# ROOT primer

# Get Started

- Open the web page:
  https://root.cern.ch/root/htmldoc/guides/primer/ROOTPrimer.html

- In the following, we will go step-by-step through the examples.

- Example macros and input files can be found in:
  $ALIBUILD_WORK_DIR/SOURCES/ROOT/v6-16-00/v6-16-00/documentation/primer/macros/

# Errata in the primer tutorial

- macro5.C → 16bins instead of 100

- 7.2.4.
  for i in 0 1 2 3 4 5;  do root -l -x -b -q
  "write_ntuple_to_file_advanced.C(\"conductivity_experiment_${i}.root\", 100)"; done

# Python

- Execute “python3” in terminal

```
import ROOT
f1 = ROOT.TF1("f2","[0]*sin([1]*x)/x",0.,10.)
f1.SetParameter(0,1);
f1.SetParameter(1,1);
f1.Draw();
```

- Exectute “python3 macro3.py”

# Additional slides

# 2.1 ROOT as calculator

You can even use the ROOT interactive shell in lieu of a calculator! Launch the ROOT interactive shell with the command

```
> root
```

on your Linux box. The prompt should appear shortly:

```
root [0]
```

and let's dive in with the steps shown here:

```
root [0] 1+1
(int) 2
root [1] 2*(4+2)/12.
(double) 1.000000
root [2] sqrt(3.)
(double) 1.732051
root [3] 1 > 2
(bool) false
root [4] TMath::Pi()
(double) 3.141593
root [5] TMath::Erf(.2)
(double) 0.222703
```

# 2.3 ROOT as function plotter

Using one of ROOT's powerful classes, here `TF1`,[1] will allow us to display a function of one variable, *x*. Try the following:

```
root [11] TF1 f1("f1","sin(x)/x",0.,10.);
root [12] f1.Draw();
```

`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.

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(<parameter_number>,<parameter_value>)` of class `TF1`. Here is an example:

```
root [13] TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
root [14] f2.SetParameter(0,1);
root [15] f2.SetParameter(1,1);
root [16] f2.Draw();
```

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.

```
// Example drawing the interference pattern of light
// falling on a grid with n slits and ratio r of slit
// width over distance between slits.

auto pi = TMath::Pi();

// function code in C
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);
}

// This is the main program
void slits() {
  float r,ns;

```

```
24    // request user input
25    cout << "slit width / g ? ";
26    scanf("%f",&r);
27    cout << "# of slits? ";
28    scanf("%f",&ns);
29    cout <<"interference pattern for "<< ns
30         <<" slits, width/distance: "<<r<<endl;
31
32    // define function and set options
33    TF1 *Fnslit  = new TF1("Fnslit",nslit,-5.001,5.,2);
34    Fnslit->SetNpx(500);
35
36    // set parameters, as read in above
37    Fnslit->SetParameter(0,r);
38    Fnslit->SetParameter(1,ns);
39
40    // draw the interference pattern for a grid with n slits
41    Fnslit->Draw();
42  }
```

# 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.

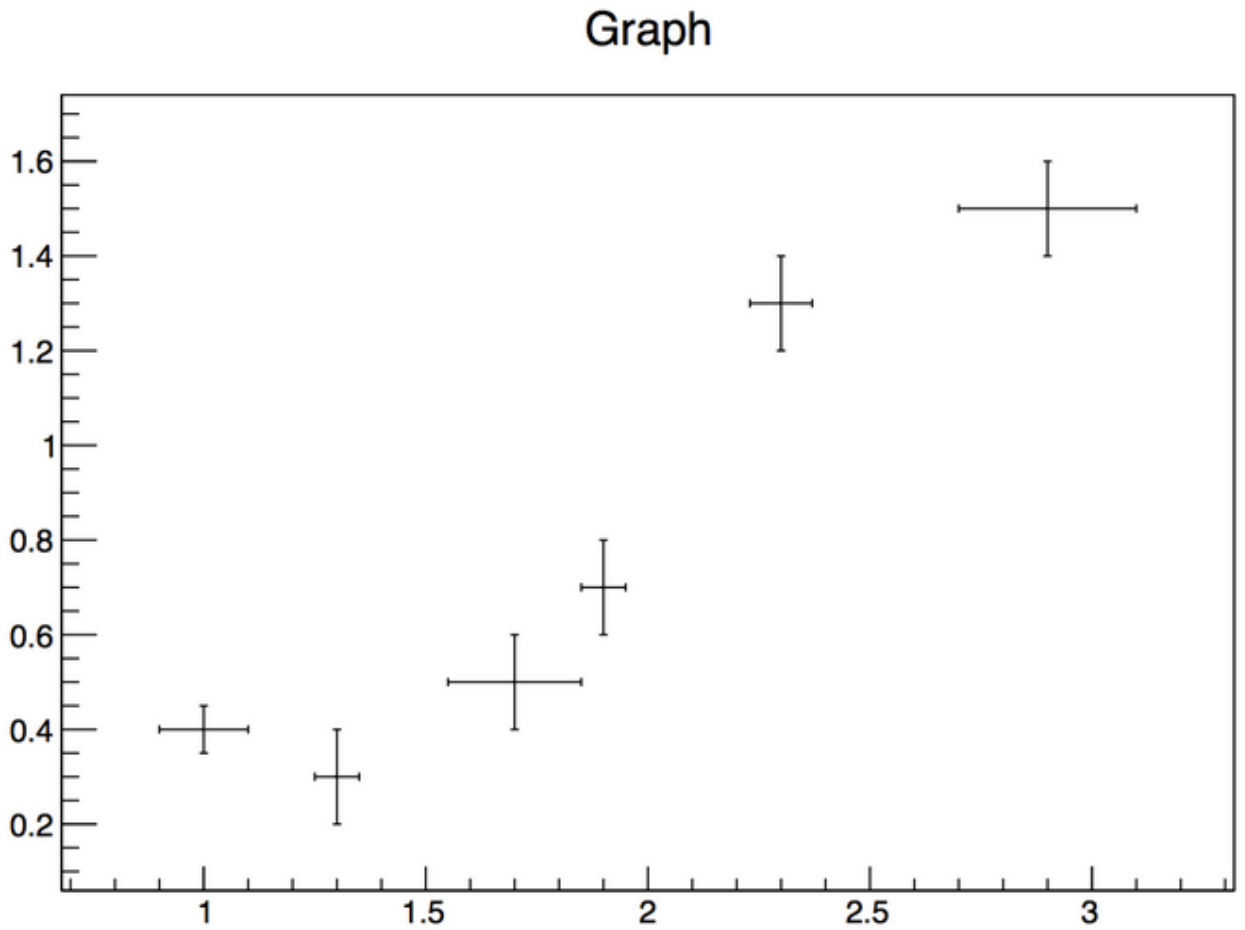
Use `.help` at the prompt to inspect the full list.

