ROOT Users Guide

Introduction:

Root was developed by Rene Brun (author of PAW) and others as an object-oriented analysis framework for particle physics. Like PAW, it is a general-purpose program to fit arbitrary functions to your data via the method of Chi-square minimization (or maximum likelihood), and offers a variety of plotting options. The main advantage over a program like Microsoft Excel is that it can fit nonlinear functions, and it returns the errors on the fitted parameters. Also, it is distributed free of charge.

Root differs from PAW in that it is based on the modern C++ programming language rather than Fortran or the KUMAC scripting language. So you don't have to learn a new syntax. It can run compiled programs, so it is faster, and it has a nice GUI for manipulating plots. On the other hand, you have to learn the many class names that are available, which in a sense is like learning new commands. Also, I find that you have to write more code to accomplish the same thing as in PAW.

This manual is written as a guide to Root for PAW users.

Important Note: Root is case-sensitive like C++

Documentation and Help:

Root does much more than is listed in this quick tutorial. If you need advanced fitting or plotting options, consult the manual and the extensive online tutorials and How-To's, which can be found from the following web site:

`http://root.cern.ch/`

The Root manual is very good, but it is 360 pages. So this guide is meant as a starter.

Starting and Quitting Root:

From the UNIX command prompt, if your environment is set correctly, type "root".
From Windows, you must double-click on the Root icon. The command prompt doesn't seem to work. As with UNIX, your environment must be set correctly.

You exit the program by typing **".q"** from the command prompt.
You can reset Root with: `gROOT->Reset();`
A history of all commands typed in recent Root sessions is kept in the file `~/.root_hist`

Data Entry:

Interactively

To enter a set of data by hand, just create a C array:

```
     float x[5] = {10.0, 20.0, 30.0, 40.0, 50.0};
     float y[5] = { 1.1,  1.2,  1.3,  1.4,  1.5};
     float dy[5] ={ 0.1,  0.1,  0.1,  0.1,  0.1};
```

In this example, each array is of dimension 5 and contains real numbers (you can also use type “int” for integers.

From a file

To read data from a file created by another program, you need to write a piece of C code:

```
{
   FILE *fp = fopen("/user/brun/root/basic.dat","r");

   Float_t x,y,z;
   Int_t ncols;
   Int_t nlines = 0;

   while (1) {
      ncols = fscanf(fp,"%f %f %f",&x, &y, &z);
      if (ncols < 0) break;
      nlines++;
   }
   fclose(fp);
}
```

Data Plotting

Printing

You can print out to the screen the contents of a variable:

```
     cout << x[0] << “ “ << x[1] << “ “ << x[2] << “ “ << x[3] << “ “ << x[4] << endl;
```

Plotting Data Points with Error Bars

You can plot data with error bars drawn, but you need arrays for `x, y` and their associated error bars `dx, dy` (which might be zero). You also have to provide the number of points and the symbol type, and superimpose the plotting on an existing graph. For example:

```
     float x[5] = {10,20,30,40,50};
     float y[5] = {1.1,1.2,1.3,1.4,1.5};
     float dy[5] = {0.1,0.1,0.1,0.1,0.1};
     float dx[5] = {5,5,5,5,5};
     TGraphErrors gr(5,x,y,dx,dy);
```

```
gr.SetMarkerColor(2);
gr.SetMarkerStyle(20);
gr.SetMarkerSize(1.0)
gr.Draw("AP");
```

The marker attributes are similar to those in PAW. Namely, for the symbol types:
20 (solid circle), 21 (solid square), 22 (solid triangle),
24 (empty circle), 25 (empty square), 26 (empty triangle

For the marker colors: 1 (black), 2 (red), 3 (green), 4 (blue), 5 (yellow), 6 (magneta), 7 (cyan)

To make the same type of graph without the error bars, create a `TGraph` object and leave off the error arrays. If you want asymmetric error bars, create a `TGraphAsymmErrors` object and add arrays for both the low and high error bars.

To set axis labels:

```
gr.GetXaxis().SetTitle("X Axis");
gr.GetYaxis().SetTitle("Y Axis");
```

Parameter Estimation

You can fit a variety of functions to your array data using the method of Chi-square minimization (or maximum likelihood) to estimate parameters in the theory you are testing. It is assumed that your measured quantity is `y`, with uncertainty `dy`, and that it depends on variable `x`.

