2-ROOT-Basics

April 13, 2016

```
In [2]: %%jsroot on
```

Now that you have installed ROOT, what's this interactive shell thing you're running ? It's like this: ROOT leads a double life. It has an interpreter for macros Cling that you can run from the command line or like other applications. But it is also an interactive shell that can evaluate arbitrary statements and expressions. This is extremely useful for debugging, quick hacking and testing. Let us first have a look at some very simple examples.

0.1 2.1 ROOT as calculator

You can even use the ROOT interactive shell instead of a calculator! Launch the ROOT interactive shell with the command:

root

on your Linux box. The prompt should appear shortly.

```
In [4]: 1+1
```

```
(int) 2
```

```
In [5]: 2*(4+2)/12.
```

```
(double) 1.00000
```

```
In [6]: sqrt(3.)
```

```
(double) 1.73205
```

```
In [7]: 1>2
```

```
(bool) false
```

```
In [8]: TMath::Pi()
```

```
(Double_t) 3.14159
```

```
In [9]: TMath::Erf(.2)
```

```
(Double_t) 0.222703
```

Not bad. You can see that ROOT offers you the possibility not only to type in C++ statements, but also advanced mathematical functions, which live in the TMath namespace.

Now let's do something more elaborated. A numerical example with the well known geometrical series:

```
In [10]: double x=.5
```

```
(double) 0.500000
```

```
In [11]: int N=30
```

```
(int) 30
```

```
In [12]: double geom_series=0
```

```
(double) 0.00000
```

```
In [13]: for (int i=0;i<N;++i)geom_series+=TMath::Power(x,i)
```

```
In [14]: TMath::Abs(geom_series - (1-TMath::Power(x,N-1))/(1-x))
```

```
(Double_t) 1.86265e-09
```

Here we made a step forward. We even declared variables and used a for control structure. Note that there are some subtle differences between Cling and the standard C++ language. You do not need the ";" at the end of line in interactive mode – try the difference e.g. declare a different double like in the command above.

0.2 2.2 Learn C++ at the ROOT prompt

Behind the ROOT prompt there is an interpreter based on a real compiler toolkit: LLVM. It is therefore possible to exercise many features of C++ and the standard library. For example in the following snippet we define a lambda function, a vector and we sort it in different ways:

```
In [15]: using doubles = std::vector<double>;
         auto pVec = [](const doubles& v){for (auto&& x:v) cout << x << endl;};
         doubles v{0,3,5,4,1,2};
         pVec(v);
```

```
0
3
5
4
1
2
```

```
In [16]: std::sort(v.begin(),v.end());
         pVec(v);
```

```
0
1
2
3
4
5
```

Or, if you prefer random number generation:

```
In [17]: std::default_random_engine generator;
         std::normal_distribution<double> distribution(0.,1.);
         distribution(generator);
         std::cout << distribution(generator);
```

```
-0.407472
```

```
In [18]: distribution(generator);
         std::cout << distribution(generator);
```

```
0.399771
```

```
In [19]: distribution(generator);
         std::cout << distribution(generator);
```

```
0.0523187
```

0.3 2.3 ROOT as function plotter

Using one of ROOT's powerful classes, here TF1 will allow us to display a function of one variable, x. Try the following:

```
In [20]: TCanvas canvas_2;
         TF1 f1("f1","sin(x)/x",0.,10.);
```

f1 is an instance of a TF1 class, the arguments are used in the constructor; the first one of type string is a name to be entered in the internal ROOT memory management system, the second string type parameter defines the function, here sin(x)/x, and the two parameters of type double define the range of the variable x. The Draw() method, here without any parameters, displays the function in a window which should pop up after you typed the above two lines.

```
In [21]: f1.Draw();
         canvas_2.Draw();
```

```
<IPython.core.display.HTML object>
```

A slightly extended version of this example is the definition of a function with parameters, called [0], [1] and so on in the ROOT formula syntax. We now need a way to assign values to these parameters; this is achieved with the method SetParameter(,) of class TF1. Here is an example:

```
In [22]: TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
```

You can try to change the parameters of the input bellow and try the results.

```
In [23]: f2.SetParameter(0,1);
         f2.SetParameter(1,1);
         f2.Draw();
         canvas_2.Draw();
```

```
<IPython.core.display.HTML object>
```

Of course, this version shows the same results as the initial one. Try playing with the parameters and plot the function again. The class TF1 has a large number of very useful methods, including integration and differentiation. To make full use of this and other ROOT classes, visit the documentation on the Internet under http://root.cern.ch/drupal/content/reference-guide. Formulae in ROOT are evaluated using the class TFormula, so also look up the relevant class documentation for examples, implemented functions and syntax.

You should definitely download this guide to your own system to have it at you disposal whenever you need it.

To extend a little bit on the above example, consider a more complex function you would like to define. You can also do this using standard C or C++ code.

