Basic Data Analysis Using ROOT

A guide to this tutorial

If you see a command in this tutorial is preceded by "[]", it means that it is a ROOT command. You should type that command into the ROOT program as appropriate, *without* the "[]" symbols. For example, if you see

```
[] .x treeviewer.C
```

it means to type ".x treeviewer.C" at a ROOT command prompt.

If you see a command in this tutorial is preceded by ">", it means that it is a UNIX command. You should type that command into UNIX, *not* into ROOT, and without the ">" symbol. For example, if you see

```
> man less
```

it means to type "man less" at a UNIX command prompt.

Most of the lessons have time estimates at the top. These are only rough estimates; some students take 20 minutes to go through a lesson labeled "15 minutes", others take only 5. Don't be too concerned about time; the important thing is for you to learn something, not to punch a time clock.

You can find this tutorial in Postscript and PDF format (along with links to the sample files) at <http://www.nevis.columbia.edu/~seligman/root-class/>.

Getting started on Linux

If you're sitting in front of a computer running Linux, just use your account name and password to login. After you log in, if your screen is just showing text (without any graphics at all), then type "startx" to run X-windows.

Click **once** on the Browser icon (at the top or bottom of the screen) to start Mozilla or Firefox. (This looks like a sphere with a mouse around it.)

Type the following URL in the "Location" field of the web browser: http://root.cern.ch/. This is the ROOT web site. You'll be coming back here often, so be sure to bookmark this site.

You'll probably want to open a terminal session to do your work. The menu path is:

Applications -> System Tools -> Terminal

You may find it convenient to add the Terminal application icon to the menu bar. You can do this by right-clicking on the menu bar... and then I think you can figure the rest for yourself.

For those familiar with Linux-based window managers: No, I don't care whether you use GNOME or KDE. Feel free to switch.

A Brief Intro to Linux

If you're already reasonably familiar with Linux or UNIX in general, skip this section.

You can spend a lifetime learning Linux; I've been working with UNIX since 1993 and I'm still learning something new every day. The commands below barely scratch the surface.

To copy a file: use the "cp" command.

For example, to copy the file "example.C" from the directory "~seligman/root-class" to your current working directory, type:

```
> cp ~seligman/root-class/example.C $PWD
```

In UNIX, the variable $PWD means the results of the "print working directory" command. (I know that a period (.) is the more usual abbreviation, but many students kept missing the period the first time I taught this class.)

To look at the contents of a text file: Use the **less** command.

This command is handy if you want to quickly look at a file without editing it. If the name of the command seems puzzling, it may help to know the **more** command also displays the contents of a text file, and the **less** command was created as more powerful version of **more**. So to quickly look at the contents of file example.C, type:

```
> less example.C
```

While **less** is running, type a space to go forward one screen, type "b" to go backward one screen, type "q" to quit, and type "h" for a complete list of commands you can use.

To get help on any UNIX command: type "man <command-name>".

While **man** is running, you can use the same navigation commands as **less**. For example, to learn about the **ls** command, type:

```
> man ls
```

To edit a file: use the **emacs** command. (If you're already familiar with another editor, such as **pico** or **vi**, you can use it instead.)

You will almost aways want to add an ampersand (&) to the end of any "emacs" command; the ampersand means to run the command as a separate process. So to edit the file example.C, type:

```
> emacs example.C &
```

The "emacs" environment is complex, and you can spend a lifetime learning it.[•] For now, just use the mouse to move the cursor and look at the menus. As soon as you can (probably not during this class), take the Emacs tutorial by selecting it under the "Help" menu.

Try to learn how to cut and paste in whatever editor you use. If you don't, you'll waste a lot of time today typing the same things over and over again.

• Already I've spent two of your lifetimes, and the class has just started!

Setting up ROOT (5 minutes)

ROOT is a robust, complex environment for performining physics analysis, and you can spend a lifetime learning it.• Before you start using ROOT at Nevis, you have to type the following command:

```
> setup root
```

The command **setup root** sets some Unix environment variables and modifies your command and library paths. If you feel a need to remove these changes, use the command **unsetup root**.

One of the variables that is set is $ROOTSYS. This will be helpful to you if you're following one of the examples in the ROOT Users Guide. For example, if you're told to find a file in $ROOTSYS/tutorials, you'll be able to do this only after you've typed "setup root400".

You have to execute the **setup root** command only once, but you must do it each time you login to Linux. If you wish this command to be automatically executed when you login, you can add it to the .myprofile file in your home directory.

You are going to need to have at least two windows open during this class. One window I'll call your "ROOT command" window; this is where you'll run ROOT. The other is a separate "UNIX command" window. On Unix, you can create a second window with the following command; don't forget the ampersand (&):

```
> xterm &
```

You can also just run the Terminal application again, or select "New..." from the File menu of a running Terminal application.

• Yes, that's three lifetimes so far...

Starting ROOT (5 minutes)

To actually run ROOT, just type:

```
> root
```

The window in which you type this command will become your ROOT command window.

First you'll see the white-and-blue ROOT window appear on your screen. It will then disappear, and a brief "Welcome to ROOT" display will be written on your command window.

If you grow tired of the introductory graphics window, type "root -l" instead of "root" to start the program.

Click on the ROOT window to select it, if necessary.

You can type "?" (or ".h") to see a list of ROOT commands... but you'll probably get more information than you can use right now. Try it and see.

The most important ROOT line command you need to know is how to quit ROOT. To exit ROOT, type ".q". Do this now, then start ROOT again, just to make sure you can do it.

Plotting a function (15 minutes)

> This example is based on the first example in Chapter 2 of the ROOT Users Guide (page 11). I emphasize different aspects of ROOT than the Users Guide, and it's a good idea to go over both the example in the Guide and the one below.

Let's plot a simple function. Start ROOT and type the following at the prompt:

```
[] TF1 f1("func1","sin(x)/x",0,10)
[] f1.Draw()
```

> Note the use of C++ syntax to invoke ROOT commands.
>
> If you have a keen memory (or you type ".h" on the ROOT command line), you'll notice that neither TF1 nor any of its methods are listed as commands, nor will you find a detailed description of TF1 in the Users Guide. The only place that the complete ROOT functionality is documented is on the ROOT web site.

Go to the ROOT web site at <http://root.cern.ch/> (did you remember to bookmark this site?), click on "Reference Guide", then on "The ROOT Class Categories", then on "Histogram", and finally on "TF1". Scroll down the page; you'll see the class methods, then class and method descriptions.

> Get to know your way around this web site. You'll be coming back often.
>
> Also note that when you executed "f1.Draw()", ROOT created a canvas for you named "c1". "Canvas" is ROOT's term for a window that contains ROOT graphics; everything ROOT draws must be inside a canvas.

Bring the window named "c1" to the front by left-clicking on it. As you move the mouse over different parts of the drawing (the function, the axes, the graph label, the plot edges) note how the shape of the mouse changes. Right-click the mouse on different parts of the graph and see how the pop-up menu changes.

Position the mouse over the function itself (it will turn into an arrow). Right-click the mouse and select "SetRange". Set the range to xmin=-10, xmax=10, and click "OK". Observe how the graph changes.

Let's start getting into a good habit by labeling our axes. Right-click on the x-axis of the plot, select "SetTitle", enter "x [radians]", and click "OK". Let's center that title: right-click on the x-axis again, select "CenterTitle", and click "OK".

> Note that clicking on the title gives you a "TCanvas" pop-up, not a text pop-up; it's as if the title wasn't there. Only if you right-click on the axis can you affect the title. In object-oriented terms, the title and its centering are a property of the axis.
>
> It's a good practice to always label the axes of your plots. Don't forget to include the units.

Do the same thing with the y-axis; call it "sin(x)/x". Select the "RotateTitle" property of the y-axis and see what happens.

Plotting a function (continued) (10 minutes)

You can zoom in on an axis interactively. Left-click on top of the number "2" on the x-axis, and drag to the number "4". The graph will expand its view. You can zoom in as much as you like. When you've finished, right-click on the axis and select "UnZoom."

