ROOT basics

andrea.rizzi@unipi.it

The material in this talk is mostly taken from ROOT Basic Course available at:
https://docs.google.com/presentation/d/189f0qsDEnMSk2R5KWLRPz2TdEV5kTfXH1VcuAra4cnU/edit#slide=id.g24ecd7c82b_0_94

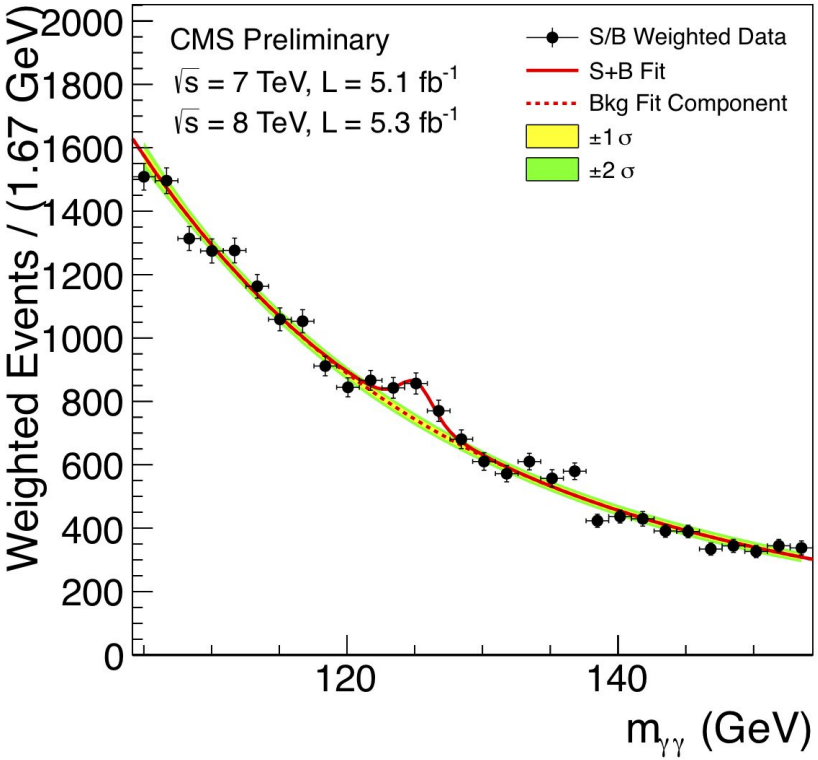
Resources

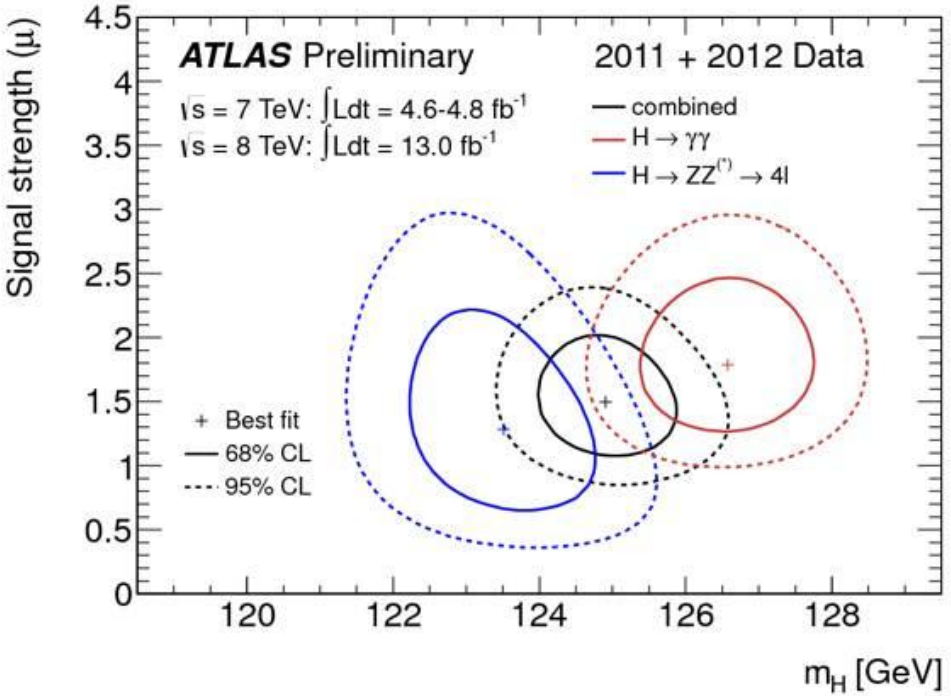
- ROOT Website: https://root.cern
- Material online: https://github.com/root-project/training
- More material: https://root.cern/getting-started
  - Includes a booklet for beginners: **the “ROOT Primer”**
- Reference Guide: https://root.cern/doc/master/index.html
- Forum: https://root-forum.cern.ch

How to Use ROOT during this lectures

- Launch your VM (where ROOT is preinstalled)
- Open a terminal and type
  - `# source ~/root/bin/thisroot.sh`
  - Now the ROOT environment is setup (echo $PATH or echo $ROOTSYS)
  - In this terminal you can now
    - Compile a program using root libraries
      - use root-config to get the gcc flags, e.g.
      - `c++ myprog.cxx -o myprog `root-config --cflags --libs``
    - Execute a program linked to root libraries
    - Run root interactive shell
    - Run a python script or python interactive session using pyRoot bindings

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 available as well

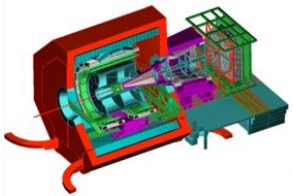
- Adopted in High Energy Physics and other sciences (but also industry)
  - 1 EB of data in ROOT format
  - Fits and parameters' estimations for discoveries (e.g. the Higgs)
  - Thousands of ROOT plots in scientific publications

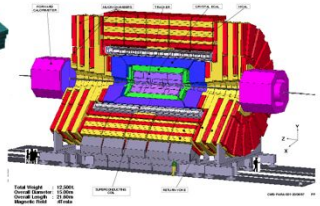
ROOT in a Nutshell

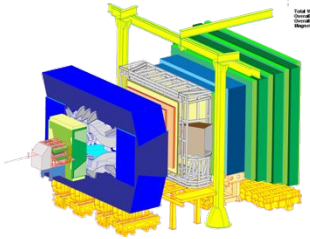
- ROOT can be seen as a collection of building blocks for various activities, like:
  - **Data analysis: histograms, graphs, functions**
  - **I/O: row-wise, column-wise** storage of any C++ object
  - **Statistical tools** (RooFit/RooStats): rich modeling and statistical inference
  - Math: **non trivial functions** (e.g. Erf, Bessel), optimised math functions
  - **C++ interpretation**: full language compliance
  - **Multivariate Analysis** (TMVA): e.g. Boosted decision trees, NN
  - **Advanced graphics** (2D, 3D, event display)
  - **Declarative Analysis**: TDataFrame
  - And more: HTTP servering, JavaScript visualisation

ROOT Application Domains

A selection of the experiments adopting ROOT







Event Filtering

Data

Analysis

Offline Processing

Event Selection, statistical treatment …

Further processing, skimming

Reconstruction

Raw

Reco

…

Analysis Formats

Images

Data Storage: Local, Network

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
  - Can use Python interpreter directly after a simple *import ROOT*
  - Possible to “mix” the two languages (see more later)

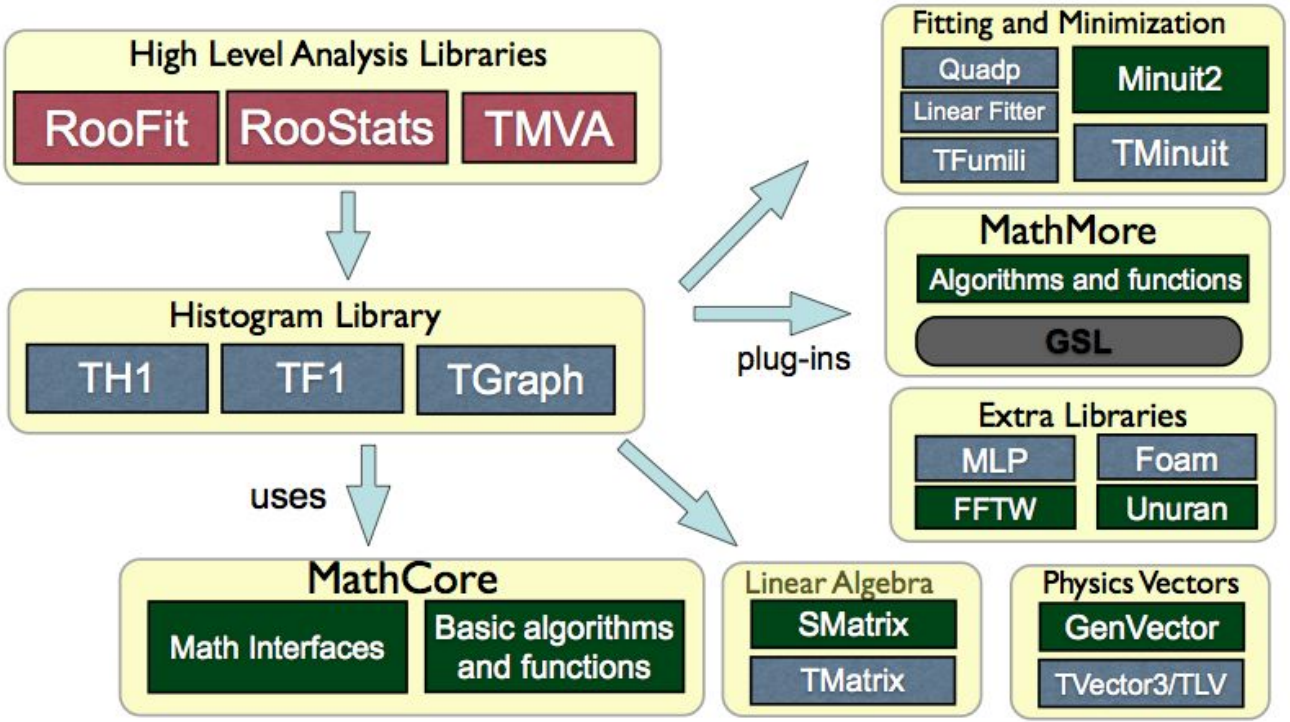
Persistency or Input/Output (I/O)

- 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

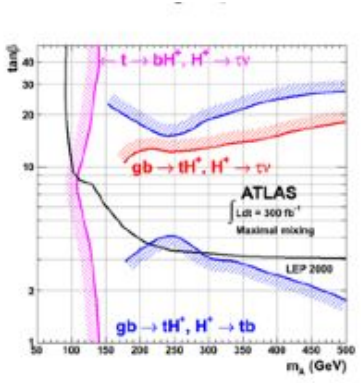
Mathematics and Statistics

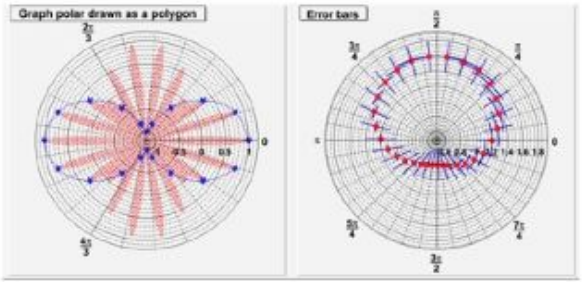
- ROOT provides a rich set of mathematical libraries and tools for sophisticated statistical data analysis

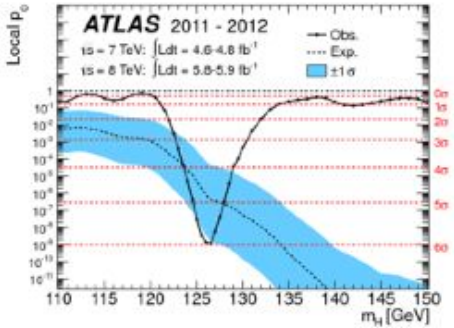


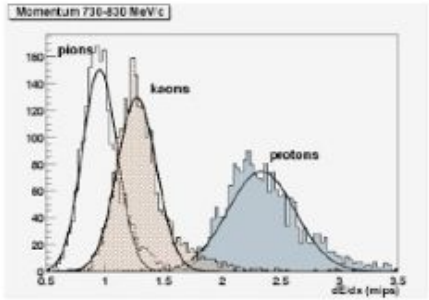
Graphics in ROOT

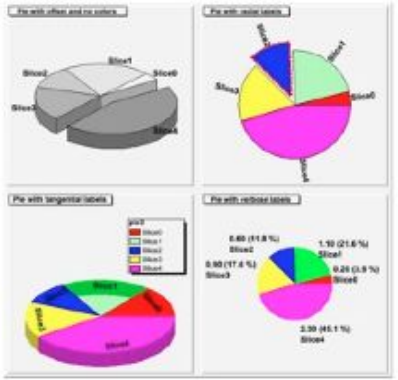
- Many formats for data analysis, and not only, plots

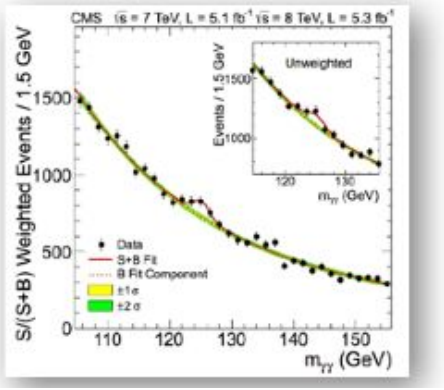




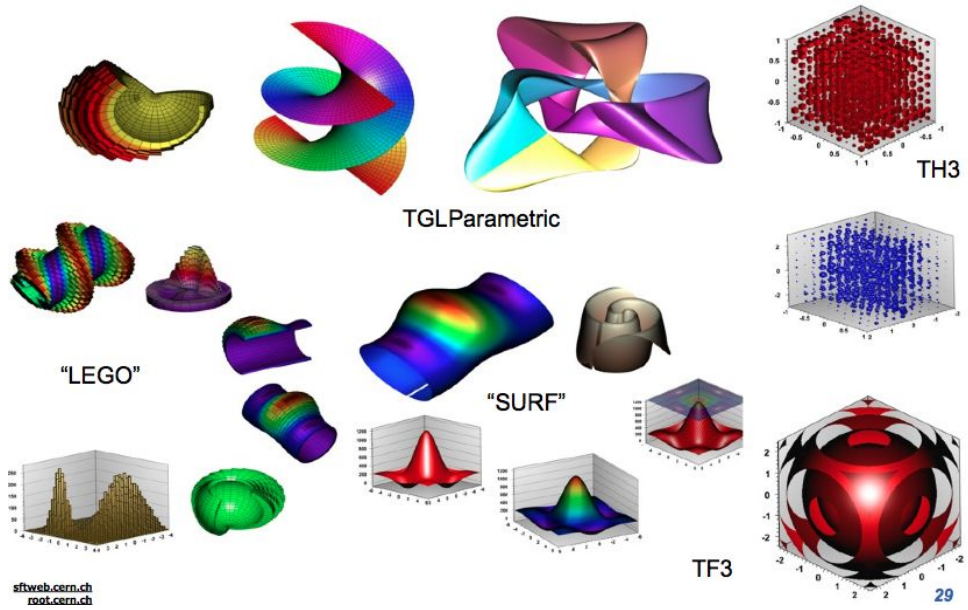


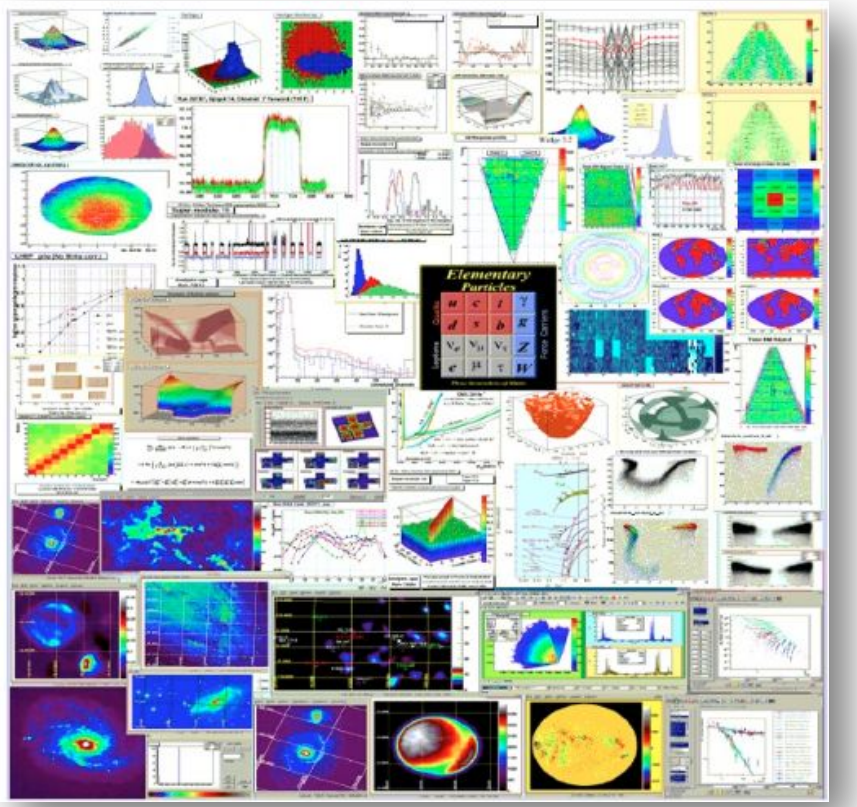






2D and 3D Graphics



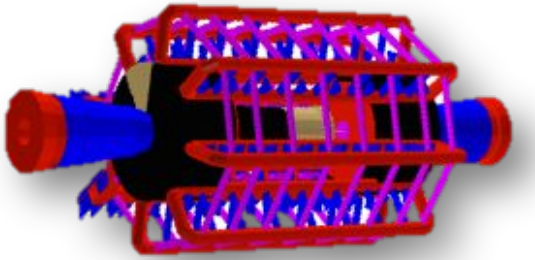


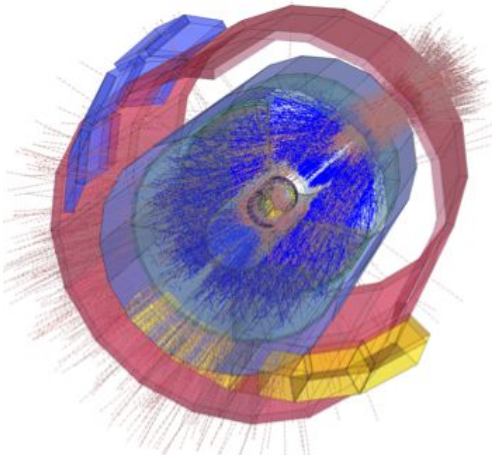
Can save graphics in many formats:
ps, pdf, svg, jpeg, LaTex, png, c, root …

Parallelism

- Many ongoing efforts to provide means for parallelisation in ROOT
- Explicit parallelism
  - **TThreadExecutor** and **TProcessExecutor**
  - Protection of resources
- Implicit parallelism
  - **TDataFrame**: functional chains
  - TTreeProcessor: process tree events in parallel
  - TTree::GetEntry: process of tree branches in parallel
- Parallelism is a fundamental element for tackling data analysis during LHC Run III and HL-LHC

Many More Features!





