MoNA ROOT Guide

Greg Christian
National Superconducting Cyclotron Laboratory

July 16, 2010

1 Introduction

ROOT is an "object oriented data analysis framework" written by programmers at CERN. It is particularly suited for analysis of the large data sets found in nuclear physics. It is capable of performing basic analysis tasks such as gating, histograming and fitting, in addition to much more complex tasks. ROOT is primarily command line driven, using the CINT C/C++ interpreter.

This manual is intended to introduce those who are new to ROOT to some of its basic functions. After reading this manual you should have a "SpecTcl like" level of functionality in using ROOT, i.e. you should be able use ROOT perform all of the tasks that are possible in SpecTcl (plus a few more).[1] At times the manual may gloss over some of the more subtle subjects involved in using ROOT; this is done in the interest of promoting simplicity and quick learning of the program. The manual also makes purposeful use of imprecise terminology as opposed to technically correct C++ jargon, with the hope that it will be better understood by readers with little or no knowledge of C++.

2 Installing ROOT

Before you can start to use ROOT at the NSCL, you need to "install" it (e.g. set the needed environment variables) on your NSCL user account.[2] If you have already set up your account to run `st_mona` simulations, then you are all set and can begin using ROOT on any of the 64-bit fishtank machines. Otherwise, open up your `~/.bashrc` file, and add the following lines to the end of the file:

```
export ROOTSYS=/user/monasoft/soft64/root/root5.26
export PATH="${ROOTSYS}/bin/:${ROOTSYS}/include/:"$PATH
export LD_LIBRARY_PATH="${ROOTSYS}/lib/"
```

Also, if you are planning to use the analysis software available for MoNA experiments (`n2analysis`), you will need to add the following lines as well:

```
export N2ANA_HOME=/where/you/have/n2analysis/located
export LD_LIBRARY_PATH="${N2ANA_HOME}/lib/root/:${LD_LIBRARY_PATH}"
```

[1] SpecTcl is just being used as an analogy here; if you aren't familiar with SpecTcl, don't worry about it.

[2] If you are logged into the NSCL servers as user `mona`, then the environment variables are already set; in this case you can go ahead and begin using ROOT on the 64-bit fishtank machines.

Then save the file and type

```
> source ~/.bashrc
```

Doing the above will get you set up to use the MoNA group's curent (July 2010) installation of root. Once you have done this, the command to begin the ROOT program is:

```
> root.exe
```

When you type this at the command line, ROOT should start up and you should see something like this:

```
   ******************************************
   *                                        *
   *        W E L C O M E  to  R O O T      *
   *                                        *
   *   Version   5.26/00  14 December 2009  *
   *                                        *
   *  You are welcome to visit our Web site *
   *          http://root.cern.ch           *
   *                                        *
   ******************************************

ROOT 5.26/00 (trunk@31882, Dec 14 2009, 20:18:36 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.17.00, Dec 21, 2008
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

ROOT's command line works much like a BASH shell: you can scroll through your most recent commands by hitting the up-arrow key, and auto-complete recognized commands using the `Tab` key. You can quit the program by typing `.q` at the command line. Also note that all commands in ROOT are case sensitive.

3 Files and Trees

ROOT reads data that is stored in a binary file with the `.root` extension. These files are capable of storing many types of objects, but for the purposes of this document we will focus on the most basic one: the `TTree`. There are two example files, `example_raw.root` and `example_cal.root` located in the `/projects/MoNA/ROOT/examples/` directory. It is suggested that you use those two example files to practice the commands being presented throughout this document.

To read data from a `root` file, you must first read the file into ROOT's memory. There are two ways to this. The easiest is to simply follow the `root.exe` command by the name of the file, e.g.