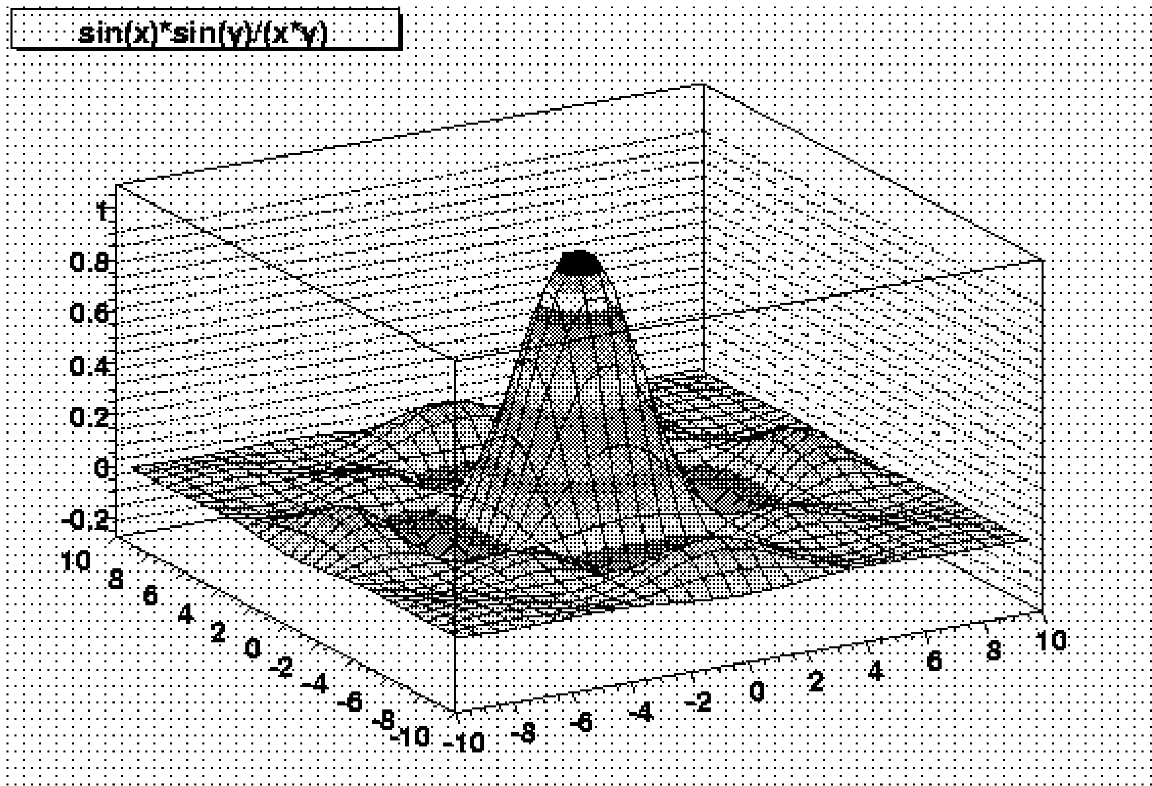
You have a lot of control over how this plot is displayed. From the "View" menu, select "Editor". Play around with this a bit. Click on different parts of the graph; notice how the options automatically change. Select "View->Toolbar"; among other options, you can see how you can draw more objects on the plot. There's no simple "Undo" command, as there might be in a dedicated graphics program, but you can usually right-click on an object and select "Delete" from the pop-up menu.

Select "Style" from the "Edit" menu. Select some different styles and hit "Apply". This can be handy if you discover a "look and feel" that you like for your plots.

> If you "ruin" your plot, you can always quit ROOT and start it again. We're not going to work with this plot in the future anyway.

Exercise 1 (10 minutes)

Try to duplicate the following plot:



Hints:

Look at the TF1 command on the last page. If class TF1 will generate a one-dimensional function, what class might generate a two-dimensional function?

If TF1 takes a function name, formula, and x-limits in its constructor, what arguments might a two-dimensional function class use?

You probably figured out how to draw something, but you got a contour plot, not a surface plot. Here's another hint: you want the give the option "surf1" to the draw command.

Don't forget that the ROOT web site is available as a resource. See you if you can trace through the class links to find the list of options that you can apply to draw a surface.

Working with Histograms (15 minutes)

Histograms are described in Chapter 3 of the ROOT Users Guide. You may want to look over this chapter later to get an idea of what else can be done with histograms other than what I cover in this class.

Let's create a simple histogram:

```
[] TH1F h1("hist1","Histogram from a gaussian",100,-3,3)
```

Let's look at what these arguments mean for a moment (and you should also look at the description of TH1F on the ROOT web site). The name of the histogram is "hist1". The title displayed when plotting the histogram is "Histogram from a gaussian". There are 100 bins in the histogram. The limits of the histogram are from -3 to 3.

Question: What should be the width of one bin of this histogram? Type the following to see if your answer is the same as ROOT thinks it is:

```
[] h1.GetBinWidth(0)
```

Note that we have to indicate which bin's width we want (bin 0 in this case), because you can define histograms with varying bin widths.

If you type

```
[] h1.Draw()
```

right now, you won't see much. That's because the histogram is empty. Let's randomly generate 10,000 values according to a distribution and fill the histogram with them:

```
[] h1.FillRandom("gaus",10000)
[] h1.Draw()
```

The "gaus" function is pre-defined by ROOT (see the TFormula class on the ROOT web site; there's also more on the next page of this tutorial). The default Gaussian distribution has a width of 1 and a mean of zero.

Note the histogram statistics in the top right-hand corner of the plot. Question (for those who've had statistics): Why isn't the mean exactly 0, or the width exactly 1?

Add another 10,000 events to histogram h1 with the FillRandom method (hit the up-arrows until you see "h1.FillRandom("gaus",10000)" again, and hit return). Click on the canvas. Note how the histogram updates immediately, without another "Draw" command.

Let's put some error bars on the histogram. Select "View->Editor", then click on the histogram. from the "Error" pop-up menu, select "Simple". Try click on the "Simple Drawing" box and see how the plot changes.

The size of the error bars is equal to the square root of the number of events in that histogram bin. With the up-arrow key in the ROOT command window, execute the FillRandom method a few more times; click on the canvas again. Question: Why do the error bars get smaller? Hint: Look at how the y-axis changes.

You will often want to draw histograms with error bars. For future reference, you could have used the following command instead of the Editor:

```
[] h1.Draw("e")
```

Working with Histograms (continued) (10 minutes)

Let's create a function of our own:

```
[] TF1 myfunc("myfunc","gaus",0,3)
```

The "gaus" (or gaussian) function is actually

$$P_0 e^{-0.5\left(\frac{(x-P_1)}{P_2}\right)^2}$$

where P_0, P_1, and P_2 are "parameters" of the function.• Let's set these three parameters to values that we choose, draw the result, then create a new histogram from our function:

```
[] myfunc.SetParameters(10.,1.0,0.5)
[] myfunc.Draw()
[] TH1F h2("hist2","Histogram from my function",100,-3,3)
[] h2.FillRandom("myfunc",10000)
[] h2.Draw()
```

Note that we could also set the function's parameters individually:

```
[] myfunc.SetParameter(1,-1.0)
[] h2.FillRandom("myfunc",10000)
```

What is the difference between the command 'SetParameters' and 'SetParameter'? If you have any doubts, check the description of class TF1 on the ROOT web site.

