc ...

Annotations (2)

In our example we added an annotation pointing the "maximum deviation". It consists of a simple arrow and a text:

```
# Draw an arrow on the canvas
arrow = ROOT.TArrow(8,8,6.2,23,0.02,"|>")
arrow.ROOT.SetLineWidth(2)
arrow.Draw()

# Add some text to the plot
text = ROOT.TLatex(8.2,7.5,"#splitline{Maximum}{Deviation}")
text.Draw()
```

⇒ Try this code adding it in the macro.

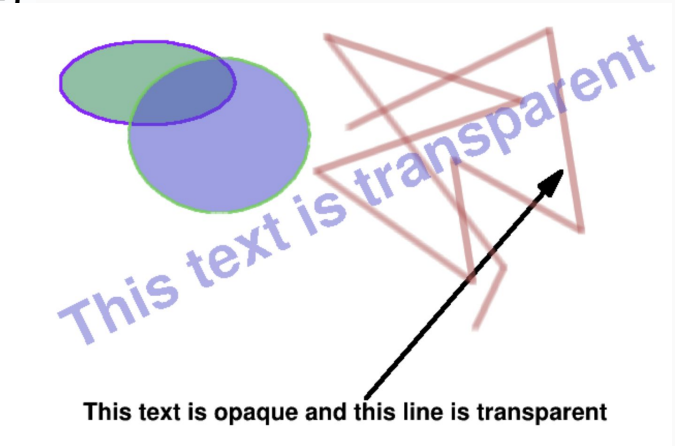
⇒ Notice changing the canvas size does not affect the pointed position.

Graphics Styles

- ROOT provides **"graphics styles"** to define the **general look** of plots.
- The **current style** can be accessed via a global pointer : **`gStyle`**.
- The styles are managed by the class TStyle.
- ROOT users can define their own style. For example an experiment can have its own style to make sure the plots produced by its scientists have the same look.
- There is a series of predefined style "Plain", "Bold", "Pub, "Modern" etc... the default style is "Modern".

Colors

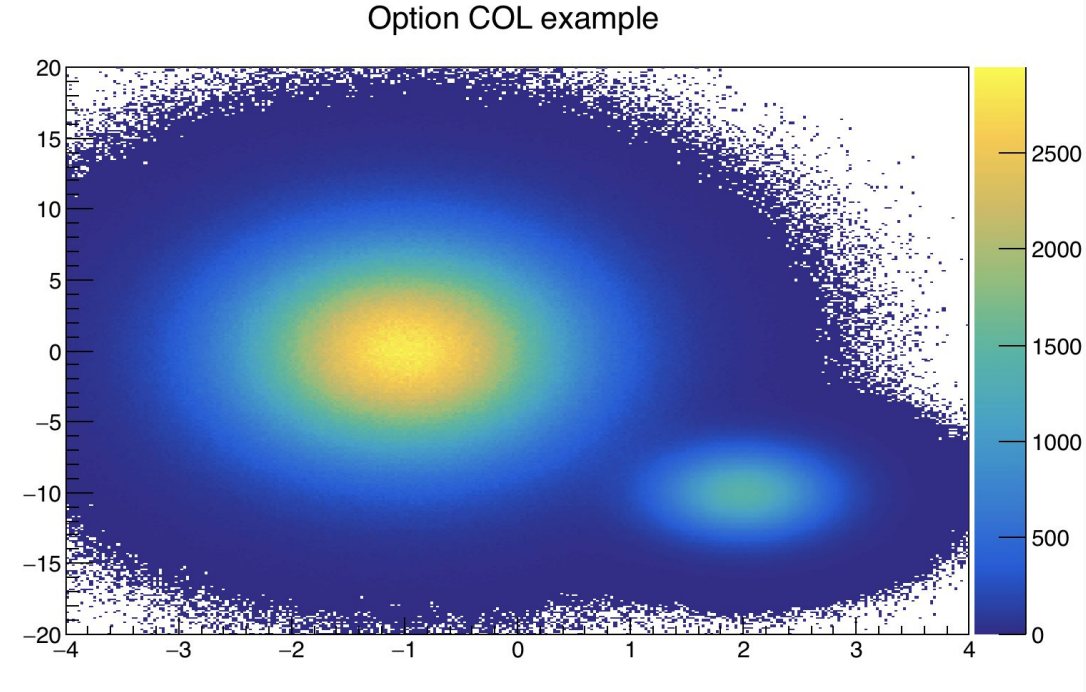
- **The transparency** is set via similar setters but with the keyword "Alpha" at the end. For instance `histo.SetFillColorAlpha(ROOT.kBlue,0.35)` will set the color of the histogram histo to red with and alpha channel equal to 0.35 (0 = fully transparent and 1 = fully opaque)

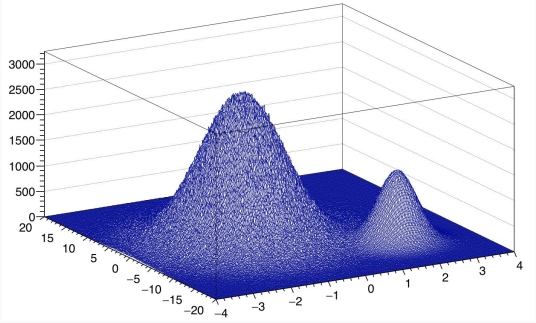


Note: The transparency is available on all platforms when the flag `OpenGL.CanvasPreferGL` is set to 1 in `$ROOTSYS/etc/system.rootrc`, or on Mac with the Cocoa backend. On the file output it is visible with PDF, PNG, Gif, JPEG, SVG ...

2D plots with color map (2)

The previous macro generates the following output which is a plot with two smooth 2D gaussian:

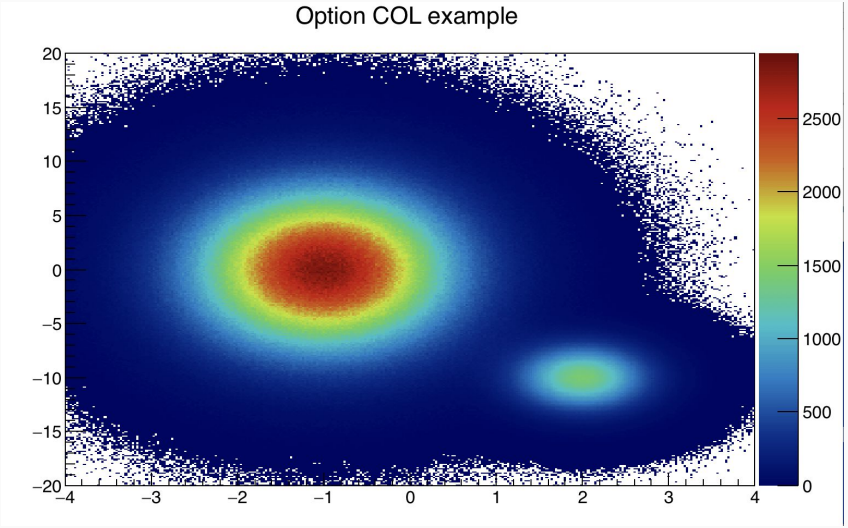




The default ROOT color map (kBird) render perfectly the smoothness of the dataset.

Danger of the Rainbow color map (1)

If instead of the default color maps we use the Rainbow one we get:

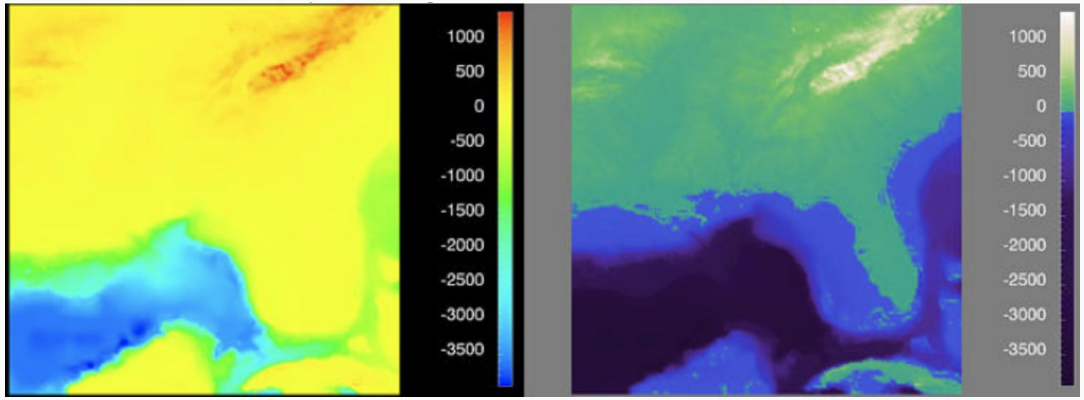


The reasons why this is not a good is explained in detail here.

For instance immediately see some "structure" appearing on the plot which does not exist in the data: just look at this yellow circle.

Danger of the Rainbow color map (2)

Another example taken from the article mentioned earlier illustrates even better why the Rainbow color map should be avoided.



These two panels show the same data, but with different colormaps. On the left, the "Rainbow" colormap provides a very colorful and vibrant image, however, it masks significant features in the data, and emphasizes less important ones.

Danger of the RainBow color map (3)

A Quick Visual Method for Evaluating color maps has been exposed in the "The 'Which Blair Project:" (Rogowitz, Bernice and Alan Kalvin)

The idea is to use a well know photography of a face presented with different colormaps. The colormaps distorting the image are not good. Clearly the Rainbow one (on the right) distorts the image !

The ROOT standard color maps (1)

ROOT provides 62 color maps (including Rainbow because people like it). Most are monotonic, but several may have quite small luminance variation from max to min.

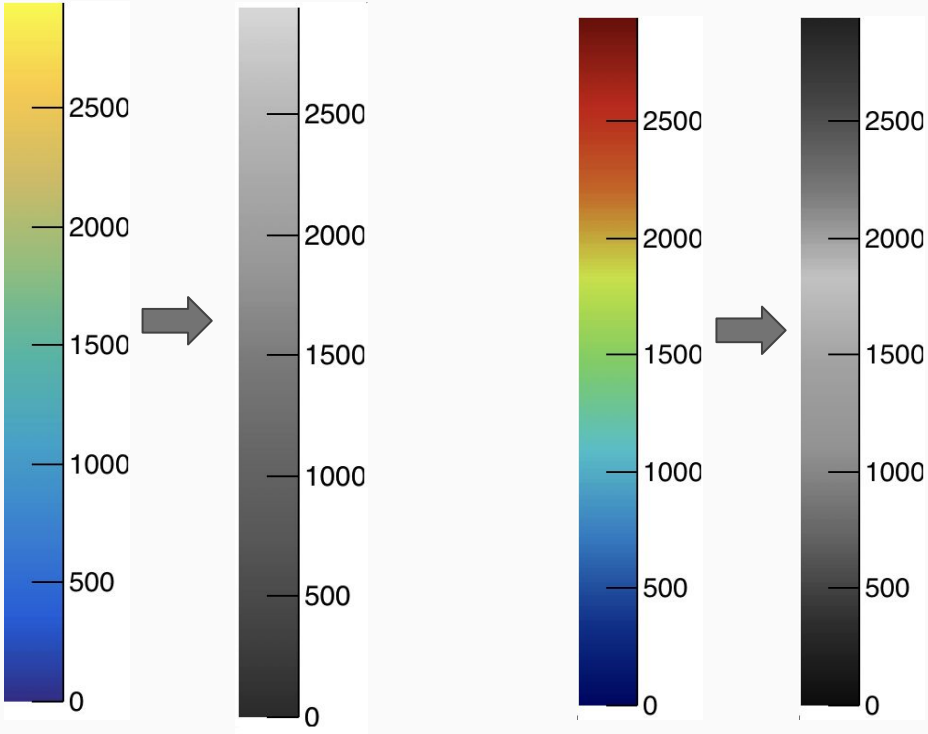
Displaying the grayscale equivalents of a color map may help people selecting ones having a large and monotonic luminance ranges.

It might be interesting to look at the ROOT color scales in this way. To do that it is enough to turn the grayscale mode on before drawing the histogram:

```
canvas.SetGrayscale();
```

The ROOT standard color maps (2)

Here we show the grayscale equivalent of the two color maps we used before:



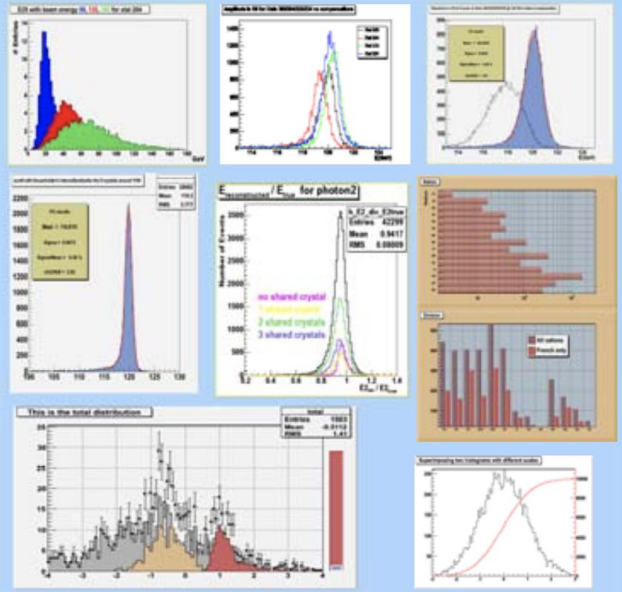
This is another good test showing the validity of a color map. In gray scale the Rainbow color map shows its luminance issue as the minimum and maximum values appear the same.

ROOT visualization techniques

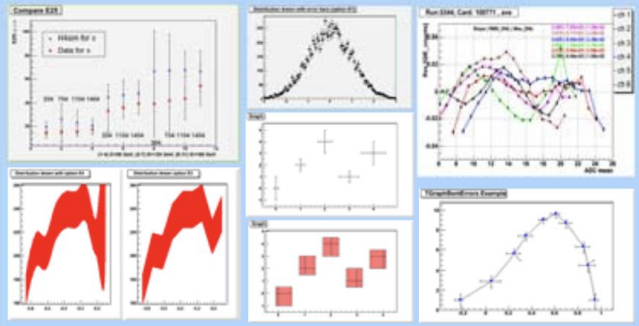
In this "*Introduction to the ROOT graphics capabilities*" we have seen a small fraction of the ROOT graphics capabilities. In the next four slides we will present a quick snapshot of what ROOT provides to visualise 2, 3, 4 and N variables data set.