You can fit several functions to a `TGraphErrors` object using the `Fit()` method of the object. For example:

`gr.Fit(“expo”)`	(exponential)
`gr.Fit(“gaus”)`	(Gaussian)
`gr.Fit(“polN”)`	(polynomial of degree “N”)

More details are described under fitting histograms, which uses the same syntax.

Notice that a plot of the fitted line is shown through your data, and that the resulting reduced Chi-square is listed in the text window. Moreover, the errors on the fitted parameters are reported. These uncertainties are determined by allowing the Chi-square to increase by 1. They only make sense if your reported Chi-square is reasonable for the number of degrees of freedom.

Histogram Creation and Filling:

1-D Histograms

To book a one-dimensional histogram, you have to give it a label, title, number of bins, and a range. For example:

```
TH1F hist("hist","Energy",100,0,1000);
```

The “1” specifies a 1-D histogram, and the “F” specifies floating-point values. You can use “D” for double precision, “S” for short integers, and “C” for characters (one byte integers).

2-D and 3-D Histograms

To book a two-dimensional histogram, you have to give it a label, title, and number of bins and range for both x and y axes:

```
     TH2F hist("hist","Time vs. Energy",50,0,1000,50,0,5.e-9);
```

The “2” specifies a 2-D histogram, and the “F” specifies floating-point values. If you replace “2” with “3”, and generalize the constructor, you can create a 3-D histogram as well.

Variable-Width Binning

To book a histogram with variable-width binning, you define an array to contain the low edge of the bins. This dimension of this array should be one more than the number of bins, so that the last bin gives the right-most edge of the histogram:

```
     float xlow[8] = {60,70,80,90,100,120,140,180};
     TH1F hist2("hist2","title2",7,xlow)
```

Filling from Arrays

If you have an array of data that you would like to make into a histogram, write a small C++ loop. For example, if the array `x` has dimension 5, and the TH1F object is called `hist`:

```
     for (int i=0;i<5;i++) hist.Fill(x[i]);
```

Histogram Plotting:

To plot a histogram, use the Draw method of the object:

```
     hist.Draw();
```

where “hist” is the label of the histogram. You can superimpose one histogram over another by adding the option “same” in the constructor:

```
     hist.Draw(“same”);
```

You can switch the line type between solid, dashed, and dotted by:

`hist.SetLineStyle(1)`	(solid line)
`hist.SetLineStyle(2)`	(dashed line)
`hist.SetLineStyle(3)`	(dotted line)
`hist.SetLineStyle(4)`	(dashed-dotted line)

You can set color of the histogram using the `SetLineColor(n)` method of the histogram class, the line width with the `SetLineWidth(n)` method, and the fill color with the `SetFillColor(n)` method. See the class descriptions for more details.

Histogram Fitting:

You can fit several functions to a histogram using the `Fit()` method of the histogram object. For example:

```
     hist.Fit(“expo”)       (exponential)
     hist.Fit(“gaus”)       (Gaussian)
     hist.Fit(“polN”)       (polynomial of degree “N”)
```

The contents of each bin of the histogram are assumed to obey Poisson statistics, where the variance is equal to the mean. Thus, the uncertainty of the contents of each bin is taken to be the square root of the number of bin entries. This uncertainty is used for the Chi-square minimization procedure. Be aware that this method breaks down when the bin has only one or zero entries. Root will ignore bins with zero entries. Generally, one should use the method of maximum likelihood for low statistics.

To display the fit results on the canvas, you can use the command:

```
     gStyle->SetOptFit(1111);
```

To fit a user-defined function, or to fit a combination of the functions listed above, or to fit a function over a subset of the histogram range, you must create a TF1 object. For example, to fit over a subset of the histogram range:

```
     TF1 g1(“g1”,”gaus”,-0.5,0.5);
     hist.Fit(“g1”,”R0”);
```

This will fit a Gaussian over the interval –0.5 to 0.5. Note that you do not specify the range in bin numbers as in PAW, but the range along the x-axis.

