1 Introduction

This is a guide to exploring some of the basic features of Root which I have found useful. It is not meant to be a complete guide, but meant to steer you in the right direction, and point out some of the basic features of Root. I chose these features because I found them useful. Root can do much more than this guide lays out. So I suggest that you use other guides as well. Usually if you are stuck on something, an internet search will provide some resources. Also, the Root manual provides a lot of useful information than given here. In addition, the tutorials can be quite useful for learning, but they are just C files, and typically do not include much in the way of explanation.

2 A Quick Example

Let’s start with an example and make a plot. Load root (in Unix) by typing “root” in the command line. Root will load, and you should see the prompt, “root [0]”. Type the following command:

```c
root [0] T F 1 f1("func1", "sin(x)*exp(-x)",0,10)
root [1] f1->Draw()
```

This will make the plot shown in Fig. 1.
The first command created a function called “f1” from \{0,10\} and the second command made a plot of the function. Typing commands directly in the command prompt is useful, but you may want to first write the code to a file. Type the code into a text file this time, enclosing the code in \{\}. See example1.C.

\begin{verbatim}
{ 
    TF1 f1 ("func1", "sin(x)*exp(-x)", 0, 10);
    f1->Draw();
}
\end{verbatim}

Then to run the file, type in the command line:

root -x <filename>

or if root is already open, just type:

-x <filename>

in this case

root -x example1.C

See file “example1.C”

3 Macros

Root can be used in C++ code. As in the program “data.C” (we’ll get to what this does later) run this as C++ code by executing the command:

root -x data.C

This executes the code, but does not compile it. If you want to compile the code, and see error messages, run:
root −x data.C

You can also just load a file, and call it later:

root .L data.C

To alert C++ to the fact that you are using Root packages, you need to include the root package used, such as:

#include <TH1.h>

Also, the filename needs to be the same as the function which you want to run. Note that people often use Double_t instead of double, Int_t instead of int etc., which just make the variables platform independent.

4 Histograms

4.1 1D Histograms

1D histograms are created using the TH1* class, where * could be “F” for floating point, “D” for double precision, etc (see the root manual). Let’s consider the histogram from the program “data.C”:

TH1 *h = new TH1F("h", "Histogram", 9, 0.05, 0.95);

TH1 *h

creates a pointer, called h, which is then filled by TH1F. The argument of the TH1F function is as follows: (“object name”, “Display Name”, # bins, xmin, xmax). After creating a histogram like this, it is blank, so we need to fill the histogram. There are several approaches to doing this. The easiest way is to use the fill command, as shown in example2.C

```c
void example2(){
    Double_t x[np] = {1.913521, 1.953769, 2.347435, 2.883654,
        3.493567, 4.04756, 4.337210,
        4.364347, 4.563004,
        5.054247, 5.194183, 5.380521,
        5.303213, 5.384578,
        5.563983};

    TH1F *h = new TH1F("h", "func t", 20, 0.6); for(Int_t i =0; i<15; i++) {
        h->Fill(x[i]);
    }
    h->Draw();
}
```

The fill command automatically divides the numbers into appropriate bins.

Suppose we wanted to fill a histogram according to a function. We could define some function (here f1), then fill:
for (Int_t i=0; i<100; i++) {
    x = static_cast<double>(i)/100.0;
    f = f1(x);       // f1 is some function which you have defined,
                     // or it could be an array
    h->Fill(f);
}

Exercise 1: Create a histogram of the function given in example 1, with 50 bins, from 0 to 10. Now try to compile your code using the command root -x <file_name>+. You will need to include several libraries, which you should specify. If you need a hint, check the example3 code. If this does not make immediate sense, try plotting the function, and see if that helps.

You can also fill a histogram with a random number distribution. For instance, you could use the fill command:

h->FillRandom("gaus",1000)

where the first argument tells the fill command to fill according to a pre-defined function (a gaussian) and the number specifies the number of points to use. More on random numbers later.

Including an extra argument in the fill command allows you to include a weight, either a number or a evaluating a function, such as:

h->Fill(f,y);       // y is a weight factor

See example3_1.C

4.2 Fitting Histograms

A histogram can be fit to a pre-defined function. See the following example:

void example5() {
    TH1F *h=new TH1F("h", "Gaussian", 50, -5,5);
    for (Int_t i=0; i<50; i++) {
        h->FillRandom("gaus"); // fills the histogram with
                               // randomly distributed numbers in a gaussian distribution
    }
    h->Fit("gaus");       // Fits the histogram to a gaussian
}

4.3 Formatting Tips

The default histogram formats can in general be quite annoying. Perhaps the least trivial formatting option is including axes. For a histogram h, you can use the following syntax to include a label on the x axis.

h->GetXaxis()->SetTitle("X Axis Title ");

To see the title, use the command:
\[ \text{h->SetTitle ("test ");} \]