- Geometry Toolkit
  - Represent geometries as complex as LHC detectors
- Event Display (EVE)
  - Visualise particle collisions within detectors

The ROOT Prompt

- C++ is a compiled language
  - A compiler is used to translate source code into machine instructions
- ROOT provides a C++ **interpreter**
  - Interactive C++, without the need of a compiler, like Python, Ruby, Haskell …
    - Code is **Just-in-Time compiled!**
  - Allows reflection (inspect at runtime layout of classes)
  - Is started with the command:

```
root
```

  - The interactive shell is also called “ROOT prompt” or “ROOT interactive prompt”

ROOT As a Calculator

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \ldots = \sum_{n=0}^{\infty} x^n$$

Here we make a step forward.
We declare **variables** and use a ***for*** control structure.

```
root [0] double x=.5
(double) 0.5
root [1] int N=30
(int) 30
root [2] double gs=0;
```

```
root [3] for (int i=0;i<N;++i) gs += pow(x,i)
root [4] std::abs(gs - (1/(1-x)))
(Double_t) 1.86265e-09
```

Controlling ROOT

- Special commands which are not C++ can be typed at the prompt, they start with a “.”

```
root [1] .<command>
```

- For example:
  - To quit root use **.q**
  - To issue a shell command use **.! <OS_command>**
  - To load a macro use **.L <file_name>** (see following slides about macros)
  - **.help** or **.?** gives the full list

Interactivity

```
root [0] #include "a.h"
root [1] A o("ThisName"); o.printName()
ThisName
root [1] dummy()
(int) 42
```

a.h

```
# include <iostream>
class A {
public:
  A(const char* n) : m_name(n) {}
  void printName() { std::cout << m_name << std::endl; }
private:
  const std::string m_name;
};

int dummy() { return 42; }
```

ROOT Macros

- We have seen how to interactively type lines at the prompt
- The next step is to write “ROOT Macros” – lightweight programs
- The general structure for a macro stored in file *MacroName.C* is:

Function, no main, same name as the file

```
void MacroName() {
         <           ...
           your lines of C++ code
                     ...              >
}
```

Unnamed ROOT Macros

- Macros can also be defined with no name
- Cannot be called as functions!
  - See next slide :)

```
{
        <           ...
          your lines of C++ code
                    ...              >
}
```

Running a Macro

- A macro is executed at the system prompt by typing:

```
> root MacroName.C
```

- or executed at the ROOT prompt using .x:

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

- or it can be loaded into a ROOT session and then be run by typing:

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

Interpretation and Compilation

- We have seen how ROOT interprets and “just in time compiles” code. ROOT also allows to compile code “traditionally”. At the ROOT prompt:

```
root [1] .L macro1.C+
root [2] macro1()
```

Generate shared library and execute function

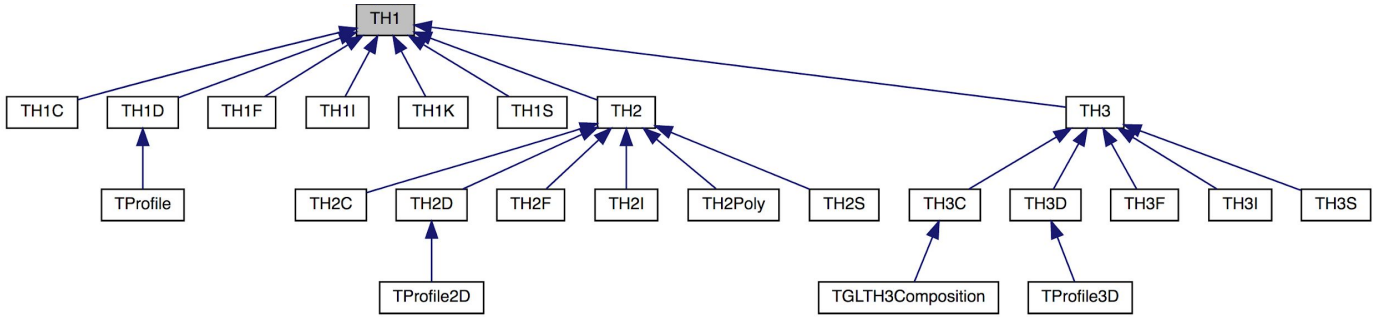
- ROOT libraries can also be used to produce standalone, compiled applications:

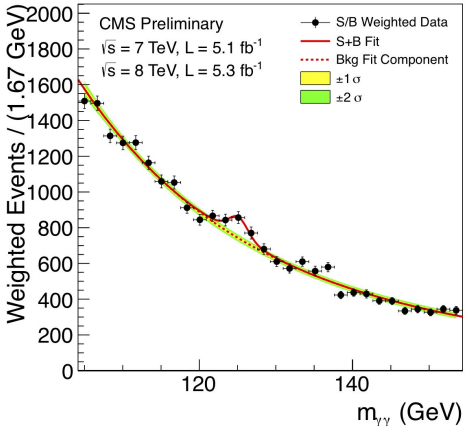
```
int main() {
  ExampleMacro();
  return 0;
}
```

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

Histograms

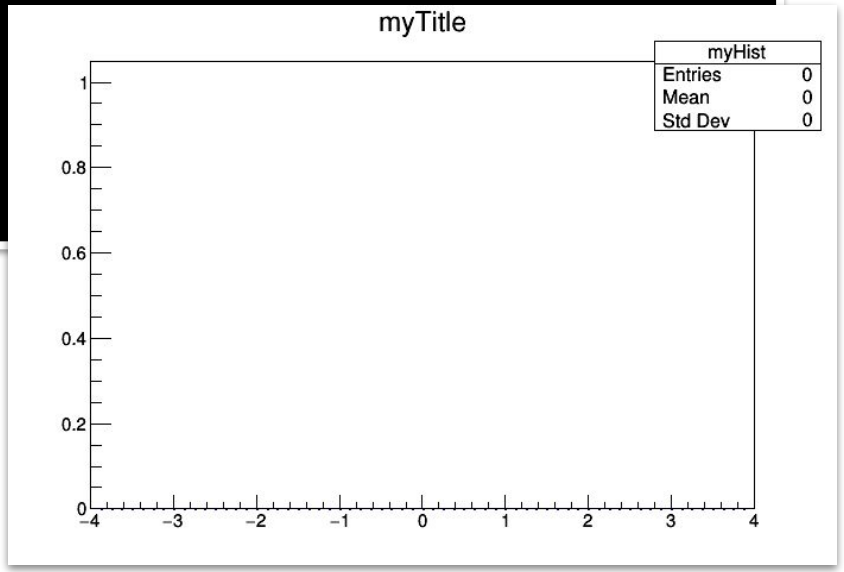
- Simplest form of data reduction
  - Can have billions of collisions, the Physics displayed in a few histograms
  - Possible to calculate momenta: mean, rms, skewness, kurtosis …
- Collect quantities in discrete categories, the bins
- ROOT Provides a rich set of histogram types
  - We'll focus on histogram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





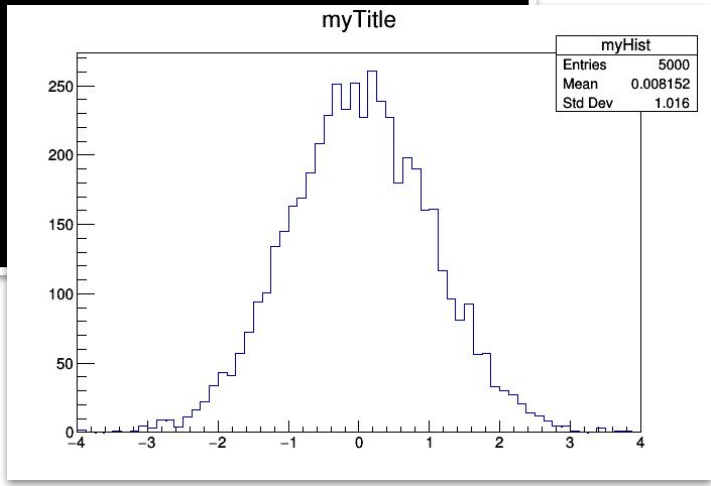
My First Histogram

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



My First Histogram

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



◆ FillRandom() [1/2]

```
void TH1::FillRandom ( const char * fname,
                       Int_t        ntimes = 5000
                     )
```

Fill histogram following distribution in function fname.

The distribution contained in the function fname (**TF1**) is integrated over the channel contents for the bin range of this histogram. It is normalized to 1.

Getting one random number implies:

- Generating a random number between 0 and 1 (say r1)
- Look in which bin in the normalized integral r1 corresponds to
- Fill histogram channel ntimes random numbers are generated

One can also call **TF1::GetRandom** to get a random variate from a function.

Reimplemented in **TH3**, and **TH2**.

Definition at line **3428** of file **TH1.cxx**.

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

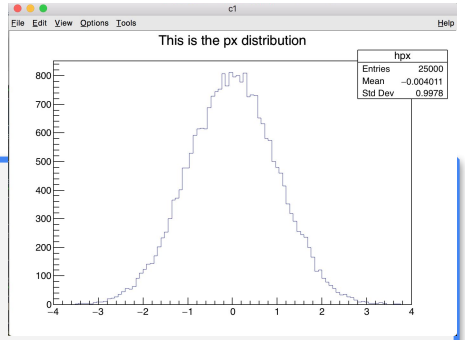
Bad for graphics:

```
// 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



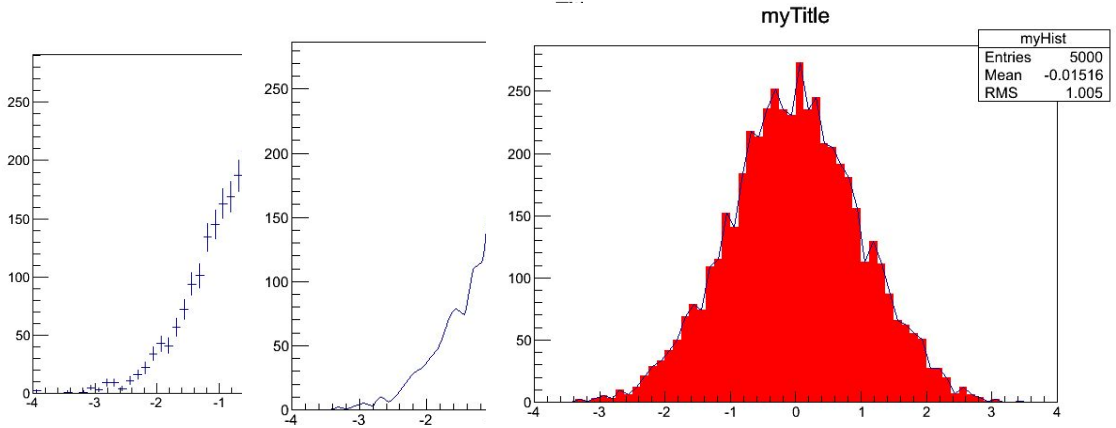

```
// makeHist.C:
void makeHist() {
  TH1F *hist = new TH1F("hist","My Histogram");
  hist->Draw(); // shows histogram
} // does not destroy hist!
```

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

TH1 options

https://root.cern.ch/doc/v608/classTHistPainter.html

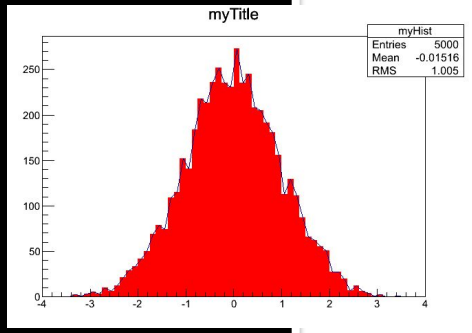
```
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw(“E”)
root [3] h.Draw(“C”)
root [4] h.SetFillColor(kRed)
root [5] h.Draw(“LF2”)
```

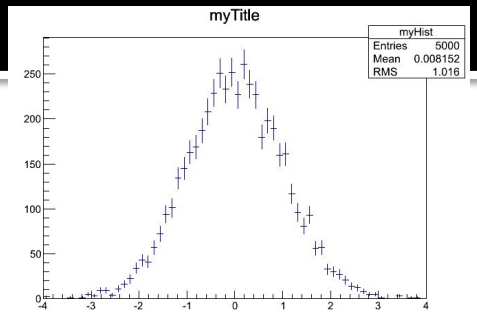


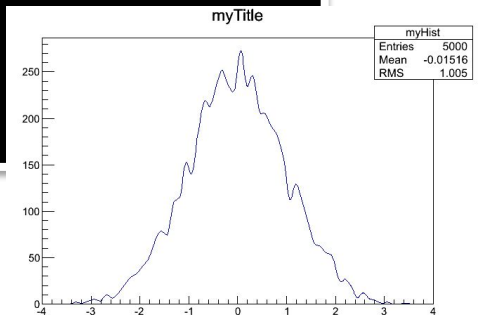
TCanvas

https://root.cern.ch/doc/v608/classTHistPainter.html