Canvas and Pad Creation and Manipulation:

So far we have been letting Root create the drawing canvas and pads for our plotting. We can control this directly:

To create a drawing canvas (the window where plots show):

```
     TCanvas c1;
```

You can control the size of the window by specifying the widths (as well as name, title, and other options):

```
     TCanvas c1(“c1”,”Title”,0,0,600,400);
```

To place more than one plot on the canvas, you can divide the plotting area (similar to the “zone” command of PAW). For two columns of 3 plots, use:

```
     c1.Divide(2,3);
```

Then to select the first of the six plots, use:

```
     c1.cd(1);
```

To set grid lines on or off:

```
     c1.SetGridx(1); c1.SetGridx(0);
     c1.SetGridy(1); c1.SetGridy(0);
```

To set logarithmic scales on or off:

```
     c1.SetLogx(1); c1.SetLogx(0);
     c1.SetLogy(1); c1.SetLogy(0);
```

You can also manipulate the axes and labels interactively. In particular, you can click on text objects and move them around. You can right-click on the space around the plot to enable/disable grid lines, logarithmic axes, etc.

Working with Trees (Ntuples)

Ntuples are basically binary files used to store data. Root allows for Ntuples like PAW, but using a file format with better I/O capabilities. Ntuples in Root are referred to as “Trees”. You can convert an Ntuple in the Zebra format used by PAW to a Tree in the Root format using the command:

```
h2root file.hbook file.root
```

This command is done at the Unix prompt, not within Root. The numerical label for the Ntuple is prefixed by an “h”. Root Trees are very similar to the “column-wise” Ntuples used in PAW.

Opening a Root file:

You open a Root file within Root by creating an object of that type, with the filename as an argument to the object constructor:

```
TFile f1("junk1.root")
TFile f2("junk2.root")
```

If you open a file using the PC version of Root, be sure to use the Unix directory format (forward slashes rather than backward slashes):

```
TFile f1("c:/Acosta/temp/junk1.root")
```

To view any subdirectories within a file, type:

```
f2.ls();
```

To change to one of the subdirectories, try something like:

```
f2.cd(“mydirectory”);
```

To set the default directory to the first file given in the example above, type:

```
f1.cd()
```

If you have several files with identical format ntuples, you can chain them together:

```
TChain chain("h10");
chain.Add("junk1.root");
chain.Add("junk2.root");
```

In the previous example it is assumed that “h10” is the Ntuple label. If the Tree is in a subdirectory of the Root file, you must do something like the following:

```
TChain chain("CLCANALYSISMODUL/h1");
chain.Add("junk1.root");
chain.Add("junk2.root");
```

If you would rather merge all the files into one file, you presently need to use a macro provided on the Root web page. If the files were generated in Zebra format originally, you could merge the files with PAW before converting to Root.

Browsing Root files:

To see what ntuples exist in the current file and subdirectory, use the `ls()` method of the TFile object:

```
      f2.ls();
```

You generally get a list of labels for all Trees and histograms stored in the file. You can get an ASCII dump of the variables in a Tree using the `Print()` method:

```
      h10.Print();
```

You can use the Root Browser for a GUI to the Tree. Select the "inspect" pull-down menu from your canvas, and choose "Start Browser". Alternatively, create the object from the command line:

```
      TBrowser b;
```

This gives you a browser like the Windows Explorer. Double-click "ROOT Files", and then double-click on the name of your file. You should then see a list of all Trees and histograms stored in the file. Double-click on one of the Tree names, and you should then see the variables (or "leafs") of that Tree. Click on the detailed-view button (like in Windows) to see the variable properties such as if it is a `float` or `int`, and whether it is an array.

An even better viewer can be started once you know the name of the Tree you want to inspect. For example, if your Tree is "h10", you can browse it with:

```
      h10.StartViewer();
```

You can also start this viewer by right-clicking on the name of the Tree from within the Object Browser. If you double-click on a leaf, you will automatically get a histogram of the variable.

To scan particular events (equivalent to ntuple/scan in PAW), see the discussion of MakeClass below.

Ntuple Plotting

To plot one of the variables in a Tree (Ntuple) type:

```
      h10.Draw(“x”);
```

where "x" is the variable you want to see plotted and "h10" is the label of the Tree. Root will create a histogram automatically for you (labeled "`htemp`"), and set the range and binning as well. It also creates a default canvas and pad for you as well.

