ROOT Workshop 2015

Basic Data Analysis Using ROOT

Introduction

This tutorial started as a one-day class I taught in 2001. Over the years, I've revised it as different versions of ROOT came out, and in response to comments received from the students.

The lessons have time estimates at the top. These are only rough estimates. Don't be concerned about time. The important thing is for you to learn something, not to punch a time clock. To put it another way: Don't worry if it takes you longer than the time estimate to complete a task. No one expects you to get through all 86 pages before you start your physics work for the summer.

If you're programming for the first time, then it will probably take you more than a half-day per part. Someone with years of prior experience in ROOT and C++ might barely get through all five parts in two days.

On the other hand, if the class seems too easy, just keep going; I gradually ramp up the difficulty. The lessons do not stay at the level of "ROOT does what physicists do."

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

At the end of the summer, let me know what you found useful or useless to you. I'll consider your suggestions for next year's workshop.

Have fun!

What's new

2015

Many changes in response to feedback from the working groups:

- Upgrade to ROOT 6, which affected the exercises and examples for Part Four and Five.
- The TreeViewer is back in the course.
- A few more "this is what it should look like" figures added (along with a couple more xkcd cartoons).
- Most of the working groups now have their students use Python for their summer work.
- The C++ portion on creating a code skeleton for reading an n-tuple now uses the newer MakeSelector method instead of the older MakeClass method.

2014

At the request of some of the experimental groups, I added a parallel track in pyroot, the Python wrapper around ROOT. The student can choose to learn ROOT/C++, pyroot, or both. This increased the size of the tutorial to five parts, but up to three of these parts are optional.

2010

In response to student feedback, what had been one full day of work was split into two half-day classes. Instead of eliminating the advanced exercises, I divided the two days of the 2009 class into four parts, each part roughly corresponding to a half-day's work. This allows each student to set their own pace, and gives experienced programmers a challenge if they need it.

2009

I was asked to expand the class to two full days. In past years, many students weren't able to complete all the exercises that were intended to be done in a single day. I added a set of advanced exercises for students who knew enough C++ to get through the original material quickly, but allowed for the rest of the students to do in two days what earlier classes had been asked to do in one.

A guide to this tutorial

If you see a command in this tutorial that's preceded by "[]", it means that it is a ROOT command. Type that command into ROOT *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 preceded by ">" it means that it is a UNIX command. 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.

If you take the pyroot part of this tutorial, the Python prompt is ">>>". For example:

```
> python
>>> from ROOT import TH1
```

> Paragraphs in this style are hints, tips, and advice. You may be able to get through this tutorial without reading any of this text… but I wouldn't count on it!
>
> If you're sharp of eye and keen of sight, you'll also notice that I use different styles for **`Linux commands`**, `program names and variables`, and **menu items**.

Part One – The Basics

Getting started with Linux on the Nevis particle-physics systems

If you're sitting in front of a computer running Linux, just use your account name and password to login.

Click **once** on the Browser icon (at the top or bottom of the screen) to start a web browser. This either looks like the standard Firefox icon, or 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 it. You may also want to download the User's Guide for a handy reference: click on **Documents**, then on the **User's Guide** link.

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

Applications -> Accessories -> Terminal

You may find it convenient to add the Terminal application icon to the menu bar or to your desktop. You can do this by selecting **Applications->Accessories->Terminal** on the menu bar, but right-click on the word **Terminal** instead of just releasing the left button. That will give you options to create icons on the desktop or the "panel" (menu bar).

Initially (for the ROOT class), you'll probably be content just to login to the student computers and start working. When you start your real work for this summer, I suggest that you **`ssh`** to the main server for your experiment; it's the one listed on "Summer Student Accounts" sheet that I handed out before the class.

Using your laptop

Mac and Linux

You can connect to the main server for your experiment by running a terminal window and using **`ssh`** to connect that server; e.g.,[1]

```
ssh -X -Y <server-name>.nevis.columbia.edu
```

On the Mac, you'll find the Terminal application in Applications/Utilities. On Mac OS 10.7 and above, you'll need to install XQuartz: http://xquartz.macosforge.org

Windows

To connect to a Linux server from Windows, you need **`ssh`** and an X-Windows emulator. I recommend MobaXterm, which includes both: http://mobaxterm.mobatek.net/.

Installing ROOT on your laptop

ROOT is not a pre-packaged app! If you have experience using terminal commands on your laptop, you can install ROOT on your own system; it's available for Mac OS X, Linux, and Windows. You can find it at http://root.cern.ch/drupal/content/downloading-root. Download the "Pro" binaries for your operating system.

Read the installation directions carefully. In particular, in Mac OS and Linux you have to set $ROOTSYS and other environment variables properly; I'm not sure about the Windows set-up. The typical commands are:

```
export ROOTSYS=<...ROOT's location on your system...>
export PATH=$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
```

The above lines are only valid for UNIX-style systems running sh-type shells. If this is not the case for you, you have some detective work ahead!

(On Nevis particle-physics systems, these environment variables are set for you when you type **`module load root`**.)

If you're using pyroot, you'll also need to install python. On Mac OS X and Linux this is probably already installed; in Windows you'll have to look up how to install it.

On the Mac, you'll probably have to install Xcode (available for free in the App Store). You'll also have to install XQuartz on OS X 10.7 and above: http://xquartz.macosforge.org.

---

[1] If you get tired of remembering to type **`-X -Y`** whenever you type **`ssh`**, edit the file `~/.ssh/config` in your home directory and add the lines:

```
ForwardX11 = yes
ForwardAgent = yes
```

These options are the default on Nevis particle-physics systems. You don't need to include them if you're logged onto one of the student boxes and **`ssh`**'ing into your workgroup's server.

A Brief Intro to Linux

If you're already familiar with Linux, 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.

There are links at http://www.nevis.columbia.edu/~seligman/root-class/links.html to sites that can teach you more about Linux.

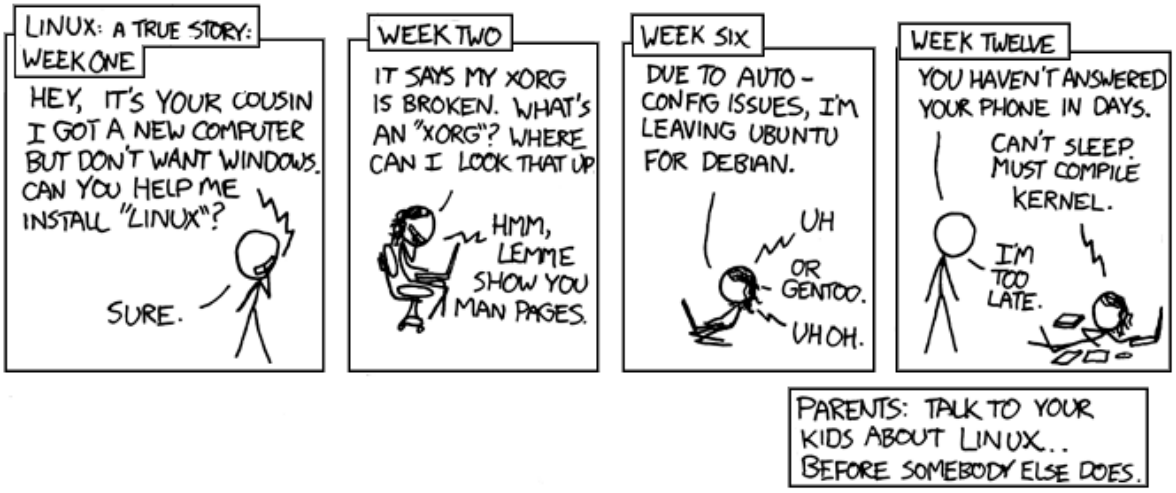

Figure 1: http://xkcd.com/456 by Randall Munroe

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

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

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

In UNIX, the variable $PWD means the results of the **pwd** ("print working directory") command.[2]

To look at the contents of a text file: Use the **less** command.[3]

This command is handy if you want to quickly look at a file without editing it. To browse the contents of file CreateSubdirectories.C, type:

```
> less CreateSubdirectories.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.

---

[2] A period (.) is the usual abbreviation in UNIX for "the current directory," but many students missed the period the first time I taught this class.

[3] If the name is confusing: the **less** command was created as a more powerful version of the **more** command.

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 **`less`** command, type:

```
> man less
```

To edit a file: I suggest you use **`emacs`**.[4]

You will almost always want to add an ampersand (&) to the end of any **`emacs`** command; the ampersand means to run the command as a separate process. To edit a file with the name CreateSubdirectories.C, type:

```
> emacs CreateSubdirectories.C &
```

The **`emacs`** environment is complex, and you can spend a lifetime learning it.[5] For now, just use the mouse to move the cursor and look at the menus. When you get the chance I suggest you take the **`emacs`** tutorial by selecting it under the "Help" menu.

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

Are you quitting **`emacs`** after you change a file, only to start up the editor again a moment later? Hint: look at the **File** menu. If you're editing many files, try opening them all with **File->Open File…** and switch between them using the **Buffers** menu.

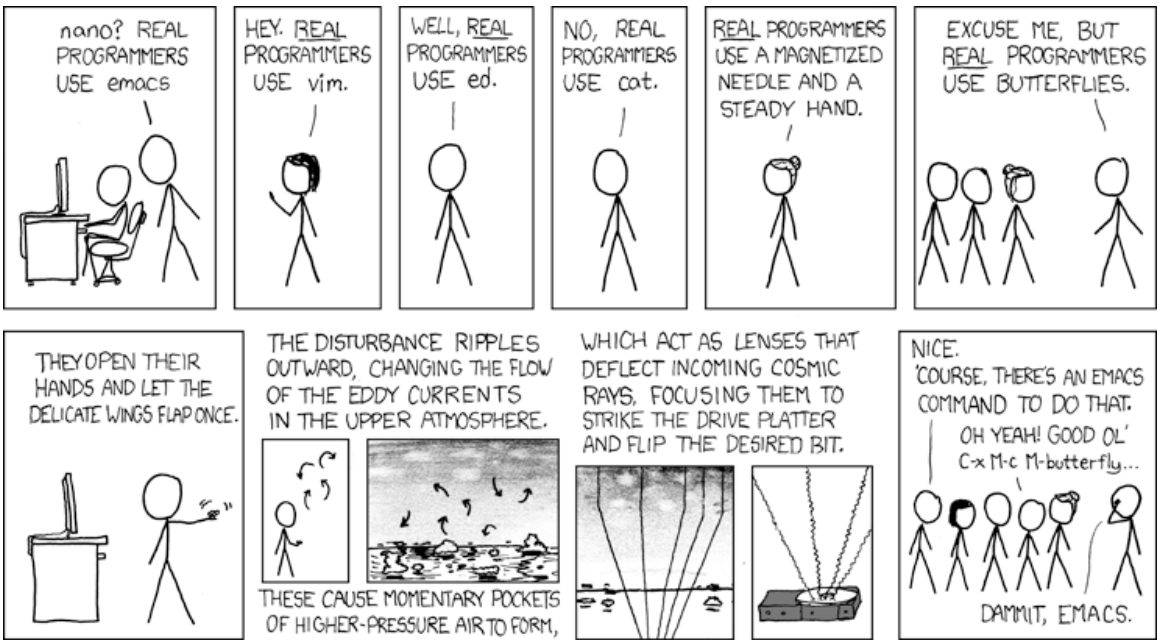

Figure 2: http://xkcd.com/378 by Randall Munroe
If you're feeling bored, type Meta-x butterfly in emacs and see what happens.

---

[4] If you're already familiar with another editor, such as **`nano`** or **`vi`**, you can use it instead.

[5] I've spent two of your lifetimes already, and the class has just started!

A Brief Intro to Linux (optional)

Here are a few Linux tricks than can make your life easier.

Using the command line

When you're typing commands into ROOT or UNIX, your two best friends are the TAB key and the up-arrow key.

Try it: On the UNIX command line, type (<TAB> means to hit the TAB key):

```
> cp ~seli<TAB>roo<TAB>Cre<TAB>S<TAB> $PWD
```

You'll see how UNIX does its best to fill in the remainder of a word, up to the point for which there's a choice.

Now list the contents of files in your current directory:

```
> ls
```

Let's execute that copy command again. You don't have to type it again, even with the help of tab-completion; just hit the up-arrow key twice and press ENTER.

Did you look at the **`emacs`** tutorial I mentioned on the previous page? If you did, you saw that it starts with a discussion of using special keypresses for cursor navigation. Perhaps you thought, "Have they never heard of a mouse?" If you did, you were right: the **`emacs`** tutorial was written before GUIs and computer mice existed.

Why is that tutorial useful, even though it's no longer the 1970s? Because those same key-based navigation commands work on the UNIX and ROOT command lines.[6]

You don't have to type the long commands in this tutorial, at least not more than once. With the help of tab-completion, the up-arrow key, and navigation keypresses, you can edit your previous commands for new tasks.

Don't get GUI

You're probably used to a graphical user interface (GUI) instead of the command line; for example, opening a file with an appropriate application by double-clicking on its icon in a window. For copying and editing files, or developing code, I recommend against a GUI; almost all physics development work is done on the command line.

However, if all you're going to do is read a file, it's OK to double-click it in a browser window and let UNIX pick an application for you.

---

[6] If you ask me to help you with a problem during the class and I start typing commands for you, you're going to see me use the up-arrow key, then Ctrl-A, Ctrl-E, Meta-F, and Meta-B to jump the cursor through the commands you've typed and make changes.

I've grown so used to those navigation commands that when I edit a file, I use **`emacs -nw`** (for "no windows") and skip the GUI features like menus and mouse-clicks. It's faster for me to keep my hands on the keyboard most of the time.

Walkthrough: Setting up ROOT (5 minutes)

ROOT is a robust, complex environment for performing physics analysis, and you can spend a lifetime learning it.[7] Before you start using ROOT on the Nevis particle-physics systems, you have to type the following command:

