# Introduction to ROOT

Steven Schramm

HASCO 2018

July 26, 2018

# Introduction

Bourses postdoctorales Banting Postdoctoral Fellowships

UNIVERSITÉ DE GENÈVE

- Programming is an essential part of modern data analysis
  - It is difficult to get through an MSc without programming
  - It is near-impossible to get through a PhD without programming
- ROOT is one of the key tools for data analysis in high energy physics
  - It is not the only tool, depending on what you do
  - However, you are extremely likely to use it at some point
- Programming is a skill that is best learned by practice
  - Don't just look at the slides - try it out yourself!
  - If you find it hard or frustrating at first, don't worry
  - Just like human languages, you will improve with use
- If you get stuck, ask questions!
  - The internet is full of questions and sample code
    - There is even an entire forum dedicated to ROOT questions
  - Your colleagues probably also use ROOT and can help

# Overview




1. Introduction to ROOT
2. Key programming concepts
3. Running ROOT macros and scripts
4. The ROOT interpreter
5. Overview of common ROOT tasks
6. Making good plots
7. PyROOT tricks

# Introduction to ROOT

# What is ROOT?

- ROOT is a powerful scientific software framework which is...
  - Likely older than many of you (1994, 24 years old)
  - Developed by CERN (mostly the EP-SFT group)
  - Written in C++, but with interfaces to other languages
  - Popular enough to have its own wikipedia page (11 languages)
  - Widely used in particle physics, but also used externally
  - Also a data format tailored to particle physics needs

# Is ROOT an acronym?

- Not the most important question, but a very common one
- There is no original acronym for the toolkit
  - When asked in 1998, the original author replied that it was the "roots" (like a tree root) upon which future work would be based
  - He suggested a possible backronym: Rapid Object-Oriented Technology
  - I have never heard this name being used outside of that email
- In short, just call it ROOT

# PyROOT

- As mentioned, ROOT has interfaces to several other languages
  - The one with the largest support and usage is python (PyROOT)
  - Note: HEP is still almost exclusively python2.7, not python3
- Essentially everything you can do in ROOT is possible in PyROOT
  - In rare cases it requires a few tricks to handle the python/C++ barrier
- In many cases, PyROOT is much easier to use than ROOT
  - However, compiled C++ code is usually faster to run
- You will likely need to use a mix of the two
  - C++ is commonly used in CPU-time-sensitive tasks: reconstruction, data processing, etc
  - Python is usually used in person-time-sensitive tasks: data analysis, plots for physics results, etc
- I will present both; start with the one that you prefer

# A note on ROOT developments

- If you are familiar with C++, you may be confused at times
  - Some of the ROOT concepts and objects seem to be redundant
- It's important to keep in mind how old this package is
  - Example: ROOT TString and std::string do have a large overlap
  - However, ROOT started in 1994, and std::string is from 1998
- ROOT is continually undergoing modernization and updates
  - However, it is difficult to make major changes when experiment lifetimes can be in the decades (such as the LHC experiments)
  - Backwards-compatibility is thus nearly always required
    - If your code works in release X, it should also work in X+1
    - Removing TString and similar is thus not easy to do

# Finding information on ROOT



- The ROOT website contains lots of useful information
  - Getting started
  - Downloading instructions, pre-compiled for many popular systems
  - User's guide
  - Reference guide (documentation of objects, functions, and more)
  - Discussion forum
- ROOT shares a name with the linux super-user/admin account
  - If you are searching on the internet for a ROOT question or feature, you probably want to search for "CERN ROOT"

# Key programming concepts

# Brief reminders

- Hopefully you already have some programming experience
- I will **very briefly** remind you of a few key points:
  - Primitive types vs complex types
  - Conditional statements
  - Iterative statements (loops)
  - Arrays/lists/vectors/etc
  - Functions
  - Classes

# Primitive types vs complex types

- "Primitive types" are the types that are "native" to a language
  - You do not need to include any libraries to use them
  - They generally have a fixed interpretation
- In C++, the primitive types are very simple:
  - Integer numbers: (unsigned) char, short, int, long, long long
  - Real numbers: float, double, long double
  - Conditions: bool = "true" or "false" = 1 or 0
  - That's it - even the std::string is a complex type
- In python, types are inherently dynamic
  - The type is not even evaluated until the command is executed
  - This is one area where python and C++ are completely different

# Conditional statements



- The conditional statement allows you to control code flow
  - Code no longer needs to be designed for a single fully-determined task
  - Arguably the most important development in programming

C++

```
if (condition1)
{
    // Passes condition1
}
else if (condition2)
{
    // Fails condition 1
    // Passes condition 2
}
else if (condition3)
{
    // Fails conditions 1 and 2
    // Passes condition 3
}
//Other conditions here
else
{
    // Fails all previous conditions
}
```

Python

```
if condition1:
    # Passes condition1
elif condition2:
    # Fails condition1
    # Passes condition2
elif condition3:
    # Fails conditions 1 and 2
    # Passes condition3
# other conditions here
else:
    # Fails all previous conditions
```

# Iterative statements (loops)

- The iterative statement allows you to simplify code
  - No need to write the same thing over and over
  - Combined with conditions, you can iterate over many items

C++

```cpp
// typical for loop
int numIterations = 5;
for (int i = 0; i < numIterations; ++i)
{
    std::cout << i << std::endl;
}

// Typical while loop
int numIterations = 5;
int i = 0;
while (i < numIterations)
{
    std::cout << i << std::endl;
    ++i;
}
```

Python

```python
# Typical for loop
numIterations = 5
for i in range(0,numIterations):
    print i

# Typical while loop
numIterations = 5
i = 0
while i < numIterations:
    print i
    i = i + 1
```

- All of the above will print out the numbers between 0 and 4, with each number on a separate line

# Arrays/lists/vectors/etc

- One of the main uses is loops is for lists of items
  - C++: arrays and vectors are commonly used
    - Use vectors if possible, as arrays can be dangerous
  - Python: lists are a fundamental piece of the code
- Such "collection" data structures are *iterable*
  - If you don't need to know the element index, there is another loop type
- Note: recall C++ and python are 0-indexed, so numbers[0] is 1

C++

```cpp
// Create a vector containing the numbers 1 to 3
// This method only works since C++11
std::vector<int> numbers {1,2,3};

// Add the number 4 to the vector
numbers.push_back(4);

// Typical for-each loop
// Only exists since C++11
for (int aNumber : numbers)
{
    std::cout << aNumber << std::endl;
}

// Typical index-based for loop
for (size_t i = 0; i < numbers.size(); ++i)
{
    std::cout << numbers.at(i) << std::endl;
}
```

Python

```python
# Create a list containing the numbers 1 to 3
numbers = [1,2,3]

# Add the number 4 to the list
numbers.append(4)

# Typical for-each loop
for aNumber in numbers:
    print aNumber

# Typical index-based for loop
for i in range(0,len(numbers)):
    print numbers[i]
```

# Functions

- Functions must be declared **before** they are used
- Modularity (use of functions) is a key piece of good code design
  - Allows for re-use of code rather than duplication
  - Easier to read/understand the code in small pieces
  - Please use clear function+variable names (unlike the example)

C++

```cpp
#include<cmath>

double myFunction(double x)
{
    if (x >= 0)
    {
        return sqrt(x);
    }
    else
    {
        return -sqrt(-x);
    }
}

int main()
{
    std::cout << myFunction(4)  << std::endl;
    std::cout << myFunction(-4) << std::endl;
    return 0;
}
```