· Optional note for advanced users:

In ROOT's TFormula notation, this would be "[0]*exp(-0.5*((x-[1])/[2])^2)"; "[n]" corresponds to P_n. I mention this so that when you become more experienced with defining your own parameterized functions, you can use a different formula:

```
[] TF1 myGaus("user",
   "[0]*exp(-.5*((x-[1])/[2])^2)/([2]*sqrt(2.*pi))")
```

This may seem cryptic to you now. Actually, it's just a gaussian distribution with a different normalization so that P_0 divided by the bin width becomes the number of events in the histogram:

```
[] myGaus.SetParameters(10.,0.,1.)
[] hist.Fit("user")
[] Double_t numberEquivalentEvents =
   myGaus.GetParameter(0) / hist.GetBinWidth(0)
```

Working with multiple plots (optional) (5 minutes)

If you're running short on time, you can skip this page (or any of the other optional pages).

We have a lot of different histograms and functions now, but we're plotting them all on the same canvas, so we can't see more than one at a time. There are two ways to get around this.

First, we can create a new canvas by selecting "New Canvas" from the File menu of our existing canvas; this will create a new canvas with a name like "c1_n2". Try this now.

Second, we can divide a canvas into "pads". On the new canvas, right-click in the middle and select "Divide". Enter nx=2, ny=3, and click "OK".

Click on the different pads and canvases with the **middle** button (if you have a mouse with a scroll wheel, the wheel is "clickable" and serves as the middle button). Observe how the yellow highlight moves from box to box. The "target" of the Draw() method will be the highlighted box. Try it: select one pad with the middle button, then enter

```
[] h2.Draw()
```

Select another pad or canvas with the middle button, and type:

```
[] myfunc.Draw()
```

At this point you may wish that you had a bigger monitor!

Saving and printing your work (15 minutes)

By now you've probably noticed the "Save" sub-menu under the "File" menu on the canvas. There are many file formats listed here, but we're only going to use three of them for this tutorial.

Select "Save->*canvas-name*.C" from one of the canvases in your ROOT session. Let's assume for the moment that the file "c1.C" is created. In your UNIX window, type

```
> less c1.C
```

(If you get complaints about a file not found, the name of the canvas is "cee-one", not "cee-ell.") As you can see, this can be an interesting way to learn more ROOT commands. However, it doesn't record the procedure you went through to create your plots, only the minimal commands necessary to display them.

Next, select "Save->c1.pdf" from the same canvas; we'll print it later.

Finally, select "Save->c1.root" from the same canvas. Let's assume the file is named "c1.root". Now quit ROOT with the ".q" command, and start it again.

To re-create your canvas from the ".C" file, use the command

```
[] .x c1.C
```

This is your first experience with a ROOT "macro", a stored sequence of ROOT commands that you can execute at a later time. One advantage of the ".C" method is that you can edit the macro file, or cut-and-paste useful command sequences into macro files of your own.

Quit ROOT and print out your Postscript file with the command

```
> lpr -Pbw-research c1.pdf
```

This may be point at which you'll notice that the default background color for ROOT plots is not pure white. You can change the background by right-clicking on a canvas and selecting "SetFillAttributes"; you'll have to do this in the regions both outside and inside the plot. If you don't want to keep doing this with all your plot, you may want to experiment more with "Edit->Style".

If you want to print directly from the canvas using "File->Print", then type
`lpr -Pbw-research`
in the first text box and leave the second one empty.

The ROOT browser (5 minutes)

> The ROOT browser is a useful tool, and you may find yourself creating one at every ROOT session. Read pages 22-23 of the ROOT Users Guide to find out how to make ROOT start a new browser automatically each time you start ROOT.

One way to retrieve the contents of file "c1.root" is to use the ROOT browser. Start up ROOT and create a browser with the command:

```
[] TBrowser tb
```

> If you happen to have a canvas already displayed, you can also start a browser with the menu item "Inspect->Start Browser".

In the right-hand pane, double-click on the folder with the same name as your home directory. Scroll through the list of files. You'll notice special icons for any files that end in ".C" or ".root". If you double-click on a file that ends in ".C", ROOT will assume the file contains a ROOT macro and interpret the contents. Try this on "c1.C", then close the canvas window.

Now double-click on "c1.root". Nothing will appear to change. Now click on the "ROOT Files" folder in the left-hand pane; this is the list of files currently opened by ROOT. Double-click on "c1.root" in the right-hand pane, then double-click on "c1;1".

> What does "c1;1" mean? You're allowed to write more than one object with the same name to a ROOT file (this topic is part of an optional lesson later in this tutorial). The first object has ";1" put after its name, the second ";2", and so on. You can use this facility to keep many versions of a histogram in a file, and be able to refer back to any previous version.
>
> At this point, saving a canvas as a ".C" file or as a ".root" file may look the same to you. But these files can do more than save and re-create canvases. In general, a ".C" file will contain ROOT commands and functions that you'll write yourself; ".root" files will contain complex objects such as n-tuples.

Fitting a histogram (15 minutes)

I created a file with a couple of histograms in it for you to play with. Switch to your UNIX window and copy this file into your directory:

```
> cp ~seligman/root-class/histogram.root $PWD
```

Go back to your browser window. (If you've quit ROOT, just start it again and start a new browser.) Click on the folder in the left-hand pane with the same name as your home directory. If you don't see "histogram.root", select "Refresh" from the "View" menu.

Double-click on "histogram.root", click on "ROOT Files" in the left-hand pane, then double-click on "histogram.root" in the right-hand pane. You can see that I've created two histograms with the names "hist1" and "hist2". Double-click on "hist1".

> You can guess from the x-axis label that I created this histogram from a gaussian distribution, but what were the parameters? In physics, to answer this question we typically perform a "fit" on the histogram: you assume a functional form that depends on one or more parameters, and then try to find the value of those parameters that make the function best fit the histogram.

Right-click on the histogram and select "FitPanel". Click on "gaus", then click on "Fit". You'll see two changes: A function is drawn on top of the histogram, and the fit results are printed on the ROOT command window.

> Interpreting fit results takes a bit of practice. Recall that a gaussian has 3 parameters (P_0, P_1, and P_2); these are labeled "Constant", "Mean", and "Sigma" on the fit output. ROOT determined that the best value for the "Mean" was 5.96±0.03, and the best value for the "Sigma" was 2.47±0.02. Compare this with the Mean and RMS printed in the box on the upper right-hand corner of the histogram. Statistics questions: Why are these values almost the same as the results from the fit? Why aren't they identical?

On the canvas, select "Fit Parameters" from the "Options" menu. Click on "Fit" on the FitPanel again. Just as a check, click on "landau" on the FitPanel and click on "Fit" again; then click on "expo" and fit again.

> It looks like of the three choices (gaussian, landau, exponential), the gaussian is the best functional form for this histogram. Take a look at the "Chi2 / ndf" value in the statistics box on the histogram ("Chi2 / ndf" is pronounced "kie-squared per degrees of freedom"). Do the fits again, and observe how this number changes. Typically, you know you have a good fit if this ratio is about 1.

Fitting a histogram (continued) (15 minutes)

Go back to the browser window and double-click on "hist2". Uh-oh; this doesn't look like a gaussian. Right-click on the histogram and select "FitPanel" (be careful not to re-use the FitPanel from "hist1"). Click on "gaus" and then on "Fit". Uggh -- that's a terrible fit! You can try the landau and exponential functions, but they won't work much better.

> You've probably already guessed by reading the x-axis label that I created this histogram from the sum of two gaussian distributions. But you've probably also noticed a button on the FitPanel labeled "user"; this is for fitting to a user-defined function.
>
> In order to use this button, you have to define a function that has the name "user".

Define a user function with the following command:

```
[] TF1 func("user","gaus(0)+gaus(3)")
```

> Note that the internal ROOT name of the function has to be "user", but not the function object itself.
>
> What does "gaus(0)+gaus(3)" mean? You already know that the "gaus" function uses three parameters. "gaus(0)" means to use the gaussian distribution starting with parameter 0; "gaus(3)" means to use the gaussian distribution starting with parameter 3. This means our user function has six parameters: P_0, P_1, and P_2 are the "constant", "mean", and "sigma" of the first gaussian, and P_3, P_4, and P_5 are the "constant", "mean", and "sigma" of the second gaussian.