```
> module load root
```

The command **`module load root`** sets some Unix environment variables and modifies your command and library paths. If you need to remove these changes, use the command **`module unload root`**.

One of the variables that's 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 (on page 68, for example) you'll be able to do this only after you've typed **`module unload root.`**

You have to execute **`module load root`** once each time you login to Linux and use ROOT. If you wish this command to be automatically executed when you login, you can add it to the .myprofile file in your home directory (read the warning below before you do this).

Warning: Some physics groups work with software frameworks that have their own versions of ROOT built-in; e.g., Athena in ATLAS or LArSoft in MicroBooNE. If you're working with such a framework, you'll have a special set-up command to use; you must *not* use the generic Nevis **`module load root`**.

The command **`module load root`** is only relevant on the Nevis particle-physics computer systems. Other systems will have different ways of setting the environment variables to make ROOT work. If there are other ROOT users on the systems you use, ask them how they set up ROOT.

[7] That's three lifetimes so far.

Walkthrough: Starting ROOT (5 minutes)

You are going to need 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 **Open Terminal...** from the **File** menu of a running Terminal application.

> I like to use **File->Open Tab…** instead, but you can use whichever mode you prefer. I suggest you try all the methods to find out which one suits you.

To actually run ROOT, just type:[8]

```
> 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 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. That's "dash-ell," not "dash-one."

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

You can type **`.h`** to see a list of ROOT commands, You'll probably get more information than you can use right now. Try it and see.

For the moment, the most important ROOT line command is the one to quit ROOT. To exit ROOT, type **`.q`**. Do this now and then start ROOT again, just to make sure you can do it.

> Sometimes ROOT will crash. If it does, it can get into a state for which **`.q`** won't work. Try typing **`.qqq`** (three q) if **`.q`** doesn't work; if that still doesn't work, try five q, then seven q. Unfortunately, if you type ten q, ROOT won't respond, "You're welcome."
>
> OK, that's a dumb joke; I should leave the humor to xkcd. But the tip about **`.qqq`**, **`.qqqqq`**, and **`.qqqqqqq`** is legitimate. Sometimes I find just typing **`q`** or using Ctrl-C also works.

[8] I'm starting with "basic" ROOT, which has commands based on C++. For python users, we'll explore pyroot later. The basic command syntax to work with ROOT is almost the same in both languages anyway.

Walkthrough: Plotting a function (15 minutes)

This example is based on the first example in Chapter 2 of the ROOT Users Guide (page 10). 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.[9] ROOT may help you out with context-based colors for the keywords it recognizes.

When you type in commands, you may see something like

```
(TF1 &) @0x7ff4623c2328
```

Don't worry about this. It's not an error.[10]

If you have a keen memory (or you type **`.h`** on the ROOT command line), you'll see 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 **Documentation**, then **Reference Guide**, then on **Pro Version…,** then on **TF1**; you may want to use the browser menu **Edit->Find** and search on **TF1** to locate that link. Scroll down the page; you'll see some documentation and examples, the class methods, then method descriptions.

Get to know your way around this web site. You'll come back often.

Also note that when you executed `f1.Draw()` ROOT created a canvas for you named `Canvas_1`. "Canvas" is ROOT's term for a window that contains ROOT graphics; everything ROOT draws must be inside a canvas.[11]

(continued on the next page)

---

[9] I'm simplifying. ROOT doesn't use a C++ compiler, but an interpreter called "cling" that duplicates most of the C++ language specification. The previous version of ROOT used an interpreter called CINT; some of the ROOT documentation still refers to the interpreter by that name.

[10] If it's not an error, what is it? ROOT is printing out the type (TF1 &) and the computer address of the object you've just created. You can read more about this on page 28. You don't have to worry about this for any of the work you'll be asked to do this summer.

[11] I'm simplifying again. The actual rule is that everything ROOT draws must be inside a "TPad." Unless you want to add graphics widgets to a window (e.g., buttons and menus), this distinction won't matter to you.

Walkthrough: Plotting a function (continued)

Bring window `Canvas_1` 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 a pointing finger or 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 get 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**.

> Right-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.

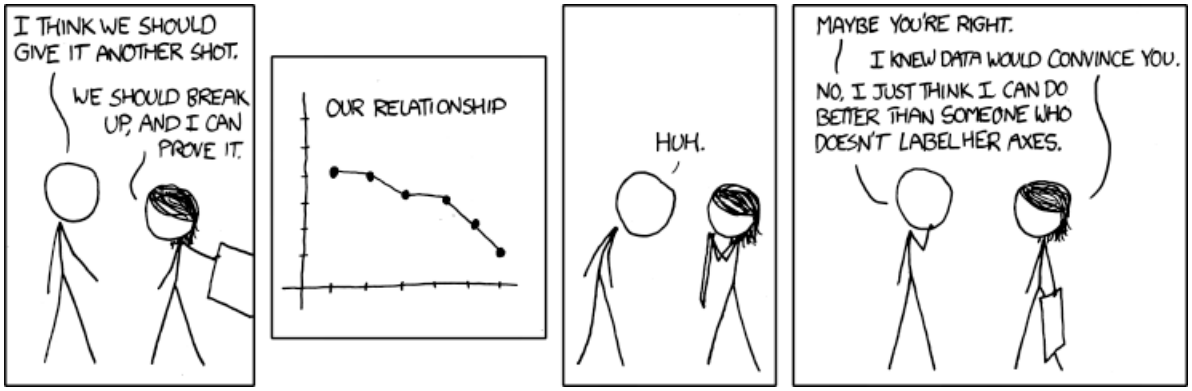

Figure 3: http://xkcd.com/833/ by Randall Munroe
Alt-text: "And if you labeled your axes, I could tell you exactly how MUCH better."

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.

You can zoom in on an axis interactively. Left-click on 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**.

Walkthrough: Plotting a function (continued)

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.

Some of the pop-up menu items have question-mark links in them. While holding down the right button (to keep the menu active), move the mouse to the "?" and press the left button. There'll be a pause for a few seconds, then you'll see a description of what the item means. You can also select an option, then click on the **online help** button. Try this for a few options.

> Note that the actual helpfulness of the descriptions varies considerably.
>
> There's also a **Help** menu on the upper-right hand corner of this window. Most ROOT windows have such a menu. Take look at its contents. I usually find that the information is enigmatic, but sometimes there's something useful.
>
> 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: Detective work (10 minutes)

Duplicate the following plot:

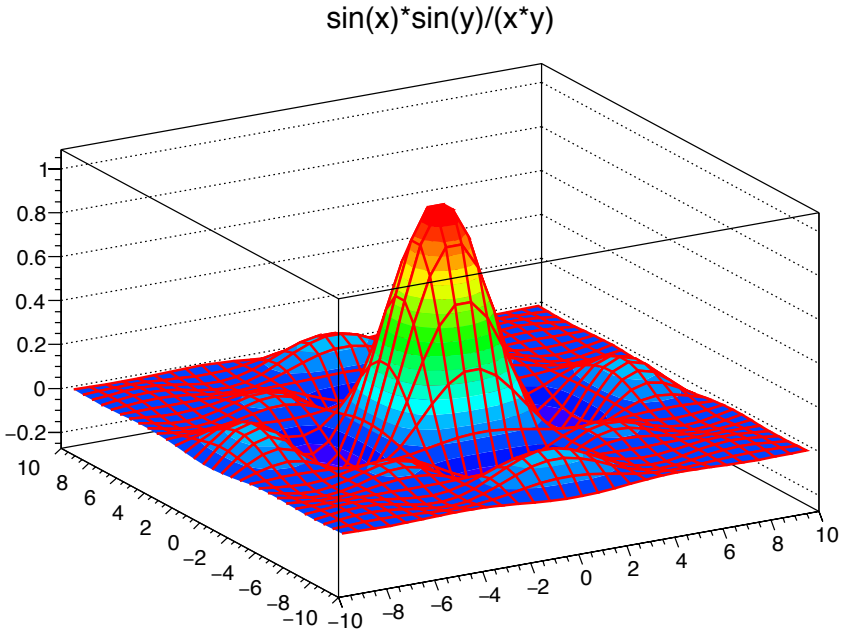

Figure 4: Some detective work is required to duplicate this plot.

Look at the `TF1` command above. 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? Where could you check your guess?

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" (with quotes) to the `Draw` method.

If you're wondering how to figure out that "surf1" was an valid option to give to `Draw()`: Unfortunately, this is not obvious in the current ROOT web site or documentation. Near the top of the `TF1` description, it states "TF1 graphics function is via the `TH1/TGraph` drawing functions." If you go to the `TH1` class and look at the `Draw()` method, it says "Histograms are drawn via the `THistPainter` class." If you go to the `THistPainter` class, you'll see all the available `Draw()` options.

It's a long chain of references, and I didn't expect you to figure it out on your own. The point is to prepare you for the kind of documentation searches you often have to do to accomplish something in ROOT; for example, the exercises in Parts Four and Five of this tutorial. Finding the "surf1" option is trivial by comparison!

Walkthrough: Working with Histograms (15 minutes)

Histograms are described in Chapter 3 of the ROOT Users Guide. You may want to look over that 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:

```
[] TH1D h1("hist1","Histogram from a gaussian",100,-3,3)
```

Let's think about what these arguments mean for a moment (and also look at the description of `TH1D` on the ROOT web site). The ROOT 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 is 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.[12]

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, nor the width exactly 1?

Add another 10,000 events to histogram `h1` with the `FillRandom` method (use up-arrow to enter the command again). Click on the canvas. Does the histogram update immediately, or do you have to type another `Draw` command?

---

[12] *For advanced users:* Why would you have varying bin widths? Recall the "too many bins" and "too few bins" examples that I showed in the introduction to the class. In physics, it's common to see event distributions with long "tails." There are times when it's a good idea to have small-width bins in regions with large numbers of events, and large bin widths in regions with only a few events. This can result in having roughly the same number of events per bin in the histogram, which helps with fitting to functions as discussed in the next few pages.

Walkthrough: Working with Histograms (continued) (10 minutes)

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 clicking 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; draw 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")
```

Let's create a function of our own:

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

The "gaus" (or gaussian) function is actually $P_0 e^{-\left(\frac{(x-P_1)}{P_2}\right)^2}$ where P_0, P_1, and P_2 are "parameters" of the function.[13] Let's set these three parameters to values that we choose, draw the result, and then create a new histogram from our function:

```
[] myfunc.SetParameters(10.,1.0,0.5)
[] myfunc.Draw()
[] TH1D 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's the difference between `SetParameters` and `SetParameter`? If you have any doubts, check the description of class `TF1` on the ROOT web site.

---

[13] *For advanced users:* In ROOT's `TFormula` notation, this would be "[0]*exp(-((x-[1])/[2])^2)" where "[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. 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)
```

Walkthrough: Working with multiple plots (optional) (5 minutes)

If you're running short on time, you can skip this.

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 `Canvas_1_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!

Walkthrough: 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 you're working with canvas `Canvas_1`, so the file "Canvas_1.C" is created. In your UNIX window, type

```
> less Canvas_1.C
```

(If you get complaints about a file not found, the name of the canvas is "Canvas-underscore-one," not "Canvas-underscore-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->Canvas_1.pdf** from the same canvas; we'll print it later.

Finally, select **Save->Canvas_1.root** from the same canvas to create the file "Canvas_1.root". Quit ROOT with the **`.q`** command, and start it again.

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

```
[] .x Canvas_1.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.[14]

You can also start ROOT and have it execute the macro all in a single line:

```
> root Canvas_1.C
```

Quit ROOT and print out your Postscript file with the command

```
> lpr -Pbw-research Canvas_1.pdf
```

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.

[14] This is still useful if you're working in pyroot, though you'll have to do some translation from C++ to python.

Walkthrough: The ROOT browser (5 minutes)

> The ROOT browser is a useful tool, and you may find yourself creating one at every ROOT session. See page 21 of the ROOT Users Guide to find out how to make ROOT start a new browser automatically each time you start ROOT.[15]

One way to retrieve the contents of file "Canvas_1.root" is to use the ROOT browser. Start up ROOT and create a browser with the command:

```
[] TBrowser tb
```

In the left-hand pane, scroll to the folder with the same name as your home directory.[16] 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": if the Editor tab is in front ROOT will display its contents in the editor window; if the Canvas tab is in front, ROOT will execute its contents. Click on the **Canvas** tab, then double-click on **Canvas_1.C** to see what happens.