Python

```python
import math

def myFunction(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return -math.sqrt(-x)

print myFunction(4)
print myFunction(-4)
```

# Classes

- Object-oriented programming is based on the notion of classes
  - Useful way of grouping similar concepts/information
  - Classes have both a state (variables) and behaviour (functions)
- You may not need to write classes too often (depends on your usage)
- However, all of the ROOT objects you work with are classes
  - histograms, files, trees, fits, etc

C++

```cpp
class MyClass
{
    public:
        MyClass(double x);
        double getX()  const { return m_x; }
        double getX2() const;

    private:
        double m_x;
};

MyClass::MyClass(double x)
    : m_x(x)
{
    std::cout << "Created with value: " << m_x << std::endl;
}

double MyClass::getX2() const
{
    return m_x*m_x;
}

int main()
{
    MyClass c(3.14159);
    std::cout << c.getX() << ", " << c.getX2() << std::endl;
    return 0;
}
```

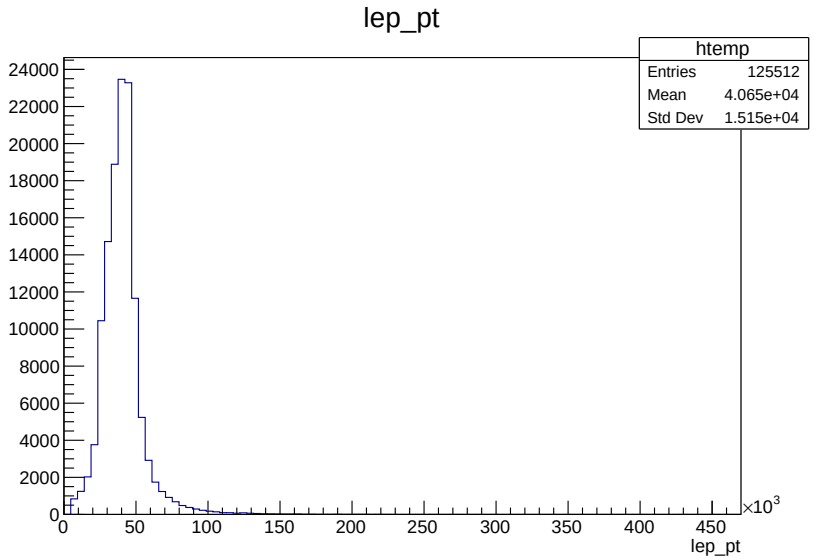
Python

```python
class MyClass:
    def __init__(self, x):
        self.x = x

    def getX(self):
        return self.x

    def getX2(self):
        return self.x*self.x

c = MyClass(3.14159)
print c.getX() , "," , c.getX2()
```

# Running ROOT macros and scripts

# Ways to use ROOT

- There are four primary ways to use ROOT
  1. Command line (Py)ROOT: quick checks and studies, in C++ or python
  2. ROOT macros: simple or moderate programs, in C++
  3. PyROOT scripts: simple or moderate programs, in python
  4. Compiled ROOT: complex or CPU-intensive programs, in C++
- We will briefly discuss these four approaches
  - Same task in each case: read a file and plot $p_{\mathrm{T}}$ of all leptons



# Command line (Py)ROOT



ROOT interpreter, quick method

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ root -l Data.root
root [0]
Attaching file Data.root as _file0...
(TFile *) 0x26c9570
root [1] .ls
TFile**         Data.root
 TFile*         Data.root
  KEY: TTree    HASCO;1 4-vectors + variables required for scaling factors
root [2] HASCO->Draw("lep_pt")
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

ROOT interpreter, detailed method

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ root -l
root [0] TFile* myfile = TFile::Open("Data.root")
(TFile *) 0x319f460
root [1] TTree* tree = (TTree*)myfile->Get("HASCO")
(TTree *) 0x34c3da0
root [2] tree->Draw("lep_pt")
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

PyROOT interpreter

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ROOT
>>> myfile = ROOT.TFile.Open("Data.root")
>>> tree = myfile.Get("HASCO")
>>> tree.Draw("lep_pt")
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

# ROOT macros



- For common tasks, you may want to put it in a macro
  - Then you don't need to type the same thing every time
- Also, if the code is anything but trivial, command-line is not optimal
- Macros: C++ files with a function named the same as the file

The macro (myMacro.cpp):

```
void myMacro(std::string fileToRead, std::string treeName, std::string varToPlot)
{
    TFile* myfile = TFile::Open(fileToRead.c_str());
    TTree* tree   = (TTree*)myfile->Get(treeName.c_str());
    tree->Draw(varToPlot.c_str());
}
```

Two ways to run the macro:

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ root -l
root [0] .x myMacro.cpp("Data.root","HASCO","lep_pt")
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ root -l "myMacro.cpp(\"Data.root\",\"HASCO\",\"lep_pt\")"
root [0]
Processing myMacro.cpp("Data.root","HASCO","lep_pt")...
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

# PyROOT scripts

- We're increasing program complexity and utility
- This is the standard method for PyROOT usage

The script (myScript.py):

```
import ROOT
import sys

if len(sys.argv) != 4:
    print "USAGE: %s <file to read> <tree name> <var to plot>"%(sys.argv[0])
    sys.exit(1)

myfile = ROOT.TFile.Open(sys.argv[1])
tree   = myfile.Get(sys.argv[2])
tree.Draw(sys.argv[3])
```

Running the script:

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ python myScript.py Data.root HASCO lep_pt
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

# Compiled ROOT



- This is much more CPU efficient, but more complex to program

The program(myCompiled.cpp):

```
#include<iostream>
#include<TFile.h>
#include<TTree.h>

int main(int argc, char* argv[])
{
    if (argc != 4)
    {
        std::cout << "USAGE: " << argv[0] << " <file to read> <tree name> <var to plot>" << std::endl;
        return 1;
    }

    TFile* myfile = TFile::Open(argv[1]);
    TTree* tree   = (TTree*)myfile->Get(argv[2]);
    tree->Draw(argv[3]);
    return 0;
}
```