```
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw(“E”)
root [3] new TCanvas()
root [4] h.Draw(“C”)
root [5] h.SetFillColor(kRed)
root [6] c= new TCanvas()
root [7] h.Draw(“LF2”)
root [7] c->SaveAs(“aa.png”)
```







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.

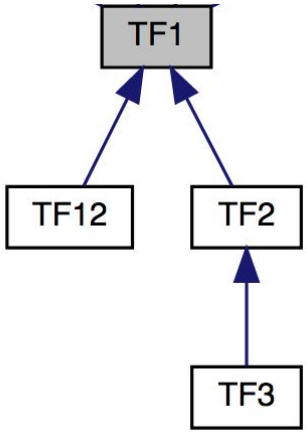
Also, after a fit, the fit result can be drawn on the plot and customized with `gStyle->SetOptFit()`

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



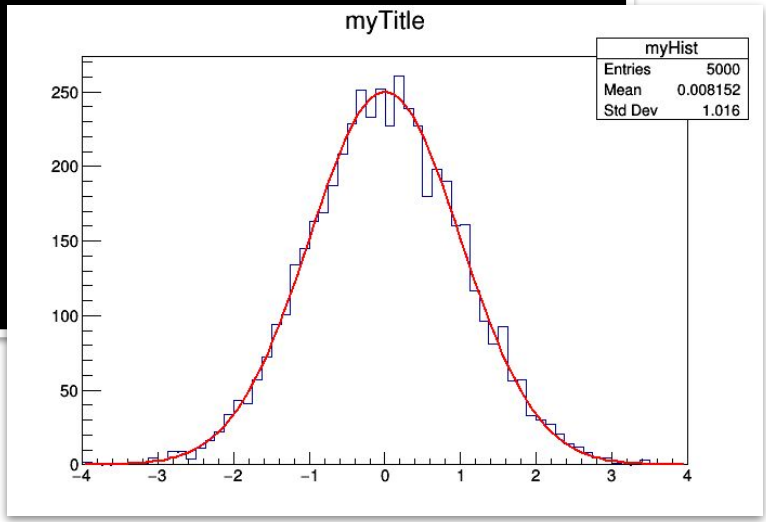
Functions

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)*):

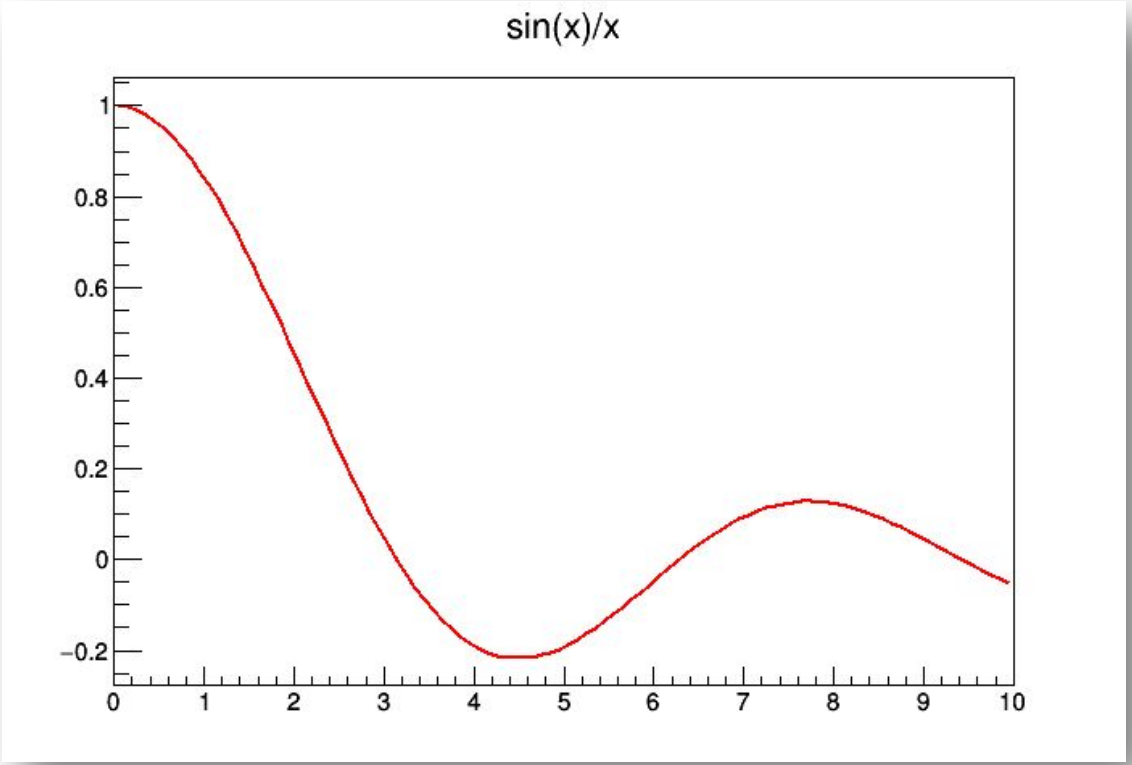
```
root [0] TF1 f1("f1","sin(x)/x",0.,10.);//name,formula, min, max
root [1] f1.Draw();
```

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

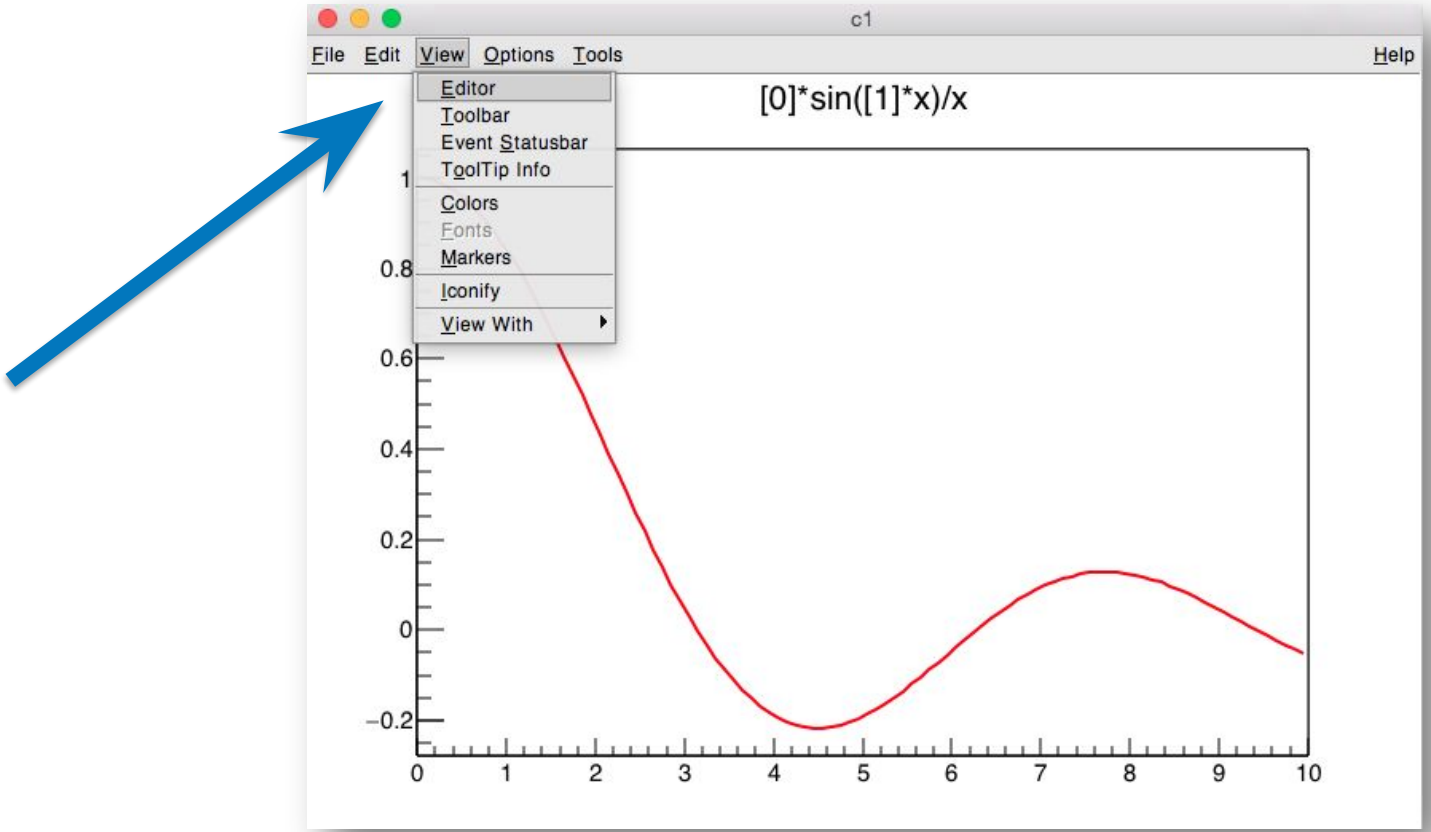
```
root [2] TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
root [3] f2.SetParameters(1,1);
root [4] f2.Draw();
```

