Hands-on ROOT

Ahmed Ali Abdelalim

Physics Department, Faculty of Science, Helwan University
Zewail City for Science and Technology

Contents

1 Introduction

1.1 About this tutorial

First information you should know is: C++ syntax is used to invoke ROOT commands. However, in this tutorial, I tried to add a set of exercises for those who are working with C++ for the first time. Most of the lessons have time estimates at the top. These are only rough estimates. Don't be too concerned about time.

If you see a command in this tutorial is preceded by ">", it means that it is a Linux command. ROOT commands are preceded by "[ ]". In both cases type the command without the ">" or "[ ]"symbol ☺.

You can find this tutorial in PDF format (along with links to the sample files) at http://dpnc.unige.ch/~abdelalim/ROOT_tutorial/ROOT_tutorial.pdf

Be sure to bookmark the ROOT web site, you will be coming back here often. You may also want to download the Users Guide for a handy reference: click on Documentation, then on the User's Guide link.

Your feedback is more than welcome, I need it for developing this tutorial. If you have any questions and/or comments, please contact me at any of these emails: ahmed.ali.abdelalim@cern.ch, aabdelalim@zewailcity.edu.eg, or ahmedali20038@gmail.com.

Now open two terminals, one for ROOT and the other for Linux, and take a breath to start the adventure ☺.

1.2 Starting ROOT

(less than a couple of minutes)

To actually run ROOT, just type:

> root

The terminal in which you type this command will become your ROOT command window. A brief "Welcome to ROOT" display will be written on your command window. If you grow tired of the introductory graphics, type "root -l" instead of "root" to start ROOT.

- **ROOT help:** you can type "?" (or ".h") to see a list of ROOT commands... but you'll probably get more information than you can use right now. Try it and see.
- **To quit ROOT:** type ".q". Do this now, then start ROOT again, just to make sure you can do it.
- **Note:** Sometimes ROOT will crash. If it does, it can get into a state for which .q doesn't work. Try typing .qqq if .q doesn't work; if that still doesn't work, try .qqqqq, then .qqqqqqq. Unfortunately, if you type ten q, ROOT won't respond, You are welcome. Sometimes just typing q or using Ctrl-C also works.

2 Getting Started with ROOT

2.1 Plotting a function

(10 minutes)

Let's plot a simple function. Start ROOT and type the following at the prompt:

[ ] TF1 f1("func1","sin(x)/x",0,10)

[ ] f1.Draw()

You see ROOT tries to help you out with context-based colors for the keywords it recognizes.

When you executed "f1.Draw()", ROOT created a canvas[1] for you named "c1". Bring the window named "c1" to the front by left-clicking on it. As you move the mouse over different parts of the drawing (the function, the axes, the graph label, the plot edges) note how the shape of the mouse changes. Right-click the mouse on different parts of the graph and see how the pop-up menu changes.
Lets play with "c1",

- **To change function range:** position the mouse over the function itself (it will turn into a pointing finger or an arrow). Right-click the mouse and select "SetRange". Set the range to xmin=-10, xmax=10, and click "OK". Observe how the graph changes.
- **Labeling axes:** right-click on the x-axis of the plot, select "SetTitle", enter "x [radians]", and click "OK". To center that title: right-click on the x-axis again, select "CenterTitle", and click "OK" [2]. Set the y-axis title; call it "sin(x)/x". Select the "RotateTitle" property of the y-axis and see what happens.

[1] "Canvas" is ROOT's term for a window that contains ROOT graphics; everything ROOT draws must be inside a canvas.

[2] Clicking on the title gives you a "TCanvas" pop-up, not a text pop-up; it's as if the title wasn't there. Only if you right-click on the axis can you affect the title. In object-oriented terms, the title and its centering are a property of the axis.

- **Zoom in on axis interactively:** left-click on the number "2" on the x-axis, and drag to the number "4". The graph will expand its view. Right-click on the axis and select "UnZoom", to restore the original range.

You have a lot of control over how this plot is displayed. From the "View" menu, select "Editor". Play around with this a bit. Click on different parts of the graph; notice how the options automatically change. Select "View→ Toolbar"; among other options, you can see how you can draw more objects on the plot. Unfortunately there is no a simple "Undo" command, but you can usually right-click on an object and select "Delete" from the pop-up menu.

Select "Style" from the "Edit" menu. Select some different styles and hit "Apply"; when you choose a style, it might de-select the window, so you may have to hit Apply twice.