2 variables visualization techniques

2 variables visualization techniques are used to display Trees, Ntuples, 1D histograms, functions y=f(x), graphs …

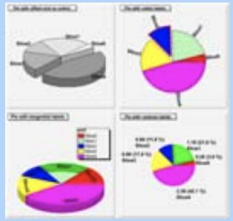


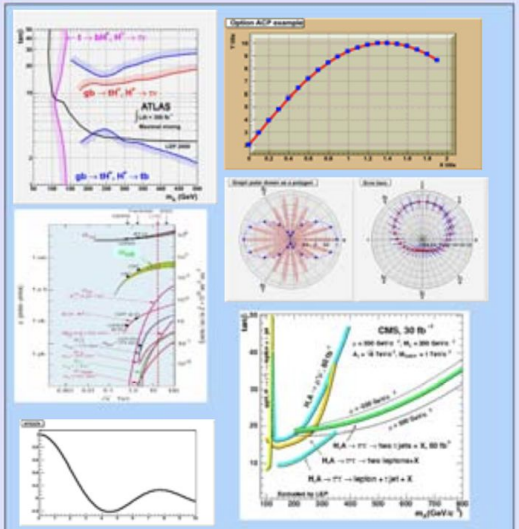
Bar charts and lines are a common way to represent 1D histograms.



Errors can be represented as bars, band, rectangles. They can be symmetric, asymmetric or bent. 1D histograms and graphs can be drawn that way.

Pie charts can be used to visualize 1D histograms. They also can be created from a simple mono dimensional vector.

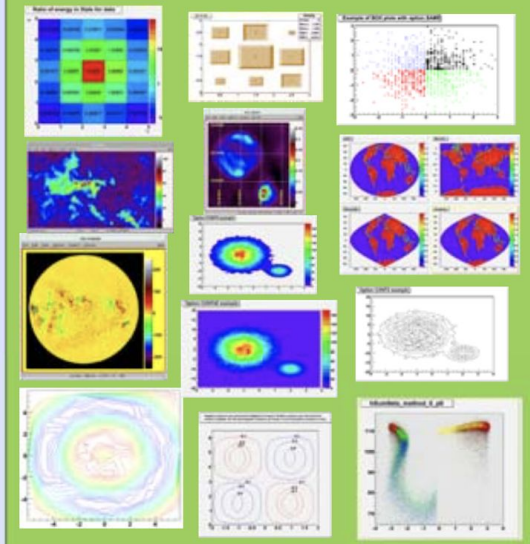




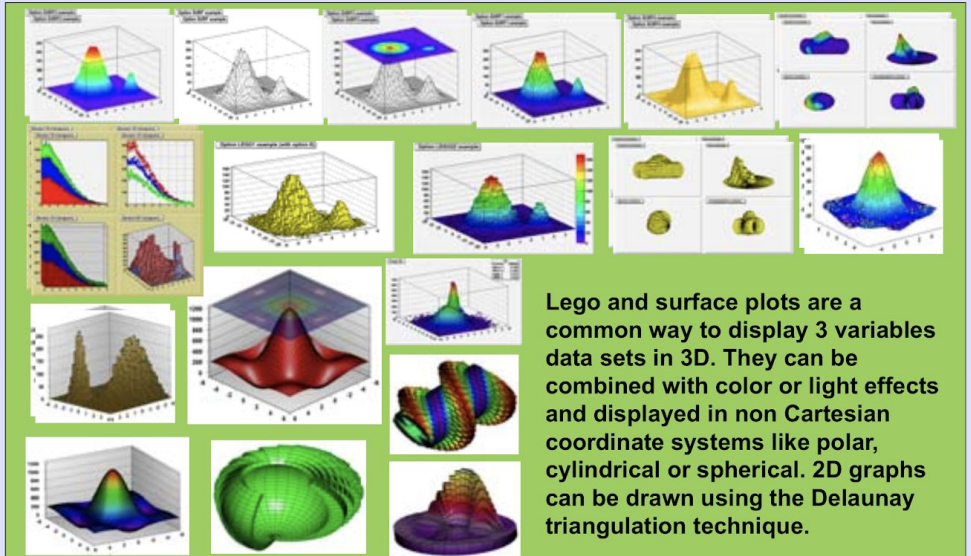
Graphs can be drawn as simple lines, like functions. They can also visualize exclusion zones or be plotted in polar coordinates.

3 variables visualization techniques

3 variables visualization techniques are used to display Trees, Ntuples, 2D histograms, 2D Graphs, 2D functions …



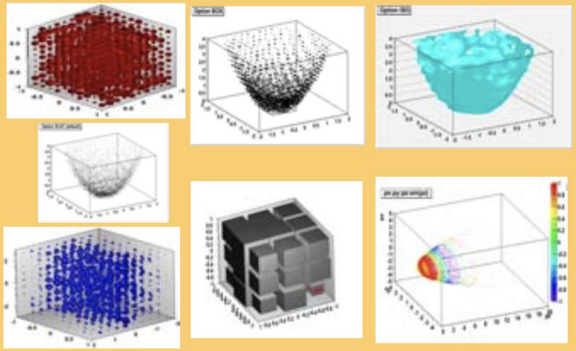
Several techniques are available to visualize 3 variables data sets in 2D. Two variables are mapped on the X and Y axis and the 3rd one on some graphical attributes like the color or the size of a box, a density of points (scatter plot) or simply by writing the value of the bin content. The 3rd variable can also be represented using contour plots. Some special projections (like Aitoff) are available to display such contours.



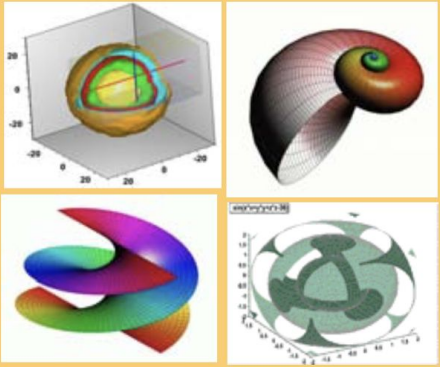
Lego and surface plots are a common way to display 3 variables data sets in 3D. They can be combined with color or light effects and displayed in non Cartesian coordinate systems like polar, cylindrical or spherical. 2D graphs can be drawn using the Delaunay triangulation technique.

4 variables visualization techniques

4 variables visualization techniques are used to display Trees, Ntuples, 3D histograms, 3D functions …



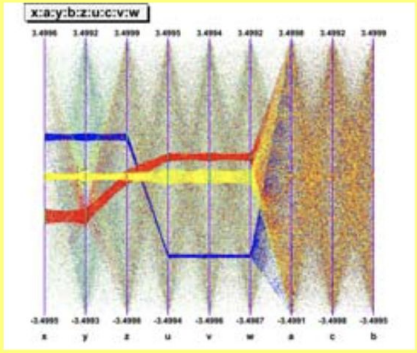
The 4 variables data set representations are extrapolations of the 3 variables ones. Rectangles become boxes or spheres, contour plots become iso-surfaces. The scatter plots (density plots) are drawn in boxes instead of rectangles. The 4th variable can also be mapped on colors. The use of OpenGL allows to enhance the plots' quality and the interactivity.



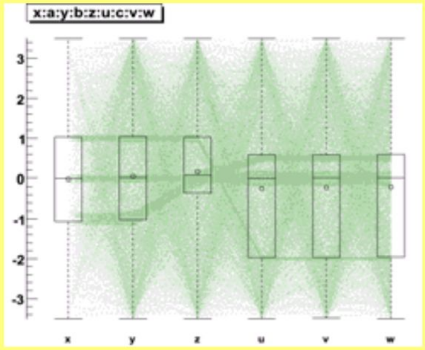
Functions like t= f(x,y,z) and 3D histograms are 4 variables objects. ROOT can render using OpenGL. It allows to enhance the plots' quality and the interactivity. Cutting planes, projection and zoom allow to better understand the data set or function.

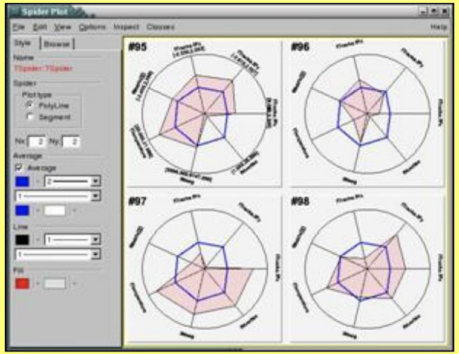
N variables visualization techniques

N variables visualization techniques are used to display Trees and Ntuples …



Above 4 variables more specific visualization techniques are required; ROOT provides three of them. The parallel coordinates (on the left) the candle plots (in the middle) which can be combined with the parallel coordinates. And the spider plot (on the right also). These three techniques, and in particular the parallel coordinates, require a high level of interactivity to be fully efficient.

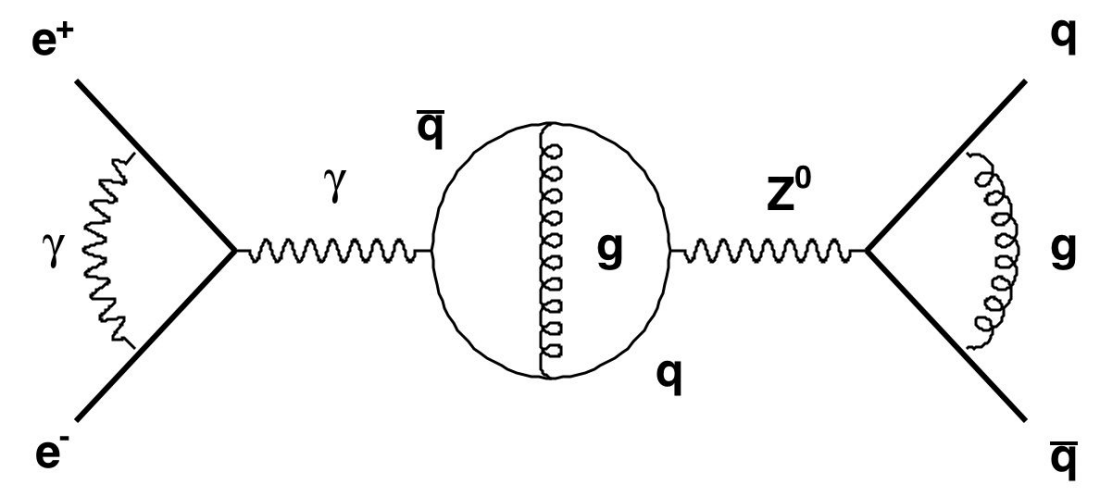




One more thing: Feynman diagrams

In theoretical physics, Feynman diagrams are pictorial representations of the mathematical expressions describing the behavior of subatomic particles.

These diagrams can also be produced by ROOT thanks to a serie of specific graphics primitives as shown in this example.



Take Home Messages

Every plot should be self-contained and deliver a clear message, even if extracted from the publication in which it's shown.

ROOT provides all the tools to make "publication ready" plots but as any tools it can be misused and some basic rules should be kept in mind.

ROOT graphics allow **great flexibility** thanks to the closeness of the data manipulated and their graphical representation.

Time for Exercises!

https://github.com/vincecr0ft/pyROOT_lectures/tree/master/ChapterOne_Syntax/2_Graphics.ipynb

Interlude: Random Number Generation

Pseudo Random Number Generation

- Crucial for HEP and science
  - E.g. Simulation, statistics
- Used in our examples
- Achieved with the TRandomX class
  - TRandom1, TRandom2, TRandom3
- Collection of algorithms available

- **TRandom1** 82 ns/call
- **TRandom2** 7 ns/call
- **TRandom3** 5 ns/call
- **TRandomMixMax** 6 ns/call
- **TRandomMixMax17** 6 ns/call
- **TRandomMixMax256** 10 ns/call
- **TRandomMT64** 9 ns/call
- **TRandomRanlux48** 270 ns/call

Generate Numbers in a Nutshell

```
root [0] TRandom3 r(1); // Marsenne-Twister generator, seed 1
root [1] r.Uniform(-3, 19)
(double) 6.17448
root [2] r.Gaus(2, 3)
(double) 4.9651
root [3] r.Exp(12)
(double) 3.93664
root [4] r.Poisson(1.7)
(int) 1
```

- Use seed to control random sequence: same seed, same sequence.
- Fundamental for reproducibility

PyROOT: The ROOT Python Bindings

Binding Python and C++

- C and C++ needed for performance-critical code
- Python enables faster development
- Ideally, we should combine them
  - Python as an interface to C++ functionality
- There are ways to invoke C and C++ from Python
  - ctypes, boost, swig
  - Cumbersome, wrappers needed, lots of work
- **In ROOT: PyROOT**

PyROOT

- Python bindings for ROOT
- Access all the ROOT C++ functionality from Python
- Dynamic, automatic
  - Include header, load library -> start interactive usage
- Communication between Python and C++ is powered by the ROOT type system
- "Pythonisations" for specific cases

Interrogating the Type System

```
auto c = TClass::GetClass("myClass")
```

- Fetch the *myClass* representation of ROOT

```
auto ms = c->GetListOfMethods()
```

- Get the methods of the class (appreciate how this is impossible in C++!)

```
auto listOfXXX = gROOT->GetListOfXXX()
```

- Same for collections of many other entities

Using PyROOT

- Entry point to use ROOT from within Python

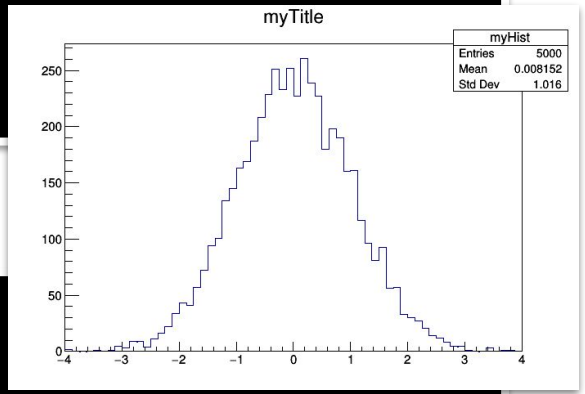
```
import ROOT
```

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

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

Example: C++ to Python

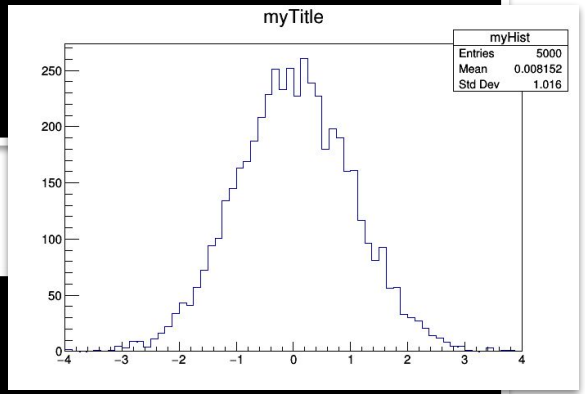
```
> root
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
```




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

Example: C++ to Python

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



also with
individual import

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

Dynamic C++ (JITting)

```
import ROOT
cpp_code = """
int f(int i) { return i*i; }
class A {
public:
  A() { cout << "Hello PyROOT!" << endl; }
};
"""
# Inject the code in the ROOT interpreter
ROOT.gInterpreter.ProcessLine(cpp_code)

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

C++ code we want to invoke from Python

Dynamic C++ (JITting)

my_cpp_library.h

```
int f(int i) { return i*i; }

class A {
public:
  A() { cout << "Hello PyROOT!" << endl; }
};
```

my_python_module.py