Now double-click on **Canvas_1.root**, then double-click on **Canvas_1;1**.

> Don't see anything? Click on the **Canvas 1** tab in the browser window.
>
> What does "Canvas_1;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.

The ROOT browser has other "gee-whiz" features. For example, if you if select **Browser->New HTML**, it will open a new tab and display the ROOT class index web page. Feel free to use this built-in web browser if you wish, though I find Firefox to be more convenient.

> As nifty as the ROOT browser is, in the work that you'll do this summer, you'll probably reach the limits of what it can do for you, especially if you have to work with large numbers of files, histograms, n-tuples, or plots.
>
> Still, it's nice to know that it's there, in case (as the name suggests) you want to browse quickly through a couple of ROOT files.

---

[15] Hmm. There's a file called `rootlogon.C` in `~seligman/root-class`. I wonder what it does?

[16] The folder hierarchy may be puzzling to you; your home directory will be in `/a/home/<server>/<account>`. For now, don't worry about this. If you'd like to know more, there's a page on automount at http://www.nevis.columbia.edu/twiki/bin/view/Nevis/Automount.

Walkthrough: 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:[17]

```
> 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.

Double-click on **histogram.root**. You can see that I've created two histograms with the names `hist1` and `hist2`. Double-click on **hist1**; you may have to move or switch windows around, or click on the **Canvas 1** tab, to see the `Canvas_1` canvas displayed.

> 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**. Under **Fit Function**, make sure that **Predef-1D** is selected. Then make sure **gaus** is selected in the pop-up menu next to it, and **Chi-square** is selected in the **Fit Settings->Method** pop-up menu. Click on **Fit** at the bottom of the panel. 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 5.98±0.03, and the best value for the "Sigma" was 2.43±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; you'll see the fit parameters displayed on the plot.

> As a general rule, whenever you do a fit, you want to show the fit parameters on the plot. They give you some idea if your "theory" (which is often some function) agrees with the "data" (the points on the plot).

(continued on the next page)

[17] If you're going through this class and you're not logged onto a system on the Nevis Linux cluster, you'll have to get all the files from my web site: http://www.nevis.columbia.edu/~seligman/root-class/files/

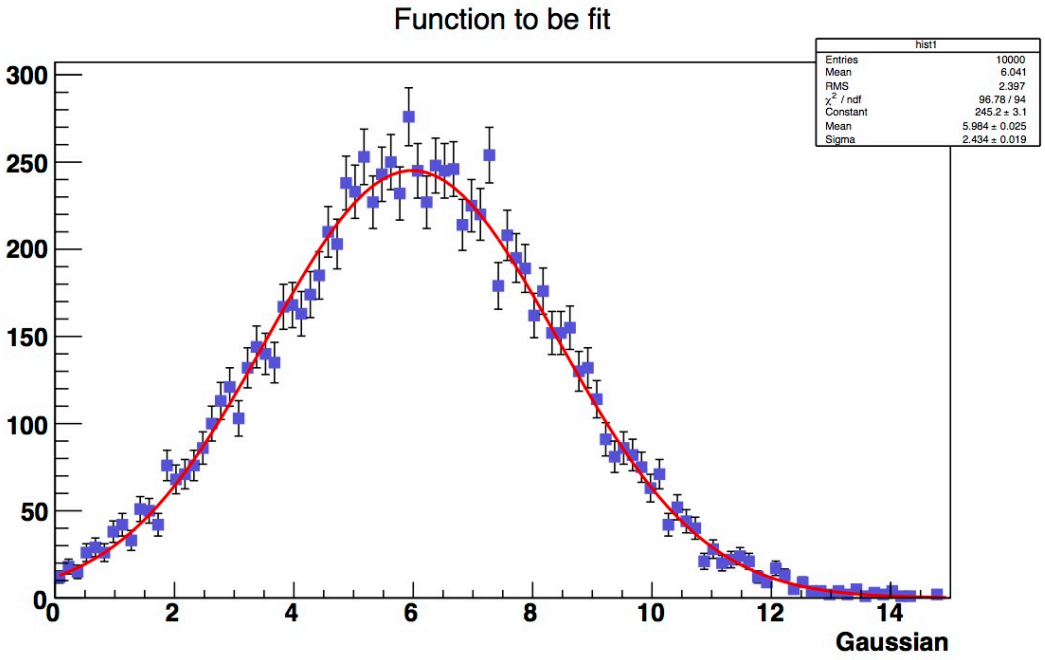

Figure 5: The resulting plot should look something like this.

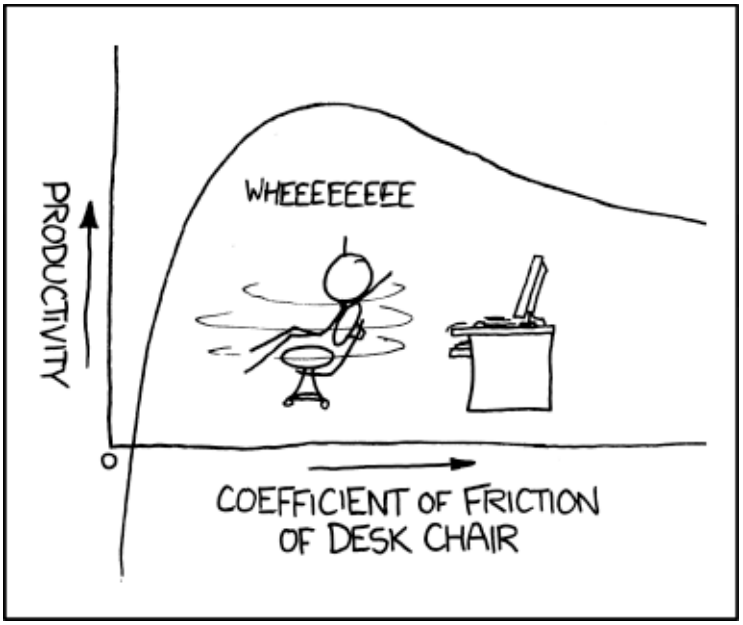

Figure 6: It will look nothing like this. This would be a poor fit for your function.
http://xkcd.com/815 by Randall Munroe
Alt-text: "As the CoKF approaches 0, productivity goes negative as you pull OTHER people into chair-spinning contests."

As a check, click on **landau** (which vaguely resembles the plot in Figure 6) on the FitPanel's **Fit Function** pop-up menu and click on **Fit** again; then try **expo** and fit again.

> You may have to click on the **Fit** button more than once for the button to "pick up" the click.
>
> It looks like of these 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 [number of] 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.
>
> The FitPanel is good for gaussian distributions and other simple fits. But for fitting large numbers of histograms (as you'd do in Parts Four and Five) or more complex functions, you want to learn the ROOT commands.

To fit hist1 to a gaussian, type the following command:[18]

```
[] hist1->Fit("gaus")
```

This does the same thing as using the FitPanel. You can close the FitPanel; we won't be using it anymore.

Go back to the browser window and double-click on **hist2**.

> You've probably already guessed by reading the x-axis label that I created this histogram from the sum of two gaussian distributions. We're going to fit this histogram by defining a custom function of our own.

Define a user function with the following command:

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

> Note that the internal ROOT name of the function is "mydoublegaus", but the name of the TF1 object is `func`.
>
> 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.

(continued on the next page)

[18] What's the deal with the arrow "->" instead of the period? It's because when you read in a histogram from a file, you get a pointer instead of an object. This only matters in C++, not in pyroot. See page 28 for more information.

Walkthrough: Fitting a histogram (continued)

Let's set the values of P_0, P_1, P_2, P_3, P_4, and P_5, and fit the histogram.[19]

```
[] func.SetParameters(5.,5.,1.,1.,10.,1.)
[] hist2->Fit("mydoublegaus")
```

It's not a very good fit, is it? 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.)
[] hist2->Fit("mydoublegaus")
```

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
```

In general, for fitting histograms in a real analysis, you'll have to define your own functions and fit to them directly, with commands like:

```
[] TF1 func("myFunction","<...some parameterized TFormula...>")
[] func.SetParameters(...some values...)
[] myHistogram->Fit("myFunction")
```

For a simple gaussian fit to a single histogram, you can always go back to using the FitPanel.

---

[19] It may help to view the PDF file with this tutorial, and cut-and-paste the commands from here into your ROOT window. You can find this file at http://www.nevis.columbia.edu/~seligman/root-class/.

Warning: Don't fall into the trap of cutting-and-pasting every command from this tutorial into ROOT. Save it for the more complicated commands like SetParameters or file names like ~seligman/root-class/AnalyzVariables.C. You want to get the "feel" for issuing commands interactively (perhaps with the tricks described on page 8), and that won't happen if you just type Ctrl-C/click/Ctrl-V over and over again.

Walkthrough: Saving your work, part 2 (15 minutes)

So now you've got a histogram fitted to a complicated function. You can use **Save as Canvas_1.root,** quit ROOT, restart it, then load canvas "Canvas_1;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 open .root 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 **Browser** menu of the browser or the **File** menu of the canvas). 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, hit the up-arrow key until you see them again. [20]

```
[] TF1 func("user","gaus(0)+gaus(3)")
[] func.SetParameters(5.,2.,1.,1.,10.,1.)
[] hist2->Fit("user")
```

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

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

Close the file to make sure you save your changes (optional; ROOT usually closes the file for you when you quit the program):

```
[] 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.

You wrote the function with `func.Write()`, but you saw `user;1` in the file. Do you see why? It has to do with the name you give to objects in your programming environment, versus the internal name that you give to ROOT. There's more about this on page 35. I wanted to point it out so that you were aware that, though they seem closely connected at times, C++/pyroot and ROOT are two *different* entities.

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

---

[20] *In case you care:* ROOT stores your last 80 or so ROOT commands in the file ".root-hist" in your home directory; that's where it gets the lines you see with the up-arrow key. Similarly, the UNIX shell stores the last 5000 commands you've typed in .sh-history in your home directory.

Walkthrough: Variables in ROOT NTuples/Trees (10 minutes)