Try it!

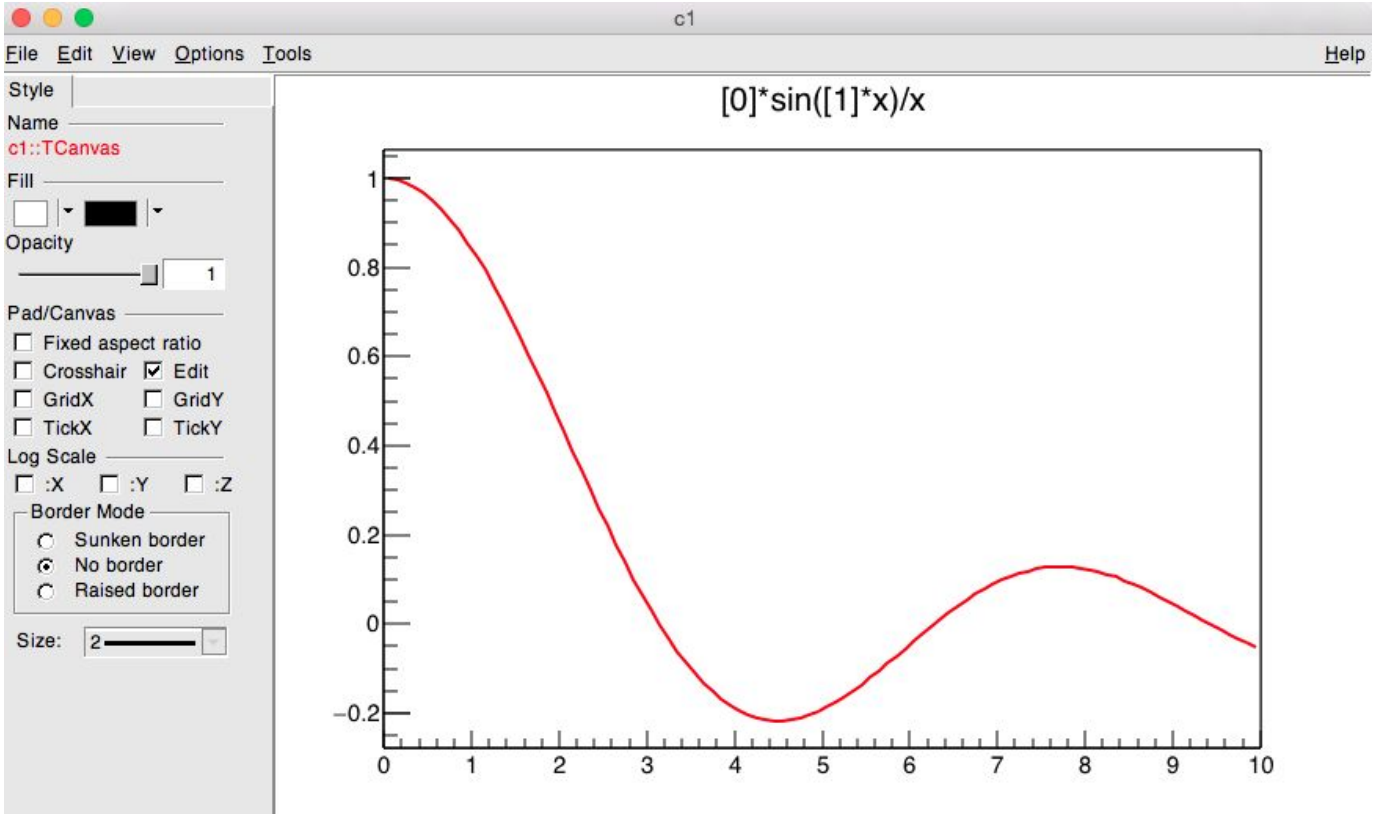
ROOT as a Function Plotter



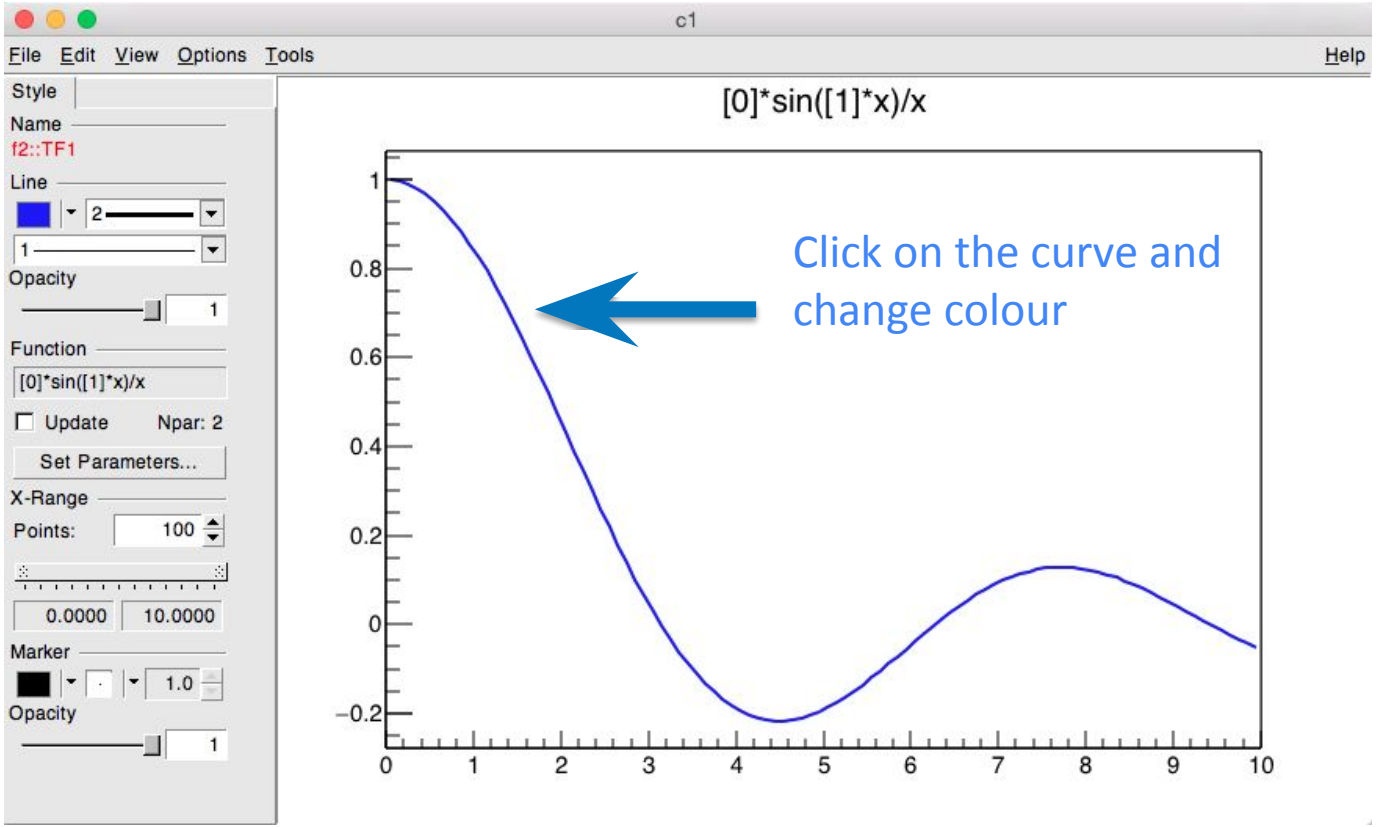
ROOT as a Function Plotter



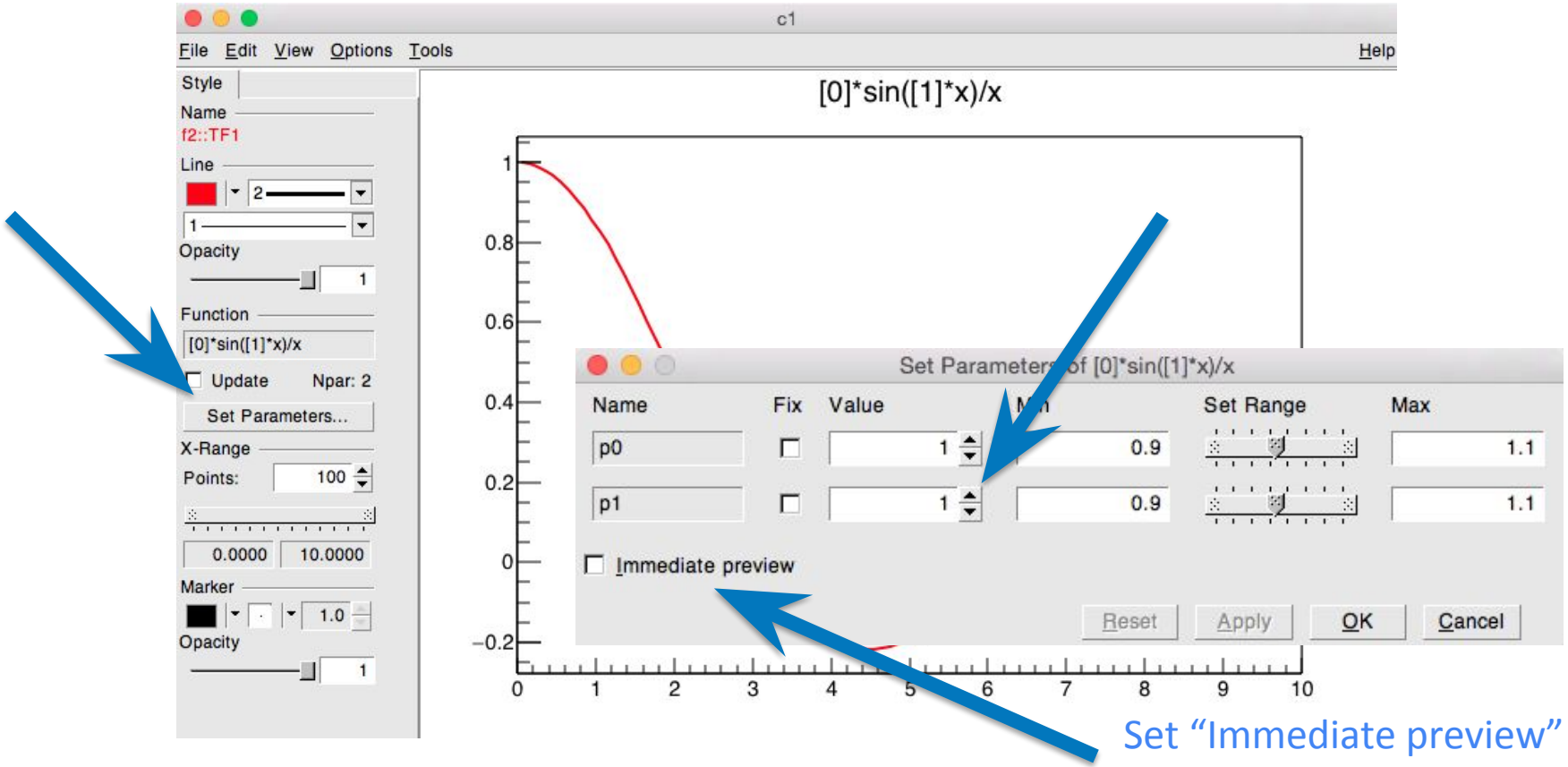
ROOT as a Function Plotter



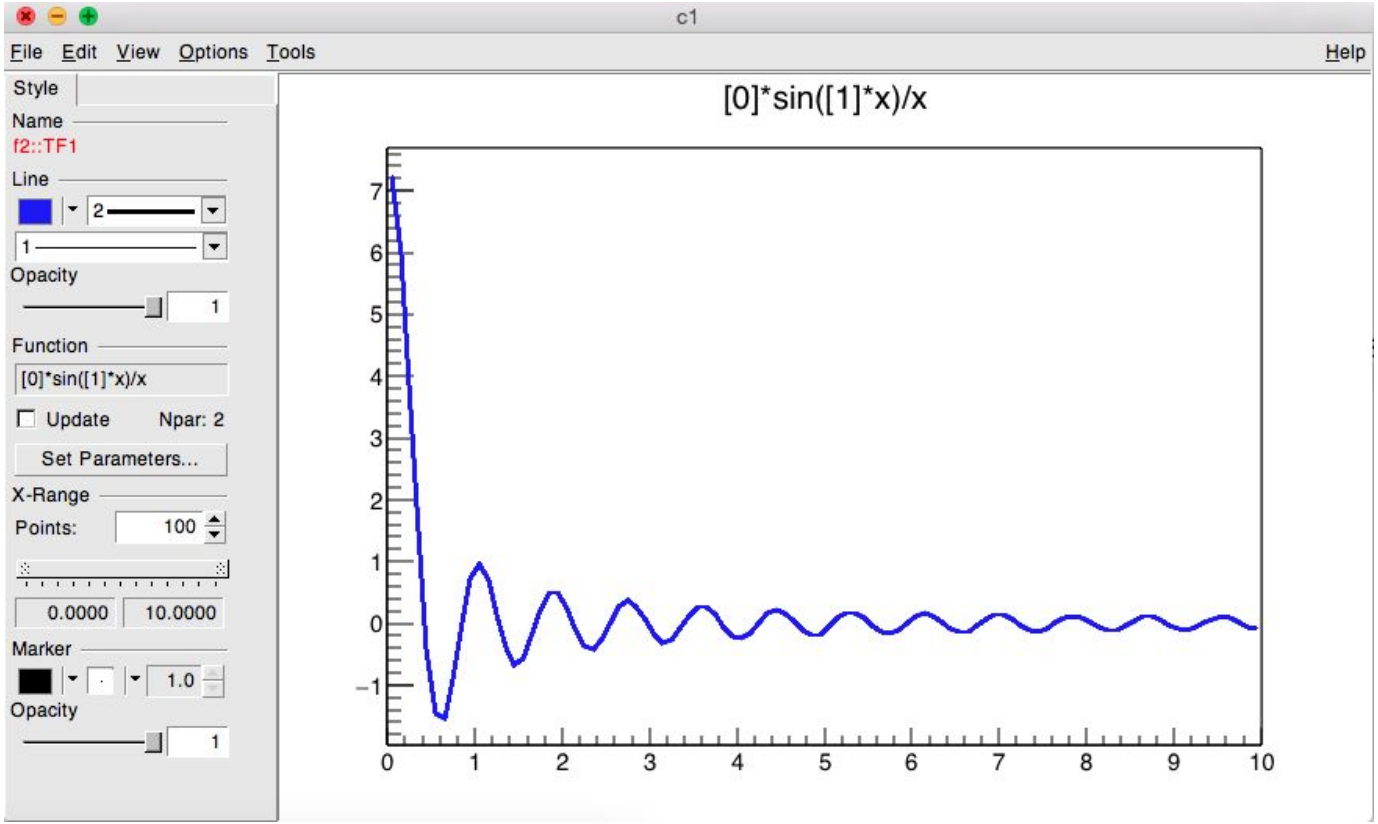
ROOT as a Function Plotter



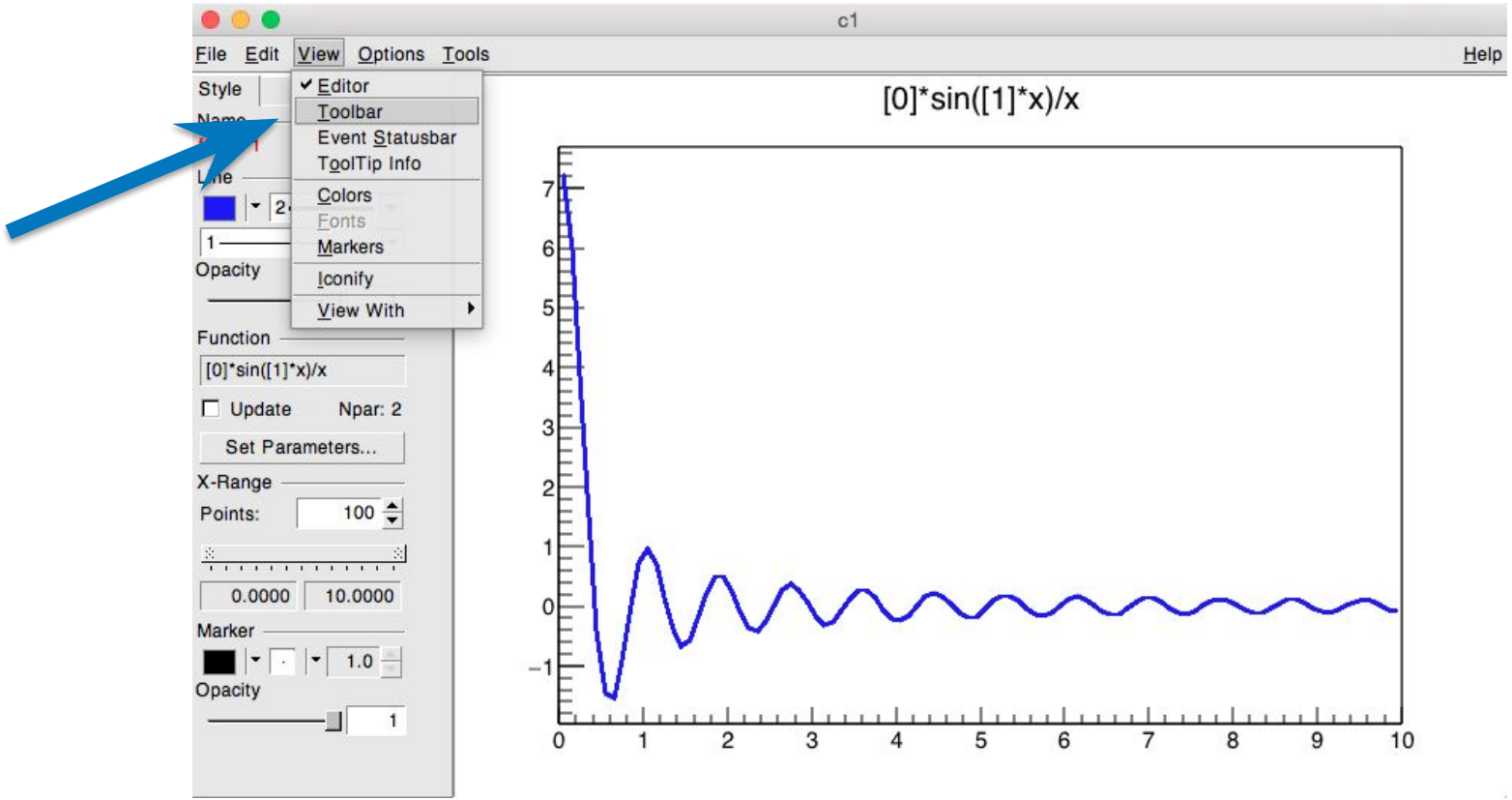
ROOT as a Function Plotter



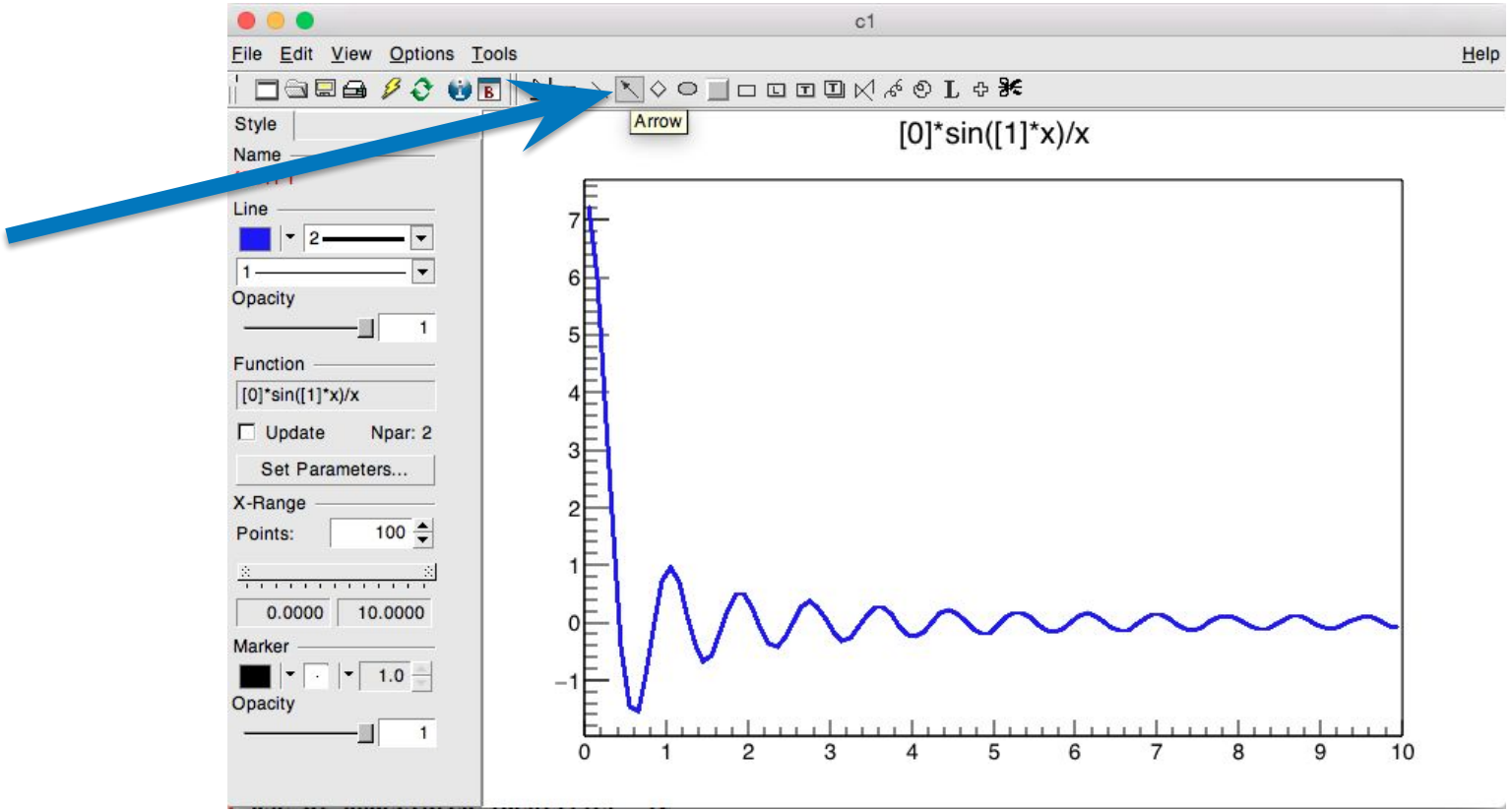
ROOT as a Function Plotter



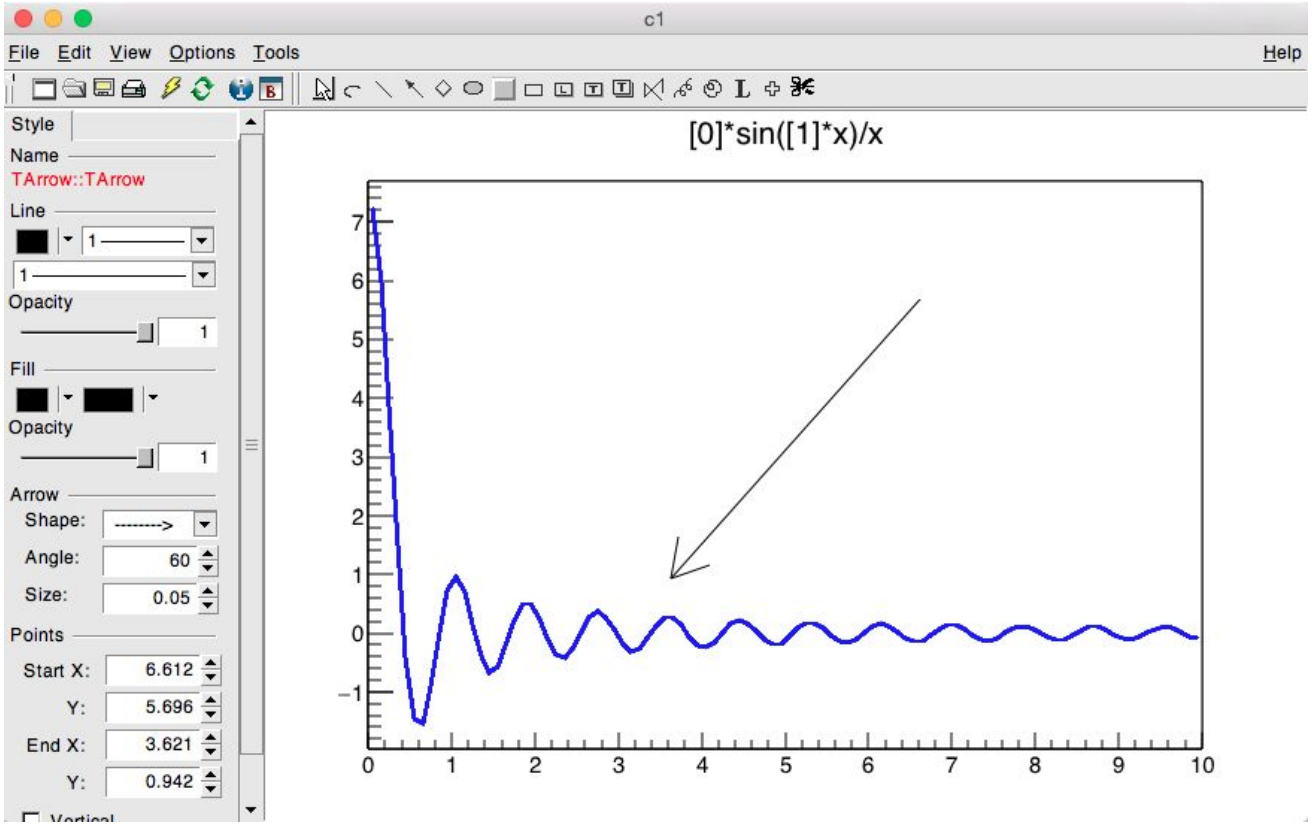
ROOT as a Function Plotter



ROOT as a Function Plotter

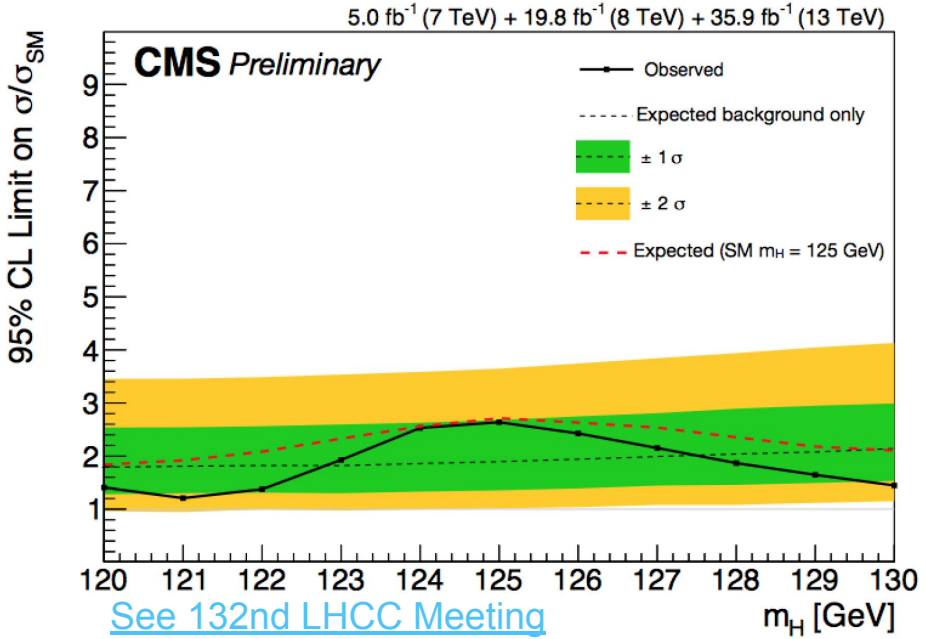


ROOT as a Function Plotter



TGraph

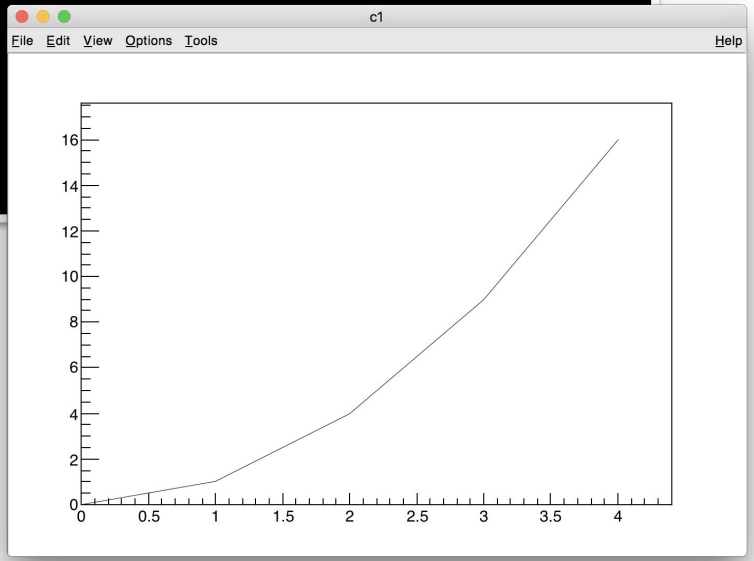
- Display points and errors
- Not possible to calculate momenta
- Not a data reduction mechanism
- **Fundamental to display trends**
- Focus on TGraph and TGraphErrors



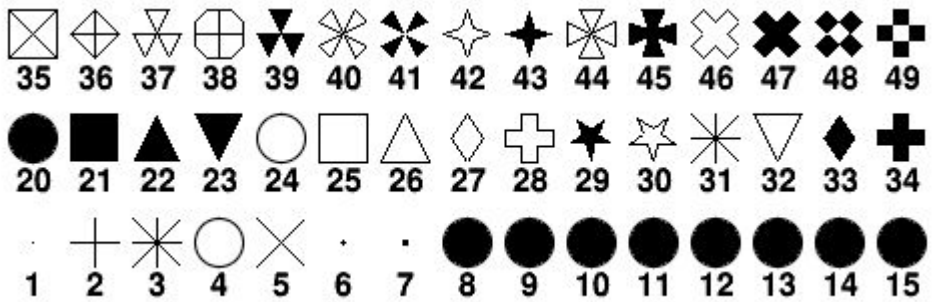
See 132nd LHCC Meeting

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")
```