Compiling and running the program (note the back-quotes):

```
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ g++ -o myCompiled myCompiled.cpp `root-config --cflags --libs`
steven@U64X-Laptop:~/ATLAS/Conferences/HASCO2018/data/sample$ ./myCompiled Data.root HASCO lep_pt
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

# The ROOT interpreter

# Special interpreter commands

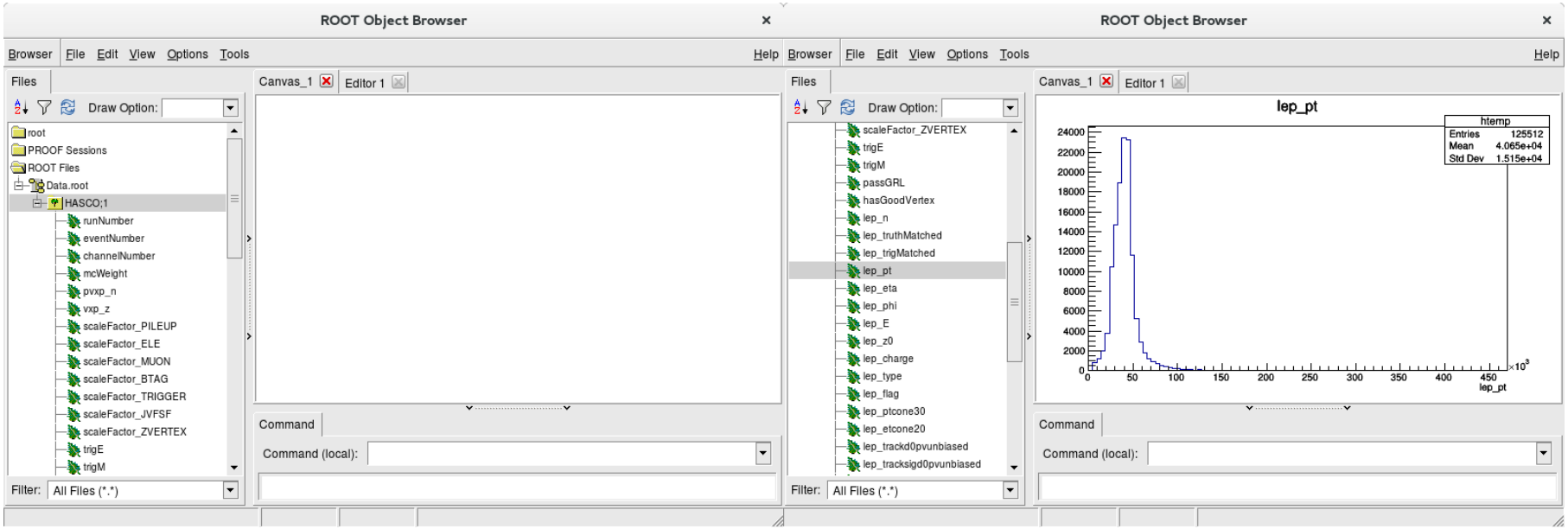
- There are tricks to using the ROOT interpreter
- When starting the interpreter, there are special commands
  - root -**l**: disable the start screen (save time, especially if remote)
  - root -**b**: disable graphics (save a lot of time if making many plots)
  - root -**h**: display the full set of options (help)
- After opening the interpreter, there are a few other tricks
  - **.L myMacro.cpp**: load but don't execute a macro
    - Call the function(s) after as you would call any function
  - **.x myMacro.cpp**: load and execute a macro
    - This requires that a function named the same as the file
  - **.!***command*: issue a console command outside of ROOT
    - .!ls: lists current directory contents
    - .!cd ABC: changes current directory to ABC
- **.q** (or **.qqqq**): quit (or force quit)

# The TBrowser

- TBrowser is a very useful ROOT application
- After opening ROOT, open by declaring a variable of type TBrowser
  - **new TBrowser** , **TBrowser t** , etc
- If you opened a file first, you will be greeted with the following:

# Plotting in the TBrowser

- With an open file, you can browse through the tree
- Double-click on a branch to see the plot
- There are other features: style editors, fitters, and more
- You can also save the current plot to many different formats



# Other useful interpreter commands

- The interpreter is really aimed at quick studies
- There are several commands to quickly look at a file
- `tree->Print()`: List all of the branches in the tree
  - Good to quickly look for branches of interest
  - Can also be filtered with wildcards: `tree->Print("lep*")`
- `tree->Scan()`: look at the contents of a given (set of) branch(es)
  - Can list multiple variables, separating with colons
  - `tree->Scan("eventNumber:lep_pt")` gives the below
    - Note that there are two values (instances) in lep_pt per event

```
root [8] HASC0->Scan("eventNumber:lep_pt")
************************************************
*    Row   * Instance * eventNumb *    lep_pt *
************************************************
*        0 *        0 *  17390906 * 30669.845 *
*        0 *        1 *  17390906 * 28722.550 *
*        1 *        0 *  17478217 * 29445.462 *
*        1 *        1 *  17478217 * 12552.986 *
```

# Selections on Draw and Scan



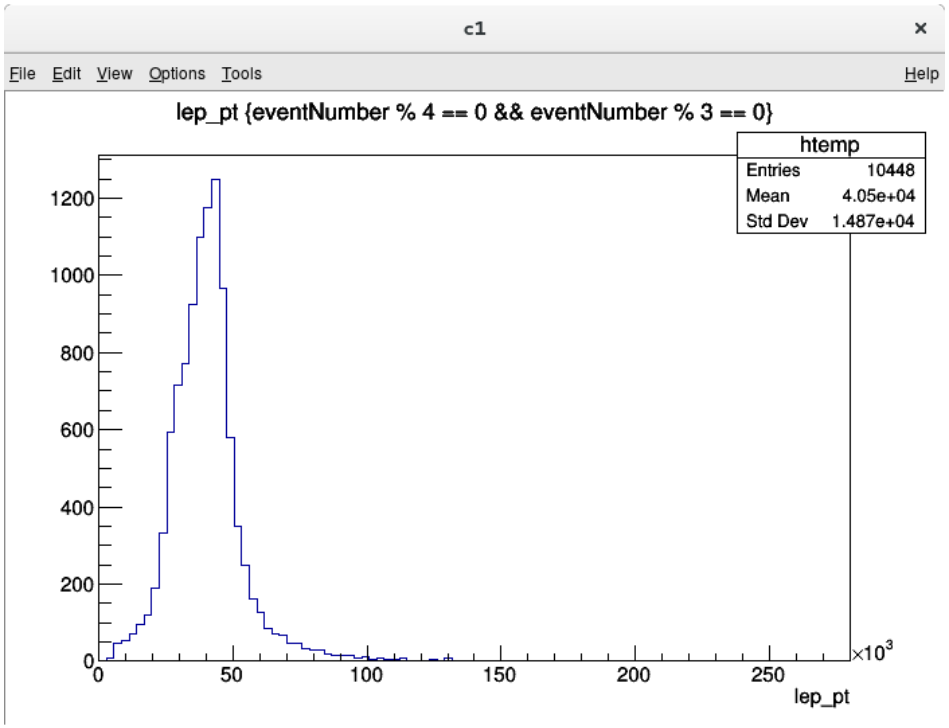
- It is possible to apply selections to the Draw and Scan commands
  - Very powerful for quick studies!
- Example: Scan results after a selection of $p_{\mathrm{T}}^{\ell 1} > 400\,\mathrm{GeV}$

```
root [11] HASCO->Scan("eventNumber:lep_pt","lep_pt[0] > 400.e3")
************************************************
*    Row   * Instance * eventNumb *    lep_pt *
************************************************
*    13157 *        0 *  27489271 * 401201.53 *
*    13157 *        1 *  27489271 * 98432.523 *
*    13668 *        0 *  18924986 * 430238.90 *
*    13668 *        1 *  18924986 * 61166.023 *
************************************************
==> 4 selected entries
```

# Selections on Draw and Scan

- It is possible to apply selections to the Draw and Scan commands
  - Very powerful for quick studies!
- Example: Scan results after a selection of $p_{\mathrm{T}}^{\ell 1} > 400\,\mathrm{GeV}$
- Example: draw lep_pt only for eventNumbers divisible by 3 and 4