```
# Make the header known to the interpreter
ROOT.gInterpreter.ProcessLine('#include "my_cpp_library.h"')

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

Dynamic Library Loading

```
int f(int i);

class A {
public:
  A();
};
```

my_cpp_library.h

```
#include "my_cpp_library.h"

int f(int i) { return i*i; }

A::A() { cout << "Hello PyROOT!" << endl; }
```

my_cpp_library.cpp

my_cpp_library.so

my_python_module.py

```
# Load a C++ library
ROOT.gInterpreter.ProcessLine('#include "my_cpp_library.h"')
ROOT.gSystem.Load('./my_cpp_library.so')

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

Exercise Dynamic JITting

- Define a C++ function that counts the characters in an std::string (hint: use std::string::size)
- Make that function known to the ROOT interpreter in a Python module
- Invoke the function via PyROOT

- *Extra*: practise the three scenarios described
  - JITted string
  - JITted header
  - Library loading

Time for Exercises!

https://github.com/vincecr0ft/pyROOT_lectures/tree/master/ChapterOne_Syntax/3_DynamicCpp.ipynb

The ROOTBooks

The Jupyter Notebook

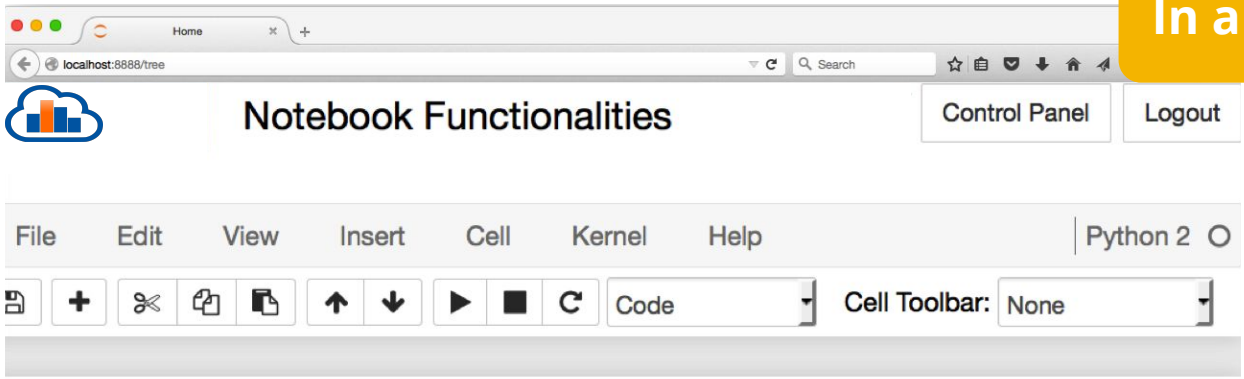
A web-based interactive computing platform that combines code, equations, text and visualisations.

Many supported languages: Python, Haskell, Julia… One generally speaks about a “kernel” for a specific language

In a nutshell: an “interactive shell opened within the browser”

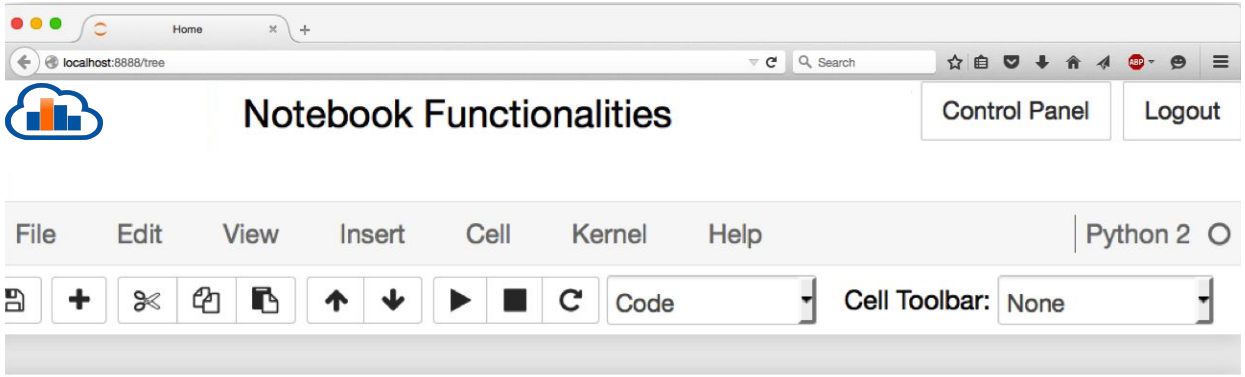
In the past also called:
iPython Notebooks

http://www.jupyter.org



In a browser

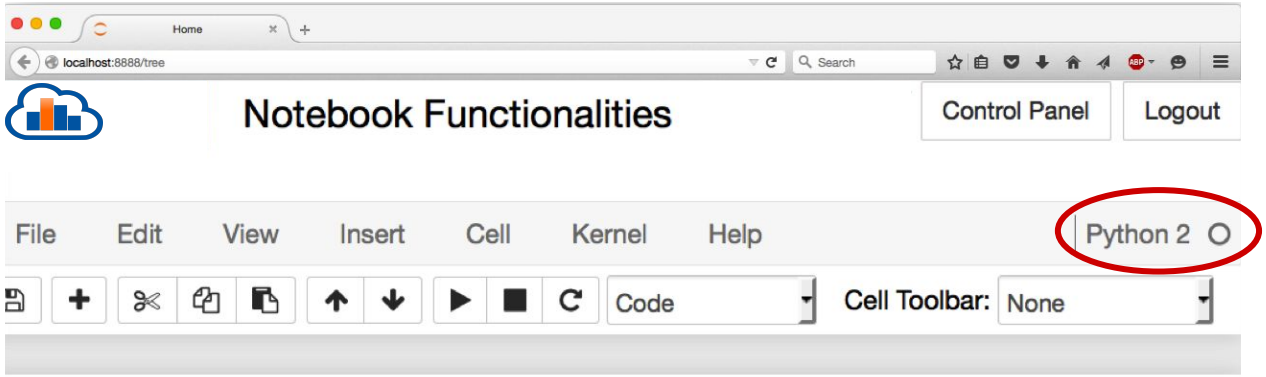
Text & Formulas



Welcome to the Notebook Technology

This is a markdown cell. You can add LaTex code: $\sum_{n=-\infty}^{\infty} |x(n)|^2$

In [1]:
```
def thisFunction():
    return 42
```



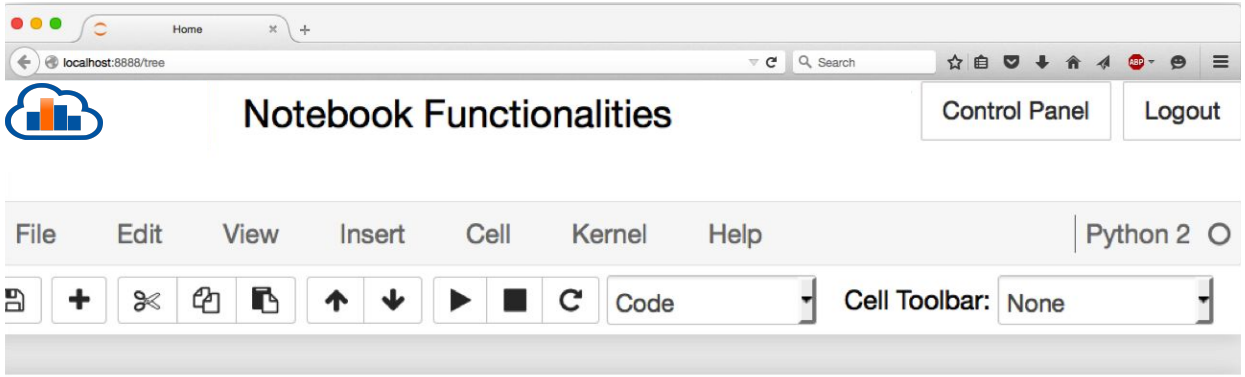
Welcome to the Notebook Technology

This is a markdown cell. You can add LaTex code: $\sum_{n=-\infty}^{\infty} |x(n)|^2$

```
In [1]: def thisFunction():
            return 42
```

Code

A *Python* notebook



Welcome to the Notebook Technology

This is a markdown cell. You can add LaTex code: $\sum_{n=-\infty}^{\infty} |x(n)|^2$

```
In [1]: def thisFunction():
            return 42

In [2]: thisFunction()

Out[2]: 42
```

Code

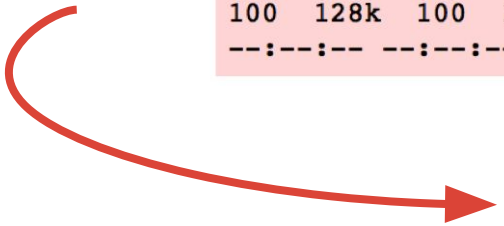
```
In [1]: def thisFunction():
            return 42

In [2]: thisFunction()

Out[2]: 42

In [3]: %%bash
        curl rootaasdemo.web.cern.ch/rootaasdemo/SaasFee.jpg \
        > SF.jpg
```

Invoke shell commands ...

Shell commands

```
In [1]: def thisFunction():
            return 42

In [2]: thisFunction()

Out[2]: 42

In [3]: %%bash
        curl rootaasdemo.web.cern.ch/rootaasdemo/SaasFee.jpg \
        > SF.jpg
```

```
  % Total    % Received % Xferd  Average Speed   Time
  Time     Time  Current
                                 Dload  Upload   Total
  Spent    Left  Speed
100  128k  100  128k    0     0  2731k      0 --:--:--
--:--:-- --:--:-- 2787k
```



... and get their output

Shell commands

In [1]:
```
def thisFunction():
    return 42
```

In [2]:
```
thisFunction()
```

Out[2]:
```
42
```

In [3]:
```
%%bash
curl rootaasdemo.web.cern.ch/rootaasdemo/SaasFee.jpg \
> SF.jpg
```

```
  % Total    % Received % Xferd  Average Speed   Time
  Time     Time  Current
                                 Dload  Upload   Total
  Spent    Left  Speed
100  128k  100  128k    0     0  2731k      0 --:--:--
--:--:-- --:--:-- 2787k
```

In [4]:
```
from IPython.display import Image
Image(filename="./SF.jpg",width=225)
```

In [1]:
```
def thisFunction():
    return 42
```

In [2]:
```
thisFunction()
```

Out[2]:
```
42
```

In [3]:
```
%%bash
curl rootaasdemo.web.cern.ch/rootaasdemo/SaasFee.jpg \
> SF.jpg
```

```
  % Total    % Received % Xferd  Average Speed   Time
  Time     Time  Current
                                 Dload  Upload   Total
  Spent    Left  Speed
100  128k  100  128k    0     0  2731k      0 --:--:--
--:--:-- --:--:-- 2787k
```

In [4]:
```
from IPython.display import Image
Image(filename="./SF.jpg",width=225)
```

Out[4]:

Figures

No excuse not to document your analysis !!

In a browser

Text & Formulas

Code

Shell commands

Figures

```
In [2]: thisFunction()

Out[2]: 42

In [3]: %%bash
        curl rootaasdemo.web.cern.ch/rootaasdemo/SaasFee.jpg \
        > SF.jpg

  % Total    % Re                    ge Speed    Time
  Time     Time                                 Upload   Total

  Spent    Left  Speed
100  128k  100  128k    0     0  2731k      0 --:--:--
--:--:-- --:--:-- 2787k

.display import Image
me="./SF.jpg",width=225)