The Markers

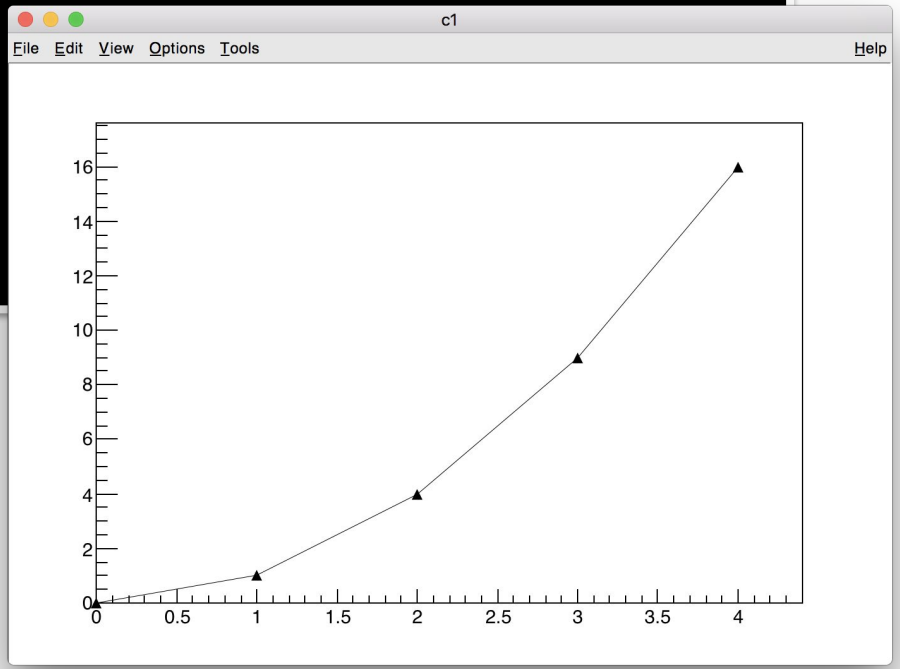



```
kDot=1, kPlus, kStar, kCircle=4, kMultiply=5,
kFullDotSmall=6, kFullDotMedium=7, kFullDotLarge=8,
kFullCircle=20, kFullSquare=21, kFullTriangleUp=22,
kFullTriangleDown=23, kOpenCircle=24, kOpenSquare=25,
kOpenTriangleUp=26, kOpenDiamond=27, kOpenCross=28,
kFullStar=29, kOpenStar=30, kOpenTriangleDown=32,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

Also available through more friendly names 😊

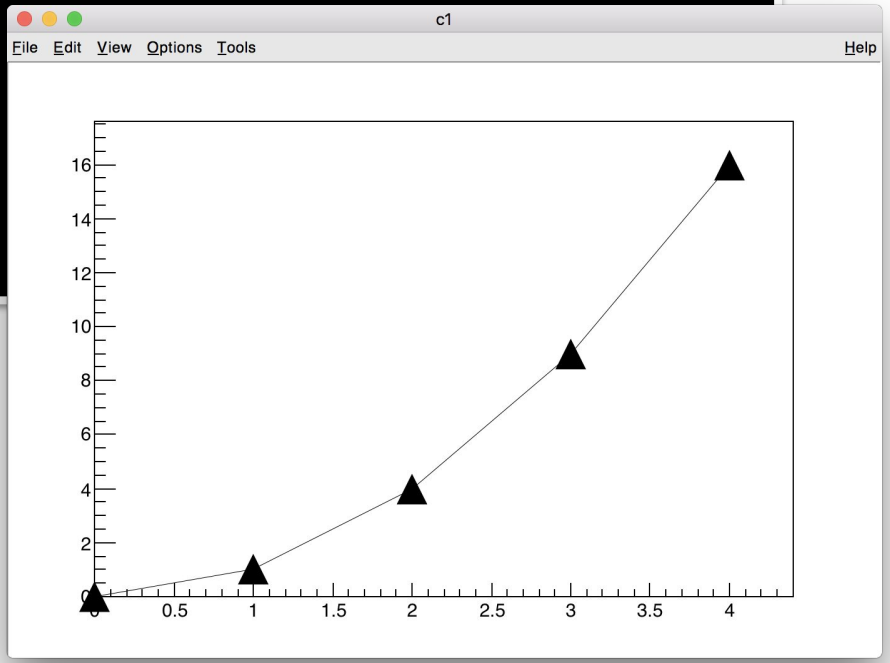
My First Graph

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

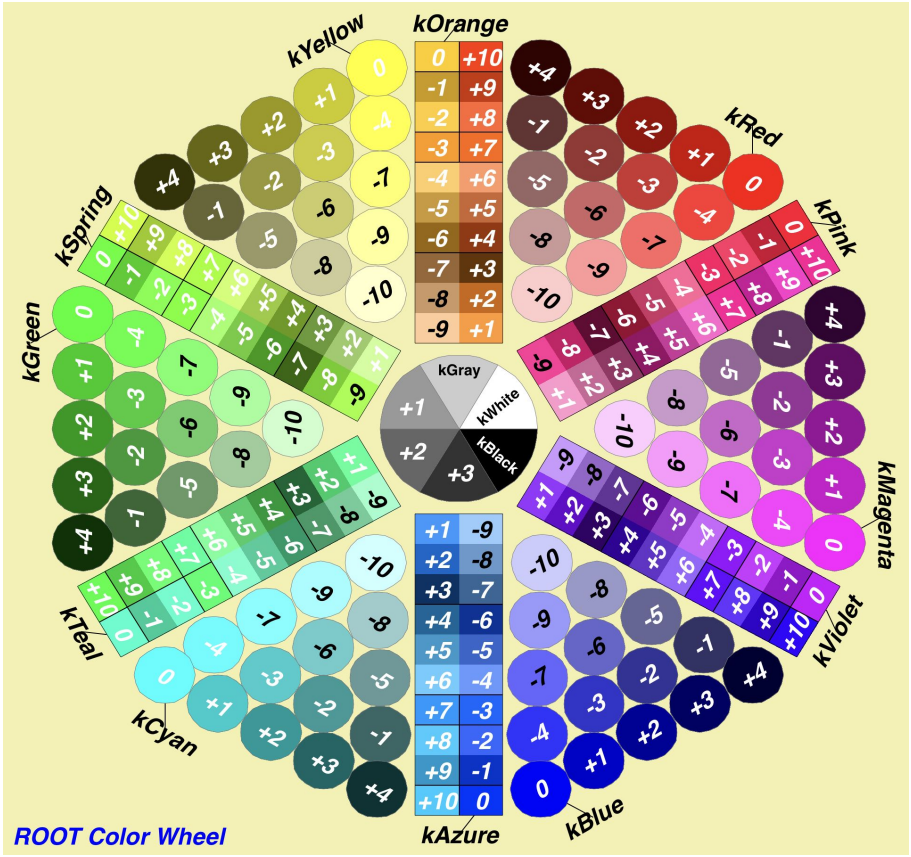


My First Graph

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

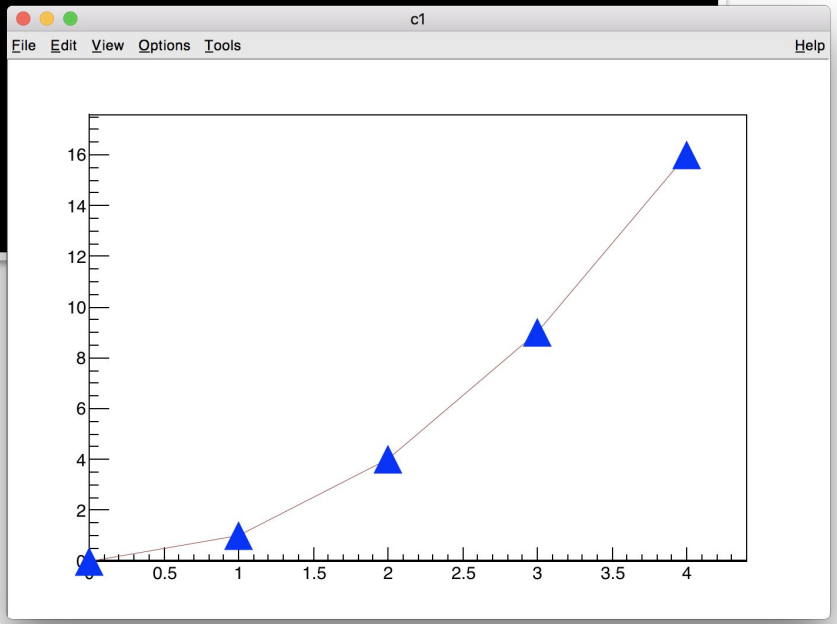


The Colors (TColorWheel)



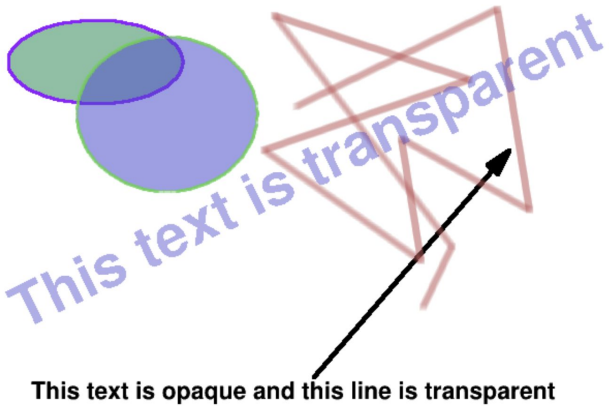
My First Graph

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



Transparency

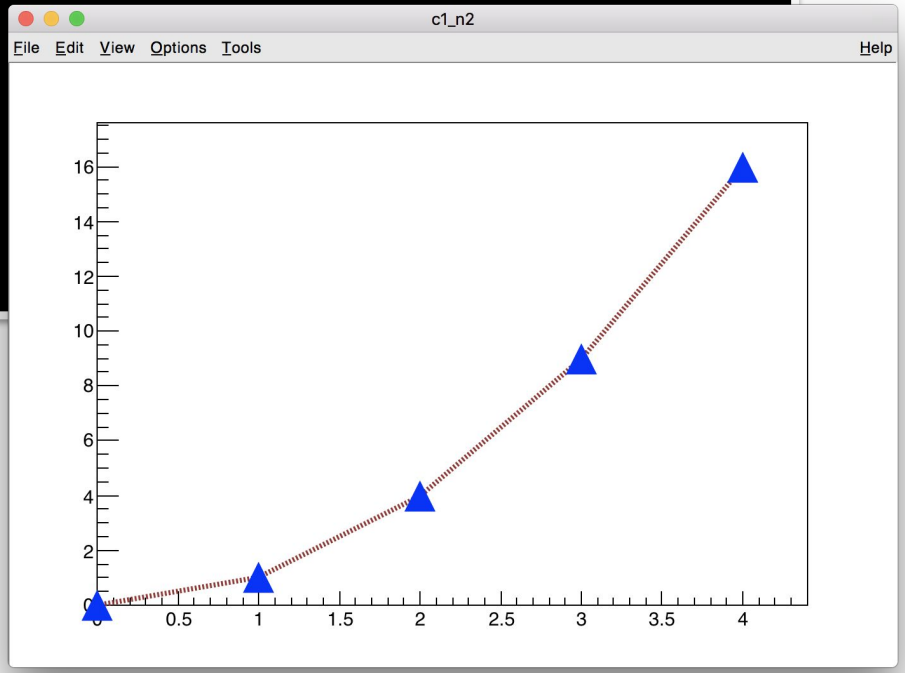
- **The transparency** is set via similar setters but with the keyword "Alpha" at the end. For instance `histo->SetFillColorAlpha(kBlue,0.35)` will set the color of the histogram histo to red with and alpha channel equal to 0.35 (0 = fully transparent and 1 = fully opaque).



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 ...

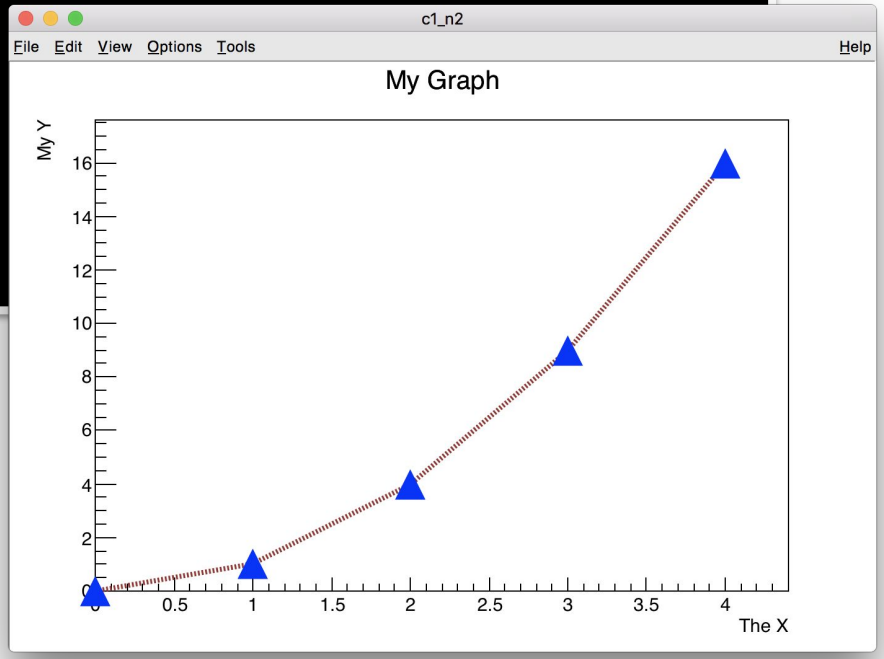
My First Graph

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



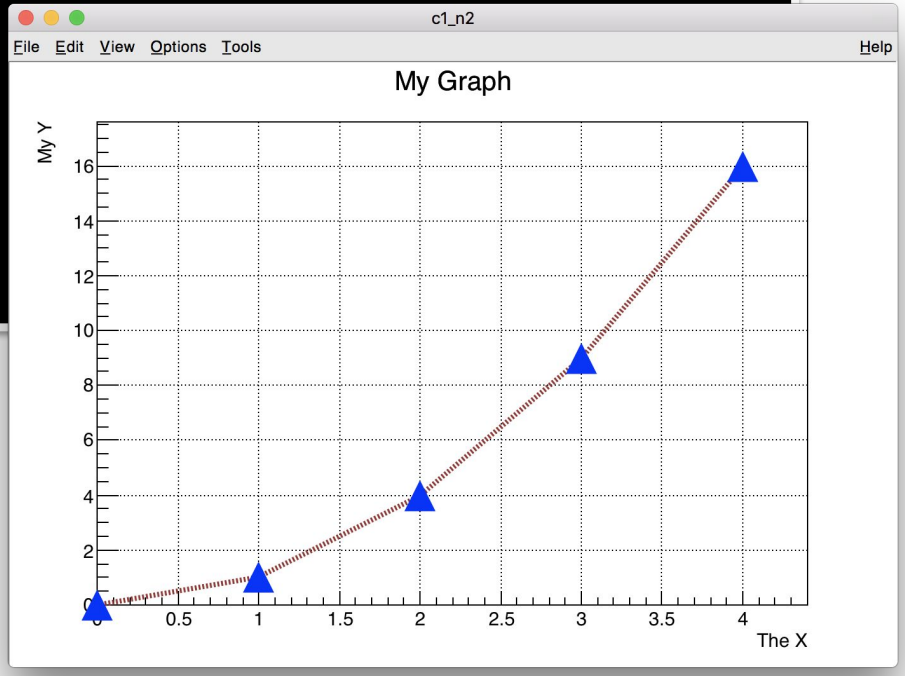
My First Graph

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



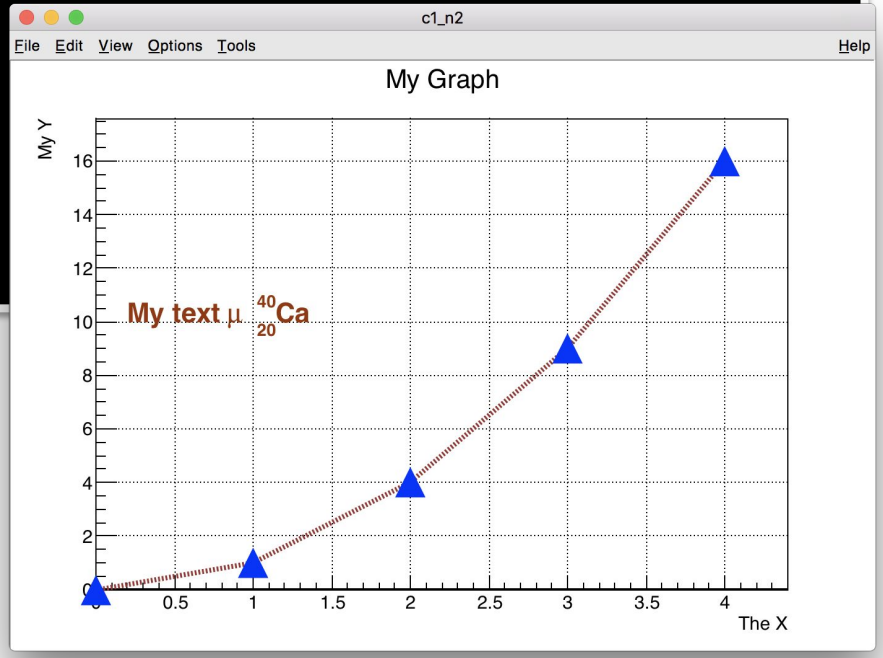
My First Graph

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



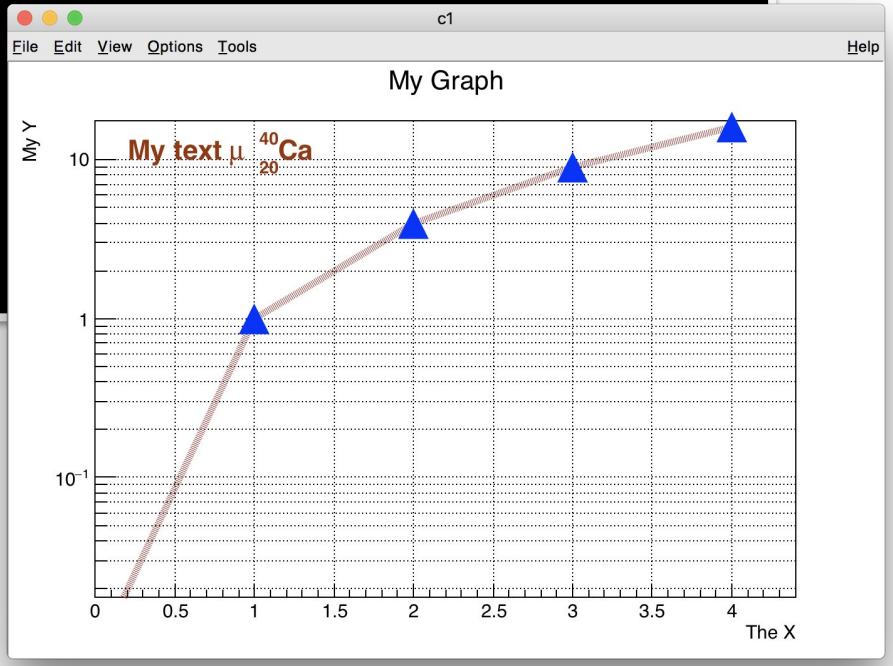
My First Graph

```
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

Features of a Good Plot

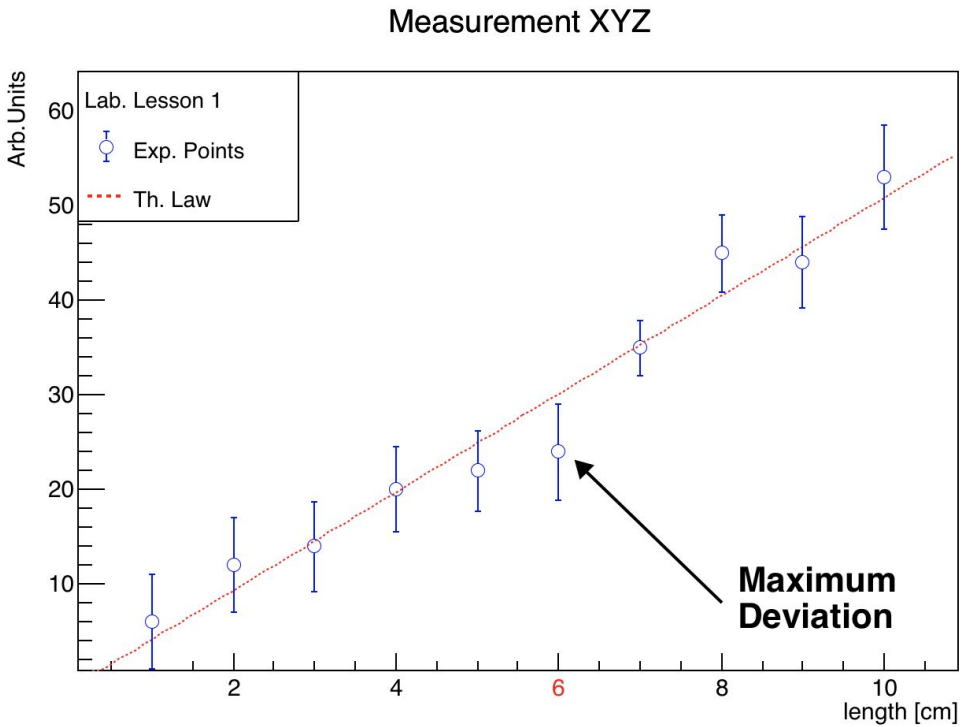
Every plot should be **self-contained** and deliver a **clear message**, even if extracted from the publication in which it's shown.

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

We will try to build a plot looking like this this afternoon



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".

2D plots with color map (1)

A very common way to represent 2D histograms is the **color plot**. We will use the following macro to illustrate this kind of plot.

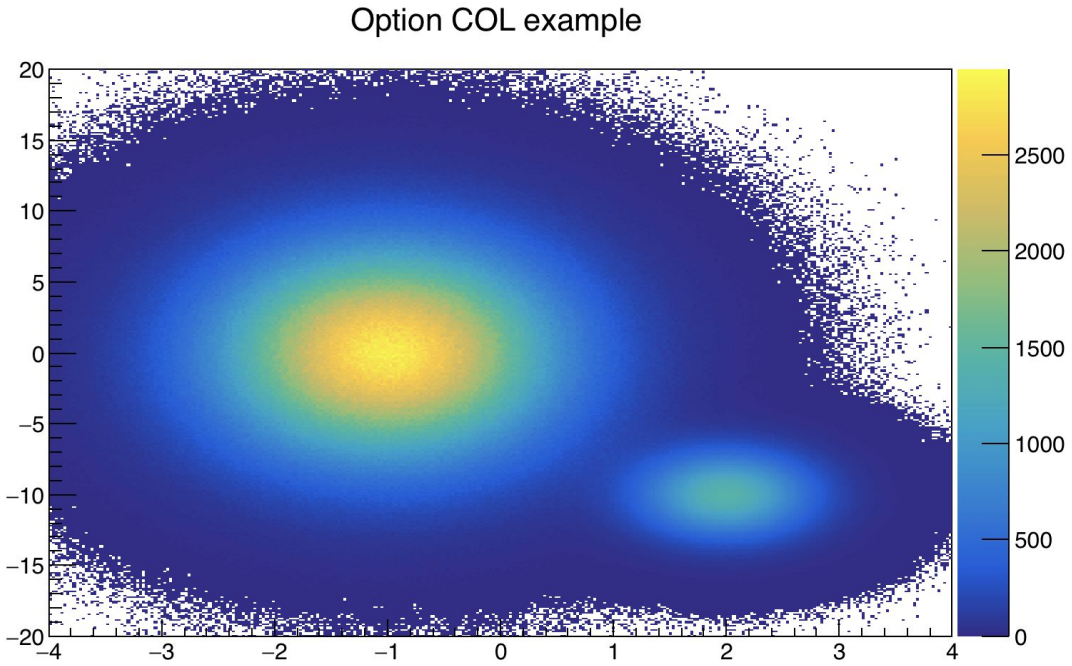
```
void macro2(){
   TH2F *h = new TH2F("h","Option COL example ",300,-4,4,300,-20,20);
   h->SetStats(0);
   h->SetContour(200);
   float px, py;
   for (int i = 0; i < 25000000; i++) {
      gRandom->Rannor(px,py);
      h->Fill(px-1,5*py);
      h->Fill(2+0.5*px,2*py-10.,0.1);
   }
   h->Draw("colz");
}
```

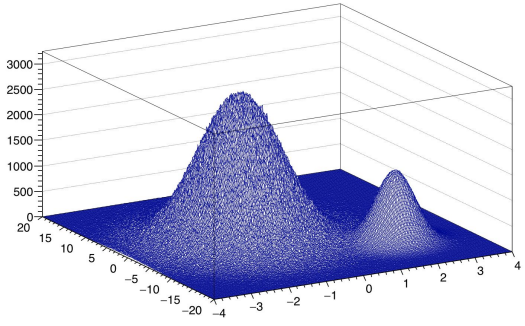
https://github.com/root-project/training/tree/master/BasicCourse/Exercises/Graphics

macro2.C

2D plots with color map (2)

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

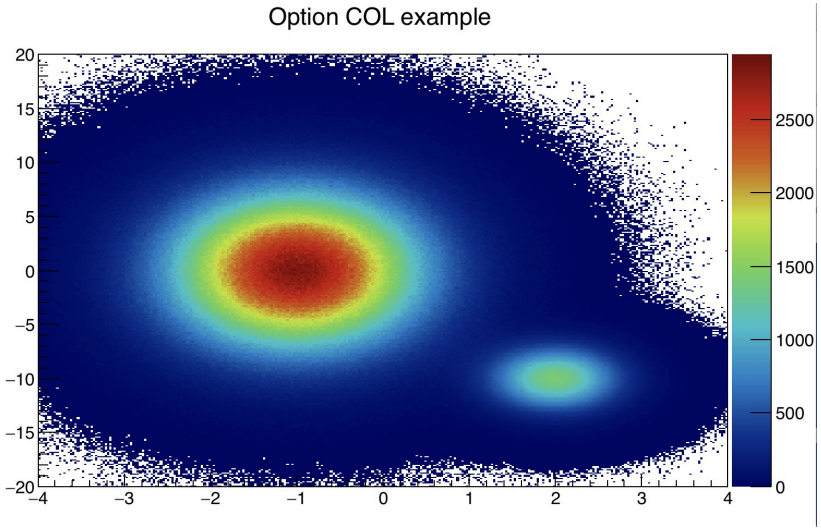




The default ROOT color map (kBird) render perfectly the smoothness of the dataset.

Danger of the Rainbow color map (1)

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

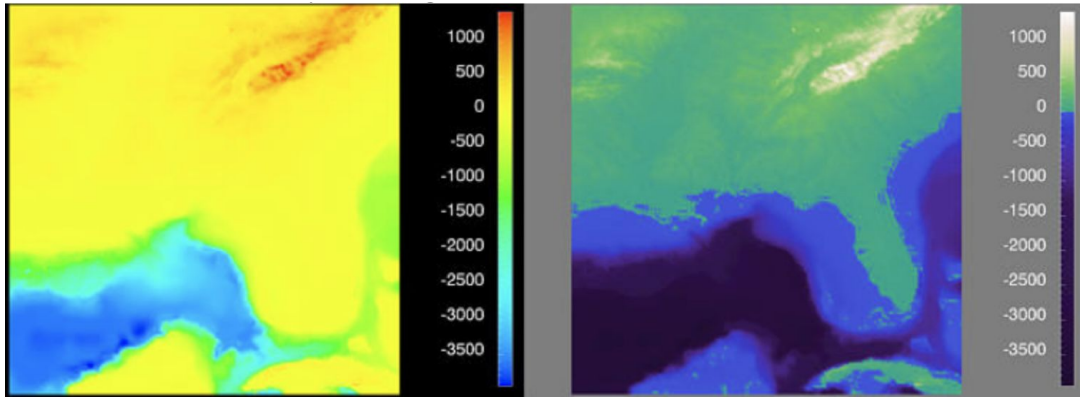


The reasons why this is not a good is explained in detail here.