```
> root.exe example_cal.root
```

Event #	Value of P1	Value of P2
1	74.689	5670.4
2	93.452	6987.6
3	45.657	3331.4
4	22.389	3787.6
5	61.723	2229.3

Table 1: Simple example of the type of data stored in a `TTree`.

The second way is to open up the file once you have already begun the program; you can do this using the following command:

```
TFile* _file0 = TFile::Open("example_cal.root");
```

One note about this method: the `_file0` can be thought of as a "variable" name (more accurately, it's the name of a pointer to the `TFile` object). It could be set to anything that you wish.

Now that you know how to read a file into ROOT, we'll go a bit into what makes up a ROOT file. As mentioned above, the most fundamental part of a ROOT file is the `TTree`; a `TTree` is essentially a collection of "events" and parameters. Each event is a collection of data values for all of the parameters present in the file. As a simple example, consider a `TTree` that stores two parameters, P1 and P2, and five events. The structure of the data might look something like what is shown in Table 1, with each event number being associated with specific values of P1 and P2.

Each `TTree` has a "name" associated with it, and in order to perform operations on the tree, you must know what that name is. To see the names of all the trees contained with the file you have open, use the command:

```
.ls
```

and you should see something like:

```
TFile**         example_cal.root
 TFile*         example_cal.root
  KEY: TTree    t;1     calibrated tree
```

Here the line `KEY: TTree  t;1  calibrated tree` tells you that you have a `TTree` with the name `t` stored within this particular ROOT file.

The *parameters* in a `TTree` are given names by the person or program who writes the ROOT file; to see a list of all of the parameters stored in your particular `TTree`, type:[3]

```
t->Print("all");
```

doing this will show you a bunch of entries that look like this:

[3]Throughout this document, we will use the tree name "t" to represent a generic TTree, and all "operations" on a TTree will assume that the tree name is "t" (unless specifically specified otherwise).

```
*.....................................................................*
*Br   95 :tof.pot_thin :                                              *
*Entries :   156690 : Total  Size=    1254998 bytes  File Size  =     523288 *
*Baskets :       12 : Basket Size=     231936 bytes  Compression=   2.40     *
*.....................................................................*
```

The lines above give information about the parameter called `tof.pot_thin`. For the purposes of this document, you just need to be able to read off the parameter name from lines like the one above: as you can see the name, `tof.pot_thin`, is located on the first row directly after the first colon. Another thing to note: if you see a paramter name that looks something like `thin.tcal[4]`, it means that particluar parameter is arranged in an array of length four (so it's really four parameters: `thin.tcal[0]`, `thin.tcal[1]`, `...`). Finally, you may soon realize that when using the method outlined above it is often annoying to scroll through many lines of output to find a particular parameter. Fortunately ROOT has a graphical "viewer" which will outline all of the parameters for you in a GUI. To use it:

```
t->StartViewer();
```

One useful feature of ROOT trees is the ability to merge `TTrees` from separate ROOT files together, by adding the second tree as a "friend tree" to the first. Essentially, when you add a friend tree to an existing tree, the existing tree looks as if it contains all of the parameters present both in itself and in the friend tree. As as example, consider the files `example_cal.root` and `example_raw.root`. Start up ROOT and load in the file `example_cal.root`. Now add as a friend the tree contained in `example_raw.root` (which is also named `t`) by doing the following:

```
t->AddFriend("traw=t","example_raw.root");
```

Once you have done this, you can work with parameters in `example_raw.root`, just as if they were contained in `example_cal.root`. However, the `AddFriend` command "renames" the tree in `example_raw.root` from `"t"` to `"traw"`, so it's parameter names must be prefaced with `traw.` For example, if I had just loaded up the `example_raw.root` file, then I would be dealing with a parameter called `pot.qraw`, but in the case where I access `example_raw.root` as a friend tree I have to access the same parameter through the name `traw.pot.qraw` instead.

In addition to the parallel merging capabilities of `AddFriend`, it is also possible to merge multiple files in series, such that a group of files can be used as if it were one large file. A grouping of files in series is called a `TChain`, and the syntax to create one is:

```
TChain* mychain = new TChain("t");
```

Note that the `t` in the expression above should be substituted with the actual name of the `TTrees` in the files that you are chaining together.

The `new TChain()` command shown above only creates a new `TChain`; you still need to add files to it using:

```
mychain->Add("file1.root");
mychain->Add("file2.root");
```

```
mychain->Add("file3.root");
mychain->Add("file4.root");
mychain->Add("file5.root");
mychain->Add("file6.root");
mychain->Add("file7.root");
.........................
```

Once you have done this you can treat `mychain` as if it were a "normal" `TTree` name.

4 Drawing Histograms

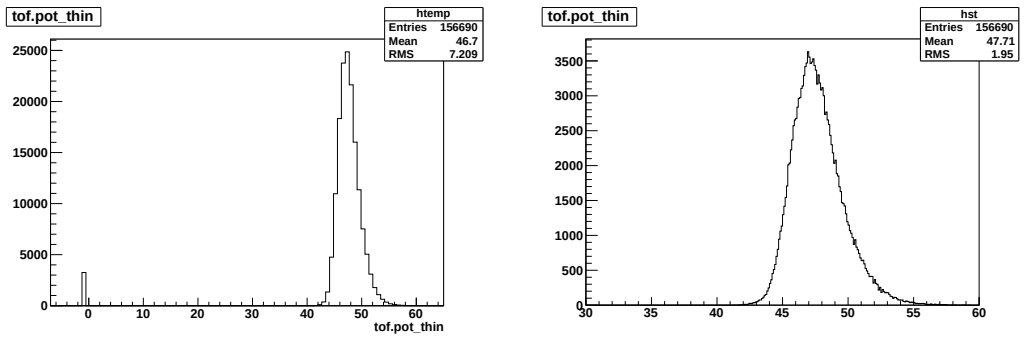
Histograms are one of the most commonly used methods of viewing data in nuclear physics. A histogram in ROOT can be thought of as the analogy of a spectrum in SpecTcl: for a given parameter it displays the number of counts falling within a user-defined bin range. The command used to draw a histogram is best illustrated with an example: lets say I am interested in seeing a histogram of the parameter `tof.pot_thin`, and I want to let ROOT set the histogram limits and bin ranges automatically. In order to do this, the command is as follows:

```
t->Draw("tof.pot_thin");
```

The histogram drawn by this command is shown in Figure 1a. Now lets say that I want to draw another histogram of `tof.pot_thin`, but I want to manually set it to display from 30 to 60 in 300 bins. The command to do this is as follows:

```
t->Draw("tof.pot_thin>>hst(300,30,60);
```

which draws the histogram seen in figure 1b. The instructions to set the binning and ranges are contained in the characters `>>hst(300,30,60)`. Here `hst` is the histogram "name," and could be set to anything.[4] The 300, 30, and 60 represent the number of bins, low bin and high bin, respectively.



(a) (b)

Figure 1: Example ROOT histograms; the histogram in the left panel is drawn with automatic binning and ranges, and the one in the right panel is manually set to go from 30 to 60 in 300 bins.

[4] Although histogram names can be reused, doing so will cause memory management issues which can eventually lead to ROOT crashing. Thus it is advisable to avoid reusing histogram names, or to explicitly get rid of the old one first using `hist->Delete()`.

The syntax to draw a two-dimensional histogram is an extension of the 1d syntax shown above. For example, if I want to plot the parameters `ic.sum` vs. `tof.pot_thin`, with `ic.sum` displayed on the y-axis from 0 to 600 in 300 bins, and `tof.pot_thin` displayed on the x-axis from 30 to 60 in 300 bins, the command is:

```
t->Draw("ic.sum:tof.pot_thin>>hst(300,30,60,300,0,600)");
```

Note that the syntax for specifying parameters is Y:X, while the syntax for setting bin and axis ranges is (somewhat unintuitively): (xbins, xlo, xhi, ybins, ylo, yhi).

The 2d draw command shown above will display the desired histogram using a 2d black and white scatterplot. It is usually desirable to view histograms in color, with different hues representing different z-axis values. Before you try drawing a 2d histogram in color, you should change the color scheme from the default (which is ugly and hard to decipher) to a more traditional blue-green-red. You can do this with the command:

```
gStyle->SetPalette(1);
```

Now that you have a nice palette, the command to draw a 2d histogram in color is:

```
t->Draw("ic.sum:tof.pot_thin>>hst(300,30,60,300,0,600)","","col");
```

It should be obvious that the `"col"` is what tells ROOT to draw the histogram in color. It should also be noted that if you want to display a color scale on the side of your histogram, you can replace `"col"` with `"colz"` in the command above. You may be a bit confused by the unfilled set of `""` in the command. These are being used as "placeholders" for the gate argument, which is described in Section 6. The results of the command above can be seen in Figure 2a.

Once you have drawn a histogram using the `Draw()` command, it is saved into memory. You can access it later using the command, `histname->Draw();`, which will re-draw the histogram named `histname` in the current window.

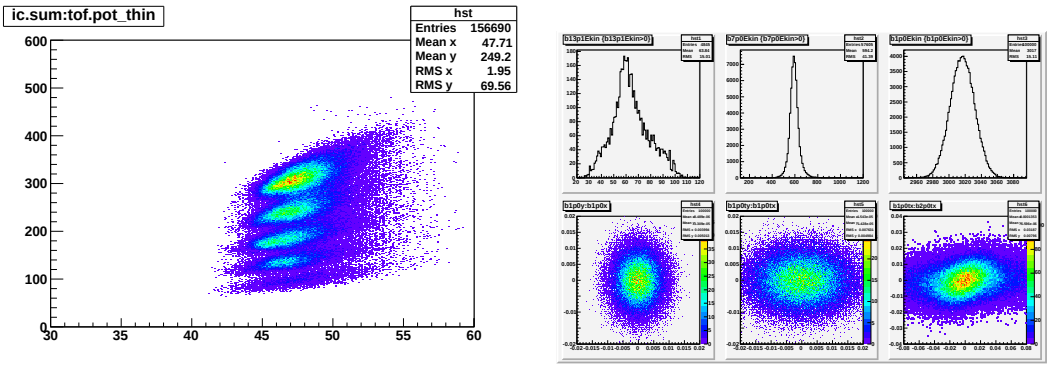
5 TCanvases

As you should have noticed, using the `t->Draw()` command creates a new graphical window in which the histogram is drawn. This window is called a `TCanvas`. You can also create a `TCanvas` manually, using the following command:

```
TCanvas* c2 = new TCanvas();
```

Here `c2` is the "name" of the `Tcanvas`, and as with histogram or tree names it could be anything you want. You can have multiple canvases open during a ROOT session, but only one of the canvases is active at a given time; the active canvas will have a yellow outline around its edges indicating that it is the active one. The command to switch between canvases is `canvasname->cd();` e.g. if I want to switch to the canvas called `c1`, then I would use:

```
c1->cd();
```



(a) Example two-dimensional ROOT histogram displaying `ic.sum` vs. `tof.pot_thin` with user defined binning of 300 bins between 30 and 60 on the x-axis and 300 bins between 0 and 600 on the y-axis.

(b) Example of a `TCanvas` divided into three pads in the x-direction and two pads in the y-direction.

Figure 2: Example 2d ROOT Histogram (left panel) and Divided Canvas (right panel).

It is also possible to divide a canvas up into "pads" or sections, allowing you to draw multiple histograms on one canvas. This is done using the `Divide()` command. For example if I want to divide `c1` so that it has three pads in the x-direction and two pads in the y-direction, then I would use:

```
c1->Divide(3,2);
```

Figure 2b shows a 3×2 divided canvas, with an example histogram drawn in each subpad.

Another useful feature is the ability to look at histograms with axes on a log scale. The command to do this in ROOT is `gPad->SetLog*()`, where `*` is the name of the axis (x, y, or z) in lowercase. The `SetLog` command only operates on the canvas or pad that is currently selected. As an example, if I have a one dimensional histogram and want to view the y-axis on a log scale, I would use:

```
gPad->SetLogy();
```

For a two dimensional histogram for which I want the z-axis on log scale, I would type:

```
gPad->SetLogz();
```

To go back to linear scale, the command is `gPad->SetLog*(0)`, e.g. if I wanted to change the y-axis back to linear, I would type:

```
gPad->SetLogy(0);
```

It is possible to modify histograms and other graphical objects using the built in GUI. To make use of this, go to the `View` menu on your canvas and select the `Editor` option. You will see a new window appear in the left edge of your canvas. Use of the editor is fairly intuitive. Note that if you click on different parts of the canvas, the object being edited (and thus the options available in the editor) change.

As a final note, you will probably want to save pictures of your canvas to file. The command to do this is:

```
c1->Print("filename.*")
```

where `*` is the file extension. ROOT will automatically set the type of file being written based on the extension you select. ROOT can save canvases to many different file types, including `.pdf`, `.ps`, `.eps`, `.svg`, `.png` and `.jpg`. You can also save your canvas using the `File->Save As` menu, or you can send it directly to a printer using `File->Print`. Histograms points can also be written out to an ASCII file, using the command:

```
hstname->Print("all"); > outputfilename.txt
```

This will write out the points of the histogram into a text file called
`outputfilename.txt`. The formatting of the output file is a bit clumsy; there is a way to write out histograms into a plain two column format, but it requires the use of custom commands; hence it will be explained later on.

6 Gates

The ability to set "gates" (logical restrictions) on parameters is needed to do any sort of nuclear physics analysis. Gates in ROOT are treated as logical C++ statements. Some of the more commonly used logical statements are: "greater than" (>), "less than" (<), "equal to" (==), "not equal to" (`!=`), AND (`&&`), and OR (`||`). Any combination of these logical statements can be used. The gate command is the second argument of the `Draw()` command and is surrounded by `" "`. As an example, if I want to view a histogram of the parameter `crdc1.x`, with the restriction that another parameter, `crdc1.y`, is greater than 0, I would use the following command:

```
t->Draw("crdc1.x", "crdc1.y > 0");
```

As a more complicated example, if I wanted to see a histogram of `crdc1.x` subject to the following conditions:

–crdc1.y is greater than 0.
–tof.pot_thin is less than 50.
–crdc1.x is not equal to -1.,

then I would type the following:

```
t->Draw("crdc1.x", "crdc1.y > 0 && tof.pot_thin < 50 && crdc1.x !=-1");
```

In additional to one dimensional gates, ROOT can also create two dimensional gates which constrain the values of two parameters to lie within a certain shape. Two dimensional gates are given a name, and can be used just like any other logical statement. For example, if I want to apply a 2d gate called "my2dgate" to a histogram containing the parameter `crdc1.x`, I would do:

```
t->Draw("crdc1.x","my2dgate");
```

Two dimensional gates can be combined with other logical statements as well, so if I wanted to draw `crdc1.x` subject to both "my2dgate" and the requirement that `crdc1.tx` be greater than zero, then I would do:

```
t->Draw("crdc1.x","my2dgate && crdc1.tx > 0");
```

The easiest way to set a 2d gate is to make use of the `TCanvas` GUI. The steps to do this are outlined below:

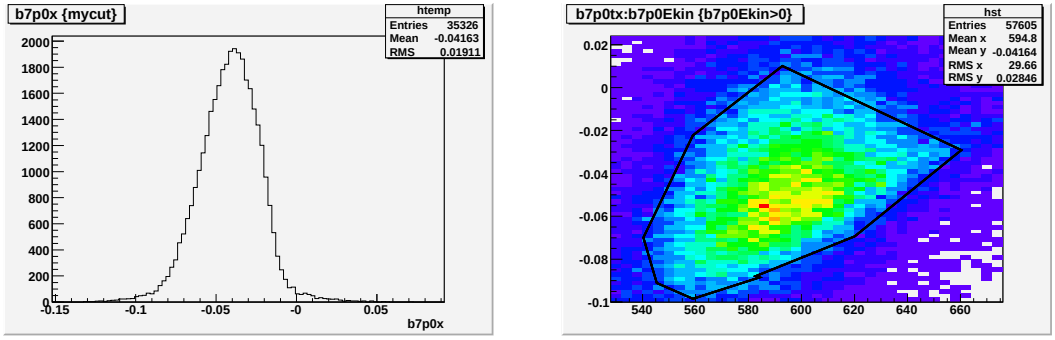
1. Draw the 2d histogram.
2. In the `TCanvas` containing the new histogram, select `View->Toolbar`.
3. In the toolbar, click on the cut tool. It has a picture of scissors on it and is the farthest right tool on the toolbar.
4. Go onto the histogram and start drawing your gate. Single clicking sets a new point in the contour, and double clicking closes the gate.
5. Move the mouse cursor over the gate (the cursor will turn into a hand with a pointing index finger when you are in the right place), and right click. Then select the `SetName` option, and type your desired name of the gate in the pop-up window.
6. If you would like to modify your gate, you can do so by left clicking on one of the gate points and moving it around.

Figure 3 shows an example of a 2d gate created using the procedure outlined above. Unlike in SpecTcl, your 2d gate will not automatically be drawn every time that you re-draw a histogram, but the gate is still stored into memory until you quit the program. To re-draw a gate the command is:

```
mygate->Draw();
```

It is often the case that you forget exactly which gates you have defined or what you have decided to name them. To avoid this, if you have loaded the `root_logon.C` macro (as outlined in Section 8) then you can use the command:

```
list_cuts();
```



(a) Spectrum showing a parameter with the gate drawn in panel (b) applied to it.

(b) Example of a two dimensional gate. The events outlined in black are included within the gate.

Figure 3: Example 2d Gate (right panel) and histogram with the gate applied to it (left panel).

to see all of the cuts currently in your ROOT session.

You will likely find yourself typing the same gate conditions over and over, which can get tedious. You can store these logical statements in a `TCut` object, and when you do so, the gate conditions can be accessed just by invoking the name of the `TCut`. The syntax to create a TCut is explained with an example: if I wanted to create a `TCut`, named `mycut`, which requires that `crdc1.x` and `crdc1.y` both be greater than zero, I would type the following:

```
TCut mycut = "crdc1.x > 0 && crdc1.y > 0" ;
```

The `TCut` can then be used like a normal logical statement, e.g.

```
t->Draw("crdc1.x", mycut);
```

Note that in the above command there are no `" "` around `mycut`. This is because the needed `" "` are contained in the definition of `mycut`. `TCuts` can also be combined with "ordinary" gates or other `TCuts` as follows:

```
t->Draw("crdc1.x",mycut && "crdc2.x > 0"); //\ combine a TCut with a "normal" gate.
t->Draw("crdc1.x",mycut && mycut2); //\ combine two TCuts.
```

7 Creating Pseudo Parameters

Quite often one wants to look at mathematical combinations of the parameters stored in a ROOT file. The simplest way to do this is to simply include the mathematical statements in the argument of the `Draw()` command. For example, if I want to look at the velocity of a particle, which for the purposes of this example is equal to `505.2 / tof.pot_thin`, then I would type the following:

```
t->Draw("505.2 / tof.pot_thin");
```

Any valid C++ math operator can be used; some of the more common examples are listed below

– add, subtract, multiply, divide: `+, -, *, /`
– square root of x: `sqrt(x)`
– raise x to the power n: `pow(x,n)`
– sin, cos, tan of x: `sin(x), cos(x), tan(x)`
– arcsin, arccos, arctan of x: `asin(x), acos(x), atan(x)`
– absolute value of x: `abs(x)`

As in the case of gates, you often do not want to type long mathematical expressions over and over again. Fortunately, ROOT allows you to store your mathematical expression in an object called an "alias." For example, if I want to store the velocity expression shown above as an alias called `frag_vel`, I would do:

```
t->SetAlias("frag_vel","505.2 / tof.pot_thin");
```

Now to draw the velocity I just have to type:

```
t->Draw("frag_vel")
```

Aliases can also be included in the definition of other aliases; for example, if I want to create an alias of 2*velocity, called `frag_velx2`, then I could do:

```
t->SetAlias("frag_velx2","2 * frag_vel");
```

Aliases can be used just like any "real" ROOT parameter, e.g. you can draw them, place gates on them (1d and 2d), and so on.

8 Using Macros

If you haven't noticed by now, using ROOT involves a lot of typing of commands, and you probably don't want to repeat a bunch of commands over and over again. ROOT allows you to save series of commands in a macro file; invoking a macro file has the same effect as typing each one of the commands on the command line individually. ROOT macro files should always have the extension `.C`, and all of the commands must be enclosed in a set of curly brackets. The command to run a macro (called e.g. `mymacro.C`) is:

```
.x mymacro.C
```

As a simple example consider the following macro, called `examplemacro.C`; it is located in the `/projects/MoNA/ROOT/examples` directory, and the code is reproduced here. Comments in the code are set off by `//` (the comment symbol in C++), and should explain to you what each line is doing.