```
root [14] HASC0->Draw("lep_pt","eventNumber % 4 == 0 && eventNumber % 3 == 0")
(long long) 10448
```



# Overview of common ROOT tasks

# Most common tasks

- In my opinion, the most common ROOT tasks are:
  - Plotting or manipulating histograms
  - Reading/writing histograms
  - Reading/writing ntuples
  - Fitting distributions
- We will practice most of these in detail tomorrow
- For now, let's briefly discuss what each task entails

# Plotting and manipulating histograms



- There are many different types of histograms for plotting, including:
  - One-, two-, and three-dimensional histograms (TH1*, TH2*, TH3*)
    - There are many types of each, but * = {I,F,D} are the most common
    - I = integer, F = float, D = double $\implies$ type of axis values
  - TProfile and TProfile2D (auto-averages)
  - TEfficiency (for efficiency calculation = fraction passing a selection)
- All of these histograms share common plotting features
  - They mainly differ in the way they are created
- Basic idea:
  - Create a histogram with a given binning (varies by type)
  - Fill histograms with each entry: `histo->Fill(x,y,z,weight)`
    - The above is for 3D, remove y and/or z for 1D or 2D
    - If weight is not specified, the default value is 1
  - Plot the histogram: `histo->Draw()`

# Reading/writing histograms and similar



- Any ROOT object can be stored in a .root file
- Outside of trees (ntuples), they are easy to read/write
- Writing basics:
  - Open a file: `TFile* myFile = TFile::Open("myFile.root","RECREATE")`
  - Enter the file: `myFile->cd()`
  - Write your histogram to the file: `myHisto->Write()`
  - Close the file: `myFile->Close()`
- Reading basics:
  - Open the file: `TFile* myFile = TFile::Open("myFile.root","READ")`
  - Retrieve from the file: `TH1D* myHisto1D = (TH1D*)myFile->Get("name")`
  - Tell the histogram to stay in memory: `myHisto1D->SetDirectory(0)`
  - Close the file: `myFile->Close()`

# Reading/writing ntuples



- TTrees are complex data structures
  - Contain "branches" representing information
  - Each branch contains one or more value(s) per event
  - Values are typically primitive types or arrays of primitive types
  - More advanced: add arbitrary objects, beyond the scope of this intro
- TTrees are highly compressed and also optimized for HEP needs
  - Most/all(?) HEP datasets (data and MC) are stored in TTrees
- If the trees are simple (no complex objects), often called ntuples
- Reading/writing ntuples is more complex and is best done by example
  - We will work with ntuples tomorrow

# Fitting distributions

- ROOT makes it very easy to fit distributions
  - People who have went to industry say they still use ROOT for fitting
- Basic idea: histograms have a `Fit()` function
  - Create the fit type we want: `TF1* myFit = new TF1("myFit","gaus",-1,1)`
    - Fit a gaussian in the range of -1 to 1
  - Optionally specify starting values: `myFit->SetParameter(0,3.14)`
    - Set the value of fit parameter 0 (the first one) to 3.14
  - Apply it to a histogram: `myHisto->Fit(myFit);`
  - Get the resulting $\chi^2/N_{\mathrm{dof}}$: `myFit->GetChisquare()/myFit->GetNDF()`
- There are several default fit functions available, more details here
  - Or specify your own: `TF1* mf = new TF1("mf","x*[0]+sqrt(x+[1])",1,5)`
  - This has two fit parameters, `[0]` and `[1]`

# Making good plots

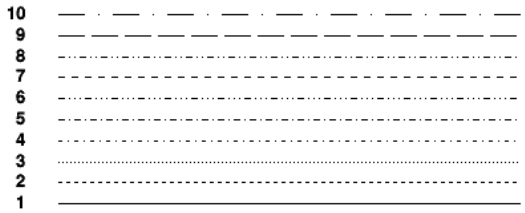
# What is a good plot?

- This is a very subjective question with no easy answer
- However, there are some agreed-upon features
  - The axis labels should be specified, including units
  - There should be a legend on the plot to explain the line(s)
  - In many cases, there should be a ratio sub-plot or similar
  - Uncertainties should be included where relevant
  - Different colours should be used, grouped where relevant
  - Different markers should be used to help in case of printing in black and white or for colourblind individuals
  - Labels should be added to the plot to make it self-explanatory
  - All text on the plot should be large enough to read on a projector
- We will briefly go through examples of how to do this

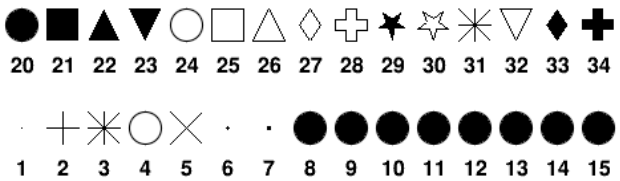
# Style references



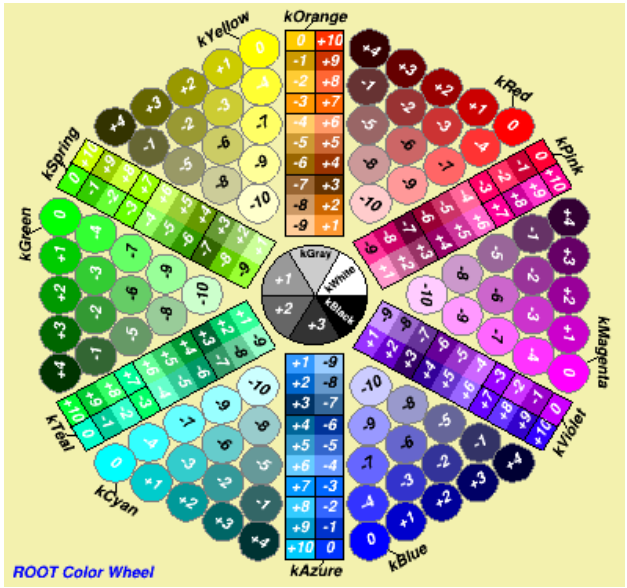
## Line types



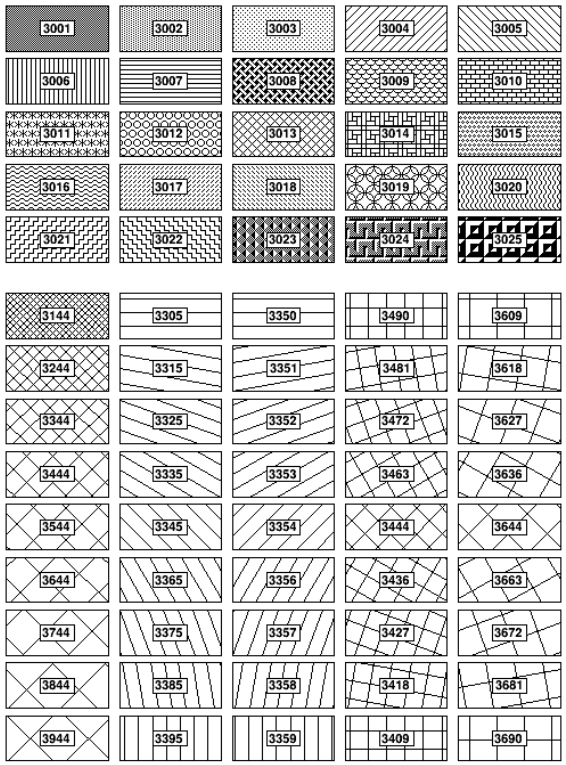

## Marker types

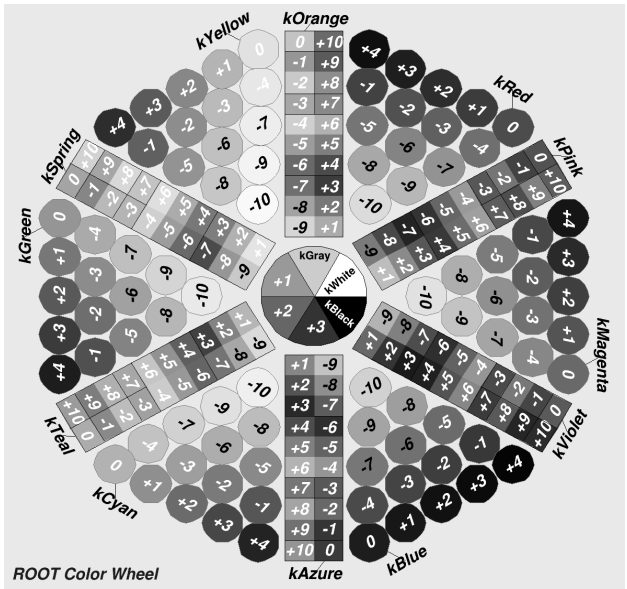



## Colours→




## Fill types



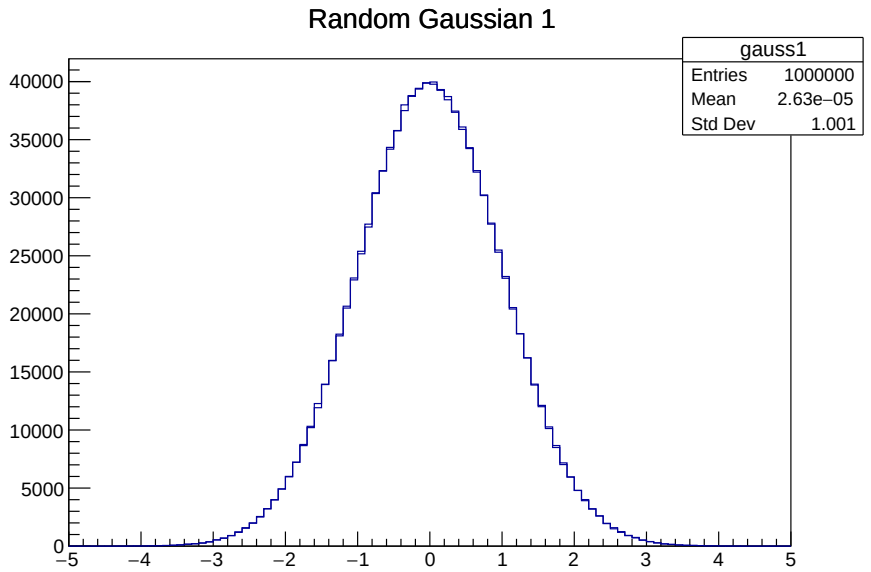



# Random Gaussian distributions



- Create two random Gaussian distributions with 1M entries each
- Draw them on the **same** plot
- They are hard to tell apart, but there are two lines
- We can look at the ratio to better differentiate between them

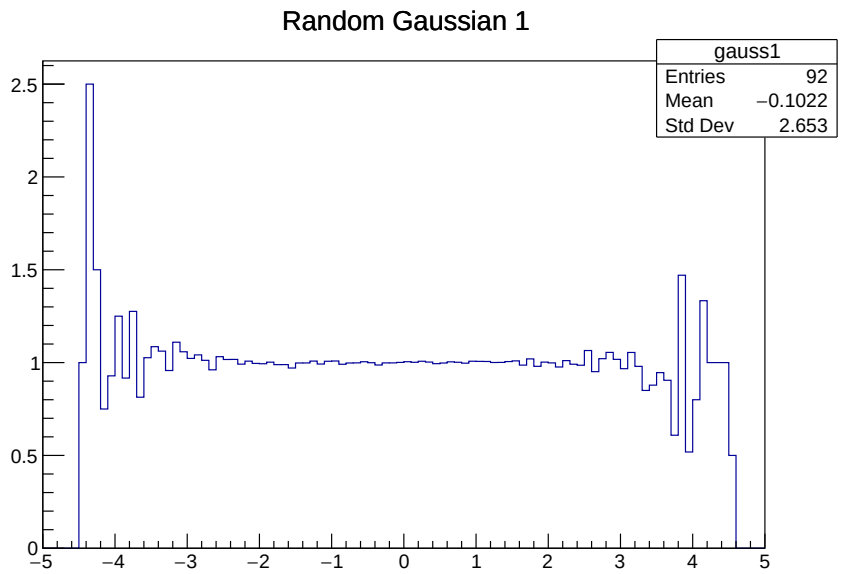
```
import ROOT