Now try to do a fit by going to the FitPanel, clicking on "user", then clicking on "Fit".

> If you look at the ROOT command window, you'll see that all six parameters have the values like zero or "nan", which means "Not A Number." For all but the simplest fits, ROOT needs to have some starting values for its fit parameters.

Let's set the values of P_0, P_1, P_2, P_3, P_4, and P_5:

```
[] func.SetParameters(5.,5.,1.,1.,10.,1.)
```

Then click on "Fit" on the FitPanel.

> The results are not much better. This is because I deliberately picked a poor set of starting values. Let's try a better set:

```
[] func.SetParameters(5.,2.,1.,1.,10.,1.)
```

These starting values don't look much different, but try to fit the histogram again.

> These simple fit examples may leave you with the impression that all histograms in physics are fit with gaussian distributions. Nothing could be further from the truth. I'm using gaussians in this class because they have properties (mean and width) that you can determine by eye.
>
> Chapter 5 of the ROOT Users Guide has a lot more information on fitting histograms, and a much more realistic example.
>
> If you want to see how I created the file histogram.root, go to the UNIX window and type:

```
> less ~seligman/root-class/CreateHist.C
```

Saving your work, part 2 (optional) (15 minutes)

So now you've got a histogram fitted to a complicated function. Try "Save->c1.C" from the "File" menu, and examine the result in your UNIX window with

```
> less c1.C
```

I know it looks like a bunch of C++ commands. But if you look carefully, you'll see that the user function you just created is **not** re-created in the macro.

> If you were to use "Save as canvas.root", quit ROOT, restart it, then load canvas "c1;1" from the file, you'd get your histogram back with the function superimposed... but it's not obvious where the function is or how to access it now.
>
> What if you want to save your work in the same file as the histograms you just read in? You can do it, but not by using the ROOT browser. The browser will only open files in read-only mode. To be able to modify a file, you have to open it with ROOT commands.

Try the following: Quit ROOT (note that you can select "Quit ROOT" from the "File" menu of the canvas or the browser). Start ROOT again, then modify "histogram.root" with the following commands:

```
[] TFile file1("histogram.root","UPDATE")
```

It is the "UPDATE" option that will allow you to write new objects to "histogram.root".

```
[] hist2.Draw()
```

For the following two commands, try hitting the up-arrow key until you see them again. ROOT stores the last 80 or so ROOT commands you've typed in the file ".root-hist" in your home directory, and let's you re-use them with the arrow keys.

```
[] TF1 func("user","gaus(0)+gaus(3)")
[] func.SetParameters(5.,2.,1.,1.,10.,1.)
```

The following command is the same as clicking on "user" and "Fit" on the FitPanel:

```
[] hist2.Fit("user")
```

Now you can do what you couldn't before: save objects into the ROOT file:

```
[] hist2.Write()
[] func.Write()
```

You should close the file to make sure you save your changes:

```
[] file1.Close()
```

Quit ROOT, start it again, and use the ROOT browser to open "histogram.root". You'll see a couple of new objects: "hist2;2" and "user;1". Double-click on each of them to see what you've saved.

Chapter 11 of the ROOT Users Guide has more information on using ROOT files.

Dealing with PAW files (optional) (5 minutes)

Before ROOT, physicists used a package called CERNLIB to analyze data. You won't be asked to work with CERNLIB while you work at Nevis (at least, I hope not), but it may be that you'll be asked to read a file created by this old program library.

Suppose someone gives you a file that contains n-tuples or histograms, and tells you that the file was created with PAW, HBOOK, or CERNLIB (actually, to first order these are three different names for the same thing). How do you read these files using ROOT?

The answer is that you can't, at least not directly. You must convert these files into ROOT format using the command "h2root".

For example, if someone gives you a file called "testbeam.hbook", you can convert it with the command

```
> h2root testbeam.hbook
```

This creates a file "testbeam.root" that you can open in the ROOT browser.

There is no simple way of converting a ROOT file back into PAW/HBOOK/CERNLIB format. You generally have to write a custom program with both FORTRAN and C++ subroutines to accomplish this task.

Note that the "h2root" command is set up (along with ROOT) with the command

```
> setup root
```

that you type when you log in. If you accidentally type "h2root" (or "root") before you set up ROOT, you'll get the error message:

```
h2root: Command not found
```

You can get more information about "h2root" by using a special form of the "man" command:

```
> man $ROOTSYS/man/h2root.1
```

There's also information on pages 22-23 of the ROOT Users Guide.

Accessing variables in ROOT NTuples/Trees (10 minutes)

I've created a sample ROOT n-tuple in ~seligman/root-class/experiment.root.

Start fresh by quitting ROOT. Copy my 2.1 MB example Tree file with the command

```
> cp ~seligman/root-class/experiment.root $PWD
```

Start ROOT again. Start a new browser with the command

```
[] TBrowser b
```

Click on the folder in the left-hand pane with the same name as your home directory. Double-click on "experiment.root", click on "ROOT Files" in the left-hand pane, then double-click on "experiment.root" in the right-hand pane. There's just one object inside: "tree1", a ROOT Tree (or n-tuple) with 100,000 simulated physics events.

Actually, there's little or no real physics associated with the contents of this tree. I created it solely to illustrate ROOT concepts, not to demonstrate real physics with a real detector.

Right-click on the "tree1" icon, and select "Scan". You'll be presented with a dialog box; just hit "OK" for now. Select your ROOT window, even though the dialog box didn't go away. At first you'll probably just notice that it's a lot of numbers. Take a look at near the top of the screen; you should see the names of the variables in this ROOT Tree.

In this overly-simple example, an imaginary particle is travelling in a positive direction along the z-axis with energy "ebeam". It hits a target at z=0, and travels a distance "zv" before it is deflected by the material of the target. The particle's new trajectory is represented by "px", "py", and "pz", the final momenta in the x-, y-, and z-directions respectively. The variable "chi2" represents a confidence level in the measurement of the particle's momentum.

Did you notice what's missing from the above description? One important omission is the units; for example, I didn't tell you whether "zv" is in millimeters, centimeters, inches, yards, etc. Such information is not usually stored inside an n-tuple; you have to find out what it is and include the units in the labels of the plots you create. For this example, assume that "zv" is in centimeters (cm), and all energies and momenta are in *GeV*.

You can hit "return" to see more numbers, but you probably won't learn much. Hit "q" to finish the scan. You'll have to hit "return" a couple of times to see the ROOT prompt again.

Simple analysis using the Draw command (10 minutes)

The title of this section is almost the same as a corresponding section in the ROOT Users Guide, beginning on page 240. I'm just going to bring up a few basic tricks before going into the topic of using C++.

It may be that all the analysis tasks that your supervisor will ask you to do can be performed using the Draw command, the TreeViewer (described near the end of this tutorial), and other simple techniques discussed in the ROOT Users Guide.

However, it's more likely that these simple commands will only be useful when you get started, For example, you can draw a histogram of just one variable to see what the histogram limits might be in C++.

If you don't already have the sample ROOT Tree file open, open it with the following command:

```
[] TFile myFile("experiment.root")
```

You can use the Scan command to look at the contents of the Tree, instead of using the TBrowser as described on the previous page:

```
[] tree1->Scan()
```

If you take a moment to think about it (a habit I strongly encourage), you may ask how ROOT knows that there's a variable named "tree1", when you didn't type in a command to create it.

The answer is that when you read a file containing ROOT objects (see "Saving your work, part 2" above) in an interactive ROOT session, ROOT automatically looks at the objects in the file and creates variables with the same name as the objects.

This is *not* standard behavior in C++; it isn't even standard behavior when you're working with ROOT macros. Don't become too used to it!

You can also display the Tree in a different way that doesn't show the data, but displays the names of the variables and the size of the Tree:

```
[] tree1->Print()
```

Either way, you can see that the variables stored in the Tree are "event", "ebeam", "px", "py", "pz", "zv", and "chi2".