2.2 Working with Histograms

(15 minutes)

To create a simple histogram:

[ ] TH1D h1("hist1","Histogram from a gaussian",100,-3,3)

Where the name of the histogram is "hist1", the histogram title is "Histogram from a gaussian". There are 100 bins in the histogram. The limits of the histogram are from -3 to 3.

Question: What is the width of one bin of this histogram? Type the following to see if your answer is the same as ROOT thinks it is:

[ ] h1.GetBinWidth(0)

Note that we have to indicate which bin's width we want (bin 0 in this case), because you can define histograms with varying bin widths. Why would you have varying bin widths? In physics, it's common to see event distributions with long "tails." Sometimes it looks a good idea to have small-width bins in regions with large numbers of events, and large bin widths in regions with only a few events. This can result in having roughly the same number of events per bin in the histogram, which helps with fitting to functions as discussed in the next few pages.

If you type

[ ] h1.Draw()

Opsss!!! the histogram is empty. To fill it let's randomly generate 10,000 values according to a distribution and fill the histogram with them:

[ ] h1.FillRandom("gaus",10000)

[ ] h1.Draw()

The "gaus" function is pre-defined by ROOT (see the TFormula class on the ROOT web site). The histogram statistics in the top right-hand corner of the plot. The default Gaussian distribution has a width of 1 and a mean of zero.

Question (for those who've had statistics): Why isn't the mean exactly 0, or the width exactly 1? Add another 10,000 events to histogram h1 with the FillRandom method (hit the up-arrows until you see "h1.FillRandom("gaus",10000)" again, and hit return). Click on the canvas. Does the histogram update immediately, or do you have to type another "Draw" command?

To add error bars on the histogram, select "View→ Editor", then click on the histogram. From the "Error" pop-up menu, select "Simple". Try clicking on the "Simple Drawing" box and see how the plot changes.

Alternatively, you could use the following command instead of the Editor:

[ ] h1.Draw("e")

Let's create a function of our own:

[ ] TF1 myfunc("myfunc","gaus",0,3)

The "gaus" (or gaussian) function is

$$P_0 e^{[\frac{(x-P_1)}{P_2}]^2} \tag{2.1}$$

where P_0, P_1, and P_2 are "parameters" of the function. Let's set these three parameters to values that we choose, draw the result, and then create a new histogram from our function:

[ ] myfunc.SetParameters(10.,1.0,0.5)

[ ] myfunc.Draw()

[ ] TH1D h2("hist2","Histogram from my function",100,-3,3)

[ ] h2.FillRandom("myfunc",10000)

[ ] h2.Draw()

You could also set the function's parameters individually:

[ ] myfunc.SetParameter(1,-1.0)

[ ] h2.FillRandom("myfunc",10000)

2.3 Working with multiple plots

(5 minutes)

You have a lot of different histograms and functions now, but you are plotting them all on the same canvas, so you can't see more than one at a time. There are two ways to

get around this.

- **1)** Create a new canvas by selecting "New Canvas" from the File menu of your existing canvas; this will create a new canvas with a name like "c1_n2". Try it now!

- **2)** Divide a canvas into "pads". On the new canvas, right-click in the middle and select "Divide". Enter nx=2, ny=3, and click "OK".

  Click on the different pads and canvases with the middle button (if you have a mouse with a scroll wheel, the wheel is "clickable" and serves as the middle button). Observe how the yellow highlight moves from box to box. The "target" of the Draw() method will be the highlighted box. Try it: select one pad with the middle button, then enter

  [ ] h2.Draw()

  Select another pad or canvas with the middle button, and type:

  [ ] myfunc.Draw()

2.4 Saving and printing your work

(15 minutes)

One way to do this is using the "Save" sub-menu under the "File" menu on the canvas. There are many file formats listed here, but we are only going to use three of them for this tutorial.

- Select "Save→ canvas-name.C" from one of the canvases in your ROOT session. Let's assume for the moment that you are working with canvas c1, so the file "c1.C" is created. In your Linux window, type:

  > less c1.C

  Looking at the "c1.C" macro, as you can see, can be an interesting way to learn more ROOT commands. However, it doesn't record the procedure you went through to create your plots, only the minimal commands necessary to display them.

- Select "Save→ c1.pdf" from the same canvas, if you want to print it later.

- Select "Save→ c1.root" from the same canvas, a new c1.root file is created in your run directory.

Now quit ROOT with the ".q" command, and start it again.

To re-create your canvas from the ".C" file, use the command

[ ] .x c1.C

This is your first experience with a ROOT "macro", a stored sequence of ROOT commands that you can execute at a later time. One advantage of the ".C" method is that you can edit the macro file, or cut-and-paste useful command sequences into macro files of your own.