canvas = ROOT.TCanvas("canvas")
hist1  = ROOT.TH1D("gauss1","Random Gaussian 1",100,-5,5)
hist2  = ROOT.TH1D("gauss2","Random Gaussian 2",100,-5,5)

randGen = ROOT.TRandom3()
for index in range(0,1000000):
    hist1.Fill(randGen.Gaus(0,1))
    hist2.Fill(randGen.Gaus(0,1))

hist1.Draw()
hist2.Draw("same")
```



# Ratios

- Clone the first histogram and then divide by the second
- Plot the resulting ratio
- Now we can see that they differ in the tails

```
ratio = hist1.Clone()
ratio.Divide(hist2)
ratio.Draw()
```



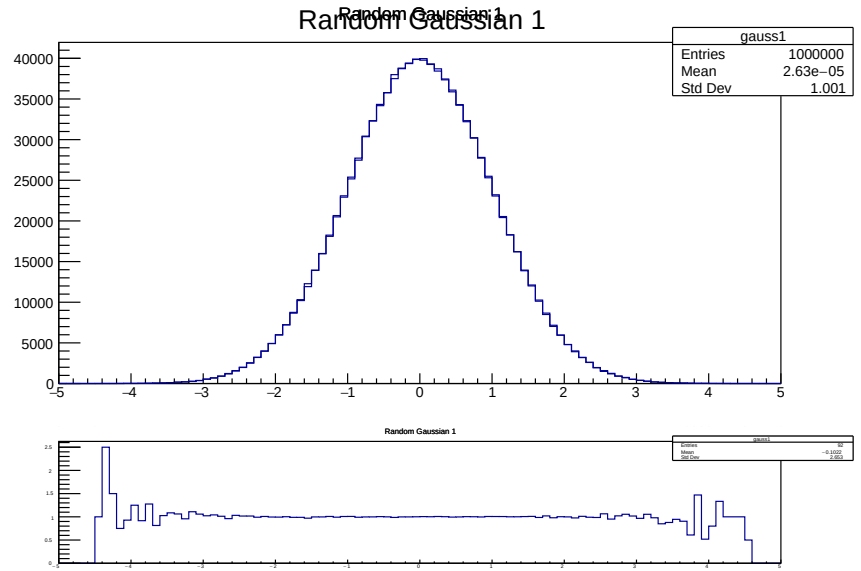
Bourses postdoctorales Banting Postdoctoral Fellowships

UNIVERSITÉ DE GENÈVE

# Two-panel plots

- The two plots we have looked at so far provide complementary info
- It would be good to see them both at once
- We can do this by splitting the canvas into two pads
- While this works, the plot style can clearly be improved

```
# Draw the normal histograms (not the ratio)
canvas.cd()
pad1 = ROOT.TPad("pad1","pad1",0,0.3,1,1)
pad1.Draw()
pad1.cd()
hist1.Draw()
hist2.Draw("same")