Consider the example below, which calculates and displays the interference pattern produced by light falling on a multiple slit. Please do not type in the example below at the ROOT command line, there is a much simpler way: Make sure you have the file slits.C on disk, and type root slits.C in the shell. This will start root and make it read the "macro" slits.C, i.e. all the lines in the file will be executed one after the other.

In this example drawing the interference pattern of light falling on a grid with n slits and ratio r of slit width over distance between slits.

```
In [24]: %%cpp -d
```

As always in the notebook envirement we need to [......................]. Something you will not need to do in your machine.

```
In [25]: auto pi = TMath::Pi();
```

Bellow you can see the function code.

We define the necessary functions in C++ code, split into three separate functions, as suggested by the problem considered. The full interference pattern is given by the product of a function depending on the ratio of the width and distance of the slits, and a second one depending on the number of slits. More important for us here is the definition of the interface of these functions to make them usable for the ROOT class TF1: the first argument is the pointer to x, the second one points to the array of parameters.

```
In [26]: %%cpp -d
         double single(double *x, double *par) {
           return pow(sin(pi*par[0]*x[0])/(pi*par[0]*x[0]),2);
         }

         double nslit0(double *x,double *par){
           return pow(sin(pi*par[1]*x[0])/sin(pi*x[0]),2);
         }

         double nslit(double *x, double *par){
           return single(x,par) * nslit0(x,par);
         }
```

Here is how the main program should look like.

It starts with the definition of a function slits() of type void. After asking for user input, a ROOT function is defined using the C-type function given in the beginning. We can now use all methods of the TF1 class to control the behaviour of our function – nice, isn't it ?

```
In [27]: %%cpp -d
           void slits() {
           float r,ns;


           r = 1;
           ns=0.45;

           /* // request user input
           cout << "slit width / g ? ";
           scanf("%f",&r);
           cout << "# of slits? ";
           scanf("%f",&ns);
           cout <<"interference pattern for "<< ns
                <<" slits, width/distance: "<<r<<endl;
           */

           // define function and set options
           TF1 *Fnslit  = new TF1("Fnslit",nslit,-5.001,5.,2);
           Fnslit->SetNpx(500);

           // set parameters, as read in above
           Fnslit->SetParameter(0,r);
           Fnslit->SetParameter(1,ns);

           // draw the interference pattern for a grid with n slits
           Fnslit->Draw();
         }
```

```
In [28]: slits();
         canvas_2.Draw();
```

```
<IPython.core.display.HTML object>
```

Output of slits.C with parameters 0.2 and 2.

In the commented out section the example asks for user input, namely the ratio of slit width over slit distance, and the number of slits. After entering this information, you should see the graphical output as above.

This is a more complicated example than the ones we have seen before, so spend some time analysing it carefully, you should have understood it before continuing.

If you like, you can easily extend the example to also plot the interference pattern of a single slit, using function double single, or of a grid with narrow slits, function double nslit0, in TF1 instances.

Here, we used a macro, some sort of lightweight program, that the interpreter distributed with ROOT, Cling, is able to execute. This is a rather extraordinary situation, since C++ is not natively an interpreted language! There is much more to say: chapter 3 is indeed dedicated to macros.

0.4 2.4 Controlling ROOT

One more remark at this point: as every command you type into ROOT is usually interpreted by Cling, an "escape character" is needed to pass commands to ROOT directly. This character is the dot at the beginning of a line:

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

This is a selection of the most common commands. * **quit root**, simply type `.q`

- obtain a **list of commands**, use `.?`
- **access the shell** of the operating system, type `.!<OS_command>`; try, e.g. `.!ls` or `.!pwd`
- **execute a macro**, enter `.x <file_name>`; in the above example, you might have used `.x slits.C` at the ROOT prompt
- **load a macro**, type `.L <file_name>`; in the above example, you might instead have used the command `.L slits.C` followed by the function call `slits();`. Note that after loading a macro all functions and procedures defined therein are available at the ROOT prompt.
  - **compile a macro**, type `.L <file_name>+`; ROOT is able to manage for you the `C++` compiler behind the scenes and to produce machine code starting from your macro. One could decide to compile a macro in order to obtain better performance or to get nearer to the production environment.

Use `.help` at the prompt to inspect the full list.