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:

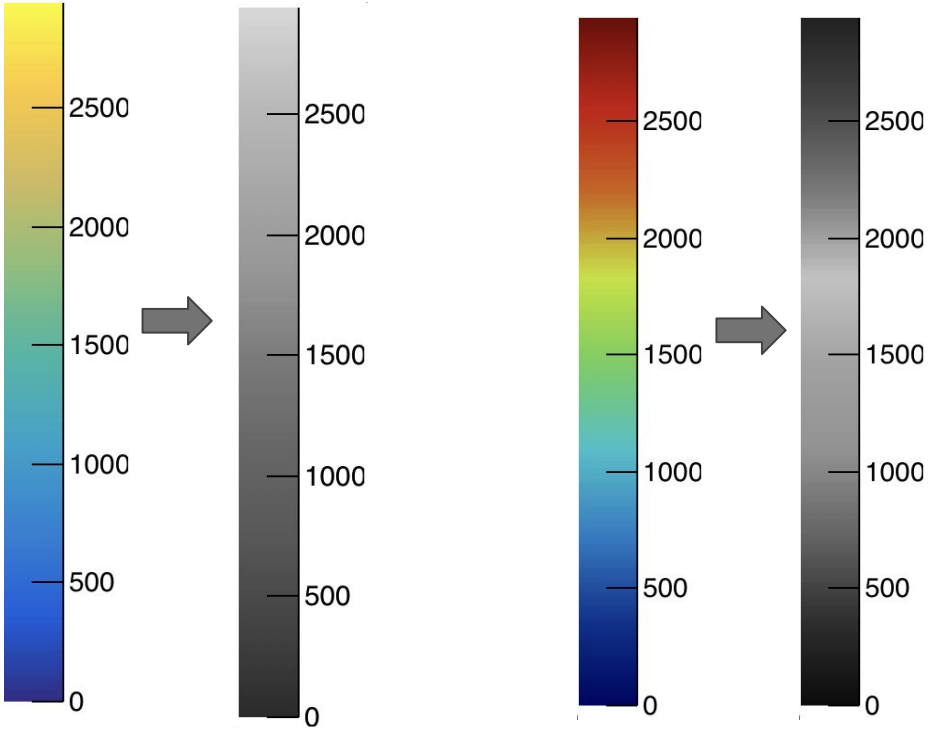
```
canvas->SetGrayscale();
```

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

macro2.C -> Create a TCanvas, draw the histogram and activate the gray scale!

The ROOT standard color maps (2)

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

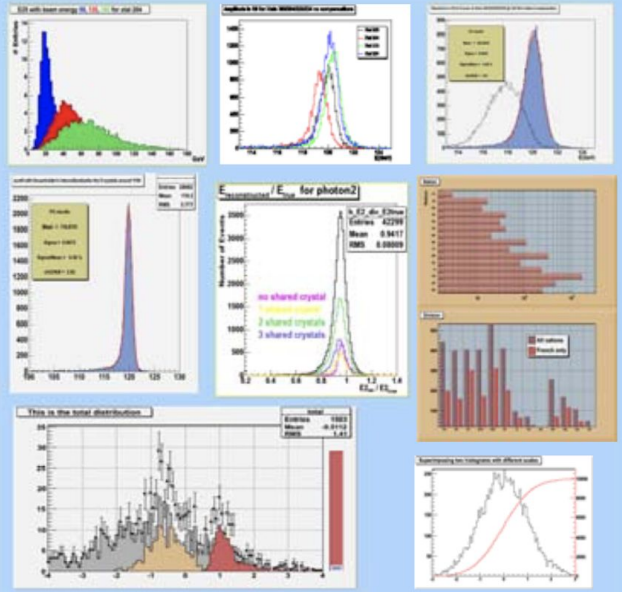


This is another good test showing the validity of a color map. In gray scale the Rainbow color map shows its luminance issue as the minimum and maximum values appear the same.

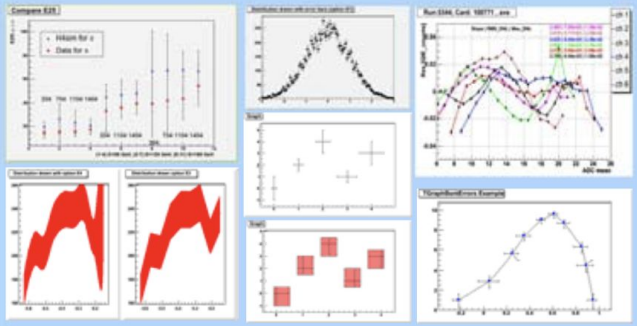
More ROOT drawing capabilities

2 variables visualization techniques

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

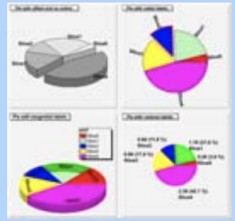


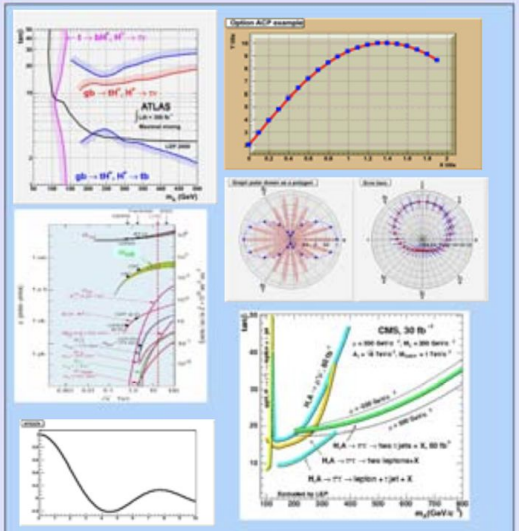
Bar charts and lines are a common way to represent 1D histograms.



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.

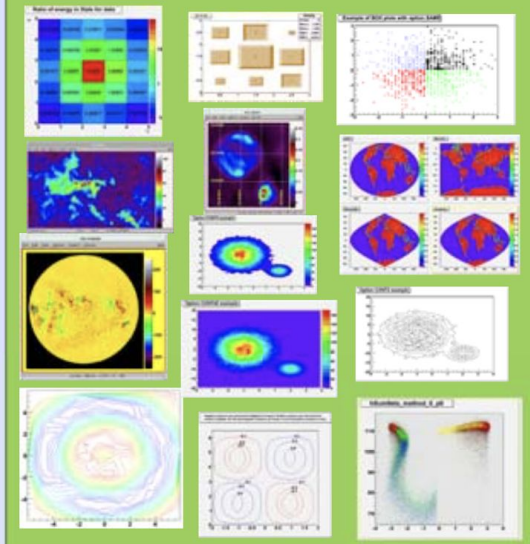




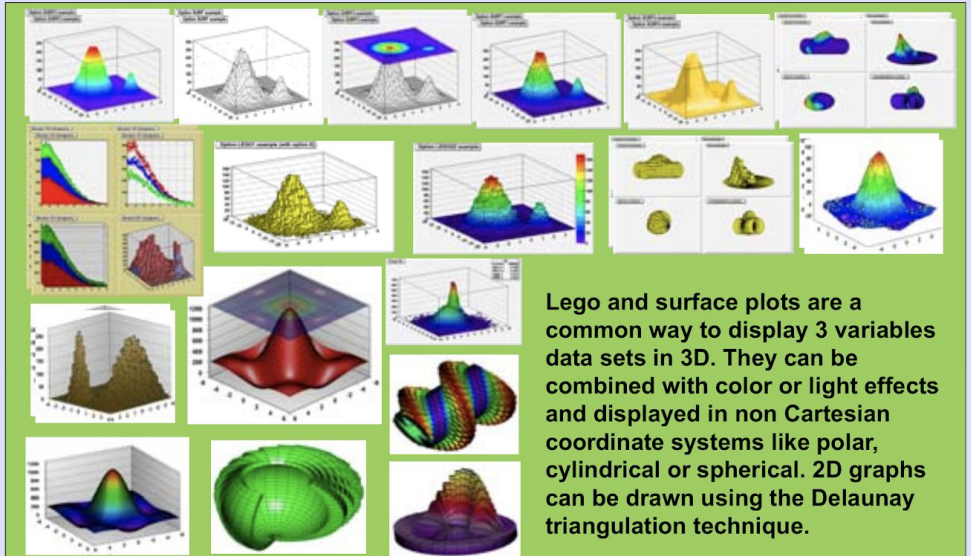
Graphs can be drawn as simple lines, like functions. They can also visualize exclusion zones or be plotted in polar coordinates.

3 variables visualization techniques

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



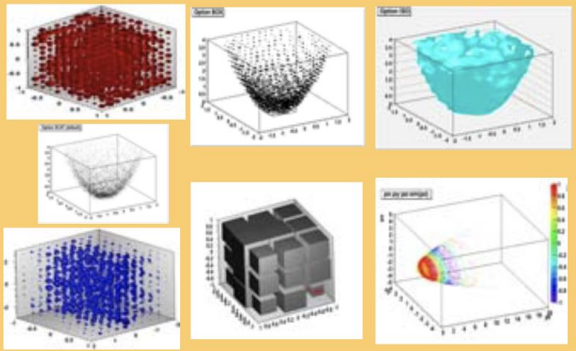
Several techniques are available to visualize 3 variables data sets in 2D. Two variables are mapped on the X and Y axis and the 3rd one on some graphical attributes like the color or the size of a box, a density of points (scatter plot) or simply by writing the value of the bin content. The 3rd variable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



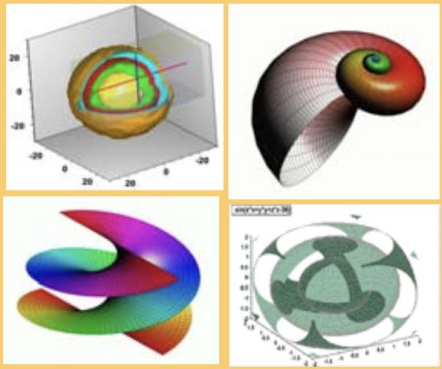
Lego and surface plots are a common way to display 3 variables data sets in 3D. They can be combined with color or light effects and displayed in non Cartesian coordinate systems like polar, cylindrical or spherical. 2D graphs can be drawn using the Delaunay triangulation technique.

4 variables visualization techniques

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



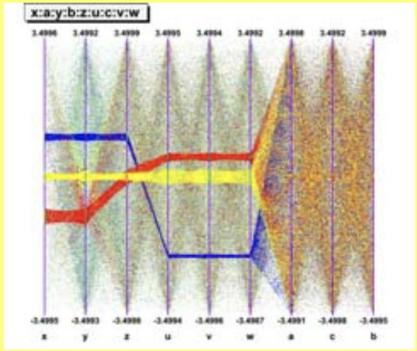
The 4 variables data set representations are extrapolations of the 3 variables ones. Rectangles become boxes or spheres, contour plots become iso-surfaces. The scatter plots (density plots) are drawn in boxes instead of rectangles. The 4th variable can also be mapped on colors. The use of OpenGL allows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



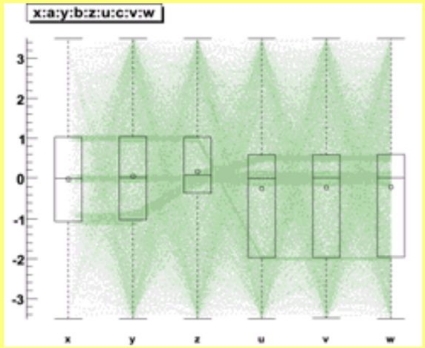
Functions like t= f(x,y,z) and 3D histograms are 4 variables objects. ROOT can render using OpenGL. It allows to enhance the plots' quality and the interactivity. Cutting planes, projection and zoom allow to better understand the data set or function.

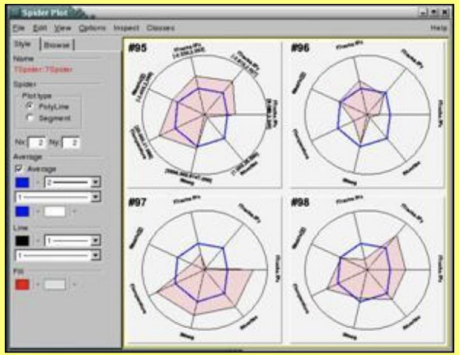
N variables visualization techniques

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



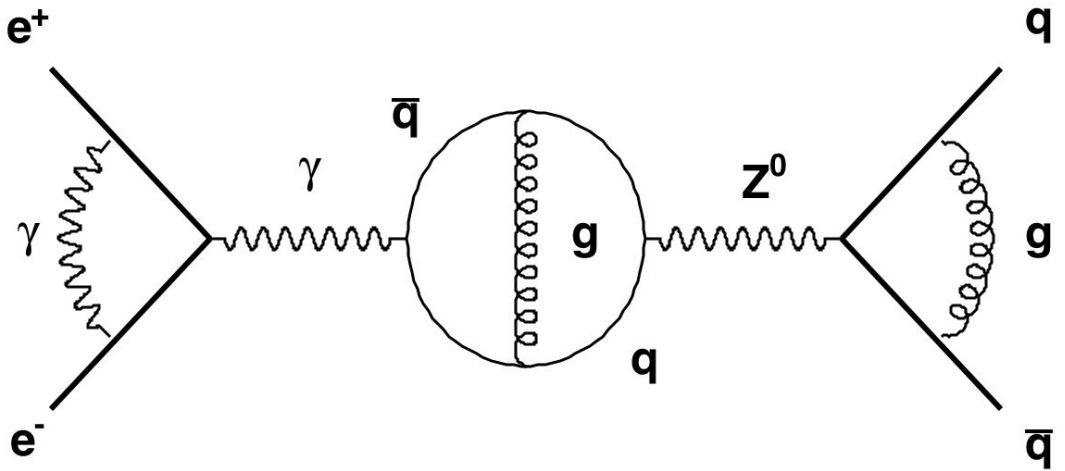
Above 4 variables more specific visualization techniques are required; ROOT provides three of them. The parallel coordinates (on the left) the candle plots (in the middle) which can be combined with the parallel coordinates. And the spider plot (on the right also). These three techniques, and in particular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





Feynman diagrams

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



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")
```

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

- Fetch the *myClass* representation of ROOT

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

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

- Same for collections of many other entities

Using PyROOT

- Entry point to use ROOT from within Python

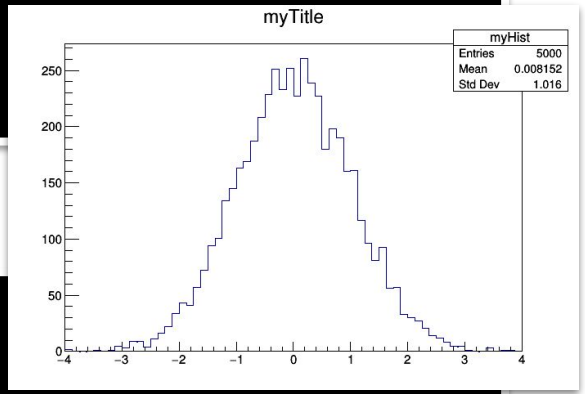
```
import ROOT
```

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

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

Example: C++ to Python

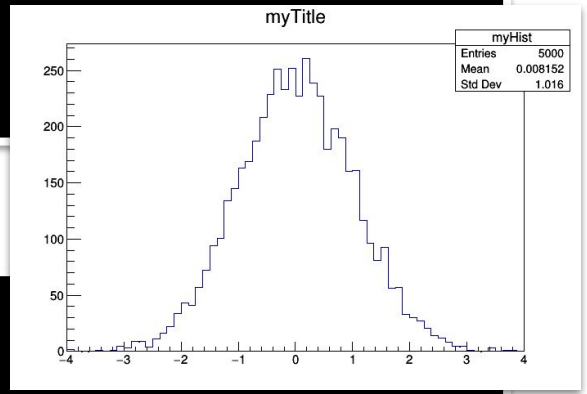
```
> root
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
```




```
> 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

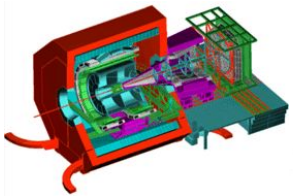
my_python_module.py

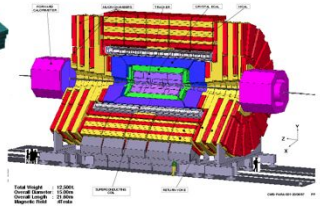
```
# Load a C++ library
ROOT.gInterpreter.ProcessLine('#include "my_cpp_library.h"')
ROOT.gSystem.Load('./my_cpp_library.so')

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

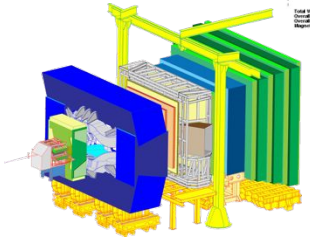
Reading and Writing Data

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

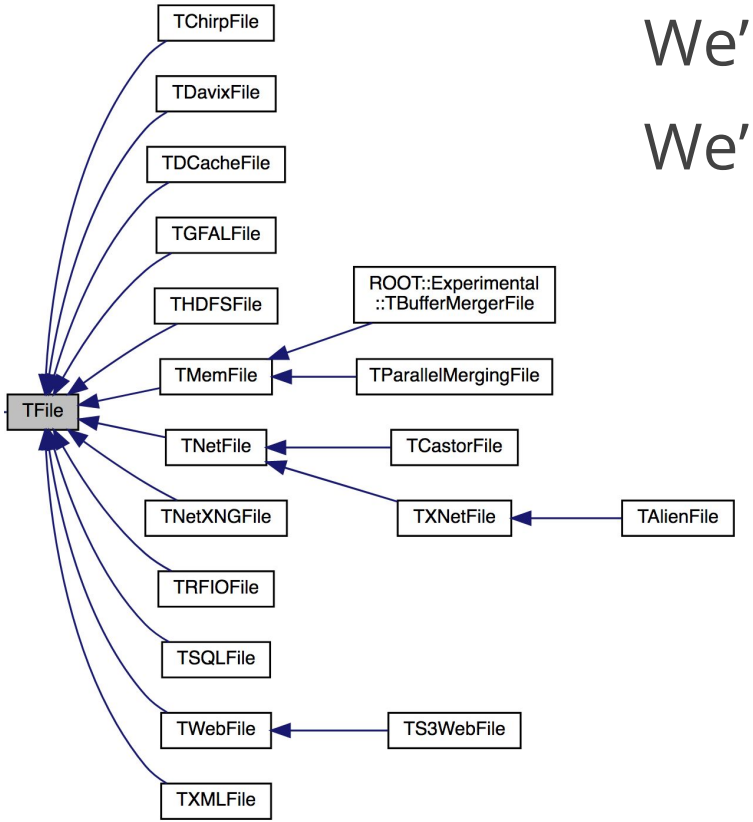
Data Storage: Local, Network

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.

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:

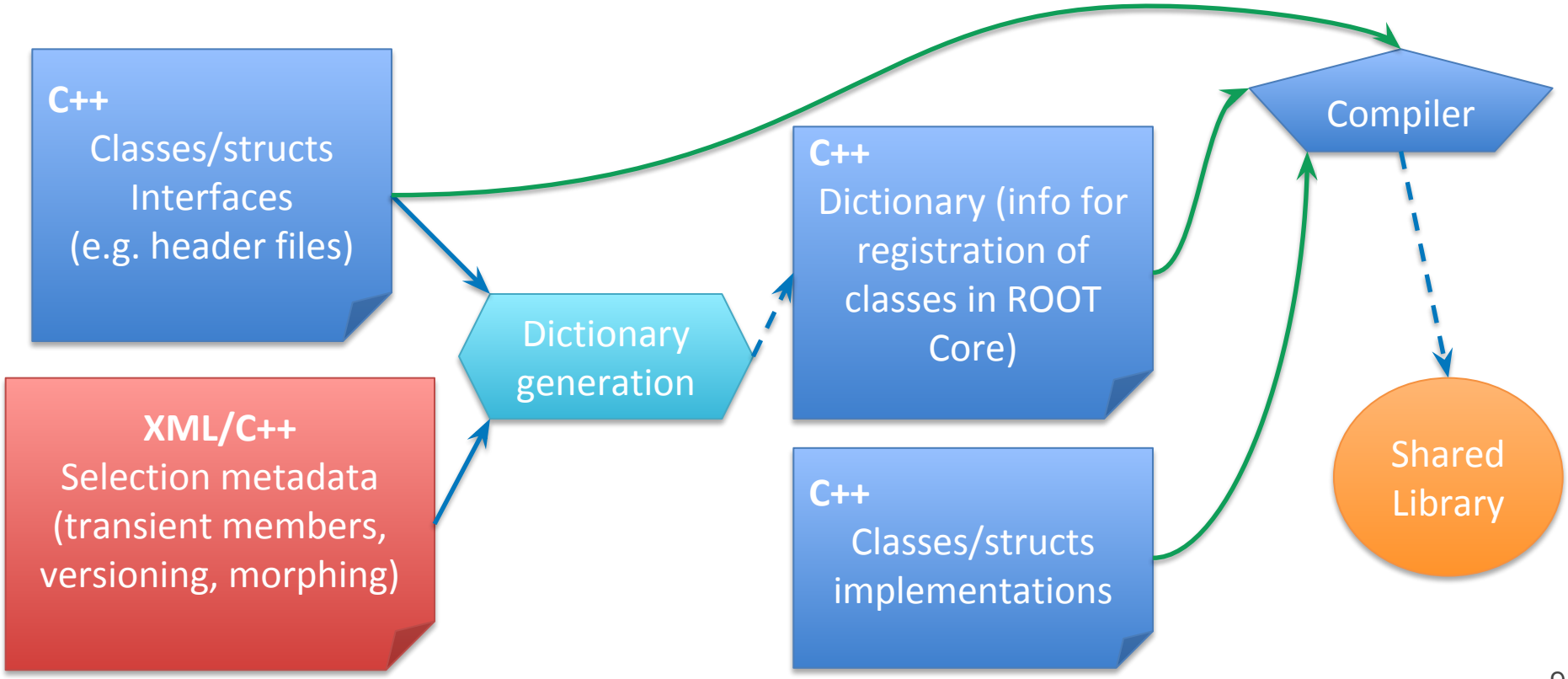
1. Serialisation: from an object in memory to a blob of bytes
2. Compression: use an algorithm to reduce size of the blob (e.g. zip, lzma, lz4)
3. “Real” writing via OS primitives

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



TFile in Action

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

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 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

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

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

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

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

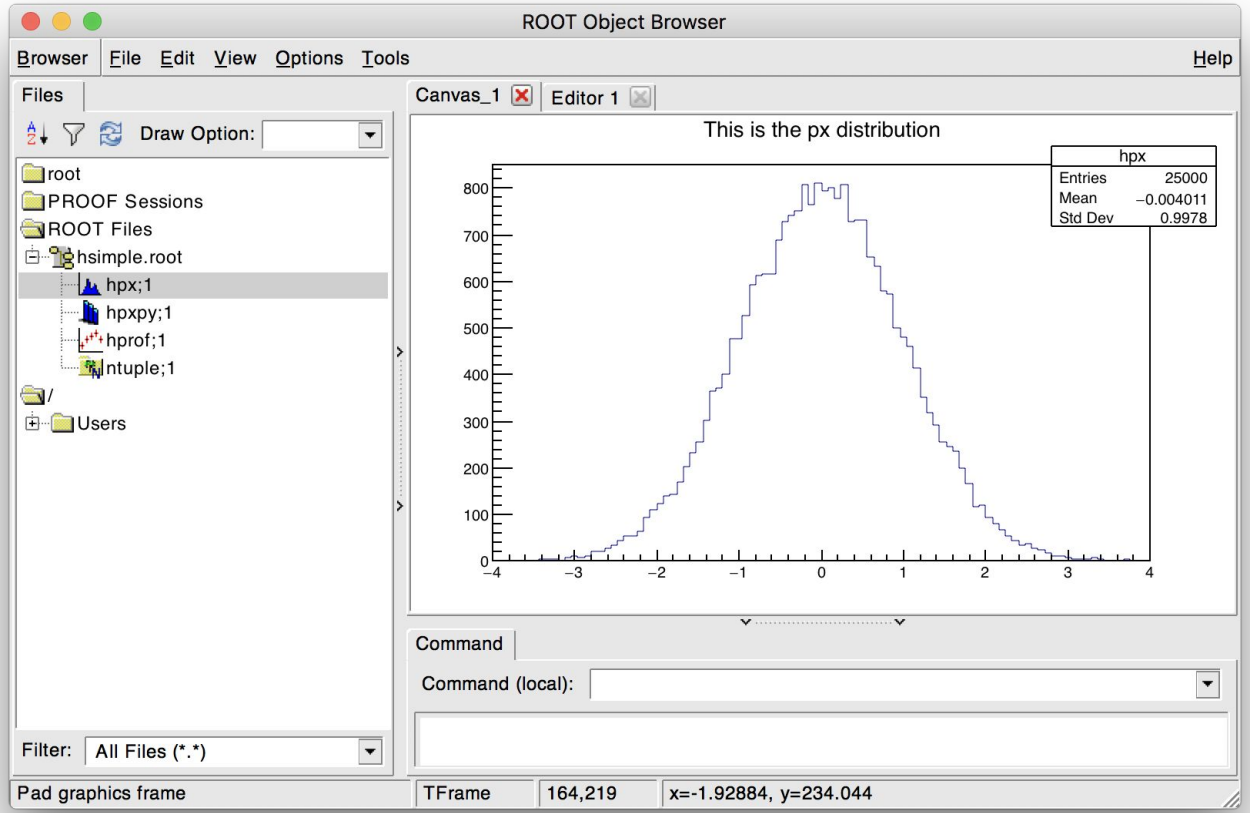
```
import ROOT
f = ROOT.TFile("myfile.root")
f.h.Draw()
```

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