# Draw the ratio
canvas.cd()
pad2 = ROOT.TPad("pad2","pad2",0,0.05,1,0.3)
pad2.Draw()
pad2.cd()
ratio.Draw()
```



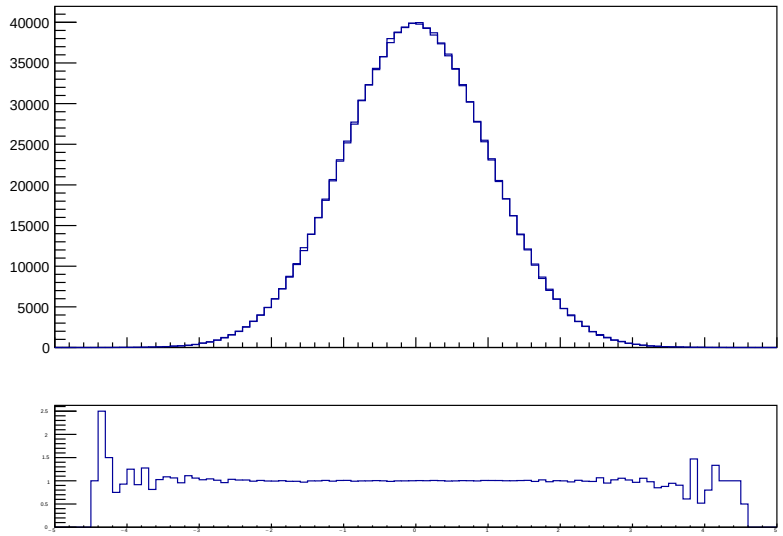
Bourses postdoctorales Banting Postdoctoral Fellowships

UNIVERSITÉ DE GENÈVE

# Cleaning up the figure

- Remove the statistics boxes
- Remove the overlapping titles
- Remove the labels on the upper x-axis
- Now that it's more clean, let's make it look better

```
hist1.SetStats(0)
hist2.SetStats(0)
ratio.SetStats(0)
hist1.SetTitle("")
hist2.SetTitle("")
ratio.SetTitle("")
hist1.GetXaxis().SetLabelSize(0)
```

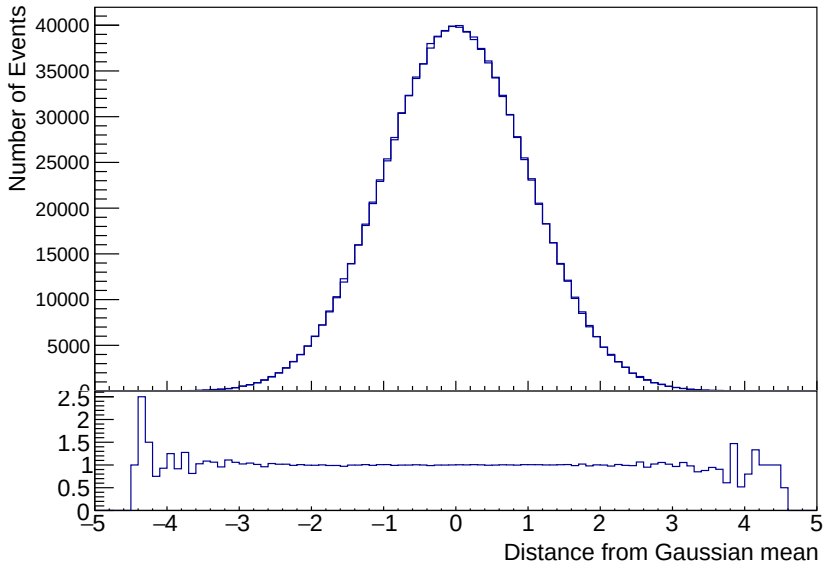


# Adjusting axes

- The plot is starting to look good in overall structure
- However, still a few issues
  - Upper y-axis is clipped at zero
    - We will ignore this for now, as it's more complicated
  - Colours and line/marker styles could be improved

```
hist1.GetYaxis().SetLabelSize(0.05)
hist1.GetYaxis().SetTitle("Number of Events")
hist1.GetYaxis().SetTitleSize(0.06)
hist1.GetYaxis().SetTitleOffset(0.9)

ratio.GetXaxis().SetLabelSize(0.15)
ratio.GetYaxis().SetLabelSize(0.15)
ratio.GetXaxis().SetTitle("Distance from Gaussian mean")
ratio.GetXaxis().SetTitleSize(0.15)
```

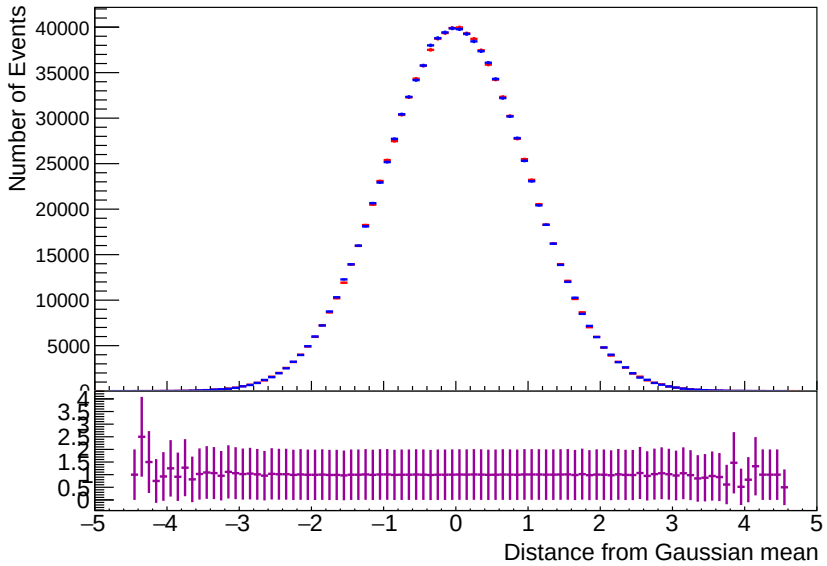
- Also changed pad boundaries
  - `pad1.SetBottomMargin(0)`
  - `pad2.SetTopMargin(0)`
  - `pad2.SetBottomMargin(0.30)`



# More discernible lines

- We've improved the style, but it's pointed out a problem
- The ratio errors are clearly wrong - what's going on?
  - We forgot to configure error handling before filling histograms
  - We need to tell the histograms to keep the (sum of weights)$^2$ for errors

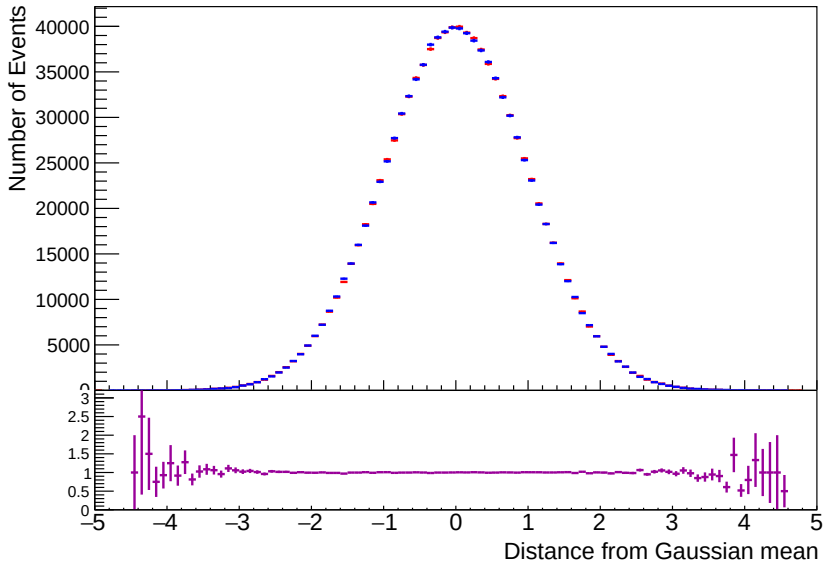
```
hist1.SetLineColor(ROOT.kRed)
hist2.SetLineColor(ROOT.kBlue)
ratio.SetLineColor(ROOT.kMagenta+2)
hist1.SetMarkerColor(ROOT.kRed)
hist2.SetMarkerColor(ROOT.kBlue)
ratio.SetMarkerColor(ROOT.kMagenta+2)
hist1.SetLineWidth(2)
hist2.SetLineWidth(2)
ratio.SetLineWidth(2)
```



# Fixed statistical uncertainties

- Uncertainties are now handled correctly
- Let's add a few more labels and a legend
- Let's also add a line to the ratio to emphasize the value of 1