Create a histogram of one of the variables. For example:

```
[] tree1->Draw("ebeam")
```

Using the Draw command, make histograms of the other variables.

By the way, the variable "event" is just the event number (it's 0 for the first event, 1 for the second event, 2 for the third event... 99999 for the 100,000th event).

Pointers: An all-too-brief explanation (optional, for those who don't know C++ or C) (5 minutes)

Notice that on the previous page we used the pointer symbols "->" (a dash followed by a greater-than sign) instead of the period "." to issue the commands to the Tree. This is because the variable "tree1" isn't really the Tree itself; it's a 'pointer' to the Tree.

The difference between an object and a pointer in C++ (and ROOT) is a key concept in programming. Unfortunately, a detailed explanation is beyond the scope of this tutorial, although I may try to say something about this in class. I strongly suggest that you look up the description of pointers that you can find in every introductory text on C++ or C.

For now, I hope it's enough to just show a couple of examples:

[] `TH1F hist1("h1","a histogram",100,-3,3)`

This creates a new histogram in ROOT, and the name of the 'histogram object' is "hist1". I must use a period to issue commands to the histogram:

[] `hist1.Draw()`

Here's the same thing, but using a pointer instead:

[] `TH1F *hist1 = new TH1F("h1","a histogram",100,-3,3)`

Note the use of the asterisk "*" when I define the variable, and the use of the C++ keyword "new".

In this example, "hist1" is not a 'histogram object,' it's a 'histogram pointer.' I must use the pointer symbols to issue commands:

[] `hist1->Draw()`

Take another look at the file c1.C that you created in a previous example. Note that ROOT uses pointers for almost all the code it creates. On the previous page, I mentioned that ROOT automatically creates variables when it opens files in interactive mode; these variables are always pointers.

It's a little harder to think in terms of pointers than in terms of objects. However, you have to use pointers if you want to take advantage of the C++ code that ROOT can generate for you automatically.

Simple analysis using the Draw command, part 2 (15 minutes)

Instead of just plotting a single variables, let's try plotting two variables at once:

```
[] tree1->Draw("ebeam:px")
```

This is a scatterplot, a handy way of observing the correlations between two variables. The Draw command interprets the variables as ("x:y") to decide which axes to use.

Be careful: it's easy to fall into the trap of thinking that each (x,y) point on a scatterplot represents two values in your n-tuple. In fact, the scatterplot is a grid and each square in the grid is randomly populated with a density of dots that's proportional to the number of values in that grid.

Try making scatterplots of different pairs of variables. Do you see any correlations between the variables?

If you just see a shapeless blob on the scatterplot, the variables are likely to be uncorrelated; for example, plot "px" versus "py". If you see a pattern, there may be a correlation; for example, plot "pz" versus "zv". It appears that the higher "pz" is, the lower "zv" is, and vice versa. Perhaps the particle loses energy before it is deflected in the target.

By the way, you can also make three-dimentional plots this way:

```
[] tree1->Draw("px:py:pz")
```

After looking at these plots, you can see why it's important to always label your axes!

Let's create a "cut" (a limit on the range of a variable to be plotted):

```
[] tree1->Draw("zv","zv<20")
```

Look at the x-axis of the histogram. Compare this with:

```
[] tree1->Draw("zv")
```

Note that ROOT determines an appropriate range for the x-axis of your histogram. Enjoy this while you can; this feature is lost when you start using analysis macros.

Note that a variable in a cut does not have to be one of the variables you're plotting:

```
[] tree1->Draw("ebeam","zv<20")
```

Try this with some of the other variables in the tree.

The symbol for logical AND in C++ is "&&". Try using this in a cut, e.g.:

```
[] tree1->Draw("ebeam","px > 10 && zv<20")
```

A note for advanced users: A "cut" is actually a weight that ROOT applies when filling a histogram; a logical expression has the value 1 if true and the value 0 if false. If you want to fill a histogram with weighted values, use an expression for the cut that corresponds to the weight. For example, a cut of "1/e" will fill a histogram with each event weighted by 1/e; a cut of "(1/e)*(sqrt(z)>3.2)" will fill a histogram with events weighted by 1/e, for those events with sqrt(z) greater than 3.2.

Using C++ to analyze a Tree (10 minutes)

You can spend a lifetime learning all the in-and-outs of Object-Oriented programming in C++.• Fortunately, you only need a small subset of this to perform analysis tasks with ROOT. The first step is to have ROOT write the skeleton of an analysis class for your n-tuple. This is done with the MakeClass command.

Let's start with a clean slate: quit ROOT and start it up again. Open the ROOT tree again:

```
[] TFile myFile("experiment.root")
```

Now create an analysis macro for "tree1" with MakeClass. I'm going to use the name 'Analyze' for this macro, but you can use any name you want; just remember to use your name instead of 'Analyze' in all the examples below.

```
[] tree1->MakeClass("Analyze")
```

Switch to the UNIX window and examine the files that were created:

```
> less Analyze.h
> less Analyze.C
```

Remember this from my introductory talk? Unless you're familiar with C++, this probably looks like gobbledy-gook to you. (I know C++, and it looks like gobbledy-gook to *me*.) Fortunately, we can simplify this by understanding the approach of most analysis tasks:

- Set-up (open files, define variables, create histograms, etc.).
- Loop (for each event in the n-tuple or Tree, perform some tasks: calculate values, apply cuts, fill histograms, etc.).
- Wrap-up (display results, save histograms, etc.)

The C++ code from Analyze.C is on the next page. I've marked the places in the code where you'd place your own commands for Set-up, Loop, and Wrap-up. Compare the code you see in Analyze.C with what I've put on the next page.

You've probably already guessed that lines beginning with "//" are comments. Comments are not executed by the computer; they're there for the benefit of humans.

You're next observation may be that the comments put there by ROOT aren't helpful to you. I agree; the target audience for these comments are people experienced with ROOT. These are the comments that ROOT automatically generates with the MakeClass command; you can edit them after they're created, but you can't easily prevent them from being created in the first place.

Note that Loop and Wrap-up are separated by a single right curly bracket "}". Make sure your commands go in the right place! Also, be careful not to accidentally delete the final "}" in the file when you edit your Wrap-up commands.

Finally, I'm sure you've noticed the comments I put in the code in the different font. That's where you're going to put your own analysis code. If you wish, edit Analyze.C and put those comments in there to act as placeholders for your code; I suggest you give the file a different name as you edit it, such as "AnalyzeComments.C". I've already done this for you, and you can copy this code if you wish:

```
> cp ~seligman/root-class/AnalyzeComments.C $PWD
```

• That's four lifetimes. And you thought you only signed up for a ten-week project!

```
#define Analyze_cxx
#include "Analyze.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>

void Analyze::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L Analyze.C
//      Root > Analyze t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;
        // The Set-up code goes here.
   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
       // The Loop code goes here.
   }
       // The Wrap-up code goes here.
}
```

Running the Analyze macro (10 minutes)

As it stands, the Analyze macro does nothing, but let's learn how to run it anyway. Quit ROOT, start it again, and enter the following lines:

```
[] .L AnalyzeComments.C
[] Analyze a
[] a.Loop()
```

Have you figured out tab-completion on your own yet? If not, try this when you type the first command above: type ".L An", then hit the tab key, then "C", then hit the tab key again. If ROOT can figure out that you're trying to type in a file name, it will try to complete that name as best it can when you hit the tab key.

By the way, it's not just ROOT that can do this. When you're in the UNIX window and you have a long file name to work with, try typing the first couple of letters and hit tab.

After the last command, ROOT will pause as it reads through all the events in the Tree. Since we haven't included any analysis code yet, you won't see anything else happen.

If you are unfamiliar with C++. you may be very confused at this point. What do any of the above commands have to do with the file "experiment.root" or the Tree inside it? And what do these commands mean?