```
TFile f("myfile1.root");
for (auto k : *f.GetListOfKeys()) {
   std::cout << k->GetName() << std::endl;
}
```

Loop on TKeys: Python

List comprehension syntax!

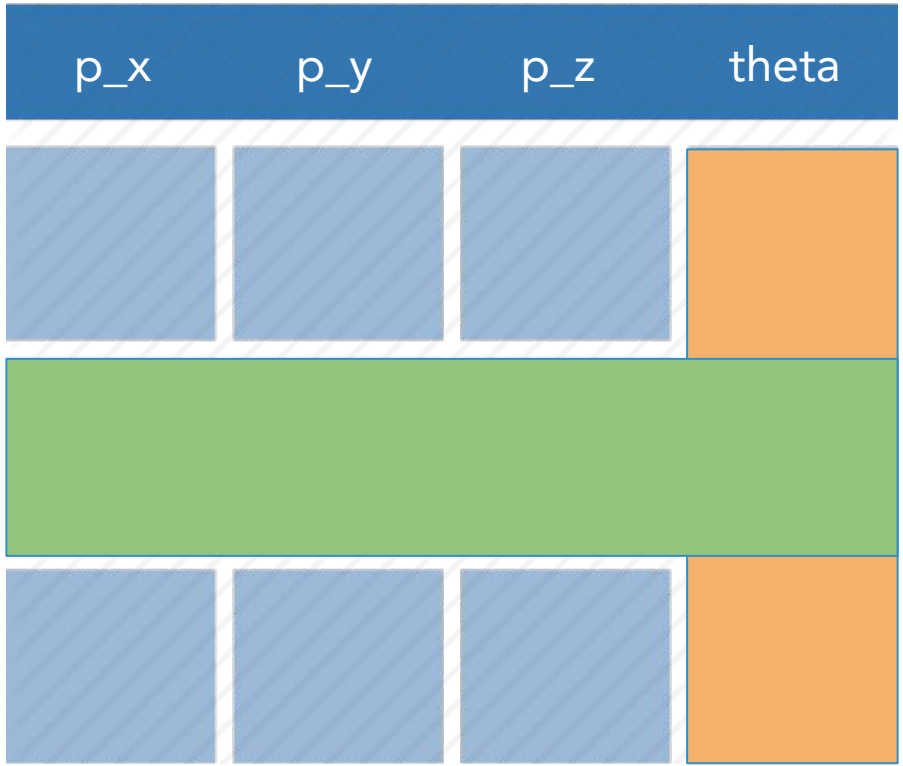
```
import ROOT
f = ROOT.TFile("myfile.root")
keyNames = [k.GetName() for k in f.GetListOfKeys()]
print keyNames
```

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?
  - Quite inefficient!
- Organise the dataset in **columns**

Columnar Representation

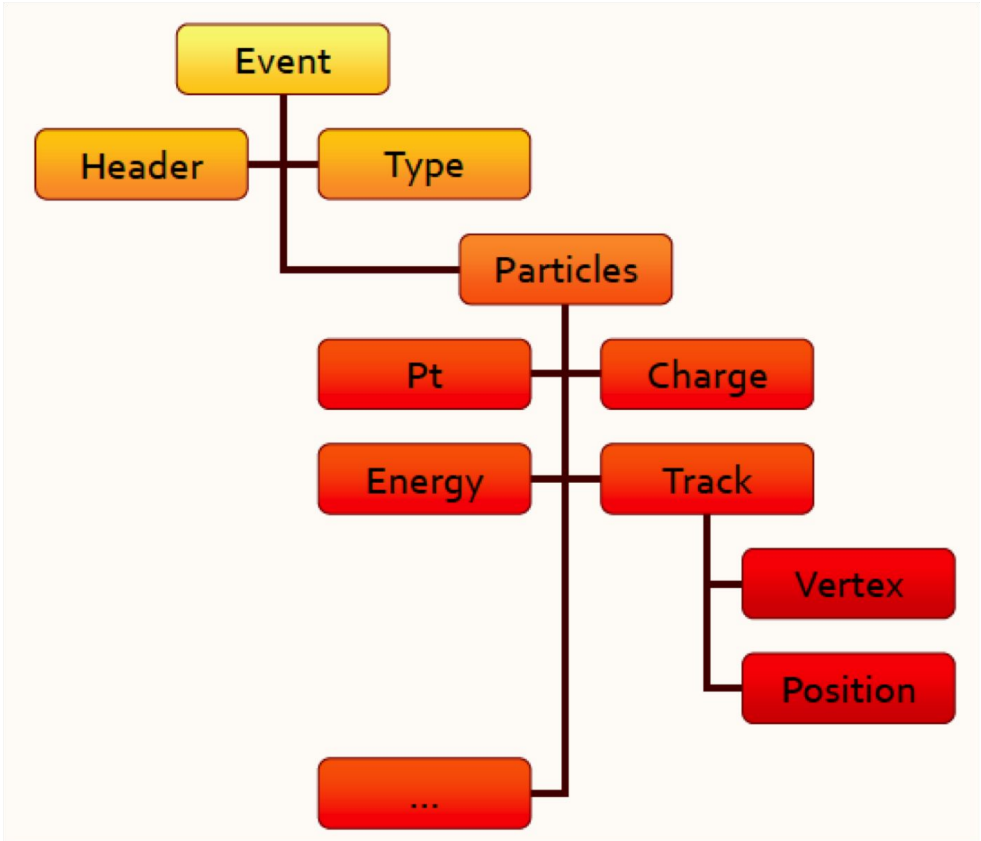


columns
or "branches"
can contain any kind
of c++ object

entries
or events
or rows

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	-0.2223	0.556881
-0.27	1.181767	470484
0.88 02	-0.65411	13209
-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

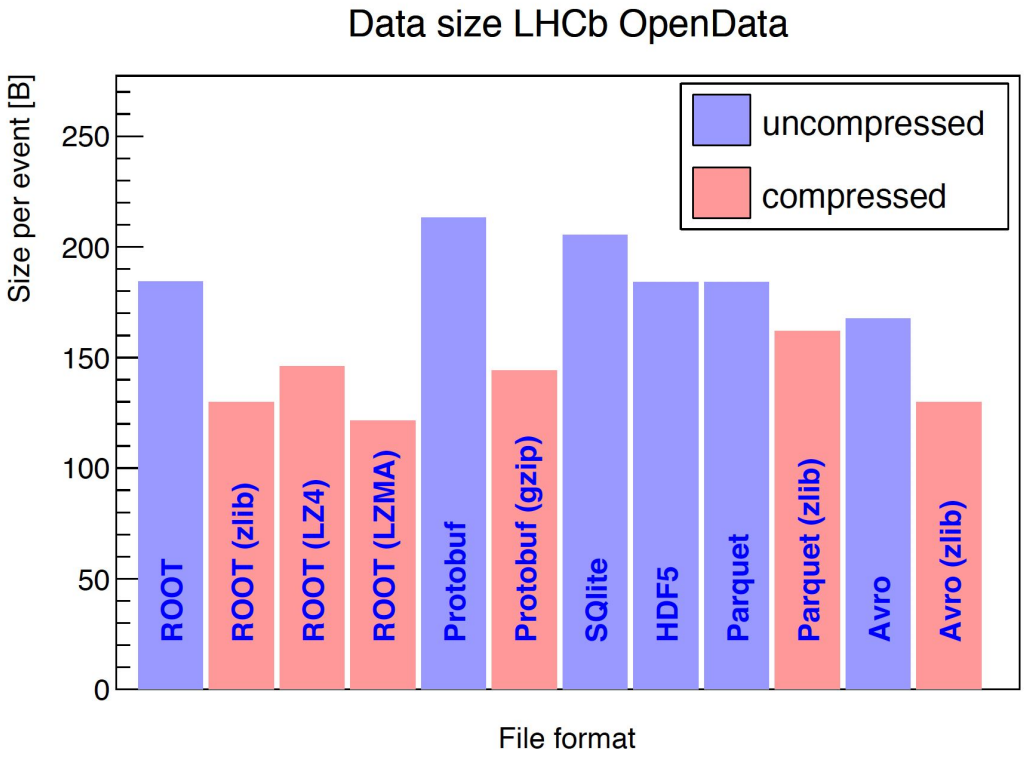
Comparison With Other I/O Systems

	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

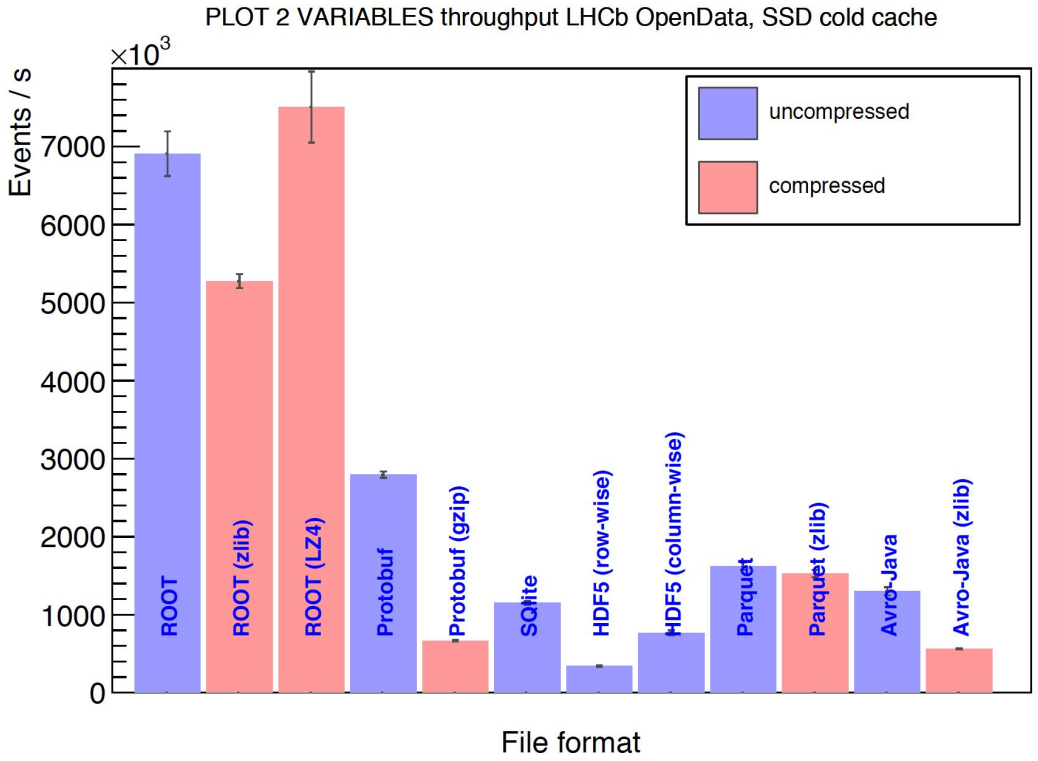
J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

Comparison With Other I/O Systems



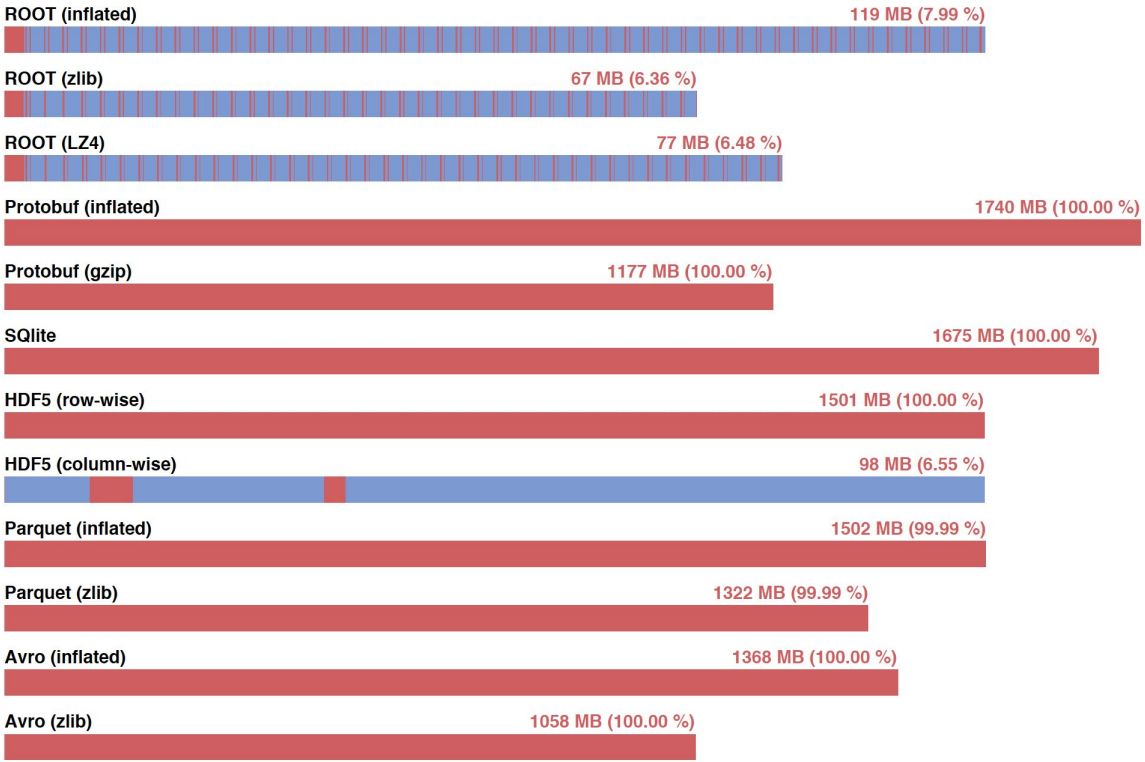
J. Blomer, [A quantitative review of data formats for HEP analyses](#) ACAT 2017

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()
```

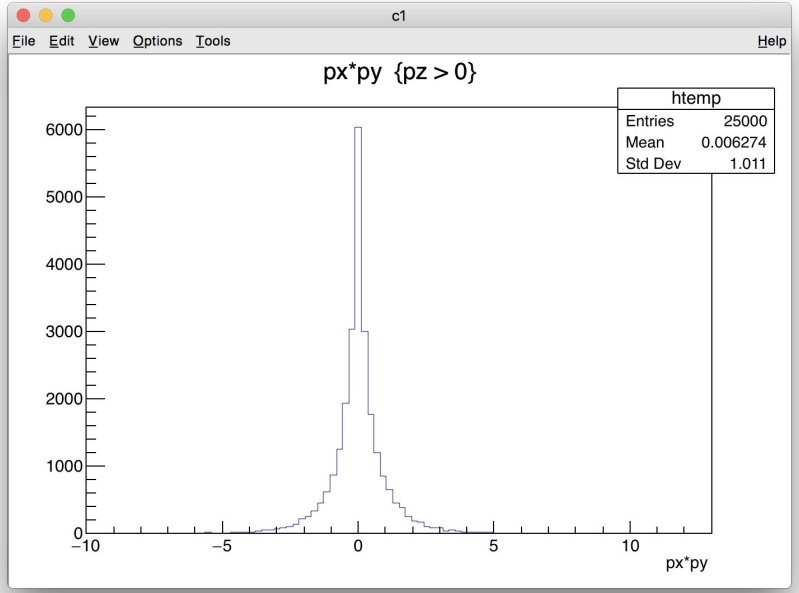
PyROOT: Works for all types of columns, not only numbers!