```
canvas = ROOT.TCanvas("canvas")
hist1  = ROOT.TH1D("gauss1","Random Gaussian 1",100,-5,5)
hist2  = ROOT.TH1D("gauss2","Random Gaussian 2",100,-5,5)

hist1.Sumw2()
hist2.Sumw2()
```

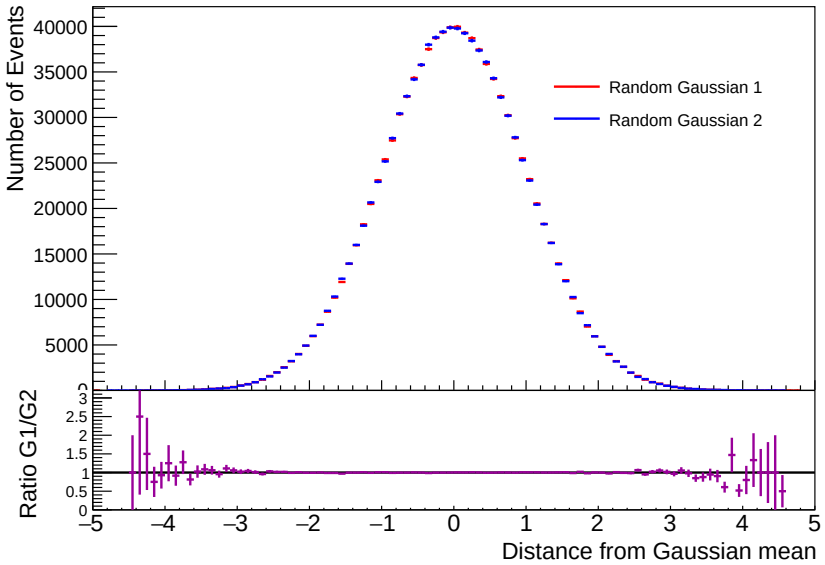


# Almost done

- We're almost there, the style is now looking quite good
  - We added the line at 1 and a legend
- As a last step, let's add some labels to the plot

```
legend = ROOT.TLegend(0.60,0.6,0.89,0.75)
legend.AddEntry(hist1,"Random Gaussian 1")
legend.AddEntry(hist2,"Random Gaussian 2")
legend.SetLineWidth(0)
legend.Draw("same")
```

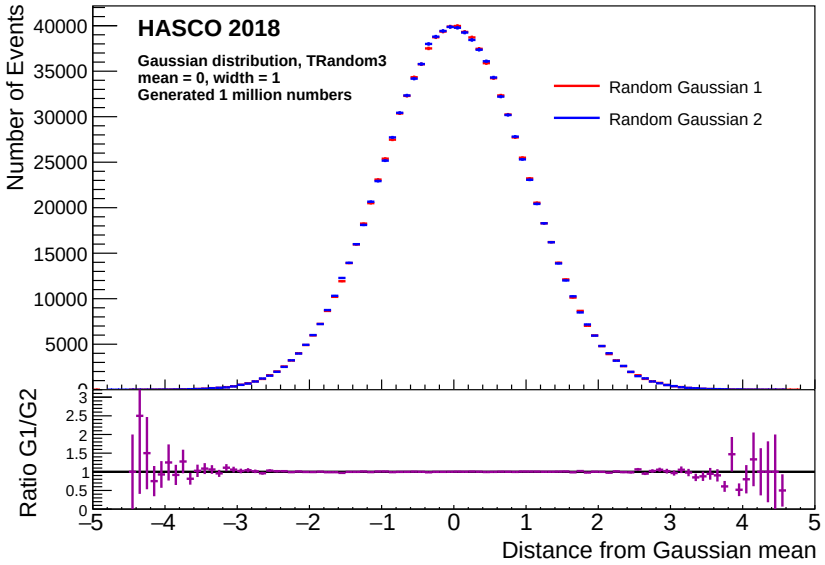
```
line = ROOT.TLine(-5,1,5,1)
line.SetLineColor(ROOT.kBlack)
line.SetLineWidth(2)
line.Draw("same")
```



# Final plot

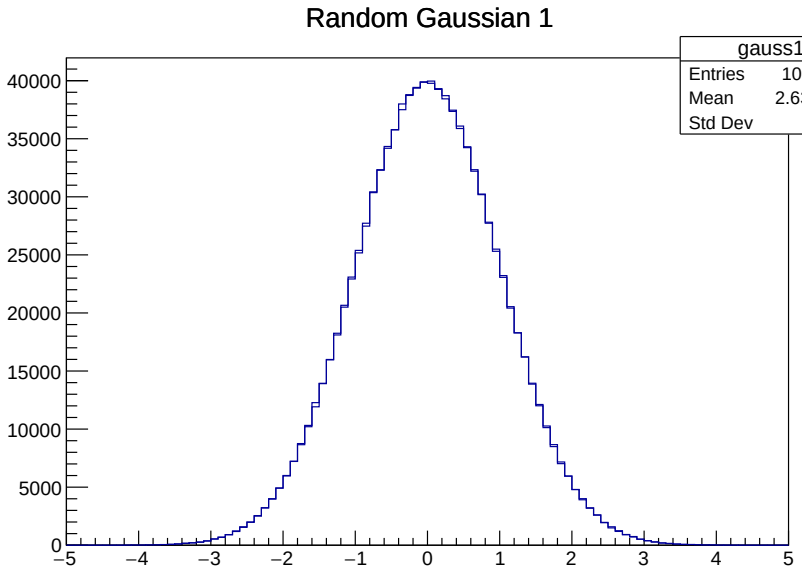
- We used TLatex to add labels where we wanted
- It can also create LaTeX-style symbols

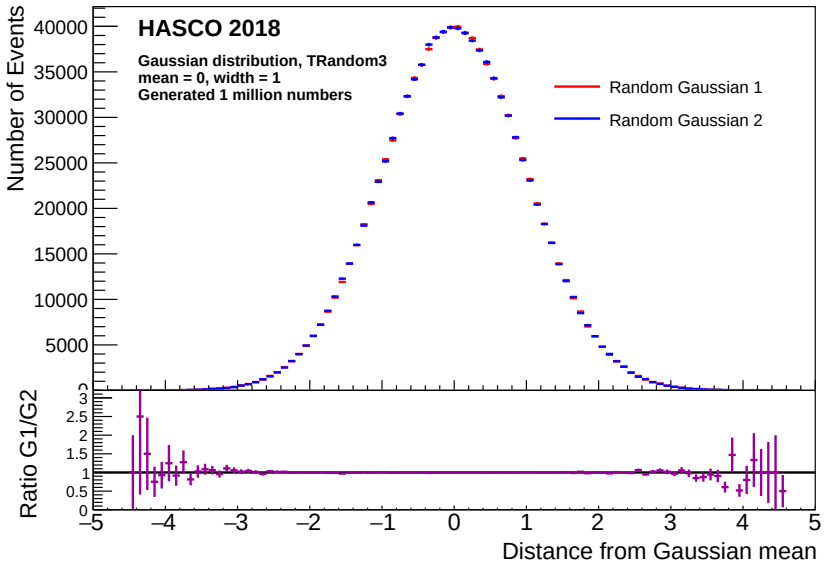
```
latex = ROOT.TLatex()
latex.SetNDC()
latex.SetTextSize(0.06)
latex.DrawText(0.15,0.83,"HASCO 2018")
latex.SetTextSize(0.04)
latex.DrawText(0.15,0.76,"Gaussian distribution, TRandom3")
latex.DrawText(0.15,0.72,"mean = 0, width = 1")
latex.DrawText(0.15,0.68,"Generated 1 million numbers")
```



# Comparison

- You have now seen how we can take a plot and make it more clear
- It may seem not worth the time, but this is very important
  - Most people listening to your talks will be busy and/or distracted
  - The more clear the plots are, the more likely people will follow along
  - Ideally, people should understand the plot without reading any caption



# PyROOT tricks

# PyROOT vs ROOT



- There are occasionally differences between ROOT and PyROOT
- Almost always related to usage of memory
  - ROOT is C++ and thus uses pointers and references
  - PyROOT is python and thus has a different memory structure
- In the rare cases where it comes up, there are workarounds
  - It is possible to declare C++-style memory in python
- There are also PyROOT tricks exploiting python's dynamic structure

# Converting lists to arrays

- Often when working with PyROOT, you need a C++ array
- Example from last class: dynamically-sized histogram bins