0.5 2.5 Plotting Measurements

To display measurements in ROOT, including errors, there exists a powerful class `TGraphErrors` with different types of constructors. In the example here, we use data from the file `ExampleData.txt` in text format:

```
In [29]: TCanvas canvas_2_5;
         TGraphErrors gr("../data/ExampleData.txt");
         gr.Draw("AP");
         canvas_2_5.Draw();
```

```
<IPython.core.display.HTML object>
```

Make sure the file ExampleData.txt is available in the directory from which you started ROOT. Inspect this file now with your favourite editor, or use the command less ExampleData.txt to inspect the file, you will see something like that:

```
# fake data to demonstrate the use of TGraphErrors

# x    y    ex    ey
  1.   0.4  0.1   0.05
  1.3  0.3  0.05  0.1
  1.7  0.5  0.15  0.1
  1.9  0.7  0.05  0.1
  2.3  1.3  0.07  0.1
  2.9  1.5  0.2   0.1
```

The format is very simple and easy to understand. Lines beginning with # are ignored. It is very convenient to add some comments about the type of data. The data itself consist of lines with four real numbers each, representing the x- and y- coordinates and their errors of each data point.

The argument of the method Draw("AP") is important here. Behind the scenes, it tells the TGraphPainter class to show the axes and to plot markers at the x and y positions of the specified data points. Note that this simple example relies on the default settings of ROOT, concerning the size of the canvas holding the plot, the marker type and the line colours and thickness used and so on. In a well-written, complete example, all this would need to be specified explicitly in order to obtain nice and well readable results. A full chapter on graphs will explain many more of the features of the class TGraphErrors and its relation to other ROOT classes in much more detail.

0.6 2.6 Histograms in ROOT

Frequency distributions in ROOT are handled by a set of classes derived from the histogram class TH1, in our case TH1F. The letter F stands for float, meaning that the data type float is used to store the entries in one histogram bin.

```
In [30]: TCanvas canvas_2_6;
         TF1 efunc("efunc","exp([0]+[1]*x)",0.,5.);
         efunc.SetParameter(0,1);
         efunc.SetParameter(1,-1);
```

The first lines of this example define a function, an exponential in this case, and set its parameters.

```
In [31]: TH1F hist_2_6_1("histogram 2.6.1","example histogram",100,0.,5.);
```

In this line a histogram is instantiated, with a name, a title, a certain number of bins (100 of them, equidistant, equally sized) in the range from 0 to 5.

We use yet another new feature of ROOT to fill this histogram with data, namely pseudo-random numbers generated with the method TF1::GetRandom, which in turn uses an instance of the ROOT class TRandom created when ROOT is started.

```
In [32]: for (int i=0;i<1000;i++) {hist_2_6_1.Fill(efunc.GetRandom());}
```

Data is entered in the histogram using the method TH1F::Fill in a loop construct. As a result, the histogram is filled with 1000 random numbers distributed according to the defined function.

```
In [33]: hist_2_6_1.Draw();
         canvas_2_6.Draw();
```

```
<IPython.core.display.HTML object>
```

The histogram is displayed using the method `TH1F::Draw()`. You may think of this example as repeated measurements of the life time of a quantum mechanical state, which are entered into the histogram, thus giving a visual impression of the probability density distribution. The plot is shown above.

Note that you will not obtain an identical plot when executing the lines above, depending on how the random number generator is initialised.

The class `TH1F` does not contain a convenient input format from plain text files. The following lines of `C++` code do the job. One number per line stored in the text file "expo.dat" is read in via an input stream and filled in the histogram until end of file is reached.

```
In [34]: TH1F hist_2_6_2("histogram 2.6.2","example histogram",100,0.,5.);
         ifstream inp;
         inp.open("expo.dat");
         while (inp >> x) { hist_2_6_2.Fill(x); }
         hist_2_6_2.Draw();
         inp.close();
         canvas_2_6.Draw();
```

```
<IPython.core.display.HTML object>
```

0.7 2.7 Interactive ROOT

Look at one of your plots again and move the mouse across. You will notice that this is much more than a static picture, as the mouse pointer changes its shape when touching objects on the plot. When the mouse is over an object, a right-click opens a pull-down menu displaying in the top line the name of the ROOT class you are dealing with, e.g. `TCanvas` for the display window itself, `TFrame` for the frame of the plot, `TAxis` for the axes, `TPaveText` for the plot name. Depending on which plot you are investigating, menus for the ROOT classes `TF1`, `TGraphErrors` or `TH1F` will show up when a right-click is performed on the respective graphical representations. The menu items allow direct access to the members of the various classes, and you can even modify them, e.g. change colour and size of the axis ticks or labels, the function lines, marker types and so on. Try it!

You will probably like the following: in the output produced by the example `slits.C`, right-click on the function line and select "SetLineAttributes", then left-click on "Set Parameters". This gives access to a panel allowing you to interactively change the parameters of the function, as shown in the figure above. Change the slit width, or go from one to two and then three or more slits, just as you like. When clicking on "Apply", the function plot is updated to reflect the actual value of the parameters you have set.

Another very useful interactive tool is the `FitPanel`, available for the classes `TGraphErrors` and `TH1F`. Predefined fit functions can be selected from a pull-down menu, including "`gaus`", "`expo`" and "`pol0`" - "`pol9`" for Gaussian and exponential functions or polynomials of degree 0 to 9, respectively. In addition, user-defined functions using the same syntax as for functions with parameters are possible.