I've created a sample ROOT n-tuple for you. Quit ROOT. Copy the example file:

```
> 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**. There's just one object inside: `tree1`, a ROOT TTree (or n-tuple) with 100,000 simulated physics events.

> There's no real physics associated with the contents of this n-tuple. I created it to illustrate ROOT concepts, not to demonstrate 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 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.

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

> In this simple example, a particle is traveling 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` (χ^2) represents a confidence level in the measurement of the particle's momentum after deflection. The variable "event" is just the event number (0 for the first event, 1 for the second event, 2 for the third event... 99999 for the 100,000th event).

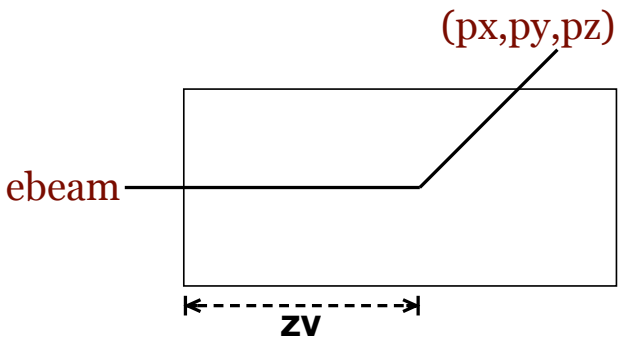

Figure 7: Sketch of the experiment and variables.

(continued on next page)

Did you notice what's missing from the above description? Answer: units. 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.[21] For this example, assume that `zv` is in centimeters (*cm*), and all energies and momenta are in *GeV*.

There's something else that's missing, but you wouldn't have noticed it unless you've performed a scientific analysis before: time. Any real experiment would have several variables relating to time (the time of the event, the time that the particle interacted in the detector, etc.) I haven't included any time-related variables in this n-tuple, with the possible exception of the event number, mainly because they wouldn't illustrate what I want to teach you in this course.

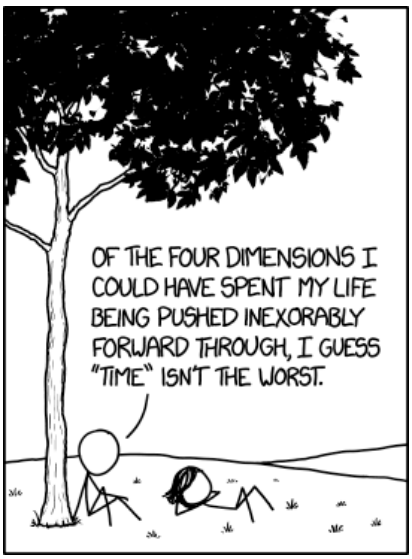

Figure 8: http://xkcd.com/1524 by Randall Munroe
Alt-text: "I would say that time is one of my top three favorite dimensions."

[21] *Advanced note:* There is a way of storing comments about the contents of a ROOT tree, which can include information such as units. However, you can't do this with n-tuples; you have to create a C++ class that contains your information in the form of comments, and use a ROOT "dictionary" to include the additional information. This is outside the scope of what you'll be asked to do this summer. If you're interested in the concept, it's described in Chapter 15 of the ROOT User's Guide. There's an example in Part Five of this class.

Using the Treeviewer[22]

Right-click the tree1 icon again and select **StartViewer**.

> You're looking at the TreeViewer, a tool for making plots from n-tuples interactively. The TreeViewer is handy for quick studies of n-tuples, but it's almost certainly not enough to get you through the work you'll have to do this summer. Any serious analysis work will involve editing ROOT macros and writing C++ code or pyroot scripts.
>
> Still, there are times when a simple tool can be useful. Let's use the TreeViewer to examine the tree1 n-tuple. Once you have an idea of what's inside tree1, you'll be ready to start writing programs to analyze it.
>
> You can 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.

In the second column of the large pane in the window, you'll see the variables in the n-tuple; they all have a "leaf" icon next to them.[23] Double-click on one of them and look the resulting histogram. Double-click on a few more variables and see how the histogram changes.

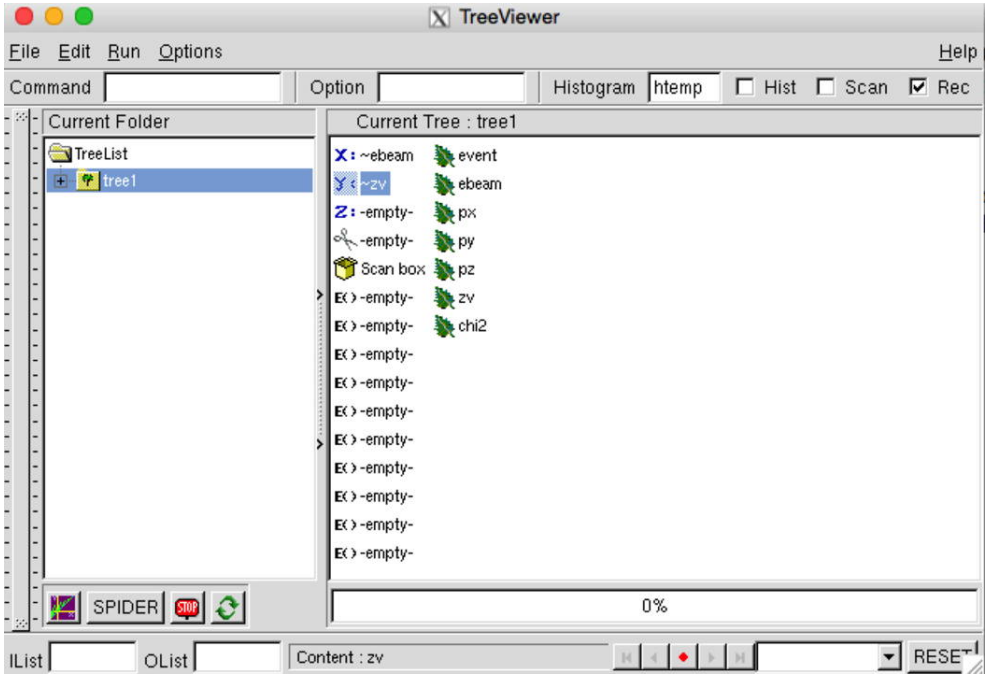

Figure 9: This is what I see when I run TreeViewer on my Macintosh.

---

[22] If you feel that this course has been too easy so far, you can skip the TreeViewer. It's easy to learn on your own if you want to. If you already know about cuts and scatterplots, skip ahead to page 34.

[23] The name of the package is ROOT, an n-tuple is a type of Tree, and the individual variables are "leaves" on the Tree. ROOT has "branches" as well; if you remember that spreadsheet model I showed you during the lecture, branches correspond to entire columns.

Correlating variables: scatterplots (10 minutes)

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 scatterplot icon in the lower left–hand corner of the TreeViewer (it's next to a button labeled "SPIDER"[24]).

> This is a scatterplot, a handy way of observing the correlations between two variables. 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.

Drag different pairs of variables to the "X" and "Y" boxes and look at the scatterplots. 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. Perhaps the particle loses energy before it is deflected in the target.

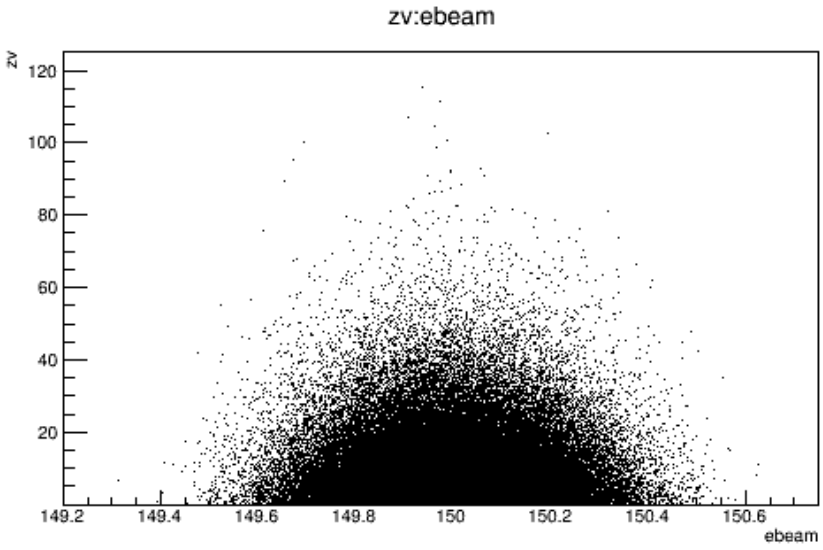

Figure 10: This is what I see when I make a scatterplot of zv versus ebeam. The variables look uncorrelated to me, subject to the restriction that we can't have zv < 0.

[24] Go ahead and click on the SPIDER button if you want. No, I don't know how to interpret those plots either.

New variables: expressions (10 minutes)

There are other quantities that we may be interested in apart from the ones already present in the n-tuple. 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.

You can use TreeViewer to 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"[25] 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.

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. Let's create a new expression to calculate theta. Double–click on a different "E()" icon with "–empty–" next to it. Type "atan2(~pt,pz)" under "Expression", and "~theta" under "Alias". Click "Done", then double–click on "~theta". [26]

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.

---

[25] That first character is a tilde (~), not a dash. On most keyboards, it's typed with SHIFT-` where ` (backtick) is on the upper-left-hand corner of the keyboard.

[26] The reason to use "atan2(y,x)" instead of just "atan(y/x)" is that the atan2 function correctly handles the case when x=0.

Restricting values: cuts (10 minutes)

Let's create a "cut" (a limit on the range of a variable to be plotted). 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 both the histogram title and 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.[27]

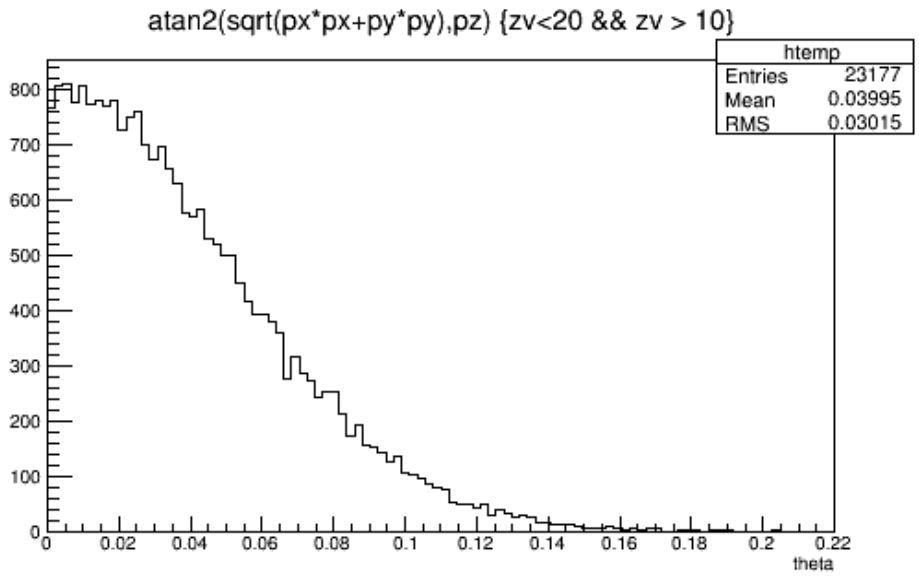

Figure 11: This is what I see when I make a plot of theta with the cut "zv<20 && zv>10".

(continued on next page)

[27] For those who know what a "weighted histogram" means: A "cut" is actually a weight 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.

Restricting values: cuts (continued) (optional)

If you wanted to display this plot in a talk, you'd have to label both axes (which you learned to do on page 14) and do something about that title. It's not clear how to fix the title of a plot from TreeViewer; if you right-click on it you see that it's a `TPaveText` with a number of options that don't seem to do what you want.

I figured this out by saving the plot as Canvas_1.C, examining that file, and looking up `TPaveText` on the ROOT web site. The simplest way to edit the title is right-click on it, select **Clear**, then select **InsertLine** and type in your new title.

Figure 12: http://xkcd.com/167 by Randall Munroe
Alt-text: "Why can't you have normal existential angst like all the other boys?"
That's why we climb (analyze) TTrees: the future is an adventure, and you don't know what you'll find.

Decision: ROOT/C++ or pyroot?

Up until this point, the commands for ROOT and pyroot were nearly identical.[28] I presented them in the context of using cling, ROOT's C++ environment.

From this point forward, using ROOT/C++ is different from using python with ROOT extensions. You have to decide: which path do you want to take learning ROOT? My initial advice is to ask your supervisor. Their response, in ascending order of likelihood, will be:

- a clear decision (C++ or pyroot);
- "I don't know. Which do you feel like learning?"
- "I have no idea what you're talking about."

If it's up to you, this may help you decide:[29,30]

In favor of pyroot:

- Learning pyroot is easier and faster than learning C++.
- Pyroot can be more appropriate for "quick-and-dirty" analysis efforts, if that's the kind of work you'll be doing this summer.

In favor of C++:

- All of the ROOT documentation, Parts Four and Five of this tutorial, and most of the tutorials included with ROOT (see page 68) are in C++.
- If you're going to be working with your experiment's analysis framework, it will almost certainly involve working in C++.
- C++ code, when compiled, is faster than python (see page 50).

Parts Two and Three of this tutorial present the same commands, exercises, and footnotes.[31] Pick which language you want to learn and go there; Part Two (ROOT/C++) starts on the next page and Part Three (pyroot) starts on page 52.

You might even be able to do both Parts Two *and* Three; once you've mastered C++, python is pretty easy!

---

[28] See page 53 for the differences when using pyroot versus interactive ROOT.

[29] Here are the areas in which neither has a clear advantage: Both C++ and python are used worldwide, so knowing either one is useful. Python's interactive development is usually cited as an advantage over C++, but ROOT offers the interactive C++ interpreter, cling. Both languages have substantive numerical computing libraries (e.g., SciPy in python, GSL in C++). For raw computing power, FORTRAN is the best, but it's no longer in style.

[30] If you're experienced with Python, you may ask why I'm not including NumPy, SciPy, matplotlib, and IPython in this tutorial. I want to focus on the ROOT toolkit, even though many tasks (especially in Parts Four and Five) can be more easily accomplished using those additional packages. But I may change my mind next year!

[31] The xkcd cartoons in the two parts are different, to give you an incentive to skim both sections.

Part Two – The C++ path

Walkthrough: Simple analysis using the Draw command (10 minutes)

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, the FitPanel 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++. Let's start with the same tasks you just did with TreeViewer.[32]

If you don't already have the sample ROOT TTree 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:

```
[] 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 a command to create it.

The answer is that when you read a file containing ROOT objects (see "Saving your work, part 2" on page 26) 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 TTree in a different way that doesn't show the data, but displays the names of the variables and the size of the TTree:

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

Either way, you can see that the variables stored in the TTree 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.

---

[32] I duplicate some of the descriptive material from the TreeViewer section, in case you decided to skip the quickie tools and get right into the programming.

Pointers: A too-short explanation (for those who don't know C++ or C) (5 minutes)

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

The detailed difference between an object and a pointer in C++ (and ROOT) is beyond the scope of this tutorial. I strongly suggest that you look this up in any introductory text on C++. For now, I hope it's enough to show a couple of examples:

```
[] TH1D 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:

```
[] TH1D *hist1 = new TH1D("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 Canvas_1.C that you created in a previous example. Note that ROOT uses pointers for almost all the code it creates. As I mentioned above, 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. But you have to use pointers if you want to use the C++ code that ROOT generates for you

You also have to use pointers to take advantage of object inheritance and polymorphism in C++. ROOT relies heavily on object inheritance (some would say too heavily); we'll get more into inheritance in Part Four of this tutorial.