Out[4]:
```

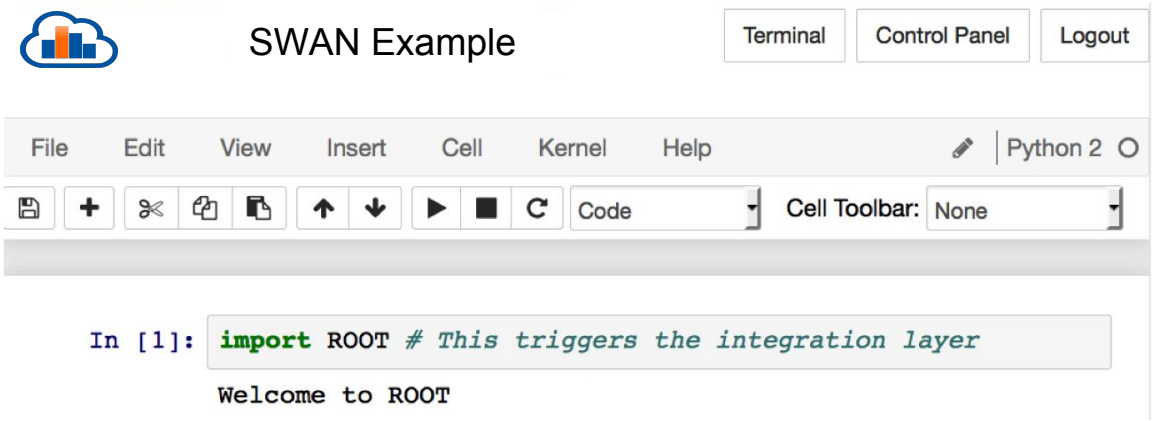
ROOTBooks

Integrate **ROOT** and note**books** - Goals:

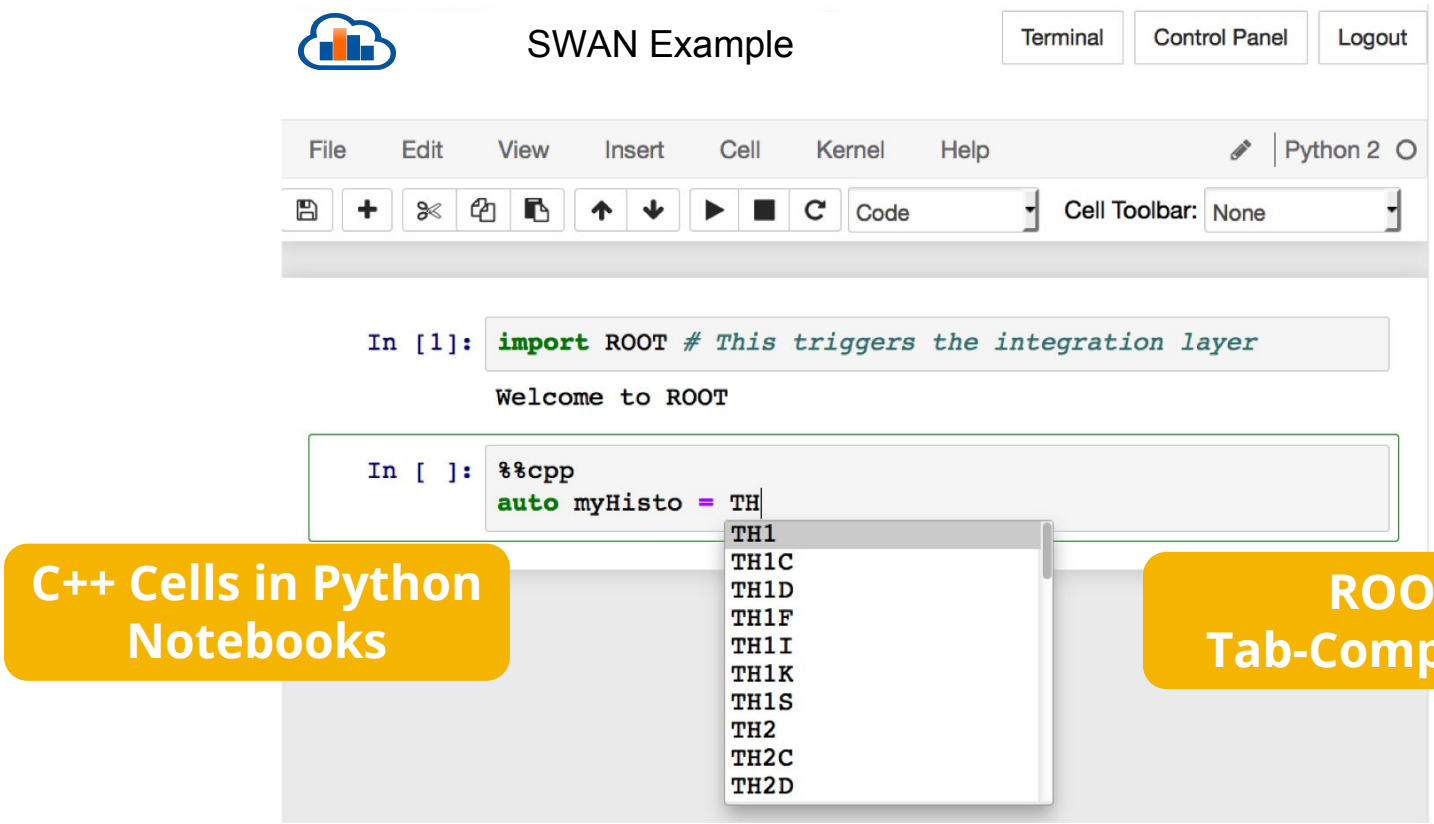
- Complement macros and command line prompts
- Encourage complete documentation
- Interoperability with other tools
  - E.g. from the Python ecosystem

Some Goodies

SWAN Example



Some Goodies

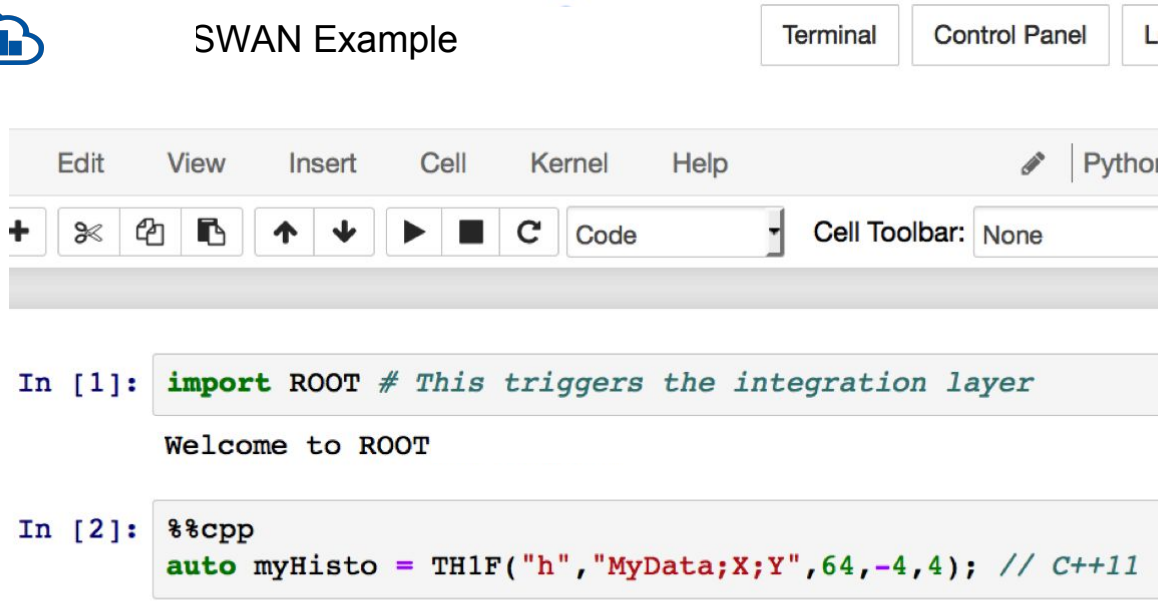


C++ Cells in Python Notebooks

ROOT Tab-Completion

Some Goodies

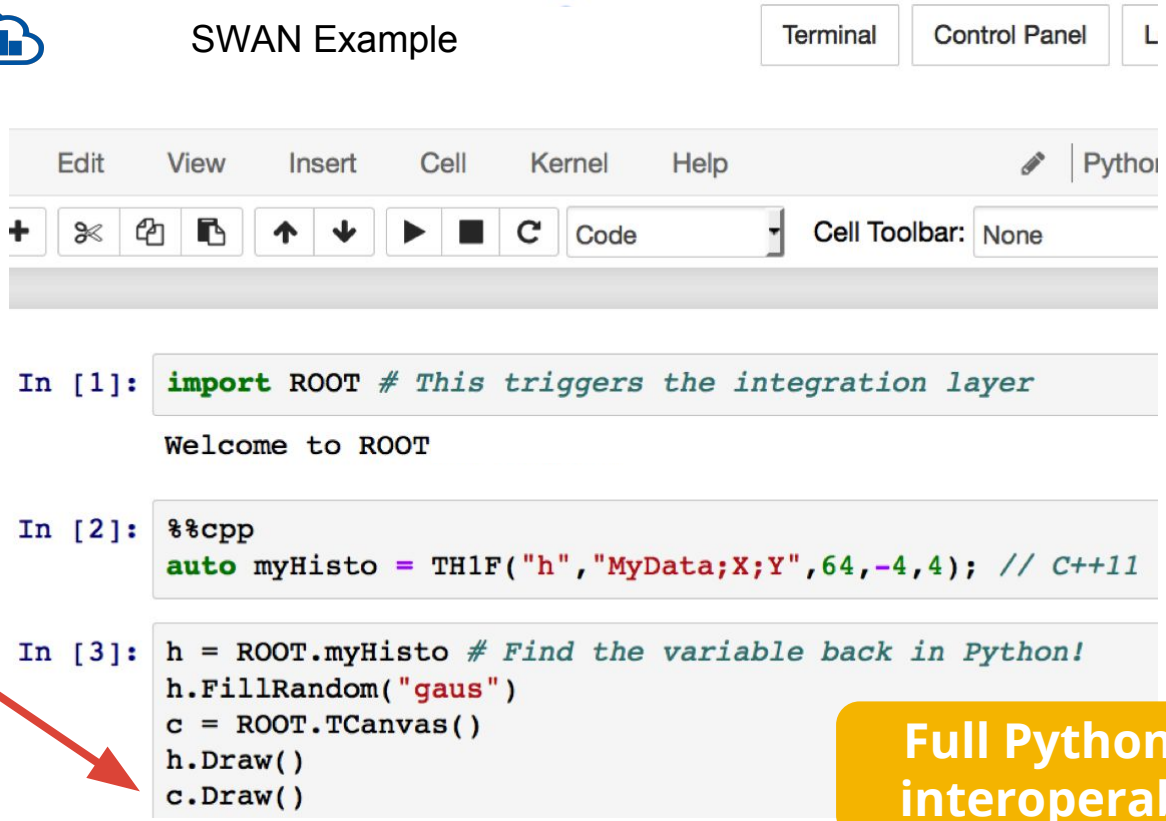
SWAN Example



I really wanted to show modern C++...

Some Goodies

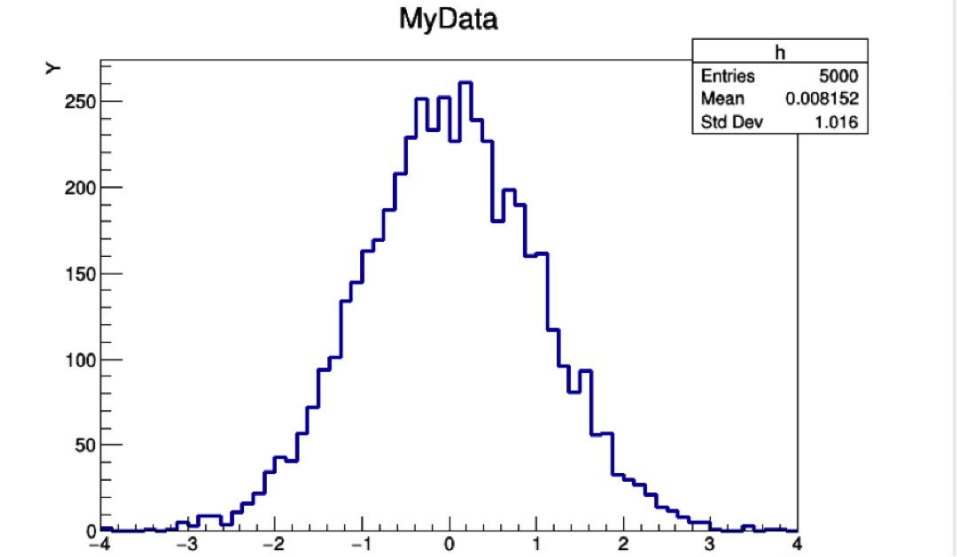
SWAN Example



Full Python-C++ interoperability

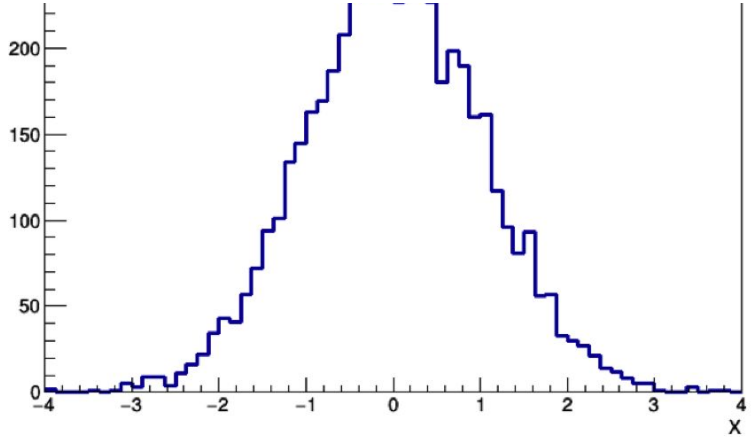
Some Goodies

```
c.Draw()
```



Seamless display of graphics

Some Goodies



In [4]:
```
%%cpp -d
double myG(double* x, double* par){
  auto res = (x[0]-par[1])/par[2];
  auto e = -.5 * res * res;
  return par[0] * exp(e); // declare function
}
```

Syntax highlighting

Some Goodies

In [4]:
```
%%cpp -d
double myG(double* x, double* par){
  auto res = (x[0]-par[1])/par[2];
  auto e = -.5 * res * res;
  return par[0] * exp(e); // declare function
}
```

In [5]:
```
f = ROOT.TF1("myGf",ROOT.myG,-5,5,3)
f.SetParameters(200,0,1);f.SetParNames("N","mu","sigma")
fr = ROOT.h.Fit(f,"S") # Capture printouts
```

Some Goodies

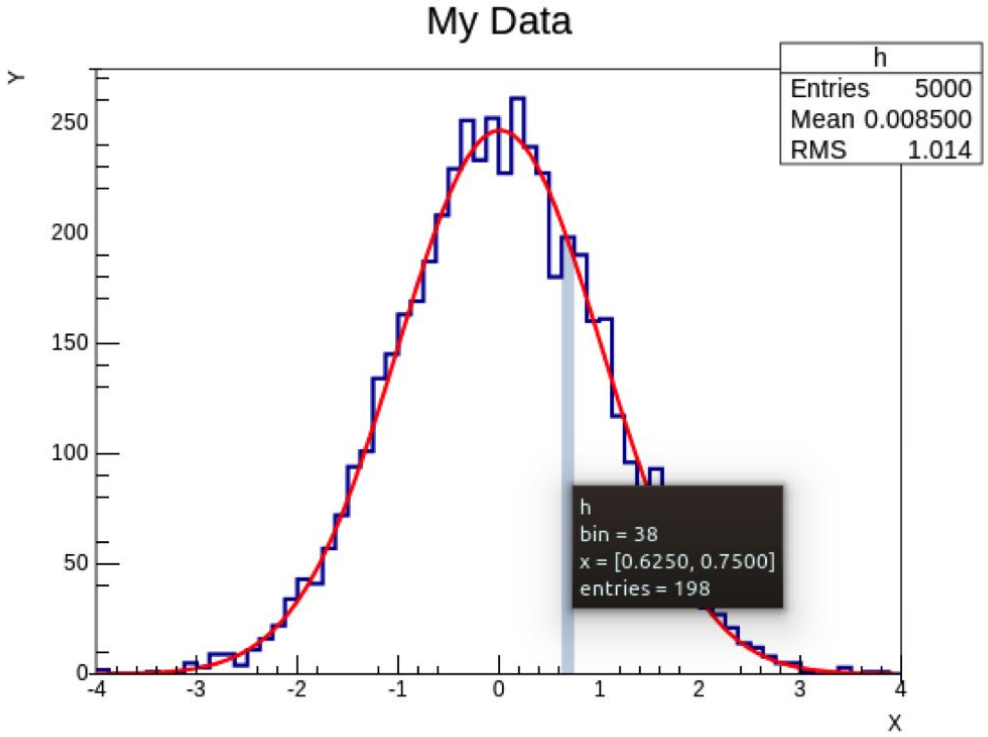
In [4]:
```
%%cpp -d
double myG(double* x, double* par){
  auto res = (x[0]-par[1])/par[2];
  auto e = -.5 * res * res;
  return par[0] * exp(e); // declare function
}
```

In [5]:
```
f = ROOT.TF1("myGf",ROOT.myG,-5,5,3)
f.SetParameters(200,0,1);f.SetParNames("N","mu","sigma")
fr = ROOT.h.Fit(f,"S") # Capture printouts
```

```
 FCN=47.4997 FROM MIGRAD    STATUS=CONVERGED      69 CALLS          70 TO
TAL
                     EDM=2.04372e-09    STRATEGY= 1      ERROR MATRIX ACC
URATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  N            2.46469e+02   4.31493e+00   1.19092e-02  -5.38026e-06
   2  mu           1.04793e-02   1.43576e-02   4.87640e-05   4.15093e-03
   3  sigma        1.00316e+00   1.03818e-02   2.86307e-05  -2.55310e-04
```

Some Goodies