If you want to make a 2-D scatter plot of "y" vs. "x", the syntax is:

```
      h10.Draw(“y:x”);
```

You can also plot any C++ arithmetic expression of the variables:

```
      h10.Draw("pow(x,2)+pow(y,2)");
```

If you want to apply selection cuts, they are entered as the second constructor argument in C++ syntax:

```
      h10.Draw(“x”, “y>10 && abs(z)<20”);
```

Note that in C++ syntax, "&&" is AND, "`||`" is OR, "==" tests equality, and "`!=`" tests inequality. You could store the cuts in a TCut object:

```
	TCut cut1 = "y>10 && abs(z)<20"
```

And pass this variable as an argument to the Draw method.

To superimpose a plot on the same histogram, add the "same" argument:

```
	h10.Draw("x", "y>10 && abs(z)<20", "same");
```

If you want to fill a specific histogram, try something like:

```
	h10.Draw("x >> myHist");
```

Indexed variables

As mentioned at the beginning of this section, Root Trees are like column-wise ntuples in PAW. Thus, you can store and access arrays of data rather than creating a variable name for each array element. If you want to access an element of an array, you need to know what the indexing variable is:

```
	h10.Draw("ptvalue[0]", "ntrack>0");
```

Here "ntrack" is the indexing variable. We have to require the existence of at least one track in order to make the plot, otherwise we would get an access violation. The above will plot the first value in the ptvalue array for all events. (Note that in C++ arrays start from 0 rather than 1 as in Fortran). If you want to plot all array entries for all events, you could just do:

```
	h10.Draw("ptvalue");
```

without the index.

MakeClass:

Root allows you to execute C++ code in place of the selection cuts and plotting variables. These functions can be interpreted by Root interactively, or compiled and run (improving execution speed). You have access to all the utility functions stored in the Root libraries as well. If you want to be able to access variables within your Tree, you need to create a class for the Tree (like defining the Fortran common blocks for a PAW Ntuple using the `ntuple/uwfunc` command). The way to do that is to execute the following command:

```
	h10.MakeClass("MyClass")
```

This command will create two files (`MyClass.h` and `MyClass.C`) that are the class definition for the Tree object labeled "h10". You can now edit the `MyClass.C` file and put whatever selection, loops, and analysis you want. For example, to dump the contents of your Tree for event 12 do the following:

```
{
	.L MyClass.C;
	MyClass t;
	t.Show(12);
}
```

This is somewhat analogous to `ntuple/scan` in PAW.

Now to loop over all events in the Tree and execute some code (such as filling histograms), you can edit the `Loop()` method of `MyClass.C` and the do the following:

```
{
     .L MyClass.C;
     MyClass t;
     t.Loop();
}
```

In addition, you can add your own method to the Tree object, and loop over all events interactively to fill histograms using this method:

```
{
     .L MyClass.C;
     MyClass t;
     TH1F hist(“hist”,”Title”,10,0,10);
     for (int i=0; i < h10.GetEntries(); i++) {
          t.GetEntry(i);
          hist.Fill(t.Cut(i));
     }
}
```

In this example we assume that the Tree is labeled “h10”, and that the method `Cut()` was defined in the MyClass object for this Tree to perform some operation based on the variables contained in the Tree.

If the Root file you want to analyze is not the same as the one you used to create `MyClass` using the `MakeClass()` method, then you must pass the Tree to the constructor of `MyClass`:

```
     TFile f("csc_track_mb_nopu.root")
     mytree = (TTree*)gDirectory->Get("h10")
     TrackCWN v(mytree)
```

Macros

As you have seen in the previous examples, Root allows you to execute code in a C++ file. If your file is “test.C”, type:

```
     .x test.C
```

To just load the file (but not execute):

```
     .L test.C
```

If you load and compile the file as a shared library, the program will run much faster:

```
     .L test.C+
```

The “+” at the end causes Root to compile the code. This is also very useful to debug your macro, because the compiler will print out a list of warnings and errors for you to fix.

Printing

Saving a plot to a file is easy, as you can do it from the canvas GUI. Choose the "File" pull-down menu and select the appropriate "Save as" option. Encapsulated Postscript is file type EPS.