Take another look at Analyze.h. If you scan through it, you'll see C++ commands that explicitly refer to the name of the file, the name of the Tree, and the variables inside it. Now go back and look at the top of AnalyzeComments.C. You'll see the line "#include Analyze.h". This means that ROOT will include the contents of the file Analyze.h when it loads AnalyzeComments.C.

".L AnalyzeComments.C" tells ROOT to load the computer code inside the file AnalyzeComments.C, and to interpret the code to create a C++ class. The name of this class will be "Analyze"; look near the top of Analyze.h, and you'll see the C++ keywords "class Analyze".

"Analyze a" creates an object whose name is "a" (I'll explain this on the next page).

"a.Loop()" tells ROOT to execute the Loop command of object "a". Look at AnalyzeComments.C again. Near the beginning, you'll see the line "void Analyze::Loop". The code in this file, and therefore the code that you'll be working with for the rest of this class, defines the Loop command.

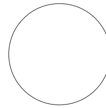
Classes and objects: An all-too-brief explanation (optional, for those who don't know C++) (5 minutes)

So what is an class and what is an object? The way I usually think of it is that a class defines an *abstract* view of a concept, but an object is a *concrete* representation of that concept.

Here's an analogy. Suppose I define the concept of a circle. What are the properties of a circle? Well, it has a radius, and the location of its center. Since we're talking about computers, we might think of giving the circle some commands to obey: tell me your circumference, tell me your area.

If I ask you where is the center of a circle, you'd have trouble answering the question numerically. I've defined the concepts of a circle, but I haven't drawn an actual circle to work with. To put it in C++ terms, I've created a circle *class*, but not a circle *object*.

Suppose I give you an actual circle. Here it is:



Now you can tell me the center of the circle in some co-ordinate system; for example, you could take a ruler and measure the center from the edge of the page. The object has the hard numbers that allow the commands to calculate values for the circumference, area, and so on.

Assume that I write C++ code to define a circle class. I'm going to put this code in a file whose name is CircleClass.C. I'm going to give the class a name: Circle. That class is going to contain a command: Area.

These are the ROOT commands that might be used to find the area of a particular circle:

```
[] .L CircleClass.C
[] Circle c
[] c.Area()
```

There's something I've avoided in the above commands: I didn't discuss how I might tell ROOT the radius or the actual (x,y) co-ordinates of the center of the circle; in C++ terms, I haven't discussed passing values to the class constructor. The reason why I've skipped over this is that it's not relevant to the Analyze example: the name of ROOT file, Tree, and variables are specified explicitly in Analyze.h.

So to do our analysis, we need a file for our C++ code. The file will define a class to perform the analysis. We need to create a concrete object of the class, and we need to issue that object the Loop command.

From a practical standpoint, this means that every time you edit the file AnalyzeComments.C, you must re-load it into ROOT, you must create a brand-new object, and you must execute the Loop command again.

Take another look at the three commands at the top of the previous page. Get used to them. You'll be typing them (or variants of them) over and over again as you do your analysis. Remember, the up-arrow key is your friend!

Making a histogram with Analyze (15 minutes)

Enough of the abstract concepts. Let's do some work.

Make a copy of the Analyze.C or AnalyzeComments.C file:

```
> cp AnalyzeComments.C AnalyzeHistogram.C
```

Edit the file AnalyzeHistogram.C. In the Set-up section, put the following code:

```
TH1* chi2Hist = new TH1F("chi2","Histogram of Chi2",100,0,20);
```

In the Loop section, put this in:

```
chi2Hist->Fill(chi2);
```

This goes in the Wrap-up section:

```
chi2Hist->Draw();
```

Don't forget the semi-colons ";" at the ends of the lines! You can omit them in interactive commands, but not in macros.

Save the file, then enter the following commands in ROOT:

```
[] .L AnalyzeHistograms.C
[] Analyze a
[] a.Loop()
```

Finally, we've made our first histogram with a C++ analysis macro. In the Set-up section, we defined a histogram; in the loop section, we filled the histogram with values; in the Wrap-up section, we drew the histogram.

How did I know which bin limits to use on "chi2Hist"? Before I wrote the code, drew a test histogram with the command:

```
[] tree1->Draw("chi2")
```

Hmm, the histogram's axes aren't labeled. How do I put the labels in the macro? Here's how I figured it out: I labeled the axes on the test histogram by right-clicking on them and selecting "SetTitle". Then I saved the canvas by selecting "Save->c1.C" from the File menu. Then I looked at c1.C and saw these commands in the file:

```
chi2->GetXaxis()->SetTitle("chi2");
chi2->GetYaxis()->SetTitle("number of events");
```

I scrolled up and saw that ROOT had used the variable "chi2" as the name of the histogram variable. I copied the lines into AnalyzeHistogram.C, but used the name of my histogram instead:

```
chi2Hist->GetXaxis()->SetTitle("chi2");
chi2Hist->GetYaxis()->SetTitle("number of events");
```

Try this yourself: add the two lines above to the Set-up section, right after the line that defines the histogram. Test the revised Analyze class.

The labels overlap the axis numbers. This is good enough for now, but you'll have to figure out how to move the labels if you were ever to publish this plot. You'll deal with this issue in Exercise 11.

Exercise 2 (5 minutes)

We're still plotting the chi2 histogram as a solid curve. Most of the time, your supervisor will want to see histograms with errors. Revise the Analyze::Loop method in AnalyzeHistogram.C to draw the histograms with error bars.

> Hint: Look back at "Working with Histograms" earlier in this tutorial.
>
> Warning: The histogram may not be immediately visible, because all the points are squeezed into the left-hand side of the plot. We'll investigate the reason why in a subsequent exercise.

Exercise 3 (15 minutes)

Revise AnalyzeHistogram.C to create, fill, and display an additional histogram of the variable "ebeam" (with error bars and axis labels, of course).

> First, some hints for those new to C++:
>
> Take care! In "Using C++ to analyze a Tree", I broke up a typical physics analysis task into three pieces: the Set-up, the Loop, and the Wrap-up; I also marked the location in the analysis macro where you'd put these steps.
>
> What may not be obvious is that *all* your commands that relate to setting things up must go in the Set-up section, *all* your commands that are repeated for each event must go in the Loop section, and so on. Don't try to create two histograms by copying the entire Analyze::Loop program and pasting it into the file more than once; it won't work.
>
> Now, some warnings for everyone:
>
> Prediction: You're going to run into trouble when you get to the Wrap-up section and draw the histograms. When you run your code, you'll probably only see one histogram plotted, and it will be the last one you plot.
>
> The problem is that when you issue the Draw command for a histogram, by default it's drawn on the "current" canvas. If there is no canvas, a default one (our old friend "c1") is created. So both histograms are being drawn to the same canvas.
>
> You can solve the problem in one of two ways: you can create a new canvas for each histogram, or you can create one large canvas and divide it into sub-pads (see the optional lesson "Working with multiple plots" earlier in this tutorial). I'll let you pick which to use, but be forewarned: working with pads is much more ambitious than creating one canvas for each plot.
>
> More clues: Look at c1.C to see an example of how a canvas is created. Look up the TCanvas class on the ROOT web site to figure out what the commands do. To figure out how to switch between canvases, look at TCanvas::cd() (that is, the cd() method of the TCanvas class).

Exercise 4 (10 minutes)

Fit the ebeam histogram to a gaussian distribution.

> OK, that part was easy, because you remembered there was a hint in "Saving your work, part 2" earlier in this tutorial. It was particularly easy because the "gaus" function is built into ROOT, so you don't have to worry about a user-defined function.

Let's make it a bit harder: the parameters from the fit are displayed in the ROOT text window; your task is to put them on the histogram as well. You want to see the parameter names, the values of the parameters, the errors on the parameters as part of the plot.

