ROOT Primer – by Jacob Ribnik (uribnik@physics.ucla.edu)

These notes contain nearly every line of code necessary to complete the assignment. Refer to the accompanying handouts for ROOT installation procedures and CINT commands. Knowledge of the C++ programming language is assumed.

The ROOT data analysis framework is a collection of C++ header files that define ROOT types and classes to use in computation and graphical rendering. It also includes CINT, a C and C++ interpretter for interactive sessions. CINT is launched by running the root executable. An effective way to use ROOT is to *load* a C++ program in CINT and call the main function yourself (type main()) to execute the program. Using ROOT interactively with CINT saves the trouble of constantly compiling programs with ROOT headers only to output ROOT files that need to be viewed in CINT; CINT loads all ROOT headers automatically.

To use ROOT, you must first initialize the Unix environment to find and run ROOT (this is already done on the machines in the 180F lab). The Physics Department default shell is csh, but you can easily change to bash (the Linux default) by running `bash`.

```
csh:      setenv PATH /usr/local/root_v4.00/bin:$PATH
          setenv ROOTSYS /usr/local/root_v4.00

bash:     export PATH=$PATH:/usr/local/root_v4.00/bin
          export ROOTSYS=/usr/local/root_v4.00
```

To automatically set these environment variables upon future logins, add the above lines to the .login or .cshrc files for csh, or to .bash_profile for bash.

If the environment variables are properly set, run interactive ROOT by typing `root`.

In order to draw a histogram, graph, formula or anything, you must prepare a canvas to draw on.

```
TCanvas* c1 = new TCanvas("c1", "My First Canvas");
```

TCanvas is a ROOT defined type for the object that IS the visual canvas accompanying anything drawn to the screen. The above code initializes the `c1` object of type TCanvas* with the name `c1` and the title 'My First Canvas'.

If you have only one canvas all objects are displayed there replacing previously drawn objects... so make more canvases.

```
TCanvas* c2 = new TCanvas("c2", "My Second Canvas");
```

Like any C++ object, the TCanvas type have a number of member functions that act on them.

How do I switch between canvases to draw on?

```
c1->cd(); // switch to canvas c1

  blah // this is where you draw something

c2->cd(); // switch to canvas c2

  blah // this is where you draw something else
```

Can I set the c2 canvas x-axis to log scale? Sure!

```
c2->SetLogx();
```

Can I change the c1 canvas window size? No. Wait, I mean yes!

```
c1->SetWindowSize(500,500); // in pixels
```

In the above examples arrows dereference (locate in memory) pointer type objects and the objects are acted upon by member functions.

Now that you have a place to draw objects, let us make a one dimensional histogram that stores floats and draw it.

```
TH1F* h1 = new TH1F(“h1”, “My First Histogram”, 100, 0, 1);
```

The h1 object of type TH1F* is initialized with the name h1, the title 'My First Histogram', and 100 bins along the x-axis that ranges from 0 to 1.

Let us draw this sucka...

```
h1->Draw(); // many object types have a member function Draw()
```

Dazzle me with a two dimensional histogram of doubles...

```
TH2D* h2 = new TH2D(“h2”, “My Second Histogram”, 100, 0, 1, 100, 0, 1);
```

The last three arguments are the number of bins and range for the y-axis. A TH3 class object requires three additional arguments for the z-axis.

Now make a random number and insert into h1...

```
double xrand = gRandom->Rndm();

h1->Fill(xrand);
```

Make another random numer and fill the random point into h2...

```
double yrand = gRandom->Rndm();

h2->Fill(xrand, yrand);
```

Objects whose names begin with g are ROOT global variables. gRandom is a ROOT global variable of type TRandom* (a random number seed), and Rndm() is a member function of the TRandom type that returns doubles between 0.0 and 1.0 when acted upon TRandom type objects.