```
%jsroot on
```



JSROOT Visualisation

Start On a Laptop

- Possible to install Jupyter as a package
- Fire up with the *jupyter notebook* command

Use Notebooks at CERN

- **SWAN**: **S**ervice for **W**eb based **AN**alysis
- Get a CERNBox (if you don't have one)
  - Visit https://cernbox.cern.ch

https://swan.cern.ch

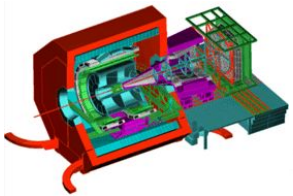
Reading and Writing Data

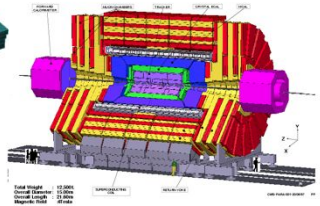
Learning Objectives

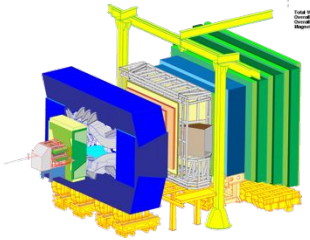
- Understand the relevance of I/O in scientific applications
- Grasp some of the details of the ROOT I/O internals
- Be able to write and read ROOT objects to and from ROOT files

I/O at LHC: an Example

A selection of the experiments adopting ROOT







Event Filtering

Data

Analysis

Offline Processing

Reconstruction

Further processing, skimming

Event Selection, statistical treatment …

Raw

Reco

…

Analysis Formats

Images

Data Storage: Local, Network

More Data in The Future?

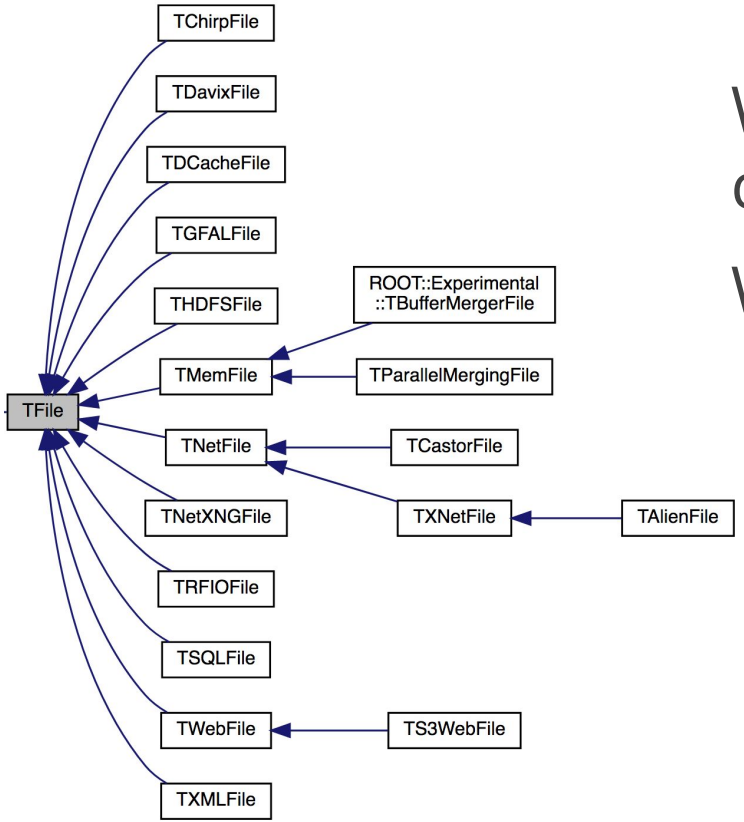
Now	1 EB of data, 0.5 million cores
Run III	LHCb 40x collisions, Alice readout @ 50 KHz (starts in 2021 already!!)
HL-LHC	Atlas/CMS pile-up 60 -> 200, recording 10x evts

The ROOT File

- In ROOT, objects are written in files*
- ROOT provides its file class: the **TFile**
- TFiles are *binary* and have: a *header*, *records* and can be compressed (transparently for the user)
- TFiles have a logical “file system like” structure
  - e.g. directory hierarchy
- **TFiles are self-descriptive**:
  - Can be read without the code of the objects streamed into them
  - E.g. can be read from JavaScript

* this is an understatement - we’ll not go into the details in this course!

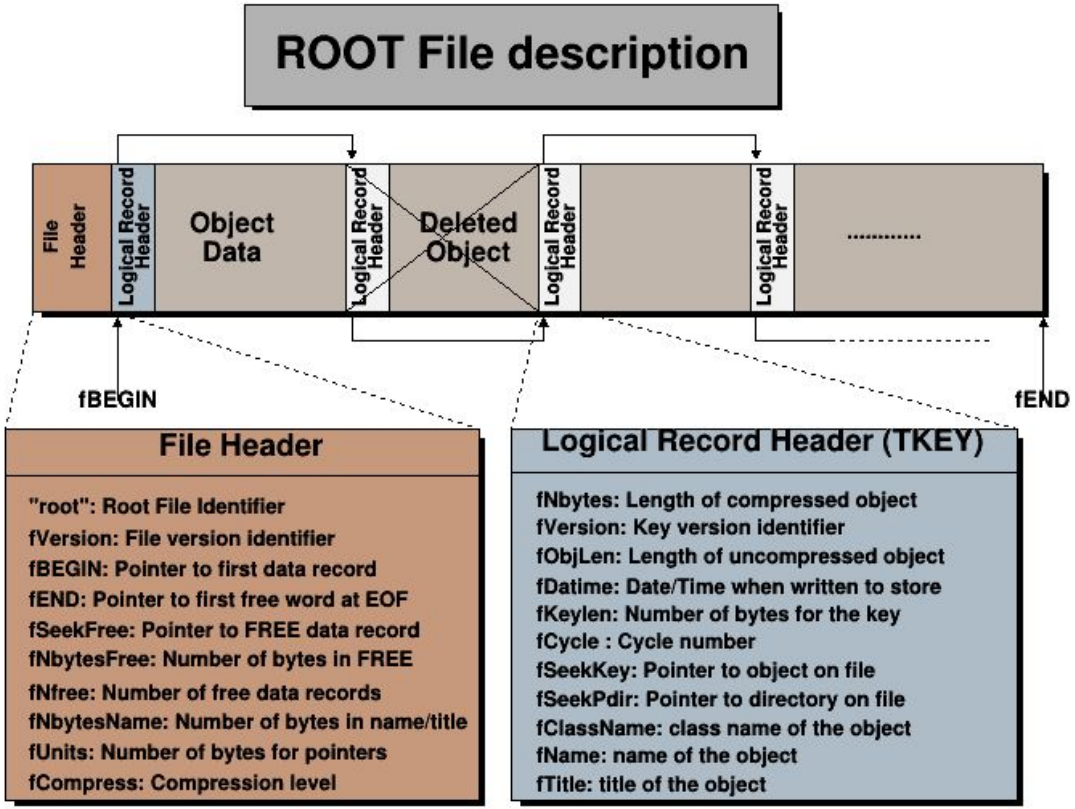
Flavour of TFiles



We'll focus on TFile only in this course.

We'll use TXMLFiles for the exercises.

ROOT File Description



A Well Documented File Format

Byte Range	Record Name	Description
1->4	"root"	Root file identifier
5->8	fVersion	File format version
9->12	fBEGIN	Pointer to first data record
13->16 [13->20]	fEND	Pointer to first free word at the EOF
17->20 [21->28]	fSeekFree	Pointer to FREE data record
21->24 [29->32]	fNbytesFree	Number of bytes in FREE data record
25->28 [33->36]	nfree	Number of free data records
29->32 [37->40]	fNbytesName	Number of bytes in **TNamed** at creation time
33->33 [41->41]	fUnits	Number of bytes for file pointers
34->37 [42->45]	fCompress	Compression level and algorithm
38->41 [46->53]	fSeekInfo	Pointer to **TStreamerInfo** record
42->45 [54->57]	fNbytesInfo	Number of bytes in **TStreamerInfo** record
46->63 [58->75]	fUUID	Universal Unique ID

How Does it Work in a Nutshell?

- **C++ does not support native I/O** of its objects
- Key ingredient: reflection information - **Provided by ROOT**
  - What are the data members of the class of which this object is instance? I.e. How does the object look in memory?
- The steps, from memory to disk:

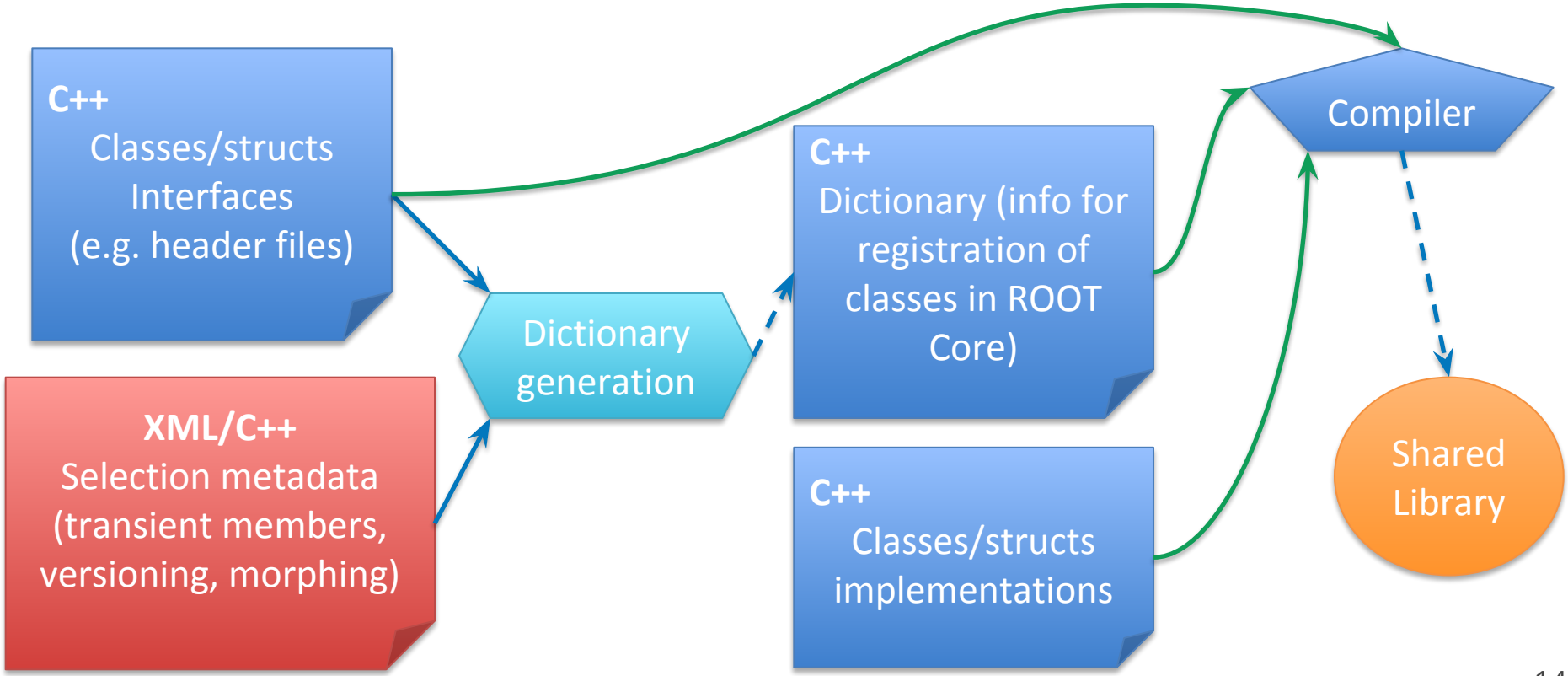
1. Serialisation: from an object in memory to a blob of bytes
2. Compression: use an algorithm to reduce size of the blob (e.g. zip, lzma, lz4)
3. “Real” writing via OS primitives

Serialisation: not a trivial task

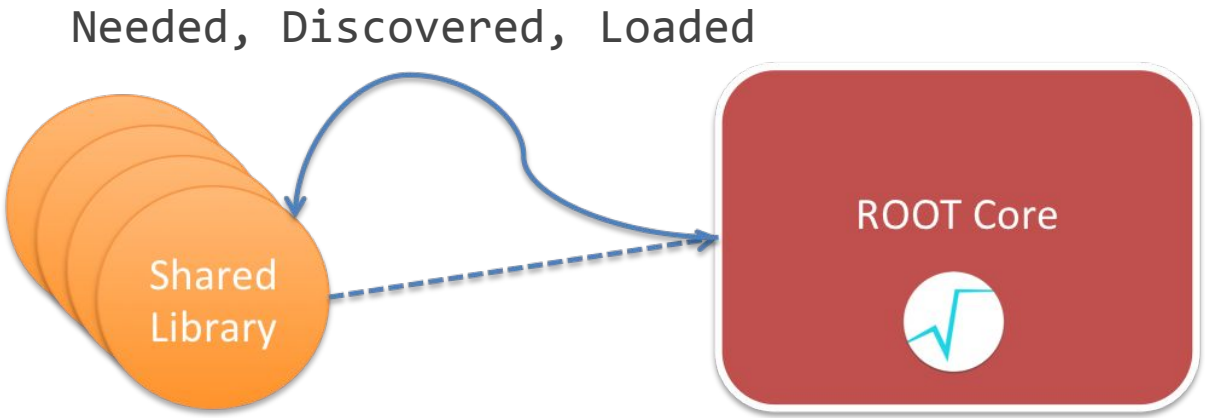
For example:

- Must be platform independent: e.g. 32bits, 64bits
  - Remove padding if present, little endian/big endian
- Must follow pointers correctly
  - And avoid loops ;)
- Must treat stl constructs
- Must take into account customisations by the user
  - E.g. skip “transient data members”

Persistency



Injection of Reflection Information



Now ROOT “knows” how to serialise the instances implemented in the library (series of data members, type, transiency) and to write them on disk in row or column format.

TFile in Action

```
f= ROOT.TFile("myfile.root", "RECREATE")
h = ROOT.TH1F("h", "h", 64, 0, 8)
h.Write()
f.Close()
```

TFile in Action

```
f = ROOT.TFile("myfile.root","RECREATE")
```

Option	Description
NEW or CREATE	Create a new file and open it for writing, if the file already exists the file 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

```
f = ROOT.TFile("myfile.root", "RECREATE")
h = TH1F("h", "h", 64, 0, 8)
h.Write()
f.Close()
```

- Write on a file
- Close the file and make sure the operation is finalised

The *gDirectory*

Wait! How does it know where to write?

- ROOT has global variables. Upon creation of a file, the “present directory” is moved to the file.
- Histograms are attached to that directory
- Has up- and down- sides
- Will be more explicit in the future versions of ROOT

```
TFile f("myfile.root", "RECREATE");
TH1F h("h", "h", 64, 0, 8);
h.Write();
f.Close();
```

More than One File

Wait! And then how do I manage more than one file?

- You can “cd” into files anytime.
- The value of the *gDirectory* will change accordingly

```
f1 = ROOT.TFile("myfile1.root", "RECREATE")
f2 = ROOT.TFile("myfile2.root", "UPDATE")
f1.cd(); h1 = ROOT.TH1F("h", "h", 64, 0, 8)
h1.Write()
f2.cd(); h2 = ROOT.TH1F("h", "h", 64, 0, 8)
h1.Write()
f1.Close(); f2.Close();
```

TFile in Action