One Line Analysis

- It is possible to produce simple plots described as strings
- **TNtuple::Draw()** method
  - or better …TT*ree::Draw*
- Good for quick looks, does not scale
  - E.g. one loop on all events per plot

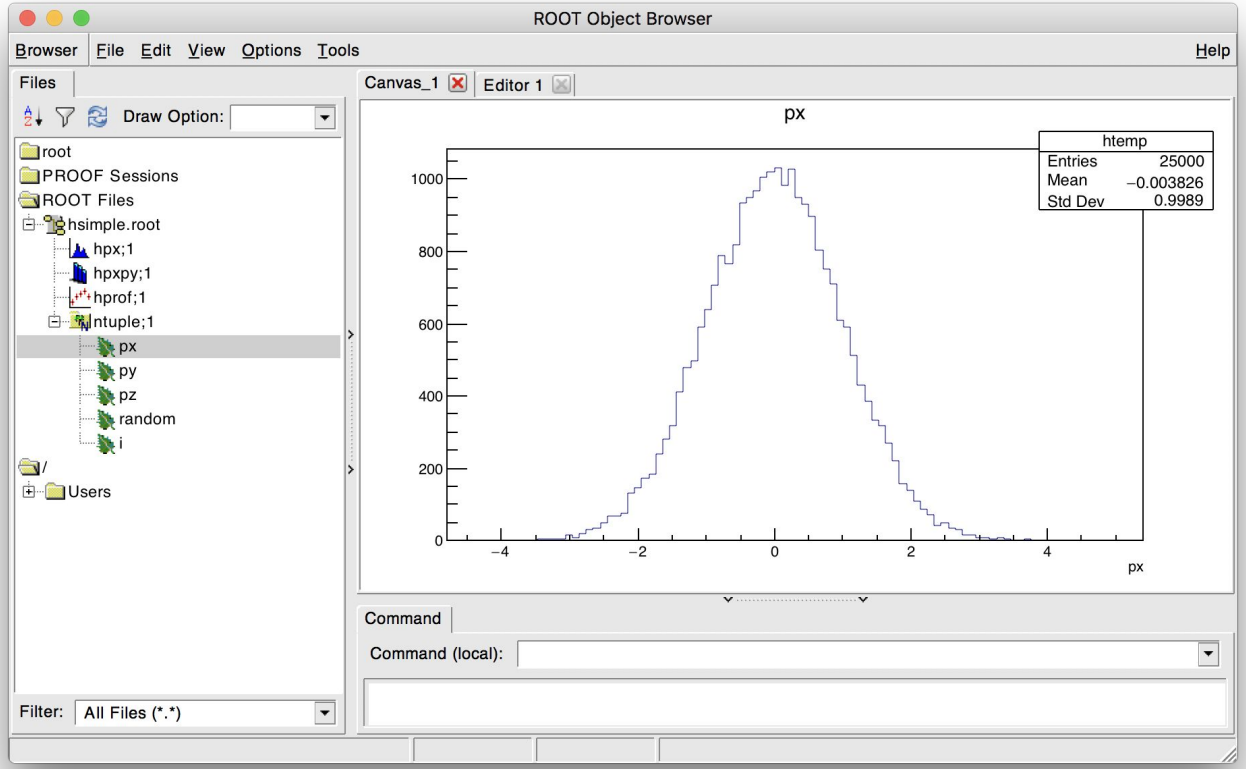
One Line Analysis

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



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();

}
```

End of morning lecture

Afternoon exercises

Exercises

Exercise 1 - check setup

- Fire up ROOT
- Verify it works as a calculator
- List the files in /etc from within the ROOT prompt
- Inspect the help
- Quit

Exercise 2 - simple math and C++ in root shell

- 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

```
root [0] double x=0
root [1] exp(-x*x*.5) – TMath::Gaus(x)
[…]
```

- Many possible ways of solving this! E.g:

```
root [0] for (auto v : {0.,1.,10.,20.}) cout << v << “ ” <<
exp(-v*v*.5) – TMath::Gaus(v) << endl
```

Exercise 3 - Make a “macro” and execute it

- Place the code you tested above into an external file named, e.g. macro1.C
- Then run it in different ways:
  - invoking from command line (# root macroname.C)
  - invoking from root prompt (root [0] .x macroname.C)
  - splitting the loading (.L) and invoking (use the function name)
  - compiling it as an executable that has a main invoking the function
- You can get the gcc flags with “root-config” app

Exercise 4 - Make histos, graphs and funcs

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

Exercise Histogram

- Open the Python interpreter
- Import the ROOT module
- Create a histogram with 64 bins and a axis ranging from 0 to 16
- Fill it with random numbers distributed according to a linear function (*pol1*)
- Change its line width with a thicker one
- Draw it!

Exercise Graph

- Create a new Python module
- In the module, create a graph (TGraph)
- Set its title to *My graph*, its X axis title to *myX* and Y axis title to *myY*
- Fill it with three points: (1,0), (2,3), (3,4)
- Set a red full square marker
- Draw an orange line between points

Exercise - Customize the look

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

macro1NoFit.C

The data set

```
void macro1(){

   // The number of points in the data set
   const int n_points = 10;
   // The values along X and Y axis
   double x_vals[n_points] = {1,2,3,4,5,6,7,8,9,10};
   double y_vals[n_points] = {6,12,14,20,22,24,35,45,44,53};
   // The errors on the Y axis
   double y_errs[n_points] = {5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};

   // Instance of the graph
   auto graph = new TGraphErrors(n_points,x_vals,y_vals,nullptr,y_errs);
}
```

This code creates the **data set**.

⇒ Create a macro called "macro1.C" 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(kOpenCircle);
graph->SetMarkerColor(kBlue);
graph->SetLineColor(kBlue);
```

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

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

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
auto f = new TF1("Linear law","[0]+x*[1]",.5,10.5);
// Let's make the function line nicer
f->SetLineColor(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 ?

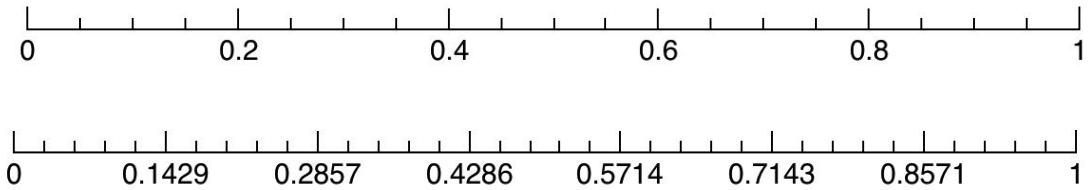
Axis labelling (2)

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,kRed);
```

Legend (1)

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

```
auto legend = new 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 **`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
auto arrow = new TArrow(8,8,6.2,23,0.02,"|>");
arrow->SetLineWidth(2);
arrow->Draw();

// Add some text to the plot
auto text = new 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.

The complete macro

```
void macro1() {
   // The values and the errors on the Y axis
   const int n_points=10;
   double x_vals[n_points] = {1,2,3,4,5,6,7,8,9,10};
   double y_vals[n_points] = {6,12,14,20,22,24,35,45,44,53};
   double y_errs[n_points] = {5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};

   // Instance of the graph
   auto graph = new TGraphErrors(n_points,x_vals,y_vals,nullptr,y_errs);
   graph->SetTitle("Measurement XYZ;length [cm];Arb.Units");

   // Make the plot esthetically better
   graph->SetMarkerStyle(kOpenCircle);
   graph->SetMarkerColor(kBlue);
   graph->SetLineColor(kBlue);

   // The canvas on which we'll draw the graph
   auto  mycanvas = new TCanvas();

   // Draw the graph !
   graph->Draw("APE");
```

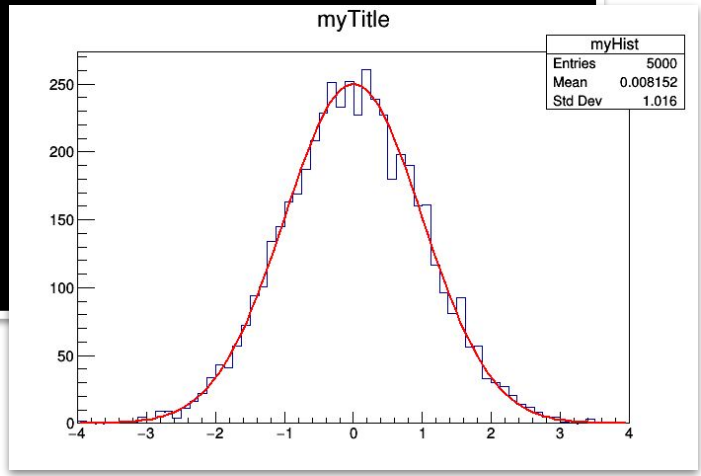
```
   // Define a linear function
   auto f = new TF1("Linear law","[0]+x*[1]",.5,10.5);

   // Let's make the function line nicer
   f->SetLineColor(kRed);
   f->SetLineStyle(2);

   // Set the function parameters
   f->SetParameters(-1,5);
   f->Draw("Same");

   // Build and Draw a legend
   auto legend = new TLegend(.1,.7,.3,.9,"Lab. Lesson 1");
   legend->AddEntry(graph,"Exp. Points","PE");
   legend->AddEntry(f,"Th. Law", "L");
   legend->Draw();

   // Draw an arrow on the canvas
   auto arrow = new TArrow(8,8,6.2,23,0.02,"|>");
   arrow->SetLineWidth(2);
   arrow->Draw();
   // Add some text to the plot and highlight the 3rd label
    auto text = new TLatex(8.2,7.5,"#splitline{Maximum}{Deviation}");
   text->Draw();
   graph->GetXaxis()->ChangeLabel(3,-1,-1,-1,kRed);
}
```

Python exercises

Exercise Function

- Complete this example in Python:

```
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")
```



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

Writing to trees

- First: try writing to trees as in the example (slide 122)
  - compile a program that fills a tree with a few floats

```
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();
```

Hint: g++ writeTest.cpp -o w `root-config --cflags --libs`

Writing objects to trees

- Try to write a vector<floats> per event instead of a single float per event (i.e. a simplified version of slide 123)
- for vectors of builtin types there is a better/alternative way

```
TFile f("test.root","RECREATE");
TTree t("tree","A tree");

float x[100];
int n;
t.Branch("n",&n,"n/I");
t.Branch("myFloats",x,"x[n]/F");

for(int i=0;i<100; i++) { // loop on events
   n=i;
   for(int j=0;j<n; j++) { // loop on particles per event
       x[j]=i*j;
   }
   t.Fill();
}
```

Autogenerate code to read back

```
TFile *_file0 = TFile::Open("test.root")

tree->MakeClass("testReader")
```

mind the name! it may overwrite your code

- Automatically creates some boilerplate code to process the data

test.h

```
class test {
public :
   TTree          *fChain;   //!pointer to
   Int_t           fCurrent; //!current Tr

// Fixed size dimensions of array or colle

   // Declaration of leaf types
   Int_t           n;
   Float_t         myFloats[99];   //[n]
```

why 99??

your analysis code goes here

test.C

```
   if (fChain == 0) return;

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
   }
```

Writing a custom object

- Let's try to write the "fourvector" class we developed a few weeks ago (e.g. https://github.com/lucabaldini/cmepda/blob/master/HEPCPPmodule/HEPCppLesson1/Assignment1/fourvector.h)
- Let's start from the example of slide 123
  - ROOT will complain with something like:

```
Error in <TTree::Branch>: The class requested (vector<FourVector>) for the branch "myParticles" is
an instance of an stl collection and does not have a compiled CollectionProxy. Ple
ase generate the dictionary for this collection (vector<FourVector>) to avoid to write corrupted
data.
```

- How do we generate dictionaries?

```
gInterpreter->GenerateDictionary( "FourVector","fourvector.h");
gInterpreter->GenerateDictionary( "FourVectors","fourvector.h");
```

a typedef to std::vector<fourvector.h>

```
#include "fourvector.h"
#include <iostream>
#include <TFile.h>
#include <TTree.h>
#include <vector>
#include <TSystem.h>
int main()
{
 std::vector<FourVector> particles;
 TFile f("testWithObj.root","RECREATE");
 TTree t("tree","A tree");
 gInterpreter->GenerateDictionary("FourVector","fourvector.h");
 gInterpreter->GenerateDictionary("FourVectors","fourvector.h");
 t.Branch("myParticles",&particles);
 for(int i=0;i<100; i++) { // loop on events
    particles.clear();
    for(int j=0;j<100; j++) { // loop on particles per event
        particles.emplace_back(i,j/2.,0,0);
    }
    t.Fill();
 }
 t.Write();
 f.Close();
}
```

Explore the tree just created

```
root [2] tree->Print ()
******************************************************************************
*Tree    :tree      : A tree                                                   *
*Entries :      100 : Total =         8243581 bytes  File  Size =     135731 *
*        :          : Tree compression factor =  61.01                       *
******************************************************************************
*Br    0 :myParticles : Int_t myParticles_                                     *
*Entries :      100 : Total  Size=     135939 bytes  File Size  =        496 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.79     *
*.............................................................................*
*Br    1 :myParticles.m_px : Float_t m_px[myParticles_]                        *
*Entries :      100 : Total  Size=    2060256 bytes  File Size  =      36916 *
*Baskets :       72 : Basket Size=      32000 bytes  Compression=  54.91     *
*.............................................................................*
*Br    2 :myParticles.m_py : Float_t m_py[myParticles_]                        *
*Entries :      100 : Total  Size=    2060256 bytes  File Size  =      59899 *
*Baskets :       72 : Basket Size=      32000 bytes  Compression=  33.84     *
*.............................................................................*
*Br    3 :myParticles.m_pz : Float_t m_pz[myParticles_]                        *
*Entries :      100 : Total  Size=    2060256 bytes  File Size  =      17836 *
*Baskets :       72 : Basket Size=      32000 bytes  Compression= 113.64     *
*.............................................................................*
*Br    4 :myParticles.m_e : Float_t m_e[myParticles_]                          *
*Entries :      100 : Total  Size=    2060178 bytes  File Size  =      17762 *
*Baskets :       72 : Basket Size=      32000 bytes  Compression= 114.11     *
*.............................................................................*
```

Autogen code from custom objects

- no “objects”

```
// Declaration of leaf types
Int_t           myParticles_;
Float_t         myParticles_m_px[kMaxmyParticles];   //[myParticles_]
Float_t         myParticles_m_py[kMaxmyParticles];   //[myParticles_]
Float_t         myParticles_m_pz[kMaxmyParticles];   //[myParticles_]
Float_t         myParticles_m_e[kMaxmyParticles];   //[myParticles_]
```

Reading as objects

https://root.cern.ch/how/how-read-tree

- Try reading the custom objects we just wrote with different methods:
  - With python
  - With TTreeReader (see example in slide 124)
  - Directly manipulating pointers to region in memory where the read object should be stored:

```
std::vector<FourVector> * particles= new  std::vector<FourVector>;
gSystem->Load("AutoDict_FourVector_cxx.so");
gSystem->Load("AutoDict_FourVectors_cxx.so");
TFile f("testWithObj.root");
TTree *t=(TTree *)f.Get("tree");


t->GetBranch("myParticles")->SetAddress(& particles);

auto nevent = t->GetEntries();
  for (int i=0;i<nevent;i++) {
     t->GetEvent(i);
     std::cout << particles->size() << std::endl;
 }
```

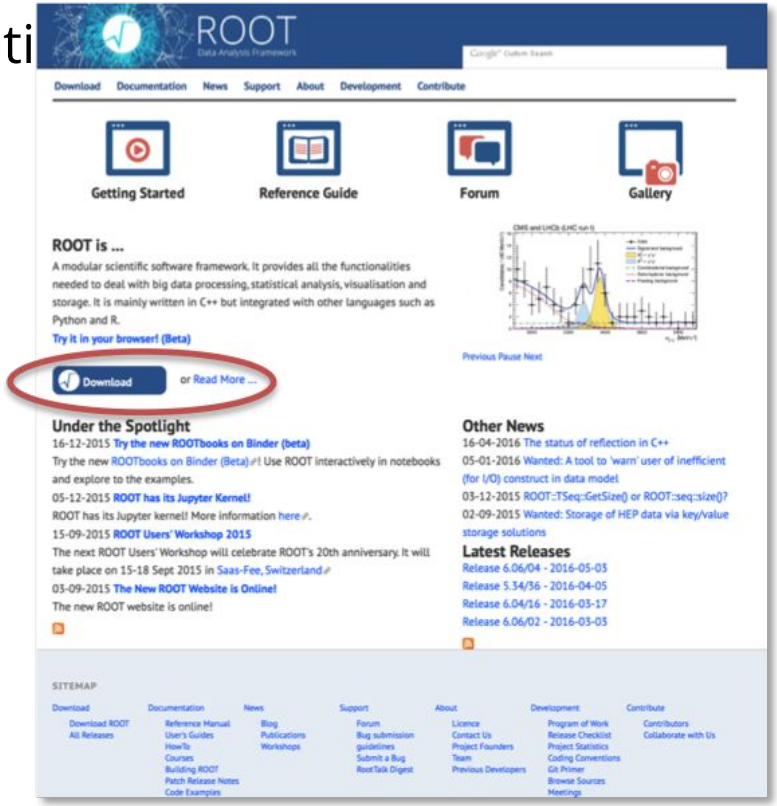
The SWAN Service

- **Data analysis with ROOT “as a service”**
- *Interface:* Jupyter Notebooks
- *Goals:*
  - Use ROOT only with a web browser
    - Platform independent ROOT-based data analysis
    - Calculations, input and results “in the Cloud”
  - Allow easy sharing of scientific results: plots, data, code
    - Through your CERNBox
  - Simplify teaching of data processing and programming

http://swan.web.cern.ch

https://root.cern

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



Ex Tempore Exercise

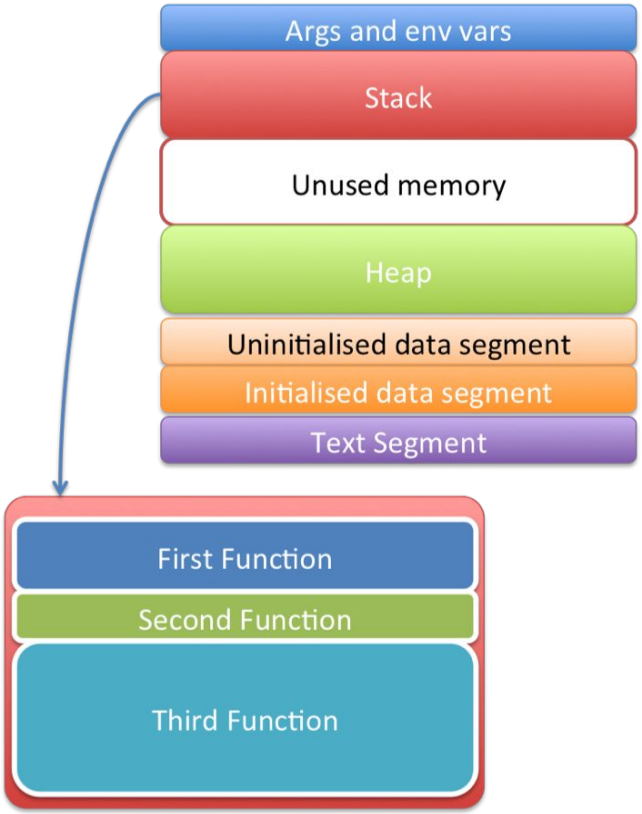
- Fire up your Virtual Machine
- Get the ROOT sources:
  - *git clone http://github.com/root-project/root*
  - *Or visit https://root.cern.ch/content/release-61102*
- Create a build directory and configure ROOT:
  - *mkdir rootBuild; cd rootBuild*
  - *cmake ../root*
- Start compilation
  - *make -j 2*
- Prepare environment:
  - *. bin/thisroot.sh*

Time For Exercises

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

A Program in Memory

- **Text Segment**: code to be executed.
- **Initialized Data Segment**: global variables initialized by the programmer.
- **Uninitialized Data Segment**: This segment contains uninitialized global variables.
- **The stack**: The stack is a collection of stack frames. It grows whenever a new function is called. "Thread private".
- **The heap**: Dynamic memory (e.g. requested with "`new`").



Time for Exercises!

https://github.com/root-project/training/tree/master/BasicCourse/Exercises/C%2B%2BInterpreter#compile-and-run-a-simple-program