## 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:

```
root [0] TGraphErrors gr("ExampleData.txt");
root [1] gr.Draw("AP");
```

You should see the output shown in Figure 2.2.




```
# 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
```

# 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.

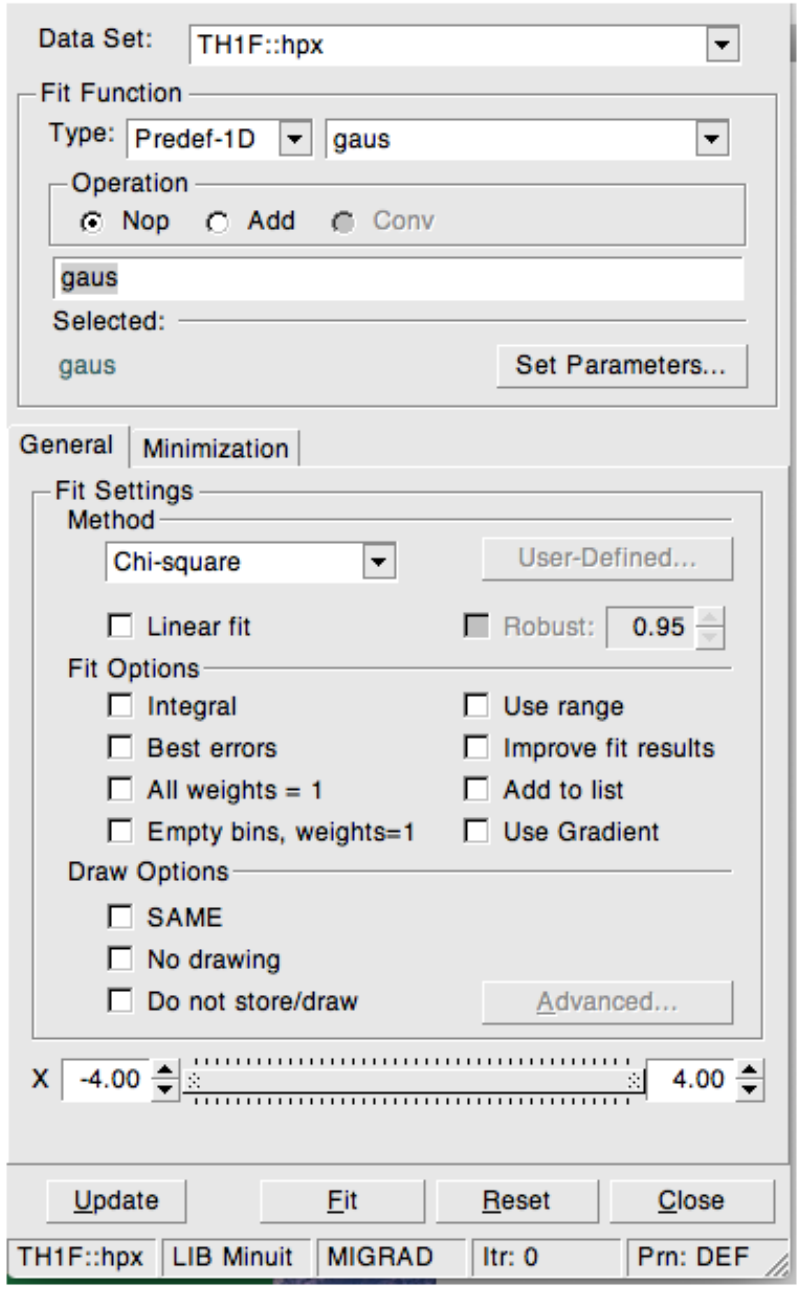
```
root [0] TF1 efunc("efunc","exp([0]+[1]*x)",0.,5.);
root [1] efunc.SetParameter(0,1);
root [2] efunc.SetParameter(1,-1);
root [3] TH1F h("h","example histogram",100,0.,5.);
root [4] for (int i=0;i<1000;i++) {h.Fill(efunc.GetRandom());}
root [5] h.Draw();
```

The first three lines of this example define a function, an exponential in this case, and set its parameters. In line *3* 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.

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.

```
root [1] TH1F h("h","example histogram",100,0.,5.);
root [2] ifstream inp; double x;
root [3] inp.open("expo.dat");
root [4] while (inp >> x) { h.Fill(x); }
root [5] h.Draw();
root [6] inp.close();
```

# Fit panel




Fit Panel.

# 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.

## 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!

### 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.

### 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.

EXTREMELY USEFUL!

```
$> tail ~/.root_hist
```

### 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.