You can also fill histograms with a 'weight factor' so that TH1 class objects appear as 2D histograms and TH2 class objects appear as 3D histograms without having to define the last axis range. Simply include an extra argument in the Fill() member function as if you were filling a histogram of one higher dimension.

```
h1->Fill(xrand, yrand); // same way one fills a 2D histogram
```

The weight factor is advantageous because instead of filling a 1D histogram with a value one hundred times, you can simply fill the value once with a weight factor of one hundred to accomplish the same thing.

That is basically all you need on the ROOT side to complete the assignment. Of course you require some C/C++ loops:

```
double squareIter;

for (squareIter = 1.0; squareIter < 100001.0; squareIter++) {
  blah // do this every time through the loop
}
```

Notice that the squareIter double is incremented by one each time through the loop by the ++ operator.

And also boolean conditionals:

```
if (radius2 <= 1) {
  blah // do this if radius2 is less than or equal to one
}
```

In C++ one outputs to the terminal by directing output to std::cout (the standard output) which is of type ostream. In addition to automatically loading ROOT and C/C++ header files, CINT automatically recognizes the std (standard) namespace; so direct output to cout:

```
double pi = 3.14;

cout << “My goal is to calculate Pi which equals “ << pi << endl;
```

where endl (std::endl) inserts a carriage return.

Things you will require outside the assignment...

The .dat files from the DOS machines are ascii (text) files with columns of numbers. Each line of a .dat file will be read into an ntuple of equally many appropriatley named columns.

For example, a .dat file from a TDC that uses five stop chanels may look like:

```
103     99      -1      -1      -1
-1      -1      1033    1034    999
```

Create an appropriate ntuple:

```
TNtuple *myNtuple = new TNtuple("myNtuple", "data from ascii file", "stop1:stop2:stop3:stop4:stop5");
```

This ntuple has five columns named stop1, stop2, stop3, stop4, and stop5.

You can fill the ntuple with the Fill() member function:

```
myNtuple->Fill(103, 99, -1, -1, -1);
myNtuple->Fill(-1, -1, 1033, 1034, 999);
```

Or fill the ntuple with arrays of values:

```
double myArray[5] = {103, 99, -1, -1, -1};

myNtuple->Fill(myArray);
```

Define a one dimensional function:

```
TF1* myFunc = new TF1(“myFunc”, “[0]+exp([1]*x)”, 0, 1000);
```

The myFunc object of type TF1* is initialized with the name myFunc, the expression 'first constant plus exponential of second constant times x', and defined on the range 0 to 1000.

Computers start counting at zero, so the first parameter is represented by [0] and the second by [1].

Predefine function parameters to speed up the fitting process. This reduces the number of calls to the fitter, thereby reducing the number of times you must call the Fit() function to get a convergent result.

```
myFunc->SetParameter(0, 50); // set parameter 0 to 50
myFunc->SetParameter(1, 1e-5); // set parameter 1 to 10^-5
```

Fitting a histogram:

```
h1->Fit("myFunc"); // fit h1 to myFunc
```

TGraph and TGraphErrors types will be particularly useful for plateau plots. Unlike histograms where you fill values, TGraph and TGraphErrors types are initialized with arrays of values.

```
const double numVals = 20;
double xVals[numVals] = {blah}; // 20 x values
double yVals[numVals] = {blah}; // 20 y values

TGraph* g1 = new TGraph(numVals, xVals, yVals);
```

Make additional arrays for error values and use the TGraphErrors type:

```
double xValsError[numVals] = {blah}; // 20 x error values
double yValsError[numVals] = {blah}; // 20 y error values

TGraphErrors* g2 = new TGraphErrors(numVals, xVals, yVals, xValsError, yValsError);
```

When you make changes to histograms, graphs or canvases refresh them graphically by dereferencing them with the Draw() function again.

Plenty more where that came from @ http://root.cern.ch

Particularly helpful is the Reference Guide that lists every ROOT type and class and their respective properties and member functions and their respective properties.