```
{

 //Macro File examplemacro.C

 TFile* _file0 = new TFile("/projects/MoNA/ROOT/examples/example_cal.root");
 //Load the file example_cal.root into memory

 TCut posenergy = "sweeper.e > 0 && mona.hit.ke[0] > 0" ;
 //Create a TCut

 t->Draw("focus.tx>>hst(100,-100,100)",posenergy);
 //draw a histogram of focus.tx, subject to the conditions of posenergy

}
```

You can also invoke other macros from within a macro; for example, if I want to call a macrofile called `macro2.C` from within another macro, I would add the line:

```
gROOT->ProcessLine(".x macro2.C");
```

You can also load in a macro file when you start up ROOT, by typing the name of the macro after the `root.exe` command. For example, if I want to automatically load the file `examplemacro.C` at startup, then the syntax is:

```
> root.exe examplemacro.C
```

There is a useful "startup" macro file located in `/projects/MoNA/ROOT/`; it is called `root_logon.C` It does a few formatting tasks like setting to default color palette to the nice looking RGB one. It also loads up a custom function that allows you to write out a histogram to a two column ASCII file. If you have loaded up `root_logon.C`, then the syntax to write a histogram to file is:

```
hstname->WriteSpec("filename.txt",xlo, xhi);
```

where `xlo` and `xhi` are lower and upper limits of the histogram.

9 Various Odds and Ends

This section contains a brief overview of some more useful commands that don't really fit anywhere else. They are listed with a brief description of what the command does followed by the command syntax.

• Draw a histogram using data points:

```
t->Draw("7p0x>>hst(100,-.1,.1)","","p");
```

This would draw `hst` with the default point size, which is very small; to change it do:

```
hst->SetMarkerStyle(20); //sets a reasonable point size
hst->Draw("p"); // redraws "hst" using points
```

• Draw a histogram with error bars (calculated as the square root of the number of counts):

```
t->Draw("7p0x>>hstnew(100,-.1,.1)","","e"); //new histogram
hstold->Draw("e"); //existing histogram
```

• Change the color of the lines or points of a histogram already in memory:

```
hst->SetLineColor(4);   //Lines
hst->SetMarkerColor(4); //Points
```

Colors in ROOT are assigned a number, so here I am setting the histograms to be "color 4" (which is blue). To see a list of all available colors and their corresponding numbers, select `View->Colors` from the `TCanvas` menu.

• Scale a histogram (by the number x):

```
hst->Scale(x);
```

• Draw an existing histogram on the same canvas, without erasing the one already there:

```
hst->Draw("same");
```

• You can also combine arguments like `p` and `same`, e.g.

```
hst->Draw("psame"); //Draw hst on the same canvas, using points.
```

• Clear your current canvas:

```
c1->Clear();
```

• Save 2D gates into a macro file (if you have loaded up `root_logon.C` macro):

```
gate->save("filename.C");
// save a single gate into a macro file

save_cuts("filename.C", "cut1", "cut2", "cut3",...);
// save multiple gates (up to 30) in the same macro file
```

The macro file `filename.C` will now contain C++ code describing the 2D gate(s) you told it to include. You can load it into your ROOT session as described in Section 8 to have access to the gates.

• Convert a SpecTcl filter file into a ROOT file:
–Run the executable `filter2root` located in `/projects/MoNA/ROOT/filter2root/`:

```
> filter2root myfilter.flt
```

This will output a new file `myfilter.root` which will contain a `TTree` with all of the parameters present in the filer. For a given event, any parameters which are not set in the filter will be put as equal to -1 in the ROOT file.

If you are interested in using the more advanced functionality of ROOT, there are many resources on the web. A good place to start is the official ROOT manual located at:

`http://root.cern.ch`

There is also an online bulletin board ("Root Talk") dedicated to helping users learn to use ROOT located at:

`http://root.cern.ch/phpBB2/`