```
TH1F* hist = new TH1F(name,title,Nbins,binEdgesArray)
TH1F* hist = new TH1F(name,title,bins.size()-1,&bins[0])
```

- You can get C++ arrays in python through array or numpy

```
import array
binArray = array.array('f',binList)
hist = ROOT.TH1F(name,title,len(binList)-1,binArray)
```

- The 'f' in the above is for float, 'd' would be double, 'i' for int, etc
- Numpy is instead (for a 32-bit float = single-precision):

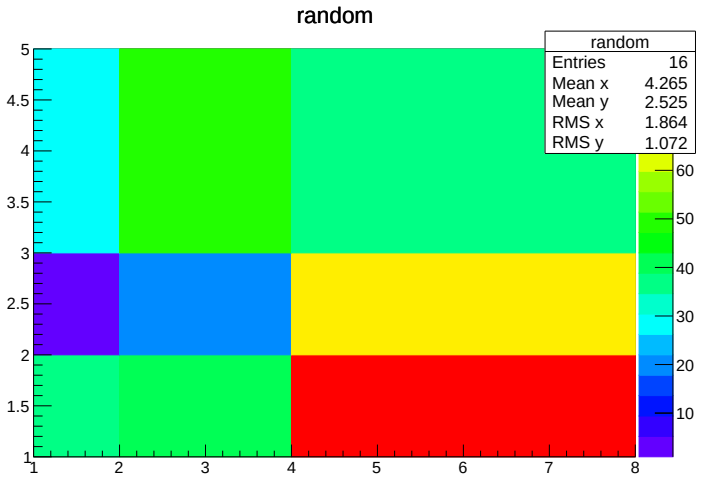
```
import numpy
binArray = numpy.array(binList,dtype=numpy.float32)
```

Bourses postdoctorales Banting Postdoctoral Fellowships

UNIVERSITÉ DE GENÈVE

# Example of 2D histogram with variable bin size

```
import ROOT
import array
import numpy
import random

binsX = [1,2,4,8]
binArrayX = array.array('f',binsX)
binsY = [1,2,3,5]
binArrayY = numpy.array(binsY,dtype=numpy.float32)

histo = ROOT.TH2F("random","random",len(binsX)-1,binArrayX,len(binsY)-1,binArrayY)

for binX in range(1,len(binsX)+1):
    for binY in range(1,len(binsY)+1):
        histo.SetBinContent(binX,binY,random.randint(0,100))

canvas = ROOT.TCanvas("mycanvas")
canvas.cd()
histo.Draw("colz")
canvas.Print("myRandomPlot.pdf")
```



# Getting C++ pointers/references

- Some ROOT functions require passing a pointer or reference
  - Example: `TGraph::GetPoint(int i, double& x, double& y)`
- The same technique, for a list of size 1, works fine
  - We're just creating a C-style memory address and passing that

```
import ROOT
import array
import numpy
import random

graph = ROOT.TGraph(3)
graph.SetPoint(0,random.random(),random.random())
graph.SetPoint(1,random.random(),random.random())
graph.SetPoint(2,random.random(),random.random())

xVal = array.array('d',[0])
yVal = numpy.array([0],dtype=numpy.float64)

print ""
for index in range(0,3):
    graph.GetPoint(index,xVal,yVal)
    print "Point %d = (%.3f,%.3f)"%(index,xVal[0],yVal[0])
print ""
```

```
Point 0 = (0.096,0.324)
Point 1 = (0.295,0.231)
Point 2 = (0.490,0.758)
```

# Iterating on a TFile

- I very often have a ROOT file containing a bunch of histograms
- Python makes it very easy to iterate on that ROOT file
  - The below prints every plot in the ROOT file to one pdf

```
import ROOT

def GetKeyNames(self,dir=""):
  self.cd(dir)
  return [key.GetName() for key in ROOT.gDirectory.GetListOfKeys()]
ROOT.TFile.GetKeyNames = GetKeyNames

myFile = ROOT.TFile.Open("myfile.root","READ")

canvas = ROOT.TCanvas("mycanvas")
plotFile="outFile.pdf"
canvas.cd()

canvas.Print(plotFile+"[")
for histName in myFile.GetKeyNames():
    histogram = myFile.Get(histName)
    if histogram.GetDimension() == 2:
        histogram.Draw("colz")
    else:
        histogram.Draw("")
    canvas.Print(plotFile)
canvas.Print(plotFile+"]")

myFile.Close()
```

# Reading ROOT trees with getattr()

- Using getattr() makes it extremely easy to read ntuples
  - When reading an event, all branches are added to the tree attributes
  - As such, they can be retrieved directly by taking advantage of python
    - No need to create branches/variables for all of them manually

```
import ROOT

myFile = ROOT.TFile.Open("ntup.root","READ")
myTree = myFile.Get("treeJets")

for entryNum in range(0,myTree.GetEntries()):
    myTree.GetEntry(entryNum)
    print "Jet pT for entry %d is %.1f GeV"%(entryNum,getattr(myTree,"jetPt"))

myFile.Close()
```

```
Jet pT for entry 2339 is 105.2 GeV
Jet pT for entry 2340 is 115.0 GeV
Jet pT for entry 2341 is 111.0 GeV
Jet pT for entry 2342 is 104.1 GeV
```

# Summary

# Summary



- (Py)ROOT is a powerful software framework
- It is a massive piece of software with many, many features
  - It will take some time, but you will improve with practice
- There are several different ways to run (Py)ROOT
  - Interpreters, macros, scripts, and compiled programs
  - There are use cases for all of these approaches
- Day-to-day, you will most likely use ROOT for four tasks
  - Plotting, reading/writing histograms, reading/writing ntuples, and fits
- Making clean plots is an important skill
  - The more clear your results, the more likely people are to pay attention

- There is lots more introductory material online - give it a look!