```
TH1F* myHist;
TFile f("myfile.root");
f.GetObject("h", myHist);
myHist->Draw();
```

TFile in Action

```
f = ROOT.TFile("myfile.root");
myHist = f.GetObject("h", myHist)
myHist.Draw()
```

TFile in Action: Python

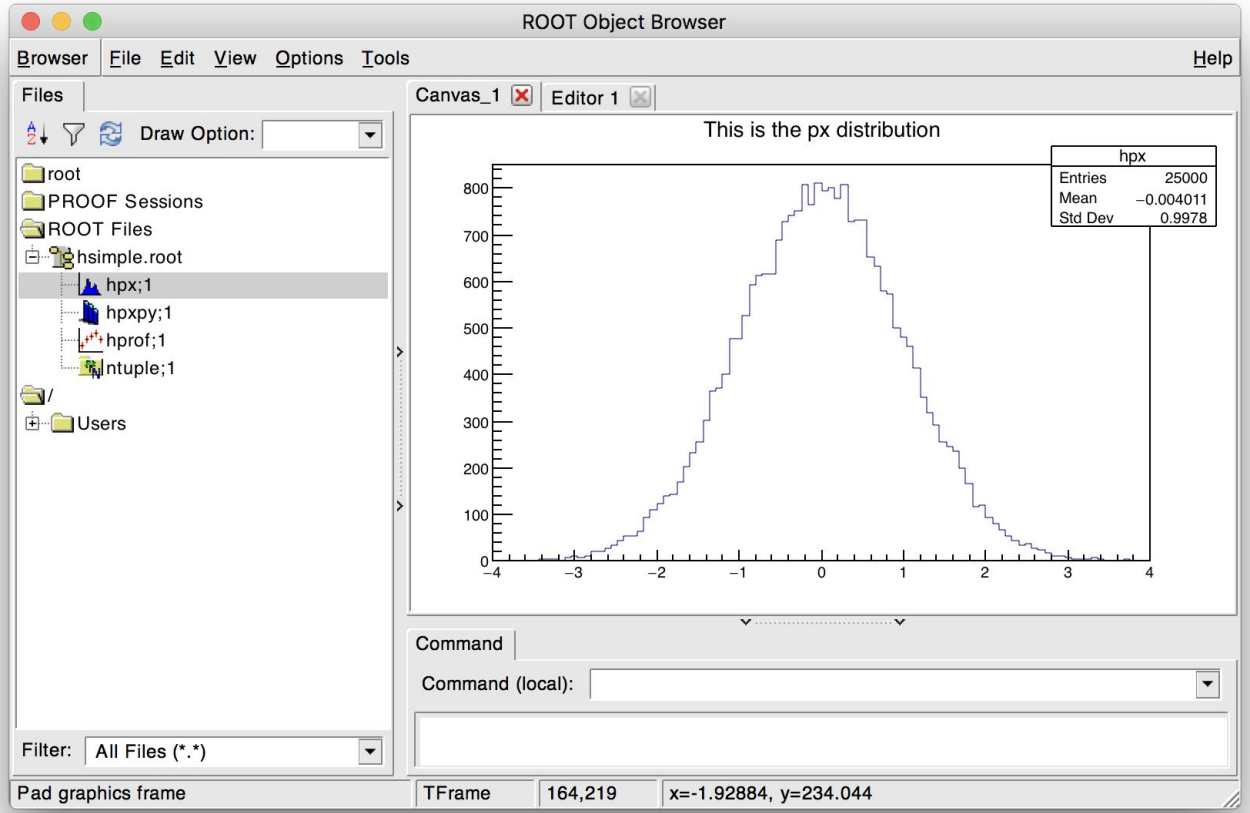
```
import ROOT
f = ROOT.TFile("myfile.root")
f.h.Draw()
```

Get the histogram by name! Possible because Python is not a compiled language

Listing TFile Content

- *TFile::ls()*: prints to screen the content of the TFile
  - Great for interactive usage
- *TBrowser* interactive tool
- Loop on the "*TKeys*", more sophisticated
  - Useful to use "programmatically"
- *rootls* commandline tool: list what is inside

TBrowser


ROOT Object Browser
Browser
File
Edit
View
Options
Tools
Help
Files
Draw Option:
root
PROOF Sessions
ROOT Files
hsimple.root
hpx;1
hpxpy;1
hprof;1
ntuple;1
/
Users
Filter:
All Files (*.*)
Canvas_1
Editor 1
This is the px distribution
hpx
Entries 25000
Mean −0.004011
Std Dev 0.9978
800
700
600
500
400
300
200
100
0
−4
−3
−2
−1
0
1
2
3
4
Command
Command (local):
Pad graphics frame
TFrame
164,219
x=-1.92884, y=234.044

Loop on TKeys: Python

List comprehension syntax!

```
import ROOT
f = ROOT.TFile("myfile.root")
keyNames = [k.GetName() for k in f.GetListOfKeys()]
print keyNames
```

Time for Exercises!

https://github.com/vincecr0ft/pyROOT_lectures/tree/master/ChapterOne_Syntax/4_IO_and_writing_files.ipynb

The ROOT Columnar Format

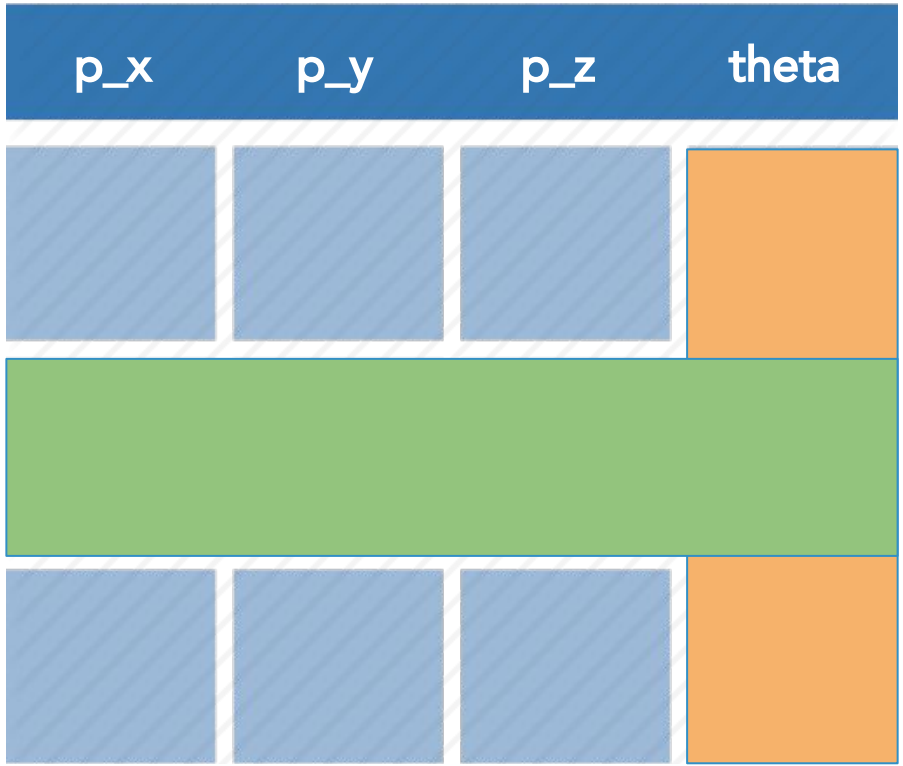
Learning Objectives

- Understand the difference between row-wise and column-wise storage
- Appreciate the difference between low and high-level interfaces for data analysis in ROOT
- Be able to write and read data in the ROOT columnar format

Columns and Rows

- High Energy Physics: many statistically independent *collision events*
- Create an event class, serialise and write out N instances on a file? No. Very inefficient!
- Organise the dataset in **columns**

Columnar Representation



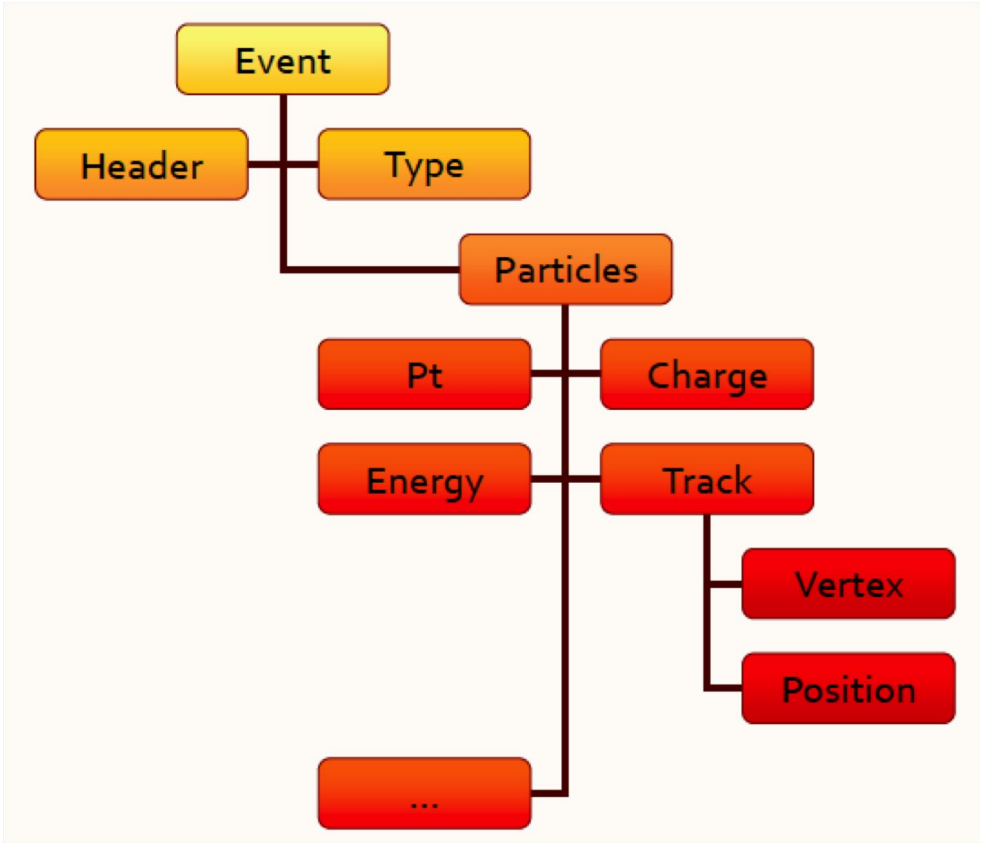
entries
or events
or rows

columns
or "branches"

can contain any kind
of c++ object

Relations Among Columns

x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
0.55	-0.21231	50281
-0.184	1.187305	.443902
0.20564	-0.770	0.635417
1.079222	0.32	1.271904
-0.27492	-	43 3.038899
2.047779	-	268 4.197329
-0.45868	4	2 2.293266
0.304731	0.884	0.875442
-0.712	-0.222	0.556881
-0.27	1.181767	470484
0.88	-0.65411	3209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
		3.562347



Optimal Runtime and Storage Usage

Runtime:

- Can decide what columns to read
- Prefetching, read-ahead optimisations possible

Storage Usage:

- Run-length Encoding (RLE). Compression of individual columns values is very efficient
  - Physics values: potentially all “similar”, e.g. within a few orders of magnitude - position, momentum, charge, index

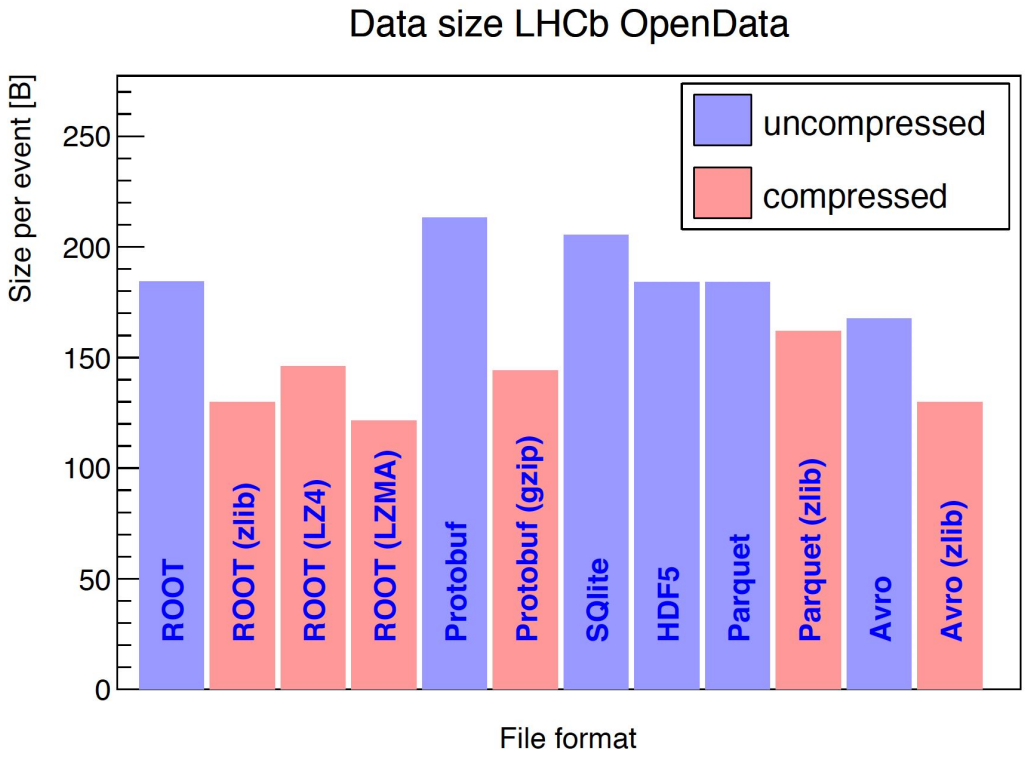
Comparison With Other I/O Systems

	ROOT	PB	SQlite	HDF5	Parquet	Avro
Well-defined encoding	✓	✓	✓	✓	✓	✓
C/C++ Library	✓	✓	✓	✓	✓	✓
Self-describing	✓	↯	✓	✓	✓	✓
Nested types	✓	✓	?	?	✓	✓
Columnar layout	✓	↯	↯	?	✓	↯
Compression	✓	✓	↯	?	✓	✓
Schema evolution	✓	↯	✓	↯	?	?

✓ = supported
↯ = unsupported
? = difficult / unclear

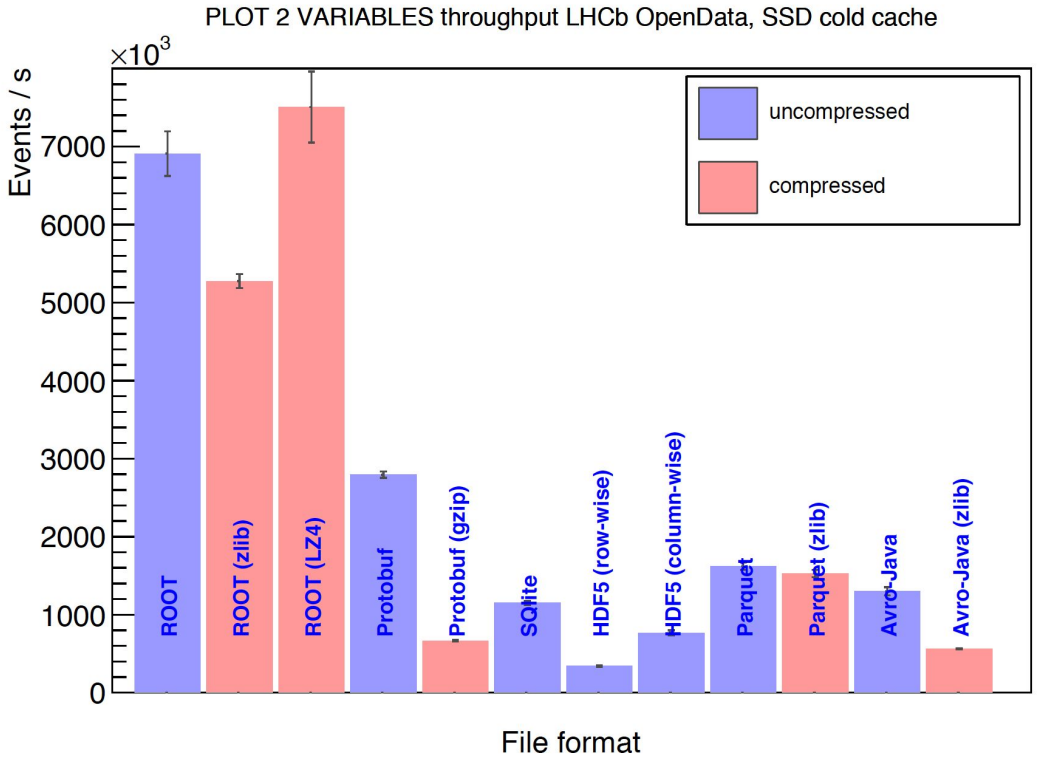
J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

Comparison With Other I/O Systems



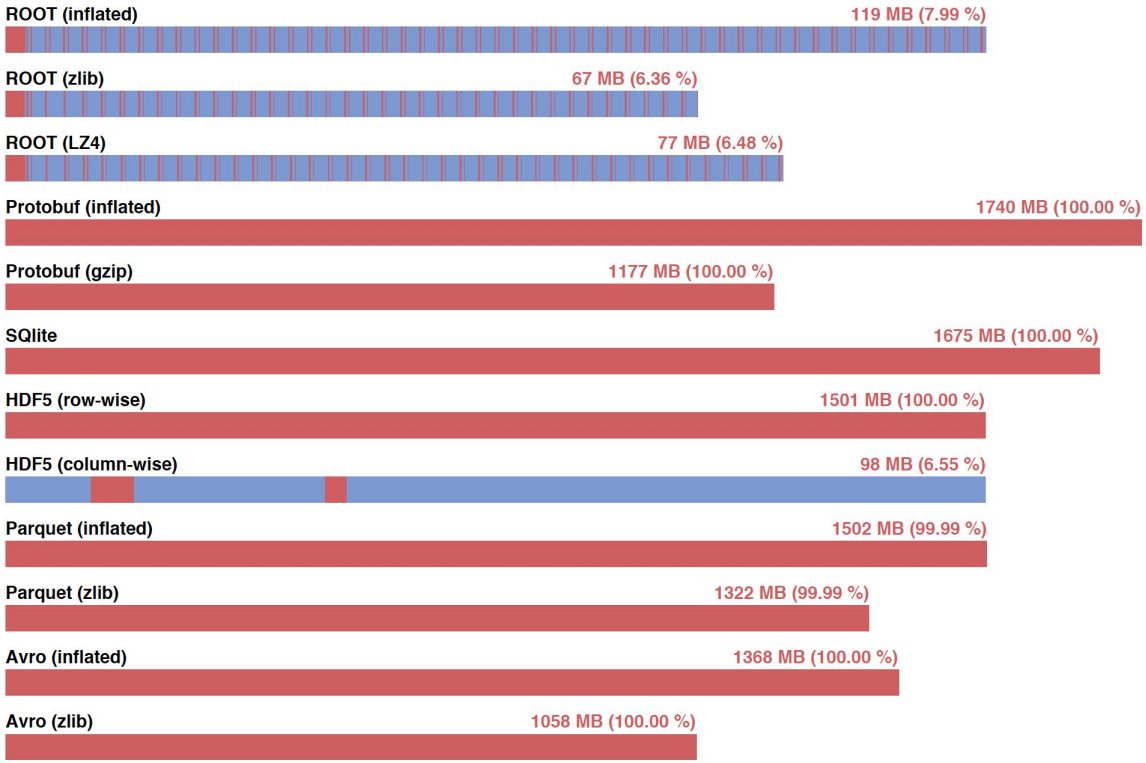
J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

Comparison With Other I/O Systems



J. Blomer, [A quantitative review of data formats for HEP analyses](#) ACAT 2017

I/O Patterns



The less you read (red sections), the faster

J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

The TTree

A columnar dataset in ROOT is represented by **TTree**:

- Also called *tree*, columns also called *branches*
- An object type per column, **any type of object**
- One row per *entry* (or, in collider physics, *event*)

If just a **single number** per column is required, the simpler **TNtuple** *can* be used.

Filling a TNtuple