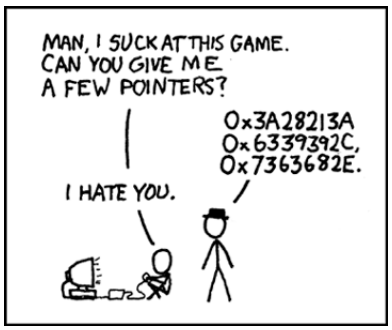

Figure 13: http://xkcd.com/138 by Randall Munroe
Alt-text: "Every computer, at the unreachable address of 0x-1, stores a secret. I have found it, and it is that all humans ar--- SEGMENTATION FAULT"

Walkthrough: Simple analysis using the Draw command, part 2 (10 minutes)

Instead of just plotting a single variable, 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 ("y:x") to decide which axes to use.

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. The scatterplot is a grid; each square in the grid is randomly populated with a density of dots proportional to the number of values in that square.

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

If you 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.

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

```
[] 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.[33]

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")
```

---

[33] *Another advanced note:* If you know what you're doing, you can use the same trick that ROOT uses when it creates the histogram you create with commands like `tree1->Draw("zv")`. The trick is:

```
TH1* hist = new TH1D(...); // define your histogram
hist->SetBit(TH1::kCanRebin); // allow the histogram to re-bin itself
hist->Sumw2(); // so the error bars are correct after re-binning
```

"Re-binning" means that if a value is supplied to the histogram that's outside its limits, it will adjust those limits automatically. It does this by summing existing bins then doubling the bin width; the bin limits change, while the number of histogram bins remains constant.

Walkthrough: 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++.[34] 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 MakeSelector command.

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

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

Now create an analysis macro for `tree1` with MakeSelector. 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->MakeSelector("Analyze")
```

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

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

Unless you're familiar with C++, this probably looks like gobbledy-gook to you. (I know C++, and it looked like gobbledy-gook to *me*… at first.) We can simplify this by understanding the approach of most analysis tasks:

- **Definition** – define the variables we're going to use.
- **Set-up** - open files, 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.

You've probably already guessed that the lines beginning with // are comments. Take a look at the comments. They describe more than we're going to use, so I'll narrow things down on the next page.[35]

(continued on the next page)

[34] That's four lifetimes. And you thought you only signed up for a ten-week project! Gosh, I wonder if it takes a lifetime to understand high-energy physics.

[35] Many of the comments, as well as the routines `SlaveBegin` and `SlaveTerminate` refer to something called PROOF. This is a method of breaking up your n-tuple into sections and analyzing each section on a separate CPU of your computer. This is handy for large n-tuples on computers with lots of CPUs, but it's outside the scope of this class. (By the way, PROOF has nothing directly to do with batch processing, which I describe on the second day of this course.)

Walkthrough: Using C++ to analyze a Tree (continued)

Here's a simplified version of the C++ code from Analyze.C is on the next page. I've removed the automatically generated comments created by ROOT, and minimized the routines `SlaveBegin` and `SlaveTerminate` which you don't need for this class. I also marked the places in the code where you'd place your own commands for Definition, Set-up, Loop, and Wrap-up. Compare the code you see in Analyze.C with what I've put below. If you wish, you can edit the contents of your Analyze.C to match what I've done; it will give you practice using **`emacs`** or whatever text editor you choose. [36]

```
#define Analyze_cxx

#include "Analyze.h"
#include <TH2.h>
#include <TStyle.h>

//******** Definition section *********

void Analyze::Begin(TTree * /*tree*/)
{
  TString option = GetOption();

  //******** Initialization section *********
}

void Analyze::SlaveBegin(TTree* tree) {}

Bool_t Analyze::Process(Long64_t entry)
{
  //******** Loop section *********
  //* You will probably want to put a GetEntry here.

  return kTRUE;
}

void Analyze::SlaveTerminate() {}

void Analyze::Terminate()
{
  //******** Wrap-up section *********
}
```

Figure 14: Example C++ TSelector macro (Analyze.C). Compare with the code in pyroot (Figure 18, page 58).

---

[36] If you're feeling lazy, you can copy the "reduced" file from my area:

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

Walkthrough: 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:

```
[] TFile myFile("experiment.root")
[] tree1->Process("Analyze.C")
```

Get used to these commands. You'll be executing them over and over again for the next several exercises. Remember, the up-arrow and tab keys are your friends![37]

Let's examine each of those commands:

- `TFile myFile("experiment.root")` – tells ROOT to load the file experiment.root into memory. This saves you from have to create the TBrowser and double-clicking on the file name every time you start ROOT (and you'll be restarting it a lot!).
- `tree1->Process("Analyze.C")` – means to load Analyze.C and run its analysis code on the contents of the tree. This means:
  - load your definitions;
  - execute your set-up;
  - execute the loop code for each entry in the tree;
  - execute your wrap-up code.

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 happen.

Take another look at Analyze.h, also called a "header file." (Analyze.C is the "implementation file.") If you scan through it, you'll see C++ commands that do something with "branches," "chains," and loading the variables from a tree. Fancy stuff, but you don't have to know about any of the nitty-gritty details. Now go back and look at the top of Analyze.C. You'll see the line

```
#include Analyze.h
```

This means ROOT will include the contents of Analyze.h when it loads Analyze.C. This takes care of defining the C++ variables for the contents of the tree.

---

[37] If you're a real ROOT jockey (and I know you want to be), there's an even faster way to do this. When I work through the exercises in this course, I start ROOT with this command:

```
> root -l experiment.root
```

This means to run ROOT without displaying the logo, and to open file experiment.root right away. This means I can omit the `TFile` command and get to work.

Walkthrough: Making a histogram with Analyze (15 minutes)

Edit the file Analyze.C. In the Definitions section, insert the following code:

```
TH1* chi2Hist = NULL;
```

This means "define a new histogram pointer and call it `chi2Hist`." Why define this as a pointer when plain ol' variables are easier to use? The short answer is that all the code ROOT uses pointers all the time; for example, if you want to read something from a file, you must always use pointers. The sooner you get used to pointers, the better.[38]

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

In the Set-up section, insert the following code:

```
chi2Hist = new TH1D("chi2","Histogram of Chi2",100,0,20);
```

This means "set this pointer to a new histogram object." We're doing this here, instead of the Definitions section, because sometimes you want quantities like histogram limits to be variable instead of being fixed.

In the Loop section, put this in:

```
tree1->GetEntry(entry);
chi2Hist->Fill(chi2);
```

The first of these two lines means "get an entry from the TTree `tree1`. Note that the variable `entry` is an argument to the `Process` method, so you don't have to set it. This line will assign values to variables defined in the n-tuple: `ebeam`, `chi2`, and so on.[39]

The second line means "add the value of `chi2` to a bin in the histogram `chi2Hist`."

(continued on next page)

---

[38] Why are we defining a pointer then setting it equal to `NULL`? I'm teaching you to avoid a common problem in programming: uninitialized pointers. If we didn't set `chi2Hist` to `NULL`, what would its value be? I don't know. It would likely be set to zero, which is also the typical value of `NULL`. But this behavior varies between different C++ compilers. It's better to be sure.

This is not an issue in the code we're writing now, but in the future you'll discover that uninitialized variables cause lots of crashes. Let's get into good programming habits and avoid them from the start.

[39] It's mildly annoying that whenever you use `MakeSelector` to create an analysis skeleton, you must remember to put a `GetEntry` line. Since `MakeSelector` is doing everything else for us, why can't it put in that one line too so we don't have to remember?

The answer is that there's more that can be done with the `TSelector` skeleton than we're doing in this course; for example, see https://root.cern.ch/root/html/tutorials/proof/ProofEventProc.C.html. Since there are times when a simple line like `tree1->GetEntry(entry)` is not what you want, or you might create an analysis skeleton for one tree and use it on another, `MakeSelector` makes you put in the `GetEntry` line manually.

This goes in the Wrap-up section:

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

You already know what this does; you've used it before!

Save the file, quit and restart ROOT, then enter the same commands as before:

```
[] TFile myFile("experiment.root")
[] tree1->Process("Analyze.C")
```

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.

"What histogram? I don't see anything!" Don't forget: if you have the TBrowser open, you may need to click on the **Canvas 1** tab.

How did I know which bin limits to use on `chi2Hist`? Before I wrote the code, I 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**. I saved the canvas by selecting **Save->Canvas_1.C** from the **File** menu. I looked at Canvas_1.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` for the name of the histogram pointer. I copied the lines into Analyze.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.

Exercise 2: Adding error bars to a histogram (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::Terminate` method in Analyze.C to draw the histograms with error bars.

Hint: Look back at "Working with Histograms" on page 18.

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.

You're going to learn this quickly: You have to restart ROOT every time before you run `tree1->Process("Analyze.C")` again. Don't forget the up-arrow key!

Exercise 3: Two histograms in the same loop (15 minutes)

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

Take care! On page 38 I broke up a typical physics analysis task into pieces: Definition, Set-up, Loop, and Wrap-up; I also marked the locations in the macro where you'd put these steps.

What may not be obvious is that *all* your commands that relate to definitions must go in the Definitions 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 program and pasting it more than once; it won't work.

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 `Canvas_1`) 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 lesson "Working with multiple plots" on page 19). I'll let you pick which to use, but be forewarned: working with pads is more ambitious than creating one canvas for each plot.

More clues: Look at Canvas_1.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).

Is the `ebeam` histogram empty? Take at look at the lower and upper limit of your histogram. What is the range of `ebeam` in the n-tuple?

Exercise 4: Displaying fit parameters (10 minutes)

Fit the `ebeam` histogram to a gaussian distribution.

> OK, that part was easy. 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, and 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 word **THistPainter**. There's lots of interesting stuff here, but for now focus on the section "Statistics Display." (This is a repeat of how I found the "surf1" option for Exercise 1 on page 16).
>
> 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 Canvas_1.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: Scatterplot (10 minutes)

Now add another plot: a scatterplot of `chi2` versus `ebeam`. Don't forget to label the axes![40]

> 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 one-dimensional histogram class is `TH1D`, so what do you think the name of the two-dimensional histogram class is? Check your guess on the ROOT web site.

---

[40] A tangent I can indulge in, now that you know about filling histograms: Suppose you're told to fill two histograms, then add them together. If you do this, you'll want to call the "Sumw2" method of both histograms before you fill them; e.g.,

```
TH1* hist1 = new TH1D(…);
TH1* hist2 = new TH1D(…);
hist1->Sumw2();
hist2->Sumw2();
// Fill your histograms
hist1->Fill(...); hist2->Fill(...);
// Add hist2 to the contents of hist1:
hist1->Add(hist2);
```

If you forget `Sumw2`, then your error bars after the math operation won't be correct. General rule: If you're going to perform histogram arithmetic, use `Sumw2` (which means "sum the squares of the weights"). Some physicists use `Sumw2` all the time, just in case.

Walkthrough: Calculating our own variables (10 minutes)

Let's calculate our own values in an analysis macro, starting with `pt` from page 30. Let's start with a fresh analysis skeleton:

```
[] tree1->MakeSelector("AnalyzeVariables.C")
```

In the `Process` section, put in the following line:[41]

```
Double_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. 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 `Process` section of `AnalyzeVariables`, we're creating a new variable named `pt`, and its type is `Double_t`.

For the purpose of the analyses that you're likely to do, there are only a few types of numeric variables that you'll have to know:

- `Float_t` is used for real numbers.
- `Double_t` is used for double-precision real numbers.
- `Int_t` is used for integers.
- `Bool_t` is for boolean (true/false) values.
- `Long64_t` specifies 64-bit integers, which you probably won't need for your work.

Most physicists use double precision for their numeric calculations, just in case.

If you already know C++: the reason why we don't just use the built-in types `float`, `double`, `int`, and `bool` is discussed on pages 18-19 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, or Chapter 13 in the ROOT User's Guide. For now, it's enough to know that `TMath::Sqrt()` computes the square root of the expression within the parenthesis "()".[42]

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

---

[41] You also have to put in that `GetEntry` line, which I complained about in Footnote 39.

[42] To be fair, there are C++ math packages as well. I could have asked you to do something like this:

```
#include <cmath>
# ... fetch px and py
pt = std::sqrt(px*px + py*py)
```

The reason why I ask you to use ROOT's math packages is that I want you to get used to looking up and using ROOT's basic math functions (algebra, trig) in preparation for using its advanced routines (e.g., fourier analysis, TMVA).

Exercise 6: Plotting a derived variable (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: Trig functions (15 minutes)

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.