You can also start ROOT and have it execute the macro all in a single line:

> root c1.C

2.5 The ROOT browser

(5 minutes)

One way to retrieve the contents of file "c1.root" is to use the ROOT browser. Start up ROOT and create a browser with the command:

[ ] TBrowser tb

In the left-hand pane, scroll to your run directory. Scroll through the list of files. You'll notice special icons for any files that end in ".C" or ".root". If you double-click on a file that ends in ".C": if the Editor tab is in front ROOT will display its contents in the editor window; if the Canvas tab is in front, ROOT will execute its contents.

Click on the Canvas tab, then double-click on c1.C to see what happens.

Double-click on "c1.root", then double-click on "c1;1"[3].

2.6 Fitting a histogram

(15 minutes)

I created a file with a couple of histograms in it for you to play with. Switch to your Linux terminal and cd to your run directory. Start ROOT and start a new browser. Click on the folder in the left-hand pane with the same name as your run directory. Double-click on "histogram.root". Double-click on "hist1"; you may have to move or switch windows around, or click on the Canvas 1 tab, to see the c1 canvas displayed.

You can guess from the x-axis label that I created this histogram from a gaussian distribution, but what were the parameters? In physics, to answer this question we typically perform a "fit" on the histogram: you assume a functional form that depends on one or more parameters, and then try to find the value of those parameters that make the function best fit the histogram.

[3] What "c1;1" mean is: you are allowed to write more than one object with the same name to a ROOT file. The first object has ";1" put after its name, the second ";2", and so on. You can use this facility to keep many versions of a histogram in a file, and be able to refer back to any previous version.

Right-click on the histogram and select "FitPanel". Under Fit Function, make sure that Predef-1D is selected. Then make sure "gaus" is selected in the pop-up menu next to it, and Chi-square is selected in the Fit Settings→ Method pop-up menu. Click on "Fit" at the bottom of the panel. You'll see two changes: A function is drawn on top of the histogram, and the fit results are printed on the ROOT command window.

Whenever you do a fit, you want to show the fit parameters on the plot. They give you some idea if your theory (which is often some function) agrees with the data (the points on the plot). On the canvas, select "Fit Parameters" from the "Options" menu; you'll see the fit parameters displayed on the plot.

Alternatively, you can fit hist1 to a gaussian, from the root command line:

[ ] hist1.Fit(gaus)

This does the same thing as using the FitPanel. You can close the FitPanel; we wont be using it anymore. Go back to the browser window and double-click on "hist2". You have probably already guessed by reading the x-axis label that I created this histogram from the sum of two gaussian distributions. We are going to fit this histogram by defining a custom function of our own. Define a user function with the following command:

[ ] TF1 func("mydoublegaus","gaus(0)+gaus(3)")

Note that the internal ROOT name of the function is "mydoublegaus", but the C++ name is func. "gaus(0)" means to use the gaussian distribution starting with parameter 0; "gaus(3)" means to use the gaussian distribution starting with parameter 3. This means our user function has six parameters: P0, P1, and P2 are the three parameters of the first gaussian, and P3, P4, and P5 are those of the second gaussian.

Let's set the values of P0, P1, P2, P3, P4, and P5, and fit the histogram.

[ ] func.SetParameters(5.,5.,1.,1.,10.,1.)

[ ] hist2.Fit(mydoublegaus)

It's not a very good fit, is it? This is because intial values of the fitting parameters were not good enough. Let's try a better set:

[ ] func.SetParameters(5.,2.,1.,1.,10.,1.)

[ ] hist2.Fit(mydoublegaus)

These simple fit examples may leave you with the impression that all histograms in physics are fit with gaussian distributions. Nothing could be further from the truth. I'm using gaussians in this class because they have properties (mean and width) that you can determine by eye. In general, for fitting histograms in a real analysis, you'll have to define your own functions and fit to them directly, with commands like:

[ ] TF1 func("myFunction","<...some parameterized TFormula...>")

[ ] func.SetParameters(...some values...)

[ ] myHistogram.Fit(myFunction)

2.7 Saving your work, part 2

(15 minutes)

So now you have got a histogram fitted to a complicated function. If you were to use "Save as c1.root", quit ROOT, restart it, then load canvas "c1;1" from the file, you would get your histogram back with the function superimposed, but it's not obvious where the function is or how to access it now.

What if you want to save your work in the same file as the histograms you just read in? You can do it, but not by using the ROOT browser. The browser will only open .root files in read-only mode. To be able to modify a file, you have to open it with ROOT commands.