```
TFile f("myfile.root", "RECREATE");
TNtuple myntuple("n", "n", "x:y:z:t");

// We assume to have 4 arrays of values:
// x_val, y_val, z_val, t_val

for (auto ievt: ROOT::TSeqI(128)) {
   myntuple.Fill(x_val[ievt], y_val[ievt],
                 z_val[ievt], t_val[ievt]);
}
myntuple.Write();
f.Close();
```

Names of the columns

Works only if columns are simple numbers

Reading a TNtuple

```
TFile f("hsimple.root");
TNtuple *myntuple;
f.GetObject("hsimple", myntuple);
TH1F h("h", "h", 64, -10, 10);
for (auto ievt: ROOT::TSeqI(myntuple->GetEntries()) {
   myntuple->GetEntry(ievt);
   auto xyzt = myntuple->GetArgs(); // Get a row
   if (xyzt[2] > 0) h.Fill(xyzt[0] * xyzt[1]);
}
h.Draw();
```

Works only if columns are simple numbers

Reading a TNtuple: PyROOT

```
import ROOT
f = ROOT.TFile("hsimple.root")
h = ROOT.TH1F("h", "h", 64, -10, 10)
for evt in f.hsimple:
   if evt.pz > 0: h.Fill(evt.py * evt.pz)
h.Draw()
```

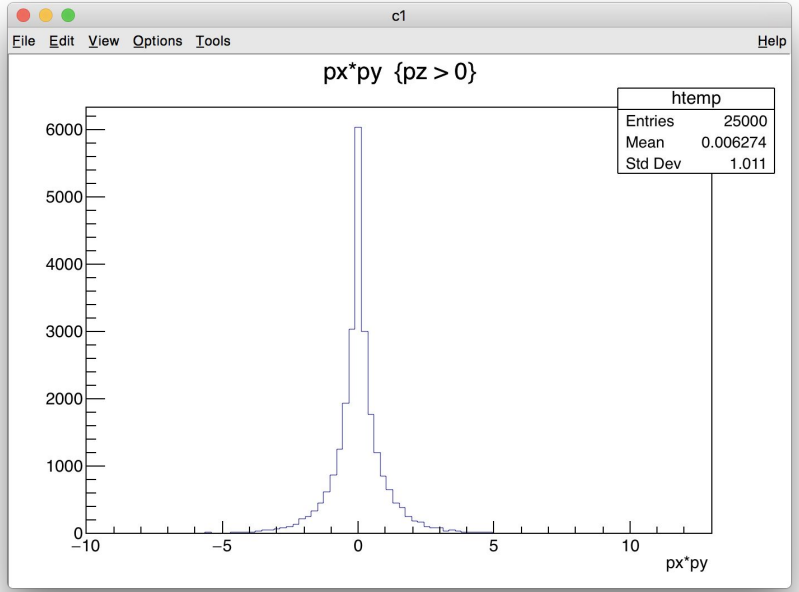
PyROOT: Works for all types of columns, not only numbers!

One Line Analysis

- It is possible to produce simple plots described as strings
- **TNtuple::Draw()** method
- Jargon: known also under the name of *ttreedraw*
- Good for quick looks, does not scale
  - E.g. one loop on all events per plot

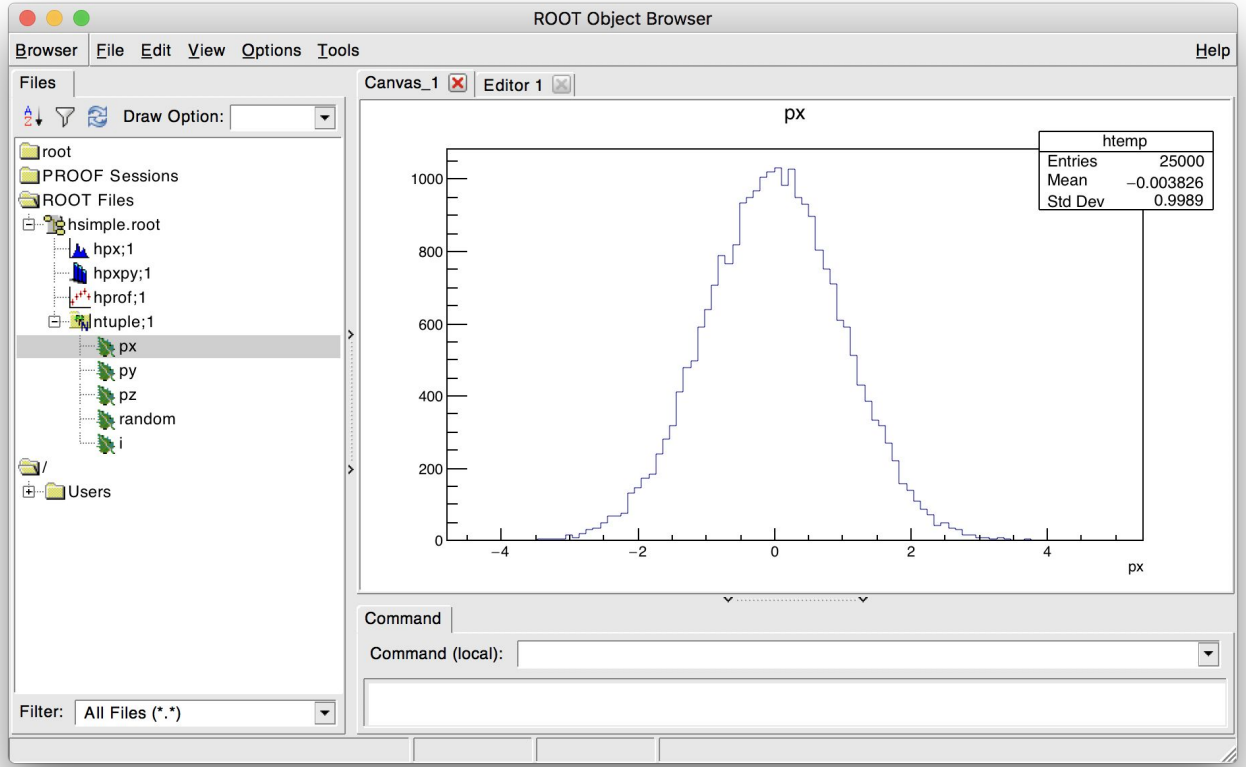
One Line Analysis

Works for all types of columns, not only numbers!



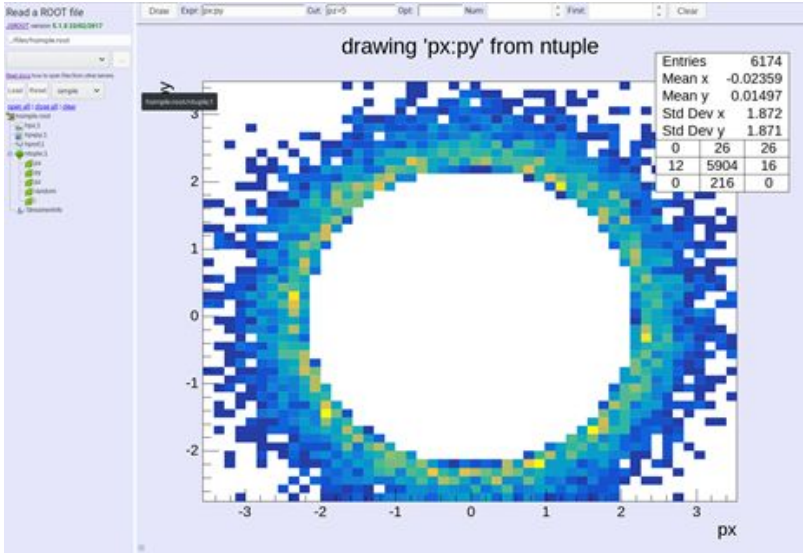

```
TFile f("hsimple.root");
TNtuple *myntuple;
f.GetObject("ntuple", myntuple);
myntuple.Draw("px * py", "pz > 0");
```

Reading a TNtuple with TBrowser



In Javascript

- Direct access to the data from the browser
- All branches types are supported
  - including split STL containers
  - and old TBranchObject
- Complex TTree::Draw syntax
  - Expressions
  - Cut conditions
  - Arrays indexes
  - Math functions
  - Class functions
  - Histogram parameters



https://root.cern.ch/js/latest/

Time For Exercises

https://github.com/root-project/training/tree/master/BasicCourse/Exercises/WorkingWithColumnarData

TTrees and TNtuples

- **TNtuple** is great, but only works if columns hold simple numbers
- If something else needs to be in the columns, **TTree** must be used
- **TNtuple** is a specialisation of **TTree**

We'll explore how to read TTrees starting from the TNtuple examples

Filling a Tree with Numbers