> This is trickier, because you have to hunt for the answer on the ROOT web site... and when you see the answer, you may be tempted to change it instead of typing in exactly what's on the web site.
>
> Take a look at the description of the TH1::Draw() method. In that description, it says "See THistPainter::Paint for a description of all the drawing options". Click on the link. There's lots of interesting stuff here, but scroll down and focus on the section on "Fit Statistics".
>
> There was another way to figure this out, and maybe you tried it: Draw a histogram, select "Options->Fit Parameters", fit a function to the histogram, save it as c1.C, and look at the file. OK, the command is there... but would you have been able to guess which one it was if you hadn't looked it up on the web site?

Exercise 5 (10 minutes)

Now add another plot: a scatterplot of "chi2" versus "ebeam". Don't forget to label the axes!

> Hint: Remember back in Exercise 1, I asked you to figure out the name "TF2" given that the name of the 1-dimensional function class was "TF1"? Well, the name of the 1-dimensional histogram class is "TH1F", so what do you think the name of the two-dimensional histogram class is? Check your guess on the ROOT web site.

Calculating our own variables (5 minutes)

The variables in the n-tuple are interesting, but there are other quantities that we may be interested in. One such quantity is p_T which is defined by:

$$p_T = \sqrt{p_x^2 + p_y^2}$$

This is the transverse momentum of the particle, that is, the component of the particle's momentum that's perpendicular to the *z*-axis.

Let's calculate our own values in an analysis macro. Start fresh by copying our AnalyzeComments example again:

```
> cp AnalyzeComments.C AnalyzeVariables.C
```

In the Loop section, put in the following line:

```
Float_t pt = TMath::Sqrt(px*px + py*py);
```

What does this mean?

Whenever you create a new variable in C++, you must say what type of thing it is. Actually, we've already done this in statements like

```
TF1 func("user","gaus(0)+gaus(3)")
```

This statement creates a brand-new variable named "func", with a type of "TF1". In the Loop section of AnalyzeVariables, we're creating a new variable named "pt", and its type is "Float_t".

For the purpose of the analyses that you're likely to do, there are only two types of numeric variables that you'll have to know: "Float_t", which is used for real numbers, and "Int_t", which is used for integers; "Long64_t" specifies 64-bit integers, which you probably won't need for your work. (For C++ experts: the reason why we don't just use the built-in types "float" and "int" is discussed on page 19 of the ROOT Users Guide.)

ROOT comes with a very complete set of math functions. You can browse them all by looking at the TMath class on the ROOT web site. For now, it's enough to know that TMath::Sqrt() computes the square root of the expression within the parenthesis "()".

Test the macro in AnalyzeVariables to make sure it runs. You won't see any output, but we'll fix that in the next exercise.

Exercise 6 (10 minutes)

Revise AnalyzeVariables.C to make a histogram of the variable "pt". Don't forget to label the axes; remember that the momenta are in *GeV*.

> If you want to figure out what the bin limits of the histogram should be, I'll permit you to "cheat" and use the following command interactively:

```
tree1->Draw("sqrt(px*px + py*py)")
```

Exercise 7 (15 minutes)

The quantity "theta", or the angle that the beam makes with the *z*-axis, is calculated by:

$$\theta = \arctan\left(\frac{p_T}{p_z}\right)$$

The units are radians. Revise AnalyzeVariables.C to include a histogram of theta.

> I'll make your life a little easier: the math function you want is TMath::ATan2(y,x), which computes the arctangent of y/x. It's better to use this function than TMath::ATan(y/x), because the atan2 function correctly handles the case when x=0.

Applying a cut (10 minutes)

The last "trick" you need to learn is how to apply a cut in an analysis macro. Once you've absorbed this, you'll know enough about ROOT to start using it for a real physics analysis.

The simplest way to apply a cut in C++ is to use the "if" statement. This is described in every introductory C and C++ text, and I won't go into detail here. Instead I'll provide an example to get you started.

Once again, let's start with a fresh Analyze macro:

```
> cp AnalyzeComments.C AnalyzeCuts.C
```

Our goal is to count the number of events for which pz is less than 145 *GeV*. Since we're going to count the events, we're going to need a counter. Put the following in the Set-up section of AnalyzeCuts.C:

```
Int_t pzCount = 0;
```

Why "Int_t" and not "Long64_t"? Well, I find that "Int_t" is easier to remember. I could even 'cheat' and just use "int", which will work for this example. You would only have to use the type Long64_t if you were counting more than 2^{32} entries. I promise you that there aren't that many entries in this file!

For every event that passes the cut, we want to add one to the count. Put the following in the Loop section:

```
if ( pz < 145 )
{
   pzCount = pzCount + 1;
}
```

Be careful: it's important that you surround the logical expression "pz < 145" with parentheses "()", but the "if-clause" must use curly brackets "{}".

Now we have to display the value. Again, I'm going to defer a complete description of formatting text output to a C++ textbook, and simply supply the following statement for your Wrap-up section:

```
std::cout << "The number of events with pz < 145 is "
   << pzCount << std::endl;
```

When I run this macro, I get the following output:

```
The number of events with pz < 145 is 14962
```

Hopefully you'll get the same answer.

Exercise 8 (15 minutes)

Go back and run the macro you created in Exercise 5. If you've overwritten it, you can copy my version:

```
> cp ~seligman/root-class/AnalyzeExercise5.C $PWD
```

The chi2 distribution and the scatterplot hint that something interesting may be going on.

The histogram, whose limits I originally got from the command tree1->Draw("chi2")', looks unusual: there's a peak around 1, but the x-axis extends far beyond that, up to chi2 > 18. Evidentally there are some events with a large chi2, but not enough of them to show up on the plot.

On the scatterplot, we can see a dark band that represents the main peak of the chi2 distribution, and a scattering of dots that represents a group of events with anomalously high chi2.

The chi2 represents a confidence level in reconstructing the particle's trajectory. If the the chi2 is high, the trajectory reconstruction was poor. It would be acceptable to apply a cut of "chi2 < 1.5", but let's see if we can correlate a large chi2 with anything else.

Make a scatterplot of "chi2" versus "theta". It's easiest if you just copy the relevant lines from your code in Exercise 7; again, there's a file AnalyzeExercise7.C in my area if that will help.

Take a careful look at the scatterplot. It looks like all the large-chi2 values are found in the region theta > 0.15 radians. It may be that our trajectory-finding code has a problem with large angles. Let's put in both a theta cut and a chi2 cut to be certain we're looking at a sample of events with good reconstructed trajectories.

Use an "if" statement to only fill your histograms if chi2 < 1.5 and theta < 0.15. You should change the bin limits of your histograms to reflect these cuts; for example, there's no point to putting bins above 1.5 in your chi2 histograms since you know there won't be any events in those bins.

It may help to remember that the symbol for logical AND in C++ is "&&".

A tip for the future: in a real analysis, you'd probably have to make plots of your results both before and after cuts. A physicist usually wants to see the effects of cuts on their data.

I must confess: I cheated when I pointed you directly to theta as the cause of the high-chi2 events. I knew this because I wrote the program that created the tree. If you want to look at this program youself, go to the UNIX window and type:

```
> less ~seligman/root-class/CreateTree.C
```

Exercise 9 (optional)

Assuming a relativistic particle, the measured energy of the particle in our example n-tuple is given by

$$E_{meas}^2 = P_x^2 + P_y^2 + P_z^2$$

and the energy lost by the particle is given by

$$E_{loss} = E_{beam} - E_{meas}$$

Create a new analysis macro to make a scatterplot of E_{loss} vs. "zv". Is there a relationship between the z-distance travelled in the target and the amount of energy lost?

Exercise 10 (optional)

In all the analysis macros we've worked with, we've drawn any plots in the Wrap-up section. Pick one of your analysis macros that makes histograms, and revise it so that it does not draw the histograms on the screen, but writes them to a file instead. Make sure that you don't try to write the histograms to "experiment.root"; write them to a different file named "analysis.root". When you're done, open "analysis.root" in ROOT and check that your plots are what you expect.