Try the following: Quit ROOT (note that you can select "Quit ROOT" from the "Browser" menu of the browser or the File menu of the canvas). Start ROOT again, then modify "histogram.root" with the following commands:

[ ] TFile file1("histogram.root","UPDATE")

It is the "UPDATE" option that will allow you to write new objects to "histogram.root".

[ ] hist2.Draw()

For the following two commands, try hitting the up-arrow key until you see them again.

[ ] TF1 func("mydoublegaus","gaus(0)+gaus(3)")

[ ] func.SetParameters(5.,2.,1.,1.,10.,1.)

[ ] hist2.Fit("mydoublegaus")

Now you can do what you couldn't before: save objects into the ROOT file:

[ ] hist2.Write()

[ ] func.Write()

Close the file to make sure you save your changes (optional; ROOT usually closes the file for you when you quit the program):

[ ] file1.Close()

Quit ROOT, start it again, and use the ROOT browser to open "histogram.root". You'll see a couple of new objects: "hist2;2" and "mydoublegaus;1". Double-click on each of them to see what you have saved.

2.8 Accessing variables in ROOT NTuples/Trees

(10 minutes)

I have created a sample ROOT n-tuple, experiment.root, for you. Start ROOT then start a new browser with the command

[ ] TBrowser b

Click on the folder in the left-hand pane with the same name as your home directory. Double-click on "experiment.root. There's just one object inside: "tree1", a ROOT Tree (or n-tuple) with 100,000 simulated physics events.

Actually, no real physics associated with the contents of this tree. It is just to illustrate ROOT concepts, not to demonstrate real physics with a real detector.

Right-click on the "tree1" icon, and select "Scan". You'll be presented with a dialog box; just hit "OK" for now. Select your ROOT window, even though the dialog box didn't go away. At first you'll probably just notice that it's a lot of numbers. Take a look at near the top of the screen; you should see the names of the variables in this ROOT Tree.

In this overly-simple example, an imaginary particle is traveling in a positive direction along the z-axis with energy "ebeam". It hits a target at z=0, and travels a distance "zv" before it is deflected by the material of the target. The particle's new trajectory is represented by "px", "py", and "pz", the final momenta in the x-, y-, and z-directions respectively. The variable "chi2" represents a confidence level in the measurement of the particle's momentum.

Did you notice what's missing from the above description? One important omission is the units; for example, I didn't tell you whether "zv" is in millimeters, centimeters, inches, yards, etc. Such information is not usually stored inside an n-tuple; you have to find out what it is and include the units in the labels of the plots you create. For this example, assume that "zv" is in centimeters (cm), and all energies and momenta are in GeV.

You can hit "return" to see more numbers, but you probably won't learn much. Hit "q" to finish the scan. You may have to hit "return" a couple of times to see the ROOT prompt again.

2.9 Simple analysis using the Draw command

(15 minutes)

If you don't already have the sample ROOT TTree file open, open it with the following command:

[ ] TFile myFile("experiment.root")

Use the Scan command to look at the contents of the Tree, instead of using the TBrowser:

[ ] tree1− >Scan()

You can also display the TTree in a different way that doesn't show the data, but displays the names of the variables and the size of the TTree:

[ ] tree1− >Print()

Either way, you can see that the variables stored in the TTree are "event", "ebeam", "px", "py", "pz", "zv", and "chi2".

Create a histogram of one of the variables. For example:

[ ] tree1− >Draw("ebeam")

Using the Draw command, make histograms of the other variables.

By the way, the variable "event" is just the event number (it's 0 for the first event, 1 for the second event, 2 for the third event... 99999 for the 100,000th event).

Instead of just plotting a single variable, let's try plotting two variables at once:

[ ] tree1− >Draw("ebeam:px")

This is a scatterplot, a handy way of observing the correlations between two variables. The Draw command interprets the variables as ("x:y") to decide which axes to use.

Be careful: it's easy to fall into the trap of thinking that each (x,y) point on a scatterplot represents two values in your n-tuple. In fact, the scatterplot is a grid and each square in the grid is randomly populated with a density of dots that's proportional to the number of values in that grid.

Try making scatterplots of different pairs of variables. Do you see any correlations between the variables?

If you just see a shapeless blob on the scatterplot, the variables are likely to be uncorrelated; for example, plot "px" versus "py". If you see a pattern, there may be a correlation; for example, plot "pz" versus "zv". It appears that the higher "pz" is, the lower "zv" is, and vice versa. Perhaps the particle loses energy before it is deflected in the target.

By the way, you can also make three-dimensional plots this way:

[ ] tree1− >Draw("px:py:pz")

After looking at these plots, you can see why it's important to always label your axes!