You can set the fill pattern and the command:

\[ \text{h->SetFillStyle(3005)} \]

where the 4 digit number corresponds to various fill patterns, and 3005 in particular corresponds to cross-hatching. See page 133 of the root manual for other fill patterns. To set the fill color:

\[ \text{h->SetFillColor(4)} \]

where 4 corresponds to blue. See page 134 of the root manual. If you have more than 1 histogram on the same plot, it can be useful to include a legend, which is another object like TH1 or TF1. See example4. By using the draw option:

\[ \text{h->Draw("same")} \]

More than 1 histogram can be included in on the same plot. To draw a normalized histogram, use:

\[ \text{h->DrawNormalized()} \]

It can also be useful to get say the mean value from a histogram, which would be done using the command:

\[
\text{Double_t mean; mean=h->GetMean(); // GetMean() returns a double}
\]

Exercise2: Add a second function to the histogram to example3 (hint: for the second histogram, use the "same" option when you draw the histogram). Now, fill them using different colors, and different fill patterns. Then include a legend. See example5.

### 4.4 Canvases

It is sometimes useful to put more than one histogram in the same window (called a canvas). To create a canvas, use the command:

\[
\text{TCanvas *c1=new TCanvas("c1", "canvas", 800,800);} \]

The second argument is the title, and the last two numbers are the size of the canvas. To divide the canvas, and set the active portion of the canvas,

\[
\text{c1->Divide(2,2); // Divide canvas into 4 quadrants}\]
\[
\text{c1->cd(1); // set active portion to quadrant 1}\]

Now when you go to plot a histogram it will be in this quadrant.
4.5 Saving Output

You can print a canvas to a postscript file using the syntax:

```c
Print("test.eps");
```

See page 138 for other output options.

4.6 2D Histograms

2D Histograms basically follow the same general principles as TH1, so this section will not be as in depth. Use the command,

```c
TH2F *h2=new TH2F("h2", "Name", xbin, xmin, xmax, ybin, ymin, ymax);
```

To fill the histogram, there are now 2 numbers you need to fill,

```c
h2->Fill(x[i], y[i]);
```

where x[i] and y[i] may be numbers from an array, or functions. If you are filling with numbers from an array, you will need to perform a double sum. You can also create projections of 2D histograms using the command:

```c
TH1D *dx=h2->ProjectionX();
```

There are a number of output formats for 2D histograms. See page 27 and following. Some personal favorites are

```c
h2->Draw("cont3");
h2->Draw("surf3");
```

5 Random Numbers

There are several random number generators in root, with the best being TRandom3. When using random numbers, you also need to include a "seed" value. The default is to re-seed each time the program compiles. The argument of 0 tells the random number generator to use a seed based on the machine clock. To make a random variable, use the syntax:

```c
TRandom3 r(0); // make a random variable r, with a seed generated by the machine clock
```

It is often useful to get random numbers in a particular distribution. For instance, this will generate a gaussian distribution with a mean of 0 and a sigma of 1.

```c
Double_t rand;
rand=r.Gaus(0, 1); // Mean 0, sigma 1
```

You can also make uniform distributions, and a number of other useful distributions (see 235).

```c
Double_t rand;
rand=r.Uniform(-1, 1); // Uniform random number distribution between -1 and 1.
```
Exercise 3: Make a function to generate 3 cartesian components of a vector which is uniformly distributed in solid angle. Note that you cannot just randomly generate x,y,z components. Hint: think of the definition of solid angle: $d\Omega = d(cos\theta)d\phi$. If you need more hints, see PDG 33.4.2.

6 Vectors

To use vectors, you need to load this package in your code,

```cpp
gSystem->Load("libPhysics");
```

where you include:

```cpp
#include <TSystem.h>
```

and

```cpp
#include <TVector3.h>
```

This allows us to define a vector:

```cpp
TVector3 r;
```

To set the components of the vector, use the command:

```cpp
r.SetX(1); // set x component to 1
r(0)=1; // set x component to 1
// or
r.SetXZY(1,2,3); // set x,y,z components to 1,2,3.
```

This library also supports a number of vector operations, such as:

```cpp
TVector3 r1, r2; // Need to set to some vector, say:
r1.SetXYZ(1,0,0);
r2.SetXYZ(0,1,0);
r1+r2; // addition
r1=r2; // comparison
r1=r2; // set r1 equal to r2
r1*r2; // scalar product
r1.Dot(r2); // scalar product
r1.Cross(r2); // scalar product
r1.Angle(r2); // Angle between r1 and r2
r1.Rotate(0.707,r2); // rotate r1 by 0.707 radians about r2;
```

To print a vector, use the following function (see example)

```cpp
void vector_print(TVector3 v, string name) {
    // Prints vector components. Use input format of (vector, name of vector)
    cout << name << "": x " << v(0) << " y " << v(1) << " z " << v(2) << endl;
}
```