After setting the initial parameters, a fit of the selected function to the data of a graph or histogram can be performed and the result displayed on the plot. The fit panel has a number of control options to select the fit method, fix or release individual parameters in the fit, to steer the level of output printed on the console, or to extract and display additional information like contour lines showing parameter correlations. As function fitting is of prime importance in any kind of data analysis, this topic will again show up later.

If you are satisfied with your plot, you probably want to save it. Just close all selector boxes you opened previously and select the menu item `Save as...` from the menu line of the window. It will pop up a file selector box to allow you to choose the format, file name and target directory to store the image. There is one very noticeable feature here: you can store a plot as a root macro! In this macro, you find the C++ representation of all methods and classes involved in generating the plot. This is a valuable source of information for your own macros, which you will hopefully write after having worked through this tutorial.

Using ROOT's interactive capabilities is useful for a first exploration of possibilities. Other ROOT classes you will encounter in this tutorial have such graphical interfaces. We will not comment further on this, just be aware of the existence of ROOT's interactive features and use them if you find them convenient. Some trial-and-error is certainly necessary to find your way through the huge number of menus and parameter settings.

0.8 2.8 ROOT Beginners' FAQ

At this point of the guide, some basic questions could have already come to your mind. We will try to clarify some of them with further explanations in the following.

0.8.1 2.8.1 ROOT type declarations for basic data types

In the official ROOT documentation, you find special data types replacing the normal ones, e.g. `Double_t`, `Float_t` or `Int_t` replacing the standard `double`, `float` or `int` types. Using the ROOT types makes it easier to port code between platforms (64/32 bit) or operating systems (windows/Linux), as these types are mapped to suitable ones in the ROOT header files. If you want adaptive code of this type, use the ROOT type declarations. However, usually you do not need such adaptive code, and you can safely use the standard C type declarations for your private code, as we did and will do throughout this guide. If you intend to become a ROOT developer, however, you better stick to the official coding rules!

0.8.2 2.8.2 Configure ROOT at start-up

The behaviour of a ROOT session can be tailored with the options in the `.rootrc` file. Examples of the tunable parameters are the ones related to the operating and window system, to the fonts to be used, to the location of start-up files. At start-up, ROOT looks for a `.rootrc` file in the following order:

- `./.rootrc //local directory`
- `$HOME/.rootrc //user directory`
- `$ROOTSYS/etc/system.rootrc //global ROOT directory`

If more than one `.rootrc` files are found in the search paths above, the options are merged, with precedence local, user, global. The parsing and interpretation of this file is handled by the ROOT class TEnv. Have a look to its documentation if you need such rather advanced features. The file `.rootrc` defines the location of two rather important files inspected at start-up: `rootalias.C` and `rootlogon.C`. They can contain code that needs to be loaded and executed at ROOT startup. `rootalias.C` is only loaded and best used to define some often used functions. rootlogon.C contains code that will be executed at startup: this file is extremely useful for example to pre-load a custom style for the plots created with ROOT. This is done most easily by creating a new TStyle object with your preferred settings, as described in the class reference guide, and then use the command `gROOT->SetStyle("MyStyleName");` to make this new style definition the default one. As an example, have a look in the file rootlogon.C coming with this tutorial. Another relevant file is `rootlogoff.C` that it called when the session is finished.

0.8.3 2.8.3 ROOT command history

Every command typed at the ROOT prompt is stored in a file `.root_hist` in your home directory. ROOT uses this file to allow for navigation in the command history with the up-arrow and down-arrow keys. It is also convenient to extract successful ROOT commands with the help of a text editor for use in your own macros.

0.8.4 2.8.4 ROOT Global Pointers

All global pointers in ROOT begin with a small "g". Some of them were already implicitly introduced (for example in the section Configure ROOT at start-up). The most important among them are presented in the following:

- **gROOT:** the `gROOT` variable is the entry point to the `ROOT` system. Technically it is an instance of the `TROOT` class. Using the gROOT pointer one has access to basically every object created in a ROOT based program. The `TROOT` object is essentially a container of several lists pointing to the main `ROOT` objects.

- **gStyle:** By default ROOT creates a default style that can be accessed via the `gStyle` pointer. This class includes functions to set some of the following object attributes.
- Canvas
- Pad
- Histogram axis
- Lines
- Fill areas
- Text
- Markers
- Functions
- Histogram Statistics and Titles
- etc ...
- **gSystem:** An instance of a base class defining a generic interface to the underlying Operating System, in our case `TUnixSystem`.
- **gInterpreter:** The entry point for the ROOT interpreter. Technically an abstraction level over a singleton instance of `TCling`.

At this point you have already learnt quite a bit about some basic features of ROOT.
Please move on to become an expert!