```
TFile f("SimpleTree.root","RECREATE"); // Create file first. The TTree will be associated to it
TTree data("tree","Example TTree");    // No need to specify column names

double x, y, z, t;
data.Branch("x",&x,"x/D");      // Associate variable pointer to column and specify its type, double
data.Branch("y",&y,"y/D");
data.Branch("z",&z,"z/D");
data.Branch("t",&t,"t/D");

for (int i = 0; i<128; ++i) {
   x = gRandom->Uniform(-10,10);
   y = gRandom->Gaus(0,5);
   z = gRandom->Exp(10);
   t = gRandom->Landau(0,2);
   data.Fill();                 // Make sure the values of the variables are recorded
}
data.Write();                   // Dump on the file
f.Close();
```

Filling a Tree with Objects

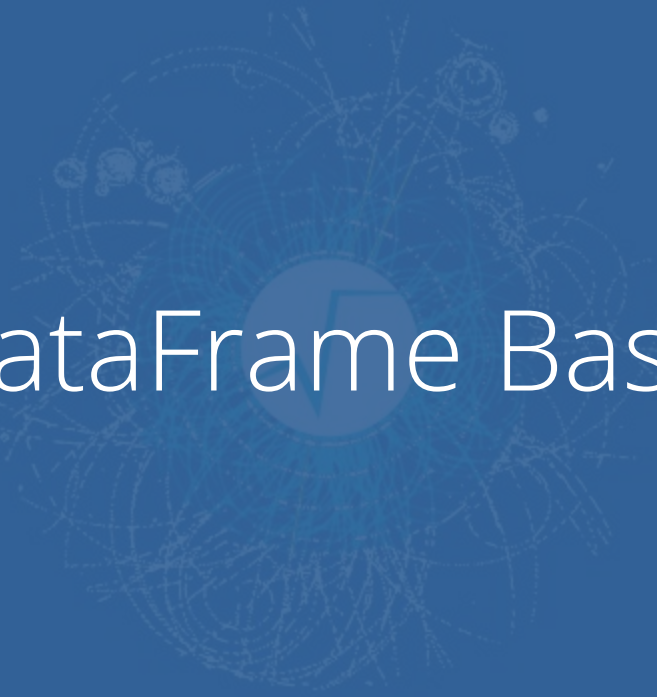
```
TRandom3 R;
using trivial4Vectors =
std::vector<std::vector<double>>;

TFile f("vectorCollection.root",
        "RECREATE");
TTree t("t","Tree with pseudo particles");

trivial4Vectors  parts;
auto partsPtr = &parts;

t1.Branch("tracks", &partsPtr);
// pi+/pi- mass
constexpr double M = 0.13957;
```

```
for (int i = 0; i < 128; ++i) {
   auto nPart = R.Poisson(20);
   particles.clear(); parts.reserve(nPart);
   for (int j = 0; j < nPart; ++j) {
      auto pt = R.Exp(10);
      auto eta = R.Uniform(-3,3);
      auto phi = R.Uniform(0, 2*TMath::Pi() );
      parts.emplace_back({pt, eta, phi, M});
    }
    t.Fill();
  }
  t.Write();
}
```

Reading Objects from a TTree

```
{

using trivial4Vector =
std::vector<double>;
using trivial4Vectors =
std::vector<trivial4Vector>;

TFile f("parts.root");
TTreeReader myReader("t", &f);
TTreeReaderValue<trivial4Vectors>
partsRV(myReader, "parts");

TH1F h("pt","Particles Transverse
Momentum;P_{T} [GeV];#", 64, 0, 10);
```

```
while (myReader.Next()) {
   for (auto &p : *partsRV ) {
     auto pt = p[0];
     h.Fill(pt);
   }
}
h.Draw();

}
```



TDataFrame Basics

Can we do Better?

simple yet powerful way to analyse data with modern C++

---

provide high-level features, e.g.
less typing, better expressivity, abstraction of complex operations

---

allow transparent optimisations, e.g.
multi-thread parallelisation and caching

Improved Interfaces

what we write

```
TTreeReader reader(data);
TTreeReaderValue<A> x(reader,"x");
TTreeReaderValue<B> y(reader,"y");
TTreeReaderValue<C> z(reader,"z");
while (reader.Next()) {
   if (IsGoodEntry(*x, *y, *z))
      h->Fill(*x);
}
```

what we *mean*

- full control over the event loop
- requires some boilerplate
- users implement common tasks again and again
- parallelisation is not trivial

TDataFrame: declarative analyses

```
TDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

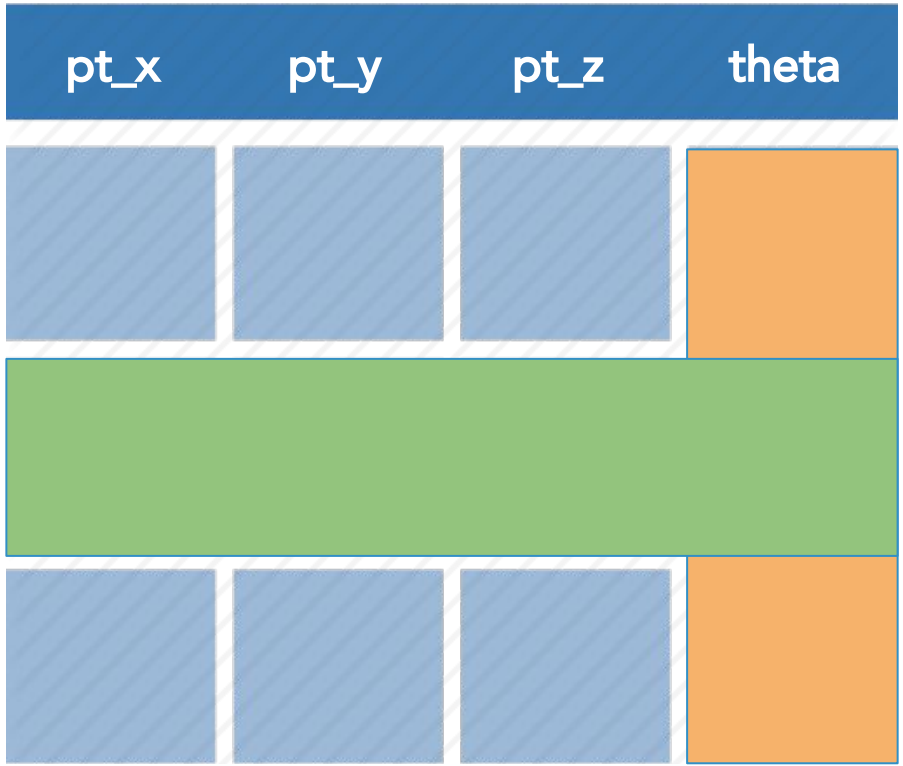
TDataFrame: declarative analyses

```
ROOT::EnableImplicitMT();
TDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

Columnar Representation



entries
or events
or rows

columns
or "branches"
can contain any kind
of c++ object

TDataFrame: quick how-to

1. build a data-frame object by specifying your data-set
2. apply a series of transformations to your data
   - filter (e.g. apply some cuts) or
   - define new columns
3. apply actions to the transformed data to produce results (e.g. fill a histogram)

Creating a TDataFrame - 1 file

```
TDataFrame d1("treename", "file.root");

auto filePtr = TFile::Open("file.root");
TDataFrame d2("treename", filePtr);

TTree *treePtr = nullptr;
filePtr->GetObject("treename", treePtr);
TDataFrame d3(*treePtr); // by reference!
```

Three ways to create a TDataFrame that reads tree "treename" from file "file.root"

Creating a TDataFrame - more files

```
TDataFrame d1("treename", "file*.root");
TDataFrame d2("treename", {"file1.root","file2.root"});

std::vector<std::string> files = {"file1.root","file2.root"};
TDataFrame d3("treename", files);

TChain chain("treename");
chain.Add("file1.root); chain.Add("file2.root);
TDataFrame d4(chain); // passed by reference, not pointer!
```

Here TDataFrame reads tree “treename” from files “file1.root” and “file2.root”

Cut on theta, fill histogram with pt

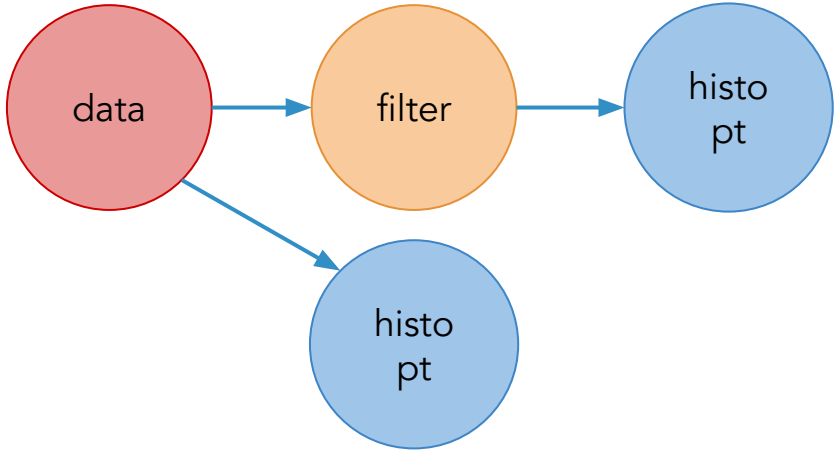
```
TDataFrame d("t", "f.root");
auto h = d.Filter("theta > 0").Histo1D("pt");
h->Draw(); // event loop is run here, when you access a result
           // for the first time
```

event-loop is run *lazily*, upon first access to the results

Time For Exercises

- Draw a plot of px + py for every pz between -2 and 2 using the hsimple.root file
  - Use TDataFrame
  - Compare with the other approaches: number of lines, readability

Think of your analysis as data-flow




```
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Histo1D("pt");
```

Using callables instead of strings

```
// define a c++11 lambda - an inline function - that checks “x>0”
auto IsPos = [](double x) { return x > 0.; };
// pass it to the filter together with a list of branch names
auto h = d.Filter(IsPos, {"theta"}).Histo1D("pt");
h->Draw();
```

any callable (function, lambda, functor class) can be used as a filter, as long as it returns a boolean

Filling multiple histograms

```
auto h1 = d.Filter("theta > 0").Histo1D("pt");
auto h2 = d.Filter("theta < 0").Histo1D("pt");
h1->Draw();          // event loop is run once here
h2->Draw("SAME"); // no need to run loop again here
```

Book all your actions upfront. The first time a result is accessed, TDataFrame will fill all booked results.

Define a new column

```
double m = 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.

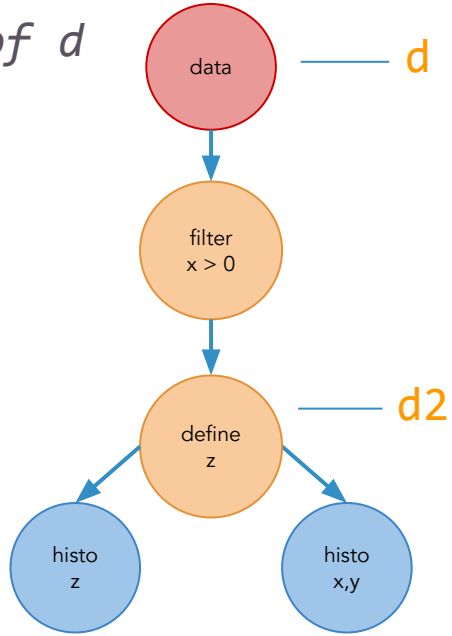
Define a new column

```
double SqrtSumSq(double, double) { return ... ; }
double m = d.Filter("x > y")
            .Define("z", SqrtSumSq, {"x","y"})
            .Mean("z");
```

Just like `Filter`, `Define` accepts any callable object (function, lambda, functor class...)

Think of your analysis as data-flow

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

// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D("x","y");
```



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

Cutflow reports

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

```
// output
xcut      : pass=49         all=100        --   49.000 %
ycut      : pass=22         all=49         --   44.898 %
```

When called on the main TDF object, `Report` prints statistics for all filters *with a name*

Running on a range of entries #1

```
// stop after 100 entries have been processed
auto hz = d.Range(100).Histo1D("x");

// skip the first 10 entries, then process one every two until the end
auto hz = d.Range(10, 0, 2).Histo1D("x");
```

Ranges are only available in single-thread executions.
They are useful for quick initial data explorations.

Running on a range of entries #2

```
// ranges can be concatenated with other transformations
auto c = d.Filter("x > 0")
          .Range(100)
          .Count();
```

This `Range` will process the first 100 entries
that pass the filter

Saving data to file

```
auto 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.

Creating a new data-set

```
TDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
              .Define("y", []() { return rand() % 10; })
              .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file

N.B. `rand()` is generally not a good way to produce uniformly distributed random numbers

Not Only ROOT Datasets

- TDataSource: Plug *any columnar* format in TDataFrame
- Keep the programming model identical!
- ROOT provides CSV data source
- More to come
  - TDataSource is a programmable interface!
  - E.g. https://github.com/bluehood/mdfds LHCb raw format - not in the ROOT repo

Not Only ROOT Datasets

```
auto fileName = "tdf014_CsvDataSource_MuRun2010B.csv";
auto tdf = ROOT::Experimental::TDF::MakeCsvDataFrame(fileName);

auto filteredEvents =
tdf.Filter("Q1 * Q2 == -1")
.Define("m", "sqrt(pow(E1 + E2, 2) - (pow(px1 + px2, 2) + pow(py1 + py2, 2) + pow(pz1 + pz2, 2)))");

auto invMass =
filteredEvents.Histo1D({"invMass", "CMS Opendata: #mu#mu mass;mass [GeV];Events", 512, 2, 110}, "m");
```

tdf014_CsvDataSource_MuRun2010B.csv:

```
Run,Event,Type1,E1,px1,py1,pz1,pt1,eta1,phi1,Q1,Type2,E2,px2,py2,pz2,pt2,eta2,phi2,Q2,M
146436,90830792,G,19.1712,3.81713,9.04323,-16.4673,9.81583,-1.28942,1.17139,1,T,5.43984,-0.362592,2.62699,-4.74849,2.65189,-1.34587,1.70796,1,2.73205
146436,90862225,G,12.9435,5.12579,-3.98369,-11.1973,6.4918,-1.31335,-0.660674,-1,G,11.8636,4.78984,-6.26222,-8.86434,7.88403,-0.966622,-0.917841,1,3.10256
```

TDataFrame
Extra features

Caching

```
TDataFrame d("mytree", "myFile.root");
auto cached_d = d.Cache();
```

All the content of the TDF is now in (contiguous) memory.
Analysis as fast as it can be (vectorisation possible too).

N.B. It is always possible to selectively cache columns to save some memory!

Creating a new data-set - parallel

```
ROOT::EnableImplicitMT();
TDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
              .Define("y", []() { return rand() % 10; })
              .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file -- in parallel

N.B. `rand()` is generally not a good way to produce uniformly distributed random numbers

More on histograms #1

```
auto h = d.Histo1D("x","w");
```

TDF can produce *weighted* TH1D, TH2D and TH3D.
Just pass the extra column name.

More on histograms #2

```
auto h = d.Histo1D({"h","h",10,0.,1.},"x", "w");
```

You can specify a model histogram with a set axis range, a name and a title (optional for TH1D, mandatory for TH2D and TH3D)

Filling histograms with arrays

```
auto h = d.Histo1D("pt_array", "x_array");
```

If `pt_array` and `x_array` are an array or an STL container (e.g. std::vector), TDF fills histograms with all of their elements. `pt_array` and `x_array` are required to have equal size for each event.

C++ / JIT / PyROOT

Pure C++

```
d.Filter([](double t) { return t > 0.; }, {"th”})
 .Snapshot<vector<float>>("t","f.root",{"pt_x"});
```

C++ and JIT-ing with CLING

```
d.Filter("th > 0").Snapshot("t","f.root","pt*");
```

---

pyROOT -- just leave out the ;

```
d.Filter("th > 0").Snapshot("t","f.root","pt*")
```

Time For Exercises

https://github.com/root-project/training/tree/master/BasicCourse/Exercises/WorkingWithColumnarData

Wrap up