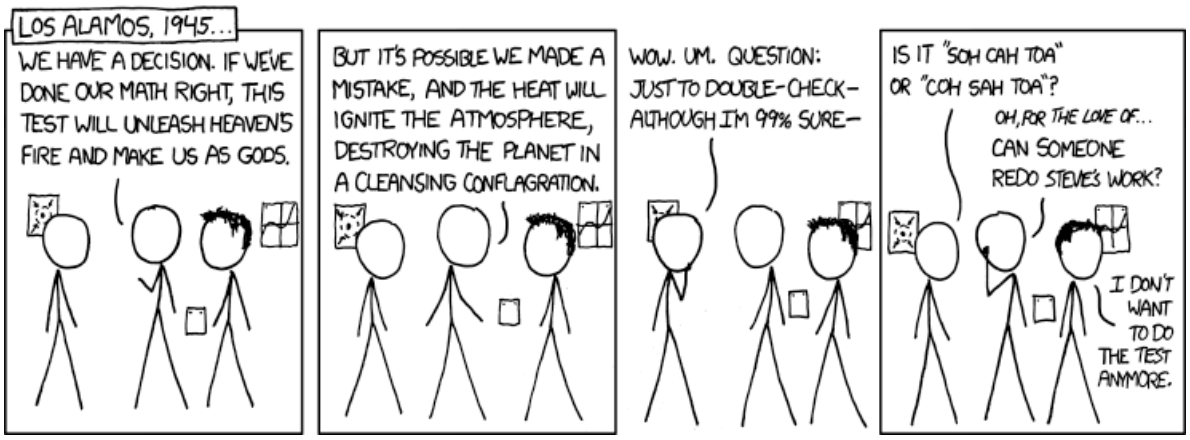

Figure 15: http://xkcd.com/809 by Randall Munroe

Walkthrough: 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 macro:

```
[] tree1->MakeSelector("AnalyzeCuts")
```

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 Definition section of AnalyzeCuts.C:

```
Int_t pzCount = 0;
```

Why `Int_t` and not `Long64_t`? 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^{31} entries. I promise you that there aren't that many entries in this file![43]

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

```
if ( pz < 145 )
{
   pzCount = pzCount + 1; // you could use "pzCount++;" instead
}
```

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.

---

[43] Recall that in the lecture I gave at the start of the class, I mentioned that other commonly used data-analysis programs couldn't handle a large number of events. Can you picture an Excel spreadsheet with more than 2^{31} rows? ROOT can handle datasets with up to 2^{63} entries!

Having trouble visualizing powers of 2? Remember that $2^{10} \approx 10^3$, so $2^{63} = 2^3 \times (2^{60}) = 2^3 \times (2^{10})^6 \approx 2^3 \times (10^3)^6 = 8*10^{18}$ or about eight quintillion, roughly the number of grains of sand in the world. My claim "ROOT can handle datasets with up to 2^{63} entries" is theoretical rather than practical.

Exercise 8: Picking a physics cut (15 minutes)

Go back and run the macro you created in Exercise 5. If you've overwritten it, you can copy my version and copy-n-paste the relevant lines to your code:

```
> 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. Evidently 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 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; there's a file AnalyzeExercise7.C in my area if it 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. 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 after cuts.

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 yourself, go to the UNIX window and type:

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

Exercise 9: A bit more physics (15 minutes)

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 (or revise one of the ones you've got) to make a scatterplot of E_{loss} vs. `zv`. Is there a relationship between the z-distance traveled in the target and the amount of energy lost?

Exercise 10: Writing histograms to a file (10 minutes)

In all the analysis macros we've worked with, we've drawn any plots in the `Terminate` method. Pick one of your analysis macros that creates 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.

> In "Saving your work, part 2" on page 26, 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: Stand-alone program (optional) (up to 60 minutes if you don't know C++)

Why would you want to write a stand-alone program instead of using ROOT interactively? Compiled code executes faster; maybe you've already learned about the techniques described on page 93 of the ROOT User's Guide. Stand-alone programs are easier to submit to batch systems that run in the background while you do something else. The full capabilities of C++ are available to you; see footnote 9 on page **Error! Bookmark not defined.**.

I'll be honest with you: I'm spending all this time to teach you about interactive ROOT, but I never use it. I can develop code faster in a stand-alone program, without restarting ROOT or dealing with a puzzling error message that refers to the wrong line in a macro.

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

So far, you've used ROOT interactively to perform 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, and run a C++ program (a ".cc" file) that does the same thing.

You can't just take Analyze.C, copy it to Analyze.cc, and hope it will compile. For one thing, Analyze.C does not have a `main` routine; you will have to write one. Also, C++ doesn't know about the ROOT classes; you have to find a way to include the classes in your program. There are links on this page that may help you:

http://www.nevis.columbia.edu/~seligman/root-class/links.html

(continued on next page)

Exercise 11: Stand-alone program (continued)

When you try to compile the program, the following simple attempt won't work:

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

You will have to add flags to the g++ command that will refer to 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/man1/root-config.1
```

Try it:

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

Is 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.cc -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 things still not working? Maybe I want you to think about adding more than one argument to a single command.

That’s enough hints.

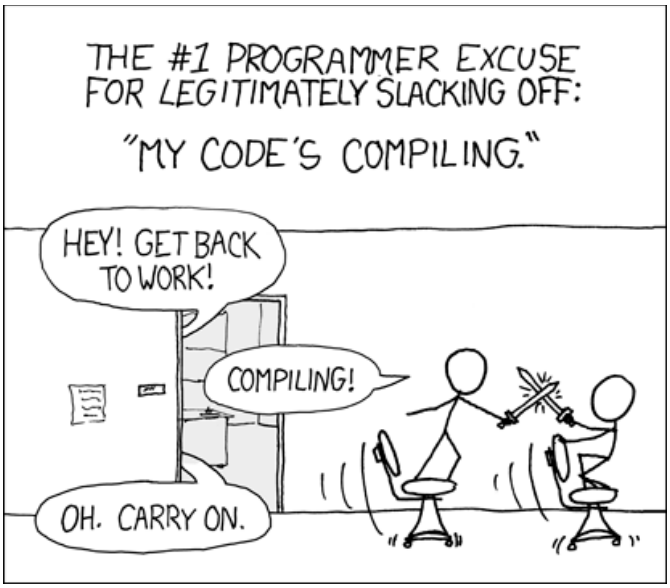

Figure 16: http://xkcd.com/303 by Randall Munroe
Alt-text: “Are you stealing those LCDs?” “Yeah, but I’m doing it while my code compiles.”

Part Three – The pyroot path

If you're not interested in pyroot or python, skip this part. Go to page 68.

Walkthrough: Starting pyroot (5 minutes)

Quit ROOT with **`.q`** if you're still in it. For Part Three, we're going to be in the python environment. Start python:

```
> python
```

You will see the python command prompt ">>>". The first thing you'll want to know is how to quit python: hit Ctrl-D (hold down the Ctrl key and press "D").

What turns python into pyroot is the inclusion of the ROOT libraries. That's done with the **`import`** command:

```
> python
>>> import ROOT
>>> # Now you can start typing ROOT commands
>>> myhist = ROOT.TH1D("example","My first pyroot histogram",100,0,100)
>>> myhist.FillRandom("gaus")
>>> myhist.Draw()
>>> myhist.Fit("gaus")
```

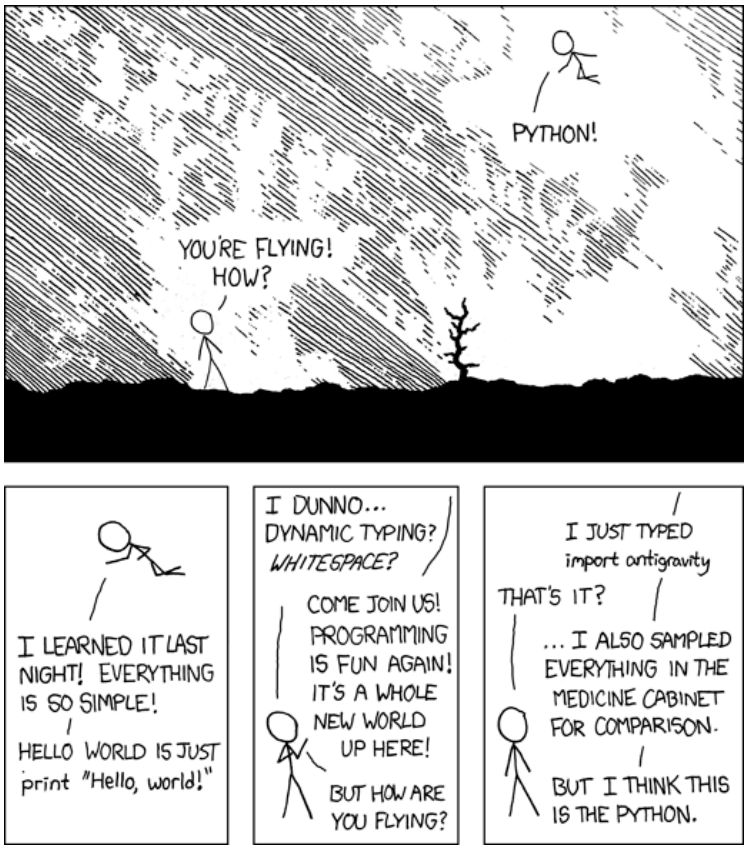

Figure 17: http://xkcd.com/353/ by Randall Munroe

Differences between C++ and python

If you already know C++, or you've already done Part Two, you should be aware of some differences between using C++ and python. Pay attention to the prompts; they tell you whether the example is in ROOT/C++ or python.

- C++ statements end with a semi-colon. Python statements end with a RETURN; no semi-colons.

```
[] myhist.FillRandom("gaus"); myhist.Fit("gaus");

>>> myhist.FillRandom("gaus")
>>> myhist.Fit("gaus")
```

- C++ control structures (e.g., `if` statements, loops) are indicated by curly braces ({}).[44] Any indentation is for the convenience of humans; the compiler doesn't need it:

```
for (Int_t jentry=0; jentry<nentries; jentry++) {
      Int_t ientry = LoadTree(jentry);
      // More stuff
}
std::cout << "The loop is over" << std::endl;
```

Python control structures are defined by indentations. The indentation is mandatory; ending (or increasing) the indentation is the same as ending (or nesting) the structure. This means that when you start working with pyroot scripts, you must be careful with the TAB and SPACE keys. Also note the colon at the end of the `for` statement; colons are also needed at the end of `if` statements:

```
for jentry in xrange( entries ):
   # get the next tree in the chain and verify
   ientry = mychain.LoadTree( jentry )
   # More stuff
print "The loop is over"
```

- C++ uses pointers, and ROOT makes liberal use of them in the code it generates for you (in .C files, etc.). Python does not use pointers, which means you don't have to remember whether to use "." or "->":

```
[] TH1* hist = new TH1D("example","my second histogram",100,-3,3);
[] hist->FillRandom("gaus");

>>> hist = ROOT.TH1D("example","my second histogram",100,-3,3)
>>> hist.FillRandom("gaus")
```

[44] I'm simplifying here. All the code in this course you've have seen so far use curly braces. I don't want to confuse you any further (except for this footnote).

- You have might picked up on this from the examples above: C++ has strict rules about types, and expects you to specify them when you create a new variable. Python determines types dynamically, and you don't have to specify them:[45]

```
[] Double_t x = 2 * 3;
[] TH1D yae = TH1D("test4","yet another example",200,-100,100);

>>> x = 2*3
>>> yae = ROOT.TH1D("test4","yet another example",200,-100,100)
```

- You may have already discovered this if you've tried typing in some of the above examples: the up-arrow key is not as good a friend as it is in the UNIX shell or in ROOT. Python will remember all the commands you've typed in your current interactive session, but it won't remember those from previous sessions.
- Finally,[46] when you're running the C++ interpreter cling within ROOT, ROOT knows the names of all its classes.

```
[] TH1D* example4 = new TH1D("example4","my fourth histogram",100,-3,3);
[] example4.FillRandom("gaus");
[] example4.Draw();
```

In python, you have to explicitly load ROOT, and then indicate that a class is part of ROOT. There are two ways to do this (see http://wlav.web.cern.ch/wlav/pyroot/using.html):

Method 1: Import all of ROOT, and indicate which classes are ROOT classes:

```
>>> import ROOT
>>> example4 = ROOT.TH1D("example4","my fourth histogram",100,-3,3)
>>> example4.FillRandom("gaus")
>>> example4.Draw()
```

Method 2: Import the classes you'll need explicitly:

```
>>> from ROOT import TH1D
>>> example4 = TH1D("example4","my fourth histogram",100,-3,3)
>>> example4.FillRandom("gaus")
>>> example4.Draw()
```

I'm typically going to use the first method in this tutorial, but you can use either one.[47] If you use the second method, be aware that if you add a new ROOT class to your python script (e.g., TCanvas), you'll have to add it to your import list:

```
>>> from ROOT import TH1D, TCanvas
```

---

[45] At least, not for the work you're likely to be asked to do with pyroot this summer.

[46] … for the purposes of this tutorial. There are many, many more differences between C++ and python!

[47] If you read up on python, you'll discover a third way: `from ROOT import *`

Never do this! It's an extremely bad programming practice that will lead you into disaster someday. In fact, forget I mentioned it. Take a marker and cross out this footnote.

Walkthrough: Simple analysis using the Draw command (10 minutes)

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, the FitPanel 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. Let's start with the same tasks you did with TreeViewer.[48]

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

```
> python
>>> import ROOT
>>> myFile = ROOT.TFile("experiment.root")
>>> tree1 = ROOT.gROOT.FindObject("tree1")
```

That second command means: Look through everything we've read in (the "everything" is gROOT) and find the object whose name is "tree1".

If you've done Part Two, note that in pyroot we have to read in the n-tuple explicitly.

You can use the Scan command to look at the contents of the Tree, instead of using the TBrowser as described on page 27:

```
>>> tree1.Scan()
```

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

```
>>> tree1.Print()
```

Either way, you can see that the variables stored in the TTree 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.