Hints:

- In "Saving your work, part 2", earlier in this tutorial, I described all the commands you're likely to need.
- Don't forget to use the ROOT web site as a reference. Here's a question that's also a bit of a hint: What would be the difference between opening your new file with "UPDATE" access, "RECREATE" access, and "NEW" access? Why might it be a bad idea to open a file with "NEW" access? (A hint within a hint: what would happen if you ran your macro twice?)

Exercise 11: Publishing your work

Let's create a sample histogram:

```
[] TH1F h1("hist","My Final Results",100,-3,3)
[] h1.FillRandom("gaus",100000)
[] h1.Draw("e1")
```

Your task is to make this plot neat enough for publication. Some things to note:

- Don't forget to label your axes. How about "x [arbitrary units]" for the x-axis, and "f(x) [arbitrary units]" for the y-axis.

- The x-axis label looks OK, but that y-axis label overlaps the numbers on the axis. You'll have to figure out how to move the axis, the label, or both. Hint: "SetTitleOffset".

- Print out the graph to check. Does anything look wrong?

- Do any of ROOT's pre-defined styles do the same thing?

> I did this in front of you at the start of the class. You will have to do it on your own in nine weeks, as you prepare your final talk or paper. The point of this exercise, as you've probably guessed, is to have you figure out how to do this using the tools and techniques you've learned so far. Hopefully, you'll still remember how to do this at the end of the summer.

Using the TreeViewer (very optional)

> The TreeViewer is a quick-and-dirty way of analyzing n-tuples without writing any C++ macros. In general, it's not useful for the types of analyses that your supervisor will ask you to do.
>
> However, the TreeViewer can be handy if you're looking over an n-tuple for the first time. It can give you a "feel" for the distributions of the variables, see what cuts might be useful, and understand what your histogram limits should be.

Start fresh by quitting ROOT and starting it again. Start a new browser with the command

```
[] TBrowser b
```

Open the file "experiment.root," and use the browser to locate "tree1." Right-click on tree1, and select "StartViewer." You're looking at the TreeViewer, an interface to help you analyze ROOT trees. Focus on the second column in the right-hand pane of the TreeViewer; you'll see the variables stored in the tree. Double-click on the names of the variables and see what happens.

> You can probably figure out how to use the TreeViewer on your own; the Help menu in the right-hand corner of the TreeViewer panel is genuinely useful. But I've decided to offer you a quick guide to get you started, repeating some of the tasks we performed on experiment.root using the TreeViewer instead.

Let's make a scatterplot. Left-click on a variable and hold the mouse down. Drag the variable next to the blue curly "X" in the first column, over the word "-empty-", and let go of the button. Now select a different variable and drag it over next to the curly "Y". Click on the PLOT icon in the lower left-hand corner of the TreeViewer (it's next to a button labeled "STOP", you may have to move the TreeViewer window).

Drag different pairs of variables to the "X" and "Y" boxes and look at the scatterplots. Is this any faster than when you used the Draw() method to make the same plots earlier?

You can also create expressions that are functions of the variables in the tree. Double-click on one the "E()" icons that has the word "-empty-" next to it. In the dialog box, type "sqrt(px*px+py*py)" in the box under "Expression", and type "pt" in the box under "Alias". Then click on "Done". Now double-click on the word "~pt" in the TreeViewer.

> When you're typing in the expression, you don't have to type the name of any variable in the tree. You can just click on the name in the TreeViewer.

Using the Treeviewer (continued, but still very optional)

Let's do this again to calculate theta and quickly re-do Exercise 7. Double-click on a different "E()" icon with "-empty-" next to it. Type "atan(~pt/pz)" under "Expression", and "theta" under "Alias". Click "Done", then double-click on "~theta".

> After an expression is no longer empty, you can't double-click on it to edit it; that will just cause the expression to be plotted. To edit an existing expression, right-click on it and select "EditExpression."
>
> Note that you can have expressions within expressions (such as "~pt" in the definition of "~theta"). All expressions that you create must have names that begin with a tilde (~), and the expression editor will enforce this. A common error is to forget the tilde when you're typing in an expression; that's the reason why it can be a good idea to insert a variable or an alias into an expression by clicking on it in the TreeViewer.

You can also use the TreeViewer to work with cuts. Edit another empty expression and give it the formula "zv < 20" and the alias "zcut".

> Note how the icon changes in the TreeViewer. ROOT recognizes that you've typed a logical expression instead of a calculation.

Drag "~zcut" to the scissor icon. Double-click on "zv" to plot it. Double-click on some of the other variables and look at the "Nent" in the statistics box of the histograms; the z-cut affects all the plots, not just the plot of "zv".

Double-click on the scissor icon to turn off the cut; note the change in the scissor icon. Double-click on the icon again to turn the cut back on.

Now edit "~zcut" by right-clicking on it and selecting "EditExpression". Edit the expression to read "zv<20 && zv>10" and click "Done." Plot "zv". Has the cut changed? Now drag "~zcut" to the scissors and plot "zv" again.

> At this point, it's probably occurred to you that Exercises 8 and 9 are easy to do using the TreeViewer instead of C++ macros. However, the real-world tasks that you'll be asked to do will probably quickly reach the limits of what the TreeViewer can do.

Using the Treeviewer (very optionally continued)

Had enough? No? OK, I'll show you one last TreeViewer trick.

After a while, you may have large number of expressions and cuts in your TreeViewer. You can save them so you don't have to type them all again. From the "File" menu, select "Save source". ROOT will create a file named "treeviewer.C".

If you want to look at the results, switch to your xterm window and type:

```
less treeviewer.C
```

Quit ROOT and start it again. Type

```
.x treeviewer.C
```

ROOT will open the tree file for you, and set up the TreeViewer with the same expressions and cuts.

If you are analyzing more than one tree, you may want to save multiple TreeViewers. A simple way to do this is to rename the treeviewer.C file after you save it. The Linux command to rename a file is "mv"; for example:

```
mv treeviewer.C dileptonviewer.C
```

You can do more with the TreeViewer, but not much more. I leave it to you to discover the rest.

Exercise 12: Stand-alone program (optional)

If it's near the end of the day, don't bother to start this exercise. But if you still have an hour or more before the end of the class -- well, then, you're pretty good. This exercise is a bit of a challenge for you.

So far, you've used ROOT interactively to perform all the exercises. Your task now is to write a stand-alone program that uses ROOT. Start with the macro you created in Exercise 10: you have a ROOT script (a ".C" file) that reads an n-tuple, performs a calculation, and writes a plot to a file. Create, compile, run, and test a C++ program (a ".cxx" file) that does the same thing.

You can't just take Analyze.C, rename it to Analyze.cxx, and hope it will compile. For one thing, Analyze.C does not have a "main" routine; you will have to write one.

For another, the following simple attempt won't work:

```
> g++ Analyze.C -o Analyze
```

You will have to add the flags that will add the ROOT header files and the ROOT libraries. You can save yourself some time by using the root-config command. Take a look a the **man** page for this command:

```
> man $ROOTSYS/man/root-config.1
```

Try it:

```
> root-config --cflags
> root-config --libs
```

If only there were a way of getting all that text into your compilation command without typing it all over again. This is where the UNIX "backtick" comes in handy. Try:

```
>  g++ Analyze.C -o Analyze `root-config --cflags`
```

Be careful as you type this; it's not the usual single quote (') but the backtick (`), which is typically located in the upper left-hand corner of a computer keyboard.

Are you having problems while linking? Try typing **setup gcc32** and try again. Are you enough of a Linux expert to guess why you might have to do this? (Hint: Some of the machines at Nevis run Redhat 9. Which version of Fedora are you running?)

That's enough hints.

Why would you ever want to write a stand-alone program instead of using ROOT interactively? First of all, compiled code executes faster; maybe you've already learned about the techniques described on page 87 of the ROOT User's Guide. Also, stand-alone programs are easier to submit to batch systems that run in the background while you do something else. For me, I find that I can develop code faster in a stand-alone program, without having to deal with frequently restarting ROOT or dealing with the occasional puzzling ROOT crash.