Let's create a "cut":

[ ] tree1− >Draw("zv","zv<20")

Look at the x-axis of the histogram. Compare this with:

[ ] tree1− >Draw("zv")

Note that a variable in a cut does not have to be one of the variables you're plotting:

[ ] tree1− >Draw("ebeam","zv<20")

Try this with some of the other variables in the tree.

Cut on two variables:

[ ] tree1− >Draw("ebeam","px>10 & & zv<20")

3 Using C++ to analyze a Tree

3.1 Using C++ to analyze a Tree

(15 minutes)

First step is to have ROOT write the skeleton of an analysis class for your ntuple. This is done with the MakeClass command. Let's start with a clean slate: quit ROOT if its running and start it up again. Open the ROOT tree again:

[ ] TFile myFile("experiment.root")

Now create an analysis macro for "tree1" with MakeClass. I'm going to use the name 'Analyze' for this macro, but you can use any name you want; just remember to use your name instead of 'Analyze' in all the examples below.

[ ] tree1− >MakeClass("Analyze")

Switch to the Linux window and examine the files that were created:

> less Analyze.h

> less Analyze.C

The approach of most analysis tasks can be simplified into three steps:

- Set-up (open files, define variables, create histograms, etc.).
- Loop (for each event in the n-tuple or Tree, perform some tasks: calculate values, apply cuts, fill histograms, etc.).
- Wrap-up (display results, save histograms, etc.).

The C++ code from Analyze.C is in the ROOT_tutorial directory in my web page http://dpnc.unige.ch/~abdelalim/ROOT_tutorial/Analyze.C. In C++ lines beginning with "//" are comments. In the analysis code generated with MakeClass() method, there are comments put there by ROOT which I think aren't helpful to you; you can edit or delete them after they are created, but you can't easily prevent them from being

created in the first place. I commented the places in the code where your C++ statements/commands for Set-up, Loop, and Warp-up should go.

The Analyze macro does nothing (it prints few statements), but let's learn how to run it anyway. Quit ROOT, start it again, and enter the following lines:

[ ] .L Analyze.C

[ ] Analyze a

[ ] a.Loop()

P.S. if ROOT can figure out that you are trying to type in a file name, it will try to complete that name as best it can when you hit the tab key. By the way, it's not just ROOT that can do this. When you're in the UNIX window and you have a long file name to work with, try typing the first couple of letters and hit tab.

Normally, after the last step (a.Loop()) ROOT will pause as it reads through all the events in the Tree. Since I added a break statement after the 10^{th} event, it will pause after only ten events, then print "Goodbye". If you are not familiar with C++, you may be very confused at this point. What do any of the above commands have to do with the file "experiment.root" or the TTree inside it? And what do these commands mean?

Take another look at Analyze.h. If you scan through it, you will see C++ commands that explicitly refer to the name of the file, the name of the Tree, and its variables. Now go back and look at the top of Analyze.C. You'll see the line

#include Analyze.h

this means that ROOT will include the contents of the file Analyze.h when it loads Analyze.C. The command:

[ ] .L Analyze.C

tells ROOT to load the computer code inside the file Analyze.C, and to interpret the code to create a C++ class. The name of this class will be "Analyze"; look near the top of Analyze.h, and you will see the C++ keywords "class Analyze".

[ ] Analyze a

creates an object whose name is "a".

[ ] a.Loop()

tells ROOT to execute the Loop command of object "a". Look at Analyze.C again. Near the beginning, you will see the line "void Analyze::Loop". The code in this file, defines the Loop command.

3.2 Making a histogram with a macro

(15 minutes)

Make a copy of the Analyze.C file:

> cp Analyze.C AnalyzeHistogram.C

Edit the file AnalyzeHistogram.C. In the Set-up section, put the following code:

TH1* chi2Hist = new TH1D("chi2","Histogram of Chi2",100,0,20);

In the Loop section, put this in:

chi2Hist-¿Fill(chi2);

This goes in the Wrap-up section:

chi2Hist-¿Draw();

Save the file, then enter the following commands in ROOT:

[ ] .L AnalyzeHistogram.C

[ ] Analyze a

[ ] a.Loop()

We just made our first histogram with a C++ analysis macro. In the Set-up section, we defined a histogram; in the loop section, we filled the histogram with values; in the Wrap-up section, we drew the histogram.

How to put labels in histogram's axes in the macro?

In the Set-up section:

chi2Hist− >GetXaxis()− >SetTitle("chi2");

chi2Hist− >GetYaxis()− >SetTitle("number of events");

Now test the revised Analyze class.

You see now the labels overlap the axis numbers ⌣̈.