[48] I duplicate some of the descriptions from the TreeViewer discussion, in case you decided to rush into programming and skip the simple tools.

Walkthrough: Simple analysis using the Draw command, part 2 (10 minutes)

Instead of just plotting a single variable, 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 ("y:x") to decide which axes to use.

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. The scatterplot is a grid; each square in the grid is randomly populated with a density of dots proportional to the number of values in that square.

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

If you 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.

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

```
>>> 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 scripts.[49]

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.

ROOT's symbol for logical AND is `&&`. Try using this in a cut, e.g.:

```
>>> tree1.Draw("ebeam","px>10 && zv<20")
```

---

[49] *Another advanced note:* If you know what you're doing, you can use the same trick that ROOT uses when it creates the histogram you create with commands like `tree1.Draw("zv")`. The trick is:

```
hist = TH1D(...) # define your histogram
hist.SetBit(ROOT.TH1.kCanRebin) # allow the histogram to re-bin itself
hist.Sumw2() # so the error bars are correct after re-binning
```

"Re-binning" means that if a value is supplied to the histogram that's outside its limits, it will adjust those limits automatically. It does this by summing existing bins then doubling the bin width; the bin limits change, while the number of histogram bins remains constant.

Walkthrough: Using python to analyze a Tree (10 minutes)

You can spend a lifetime learning all the in-and-outs of programming in python.[50] Fortunately, you only need a small subset of this to perform analysis tasks with pyroot.

In ROOT/C++, there's a method (MakeSelector) that can create a macro for you from a TTree or n-tuple. In pyroot there's no direct equivalent. However, the "analysis skeleton" for an n-tuple is much simpler in python. I've got a basic file in my area that you can copy and edit to suit your task.

Copy my example python script to your directory. Then take a look at it:

```
> cp ~seligman/root-class/Analyze.py $PWD
> less Analyze.py
```

Most analysis tasks have the following steps:

- **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 python code from Analyze.py 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.

You've probably already guessed that lines beginning with "#" are comments.

In python, "flow control" (loops, `if` statements, etc.) is indicated by indenting statements. In C++, any indentation is optional and is for the convenience of humans. In python the indentation is mandatory and shows the scope of statements like `if` and `for`.

Note that Loop and Wrap-up are distinguished by their indentation. This means that when you type in your own Loop and Wrap-up commands, they must have the same indentation as the comments I put in.

Take a look at the code `mychain.vertex`, which means "get the current value of variable `vertex` from the TTree in `mychain`." This is an example; there's no variable `vertex` in the n-tuple in experiment.root. If you want to know what variables are available, typically you'll have to examine the n-tuple/TTree in the TBrowser or display its structure with `Print` as you did on page 55.

[50] We're up to at least four lifetimes, possibly five if you completed Part Two.

```
from ROOT import TFile, gDirectory
# You probably also want to import TH1D,
# unless you're not making any histograms.
from ROOT import TH1D

# Open the file. Note that the name of your file outside this class
# will probably NOT be experiment.root.

myfile = TFile( 'experiment.root' )

# Retrieve the n-tuple of interest. In this case, the n-tuple's name is
# "tree1". You may have to use the TBrowser to find the name of the
# n-tuple that someone gives you.
mychain = gDirectory.Get( 'tree1' )
entries = mychain.GetEntriesFast()

### The Set-up code goes here.
###

for jentry in xrange( entries ):

   # Copy next entry into memory and verify.
   nb = mychain.GetEntry( jentry )
   if nb <= 0:
      continue

   # Use the values directly from the tree. This is an example using a
   # variable "vertex". This variable does not exist in the example
   # n-tuple experiment.root, to force you to think about what you're
   # doing.
   # myValue = mychain.vertex
   # myHist.Fill(myValue)

   ### The Loop code goes here.
   ###

### The Wrap-up code goes here
###
```

Figure 18: Python analysis "skeleton" for a ROOT n-tuple.
Compare with the same code in C++ (Figure 14, page 39).

Walkthrough: Using the Analyze script (10 minutes)

As it stands, the Analyze script does nothing, but let's learn how to run it anyway. Start python if you're not already in it, and enter the following line:

```
>>> execfile("Analyze.py")
```

Pyroot 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.

> The `execfile` function in python means "read this file line-by-line and execute it as if I had typed it into python." Note that at the top of the file I've imported the ROOT classes or variables that we need, so you don't have to type `import ROOT` before using this script.[51]

Make a copy of the Analyze.py file so you can edit it for this walkthrough:

```
> cp Analyze.py AnalyzeHistogram.py
```

Edit the file AnalyzeHistogram.py. In the Set-up section, insert the following code:

```
chi2Hist = TH1D("chi2","Histogram of Chi2",100,0,20)
```

In the Loop section, put this in:

```
chi2 = mychain.chi2
chi2Hist.Fill(chi2)
```

This goes in the Wrap-up section:

```
chi2Hist.Draw()
```

> Don't forget about the indentation.

Save the file, then execute the script in python:

```
>>> execfile("AnalyzeHistogram.py")
```

> Finally, we've made our first histogram with a python script. 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.
>
> "What histogram? I don't see anything!" Don't forget: if you have the TBrowser open, you may need to click on the "Canvas 1" tab.

(continued on next page)

---

[51] If you keep `execfile`-ing the same script multiple times in a single python session, after a while ROOT might start issuing error messages about duplicate objects. For the purposes of this tutorial, you can ignore this. In a larger, more formal project, you might have to start thinking about ROOT/python memory-management issues; see <http://wlav.web.cern.ch/wlav/pyroot/memory.html> for details.

You'll be using the `execfile` function a lot for the rest of Part Three. If you exit and restart python, you won't be able to use the up-arrow to recover any previous commands you've typed. If you don't want to type this command over and over again, you may want to cut-n-paste the command in your terminal window.

Note the difference between this and the C++-based `TTree::Process` method, which normally requires you to restart your ROOT session each time.

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

```
>>> import ROOT
>>> myFile = ROOT.TFile("experiment.root")
>>> tree1 = ROOT.gROOT.FindObject("tree1")
>>> 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**. I saved the canvas by selecting **Save->Canvas_1.C** from the **File** menu. I looked at Canvas_1.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` for the name of the histogram pointer. I copied the lines into AnalyzeHistogram.py, but used the name of my histogram instead, and converted the C++ lines into python. This usually means replacing "->" with ".", and removing the semi-colon from the end:

```
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 AnalyzeHistogram script.

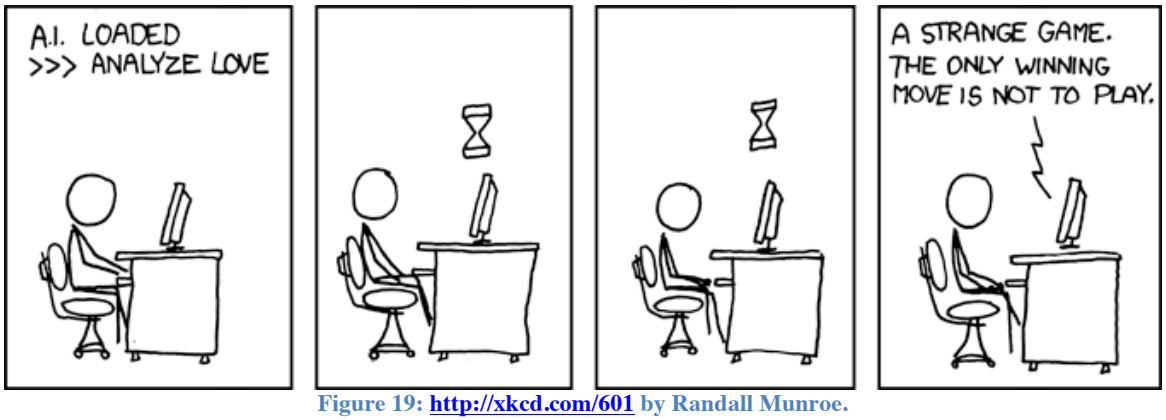

Figure 19: http://xkcd.com/601 by Randall Munroe.
Alt-text: "Wait, no, that one also loses. How about a nice game of chess?"
Fortunately it's easier to analyze histograms than it is to analyze love. At least it is for me!

Exercise 2: Adding error bars to a histogram (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 AnalyzeHistogram.py script to draw the histograms with error bars.

> Hint: Look back at "Working with Histograms" on page 18.
>
> 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: Two histograms in the same loop (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).

> Take care! On page 57 I broke up a typical physics analysis task into three pieces: the Set-up, the Loop, and the Wrap-up; I also marked the locations in the script 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 script and pasting it more than once; it may work, but it will take twice as long (because you're reading the entire n-tuple twice) and you'll be left with a single histogram at the end.
>
> 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 `Canvas_1`) 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 lesson "Working with multiple plots" on page 19). I'll let you pick which to use, but be forewarned: working with pads is more ambitious.
>
> More clues: Look at Canvas_1.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). In python, the namespace delimiter ("::" in C++) is a period ("."), so your solution will involve something like `Canvas_1.cd()`.
>
> Is the `ebeam` histogram empty? Take at look at the lower and upper limits of your histogram. What is the range of `ebeam` in the n-tuple?

Exercise 4: Displaying fit parameters (10 minutes)

Fit the `ebeam` histogram to a gaussian distribution.

> OK, that part was easy. 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, and 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 word “THistPainter”. There's lots of interesting stuff here, but for now focus on the section "Statistics Display." (This is the same procedure for figuring out the “surf1” option for Exercise 1 on page 16).
>
> 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 Canvas_1.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: Scatterplot (10 minutes)

Now add another plot: a scatterplot of `chi2` versus `ebeam`. Don't forget to label the axes![52]

> 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 one-dimensional histogram class is `TH1D`, so what do you think the name of the two-dimensional histogram class is? Check your guess on the ROOT web site.

---

[52] Now that you know about filling histograms: Suppose you're told to fill two histograms, then add them together. If you do this, you'll want to call the "Sumw2" method of both histograms before you fill them; e.g.,

```
hist1 = ROOT.TH1D(…)
hist2 = ROOT.TH1D(…)
hist1.Sumw2()
hist2.Sumw2()
# Fill your histograms
hist1->Fill(...)
hist2->Fill(...)
# Add hist2 to the contents of hist1:
hist1.Add(hist2)
```

If you forget `Sumw2`, then your error bars after the math operation won't be correct. General rule: If you're going to perform histogram arithmetic, use `Sumw2` (which means "sum the squares of the weights"). Some physicists use `Sumw2` all the time, just in case.

Walkthrough: Calculating our own variables (10 minutes)

There are other quantities that we may be interested in apart from the ones already present in the n-tuple. 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 Analyze.py script again:

```
> cp Analyze.py AnalyzeVariables.py
```

In the `Loop` section, put in the following line:

```
pt = ROOT.TMath.Sqrt(px*px + py*py)
```

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, or Chapter 13 in the ROOT User's Guide. For now, it's enough to know that `ROOT.TMath.Sqrt()` computes the square root of the expression within the parenthesis "()".[53]

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

---

[53] To be fair, there are python math packages as well. I could have asked you to do something like this:

```
import math
# ... fetch px and py
pt = math.sqrt(px*px + py*py)
```

The reason why I ask you to use ROOT's math packages is that I want you to get used to looking up and using ROOT's basic math functions (algebra, trig) in preparation for using its advanced routines (e.g., fourier transforms, multi-variant analysis).

Exercise 6: Plotting a derived variable (10 minutes)

Revise AnalyzeVariables.py 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: Trig functions (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.py to include a histogram of theta.

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

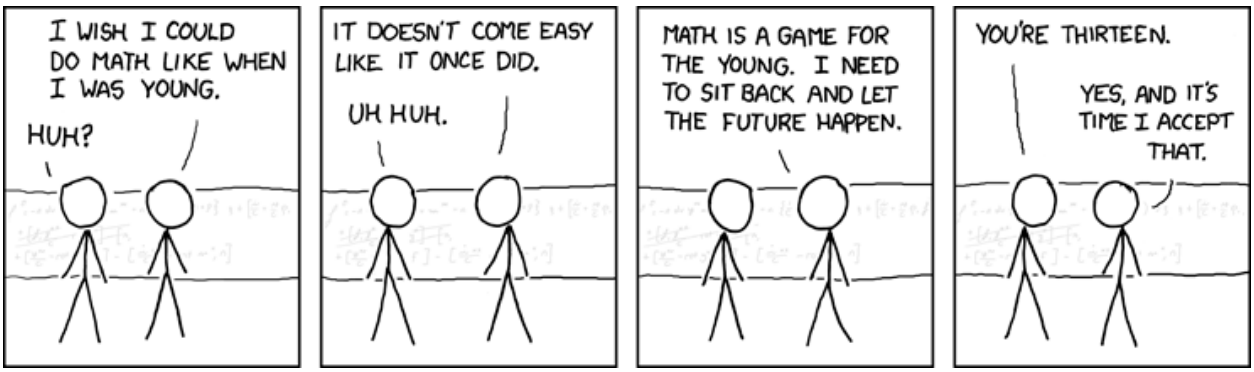

Figure 20: http://xkcd.com/447 by Randall Munroe
Alt-text: "They say that if a mathematician doesn't do their great work by age eleven, they never will."

Walkthrough: 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 is to use the `if` statement. This is described in every introductory python 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 Analyze.py AnalyzeCuts.py
```

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.py:

```
pzCount = 0
```

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: Remember that indentation is important. The next statement after `pzCount=pzCount+1` must not be indented the same amount, or it will be considered part of the `if` statement.

Now we have to display the value. Include the following statement in your Wrap-up section:

```
print "The number of events with pz < 145 is", pzCount
```

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: Picking a physics cut (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.py $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. Evidently 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 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; there's a file AnalyzeExercise7.py in my area if it 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. 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 after cuts.

It may help to remember that, in python, you'll want something like
`( chi2 < 1.5 and theta < 0.15 )`

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 yourself, go to the UNIX window and type:

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

Exercise 9: A bit more physics (15 minutes)

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 (or revise one of the ones you've got) to make a scatterplot of E_{loss} vs. `zv`. Is there a relationship between the z-distance traveled in the target and the amount of energy lost?

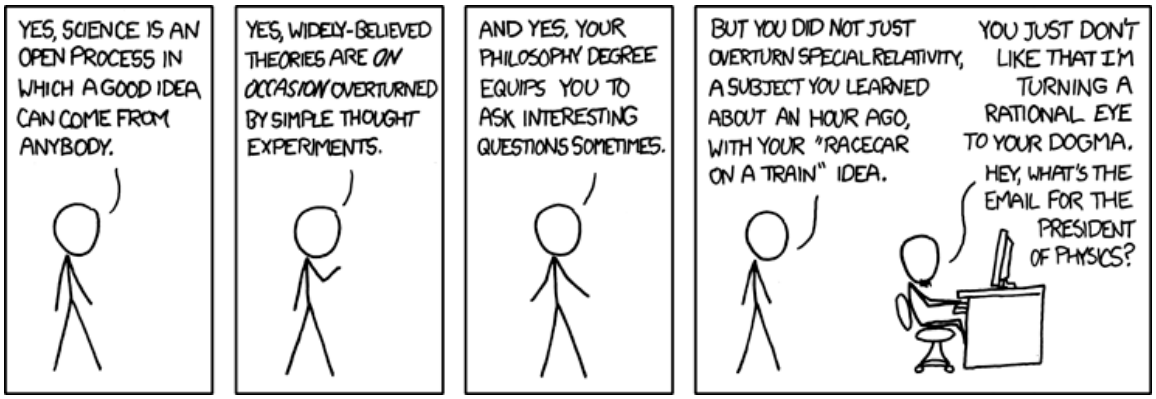

Figure 21: http://xkcd.com/675 by Randall Munroe

Alt-text: "I mean, what's more likely – that I have uncovered fundamental flaws in this field that no one in it has ever thought about, or that I need to read a little more? Hint: it's the one that involves less work."

Exercise 10: Writing histograms to a file (10 minutes)

In all the analysis macros we've worked with, we've drawn any plots in the Wrap-up section. Pick one of your analysis scripts that creates 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 the TBrowser and check that your plots are what you expect.

> In "Saving your work, part 2" on page 26, I described all the commands you'll 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's 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 script twice?)

Other Tools (for both ROOT/C++ and pyroot)

References

You've learned a few techniques to figure out how to do something in ROOT:

- The ROOT web site
- The ROOT manual
- The **Help** menu located in the upper right-hand corner of most ROOT windows
- Create something "by hand," save it as a .C file, then examine the file to see how ROOT does it

There's one other resource: the example ROOT programs that come with the package. You'll find them in $ROOTSYS/tutorials. When I ask myself the question "How do I do something complicated in ROOT?" I often find the answer in one of the examples they provide.

I've found it handy to make my own copy:[54]

```
> cp –arv $ROOTSYS/tutorials $PWD
```

Then I go into the "tutorials" sub-directory, run their examples, and look at their code:

```
> cd tutorials
> root –l demos.C
> cd graphics
> root –l first.C
> less first.C
```

> You're going to need these resources as you move into the topics for Parts Four and Five of the tutorial. I'm going to do less "hand holding" in these notes from now on, because a part of these exercises is to teach you how to use these references.[55]
>
> If the distributed nature of the information is annoying to you, welcome to the club! I often have to go hunting to find the answers I want when using ROOT, even after years of working with the package. Occasionally I've had no other choice but to examine the C++ source code of the ROOT program itself to find out the answer to a question.

---

[54] If the command doesn't work: Did you remember to type **`module load root`** in your UNIX command window? That's what sets the value of `$ROOTSYS`.

[55] You can still ask me questions during the class; I mean that any remaining written hints in this tutorial will be less detailed or require more thought.

Dealing with PAW files (optional; for reference only) (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

```
> module load 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/man1/h2root.1
```

There's also information on page 22 of the ROOT Users Guide.

Part Four – Advanced Exercises

If you still haven't finished the exercises for Parts One, Two, or Three, then keep working on them. The following exercises are relevant to larger-scale analyses, but may not be relevant to the work that you'll be asked to do this summer.

> If this class is your first exposure to programming, then these exercises are *hard*. The smart-aleck footnotes and xkcd cartoons aren't going to change that. Don't feel bound by the suggested times. Use the references to learn enough about programming to try to get the next exercise done by the end of the workshop.
>
> It's your choice whether to do the exercises in C++ or pyroot. I'm going to discuss them in C++ terms, mainly because that's my preferred programming language. Working in pyroot will pose its own set of challenges, especially when dealing with ROOT functions that take arrays as arguments. You'll learn something either way!

Before we get to those exercises, let's consider some more advanced topics in ROOT.

Working with folders inside ROOT files

> As you worked with the TBrowser, you may have realized that ROOT organizes its internal resources in the form of "folders," which are conceptually similar to the hierarchy of directories on a disk. You can also have folders within a single ROOT file.
>
> Folders are discussed in Chapter 10 in the ROOT Users Guide, but I have not seen the approach they describe (the TTask class) used in any experiment on which I've worked. Instead I'll focus on ROOT folders in the way they're more often used (if they're used at all): to organize objects within a file.

Copy the file `folders.root` from my `root-class` directory into your own, and use the ROOT TBrowser to examine its contents.

> You'll see three folders within the file: example1, example2, and example3. Each of these folders will be the basis of the next three exercises.
>
> All three exercises will require you to make a plot of data points with error bars. You'll want to use the `TGraphErrors` class for this.

A little bit more about inheritance

Look up the TGraphErrors class on the ROOT web site. After you've looked over what little help there is on this page, click on the Source tab to see an example of how the class is used. At the top of the page it says "TGraphErrors: public TGraph"; click on the TGraph link to find out where to look for options when drawing the graph.

> The "public TGraph" part means that the class TGraphErrors is inherited from the TGraph class. If you don't know about class inheritance in C++ yet, don't worry. For now, it's enough to know that if class "Derived" inherits from class "Base", class Derived will have all the methods and properties of class Base, plus some additional features.[56]
>
> Here's a simple example: If you saw "Circle: public Shape" in the ROOT reference guide, it would mean that the class Circle derives from the class Shape. If Shape had a Draw() method, then Circle would have a Draw() method as well.
>
> A more relevant example: TGraphErrors inherits from TGraph. Since TGraph has a Draw() method, so does TGraphErrors; TGraph also has a Fit() method, a GetMean() method, and a Print() method; therefore TGraphErrors has those methods too.
>
> However, TGraphErrors has a SetPointError() method, and TGraph does not. (To continue the above analogy, a Circle class might have a GetRadius() method that a more general Shape class wouldn't have.)

You may have noticed that TGraph, in turn, inherits from TNamed and some other classes. Click on the TNamed link; you'll see that TNamed inherits from TObject. Does TObject inherit from any other class?

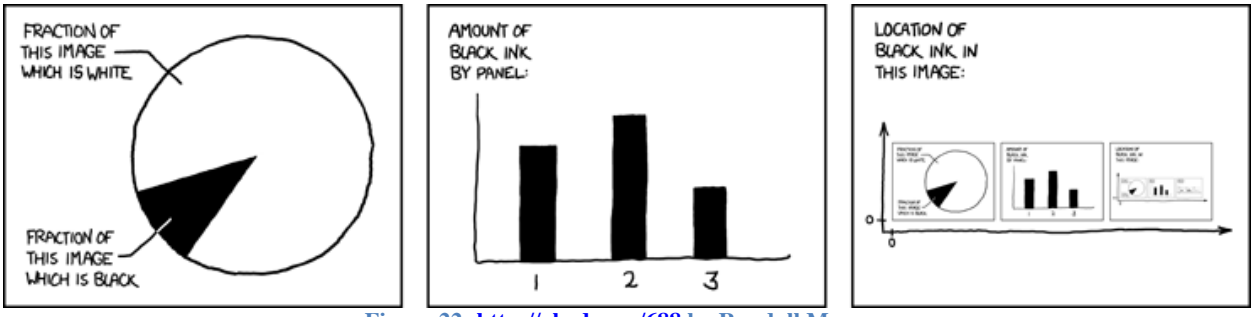

Figure 22: http://xkcd.com/688 by Randall Munroe

[56] For those familiar with the issues of public inheritance: yes, I'm skipping over a lot of details, such as the distinction between virtual versus non-virtual methods.

Container classes

Go back to the description of the TGraphErrors class. To create a TGraphErrors object, you need to supply some arguments.

These are all different ways to construct a plot with error bars:

- TGraphErrors() – This is used internally by ROOT when reading a TGraphErrors object from a file. You won't use this method directly.
- TGraphErrors(Int_t n) – You use this when you just want to supply TGraphErrors with the number of points that will be in the graph, then use the SetPoint() and SetPointError() methods to assign values and errors to the points.
- TGraphErrors(const TGraphErrors& gr) – This is called a "copy constructor" in C++, and is used when you copy a TGraphErrors object. You can ignore this.
- TGraphErrors(const TH1* h) – You use this to create a TGraphErrors plot based on values in a histogram.

Now that I've give you a guide to the first four ways to construct a TGraphErrors object, you can probably figure out what the next five are: to create one from the contents of a file, and to create plots from either float or double-precision… somethings.

Those somethings are containers. In ROOT and C++, there are three general categories of containers you have to know about.

Arrays

Do a web search on "C++ arrays" to learn about these containers. Briefly, to create a double-precision array of eight elements, you could say:

```
Double_t myArray[8];
```

To refer to the 3rd element in the array, you might use (remember, in C++ the first element has an index of 0):

```
Int_t i = 2;
myArray[i] = 0.05;
```

If you're new to C++, it won't be obvious that while myArray[2] is a Double_t object, the type of the name myArray (without any index) is Double_t*, or a pointer to a Double_t (see page 36).

Getting confused? Let's keep it simple. If you've created arrays with values and errors…

```
Double_t xValue[22];
Double_t xError[22];
Double_t yValue[22];
Double_t yError[22];
```

…and you've put numbers into those arrays, then you can create a TGraphErrors with:

```
TGraphErrors* myPlot = new TGraphErrors(22,xValue,yValue,xError,yError);
```

Did you notice a problem with that example? I had to supply a fixed value for the number of points in each array to make the plot. In general, you won't be able to do that; in fact, in exercises 15 and 16 below you *can't* do that.

In C++, one way to get around this problem is to use "dynamic arrays." I'll let you read about those on the web (search on "C++ dynamic arrays"), but I'm not going to say more about them, because I rarely use them.

ROOT's containers

Go to the Class Index page of the ROOT Reference Guide on the web. Near the top of the page there's a list of categories; click on CORE, then on CONT. You'll see a list of ROOT's container classes, along with links for information about collections and why they're used. Read the "Understanding Collections" page, and at least skim the chapter about collections in the ROOT Users Guide.

I'll be blunt here, and perhaps editorialize too much: I don't like ROOT's collection classes. The main reason is that most of them can only hold pointers to classes that inherit from TObject. For example, if you wanted to create a TList that held strings or double-precision numbers (TString and Double_t in ROOT), you can't do it.

Go back to the TGraphErrors page. The seventh way to create a TGraphErrors object has a TVectorD link; click on that link to read the description. Learn much? Try going to ROOT index of classes and clicking on TVectorT<double>.

This is ROOT's answer to the issue I just raised: they provide special containers for certain types.

You need to know a little about ROOT's collection classes to be able to understand how ROOT works with collections of objects; exercise 16 below is an example of this. For any other work, I'm going to suggest something else:

Standard Template Library (STL)

Do a web search on "standard template library". This will probably take you to SGI's web site at first. Skim a few sites, especially those that contain the words "introduction" or "tutorial". You don't have to get too in-depth; for example, you probably don't have enough time today to fully understand the concept of iterators.

Did you guess that STL is my preferred method of using containers in C++?

Vectors

> The Standard Template Library is an important development in the C++ programming language. It ties into the concepts of design patterns and generic programming, and you can spend a lifetime learning them.[57]
>
> For the work that you'll be asked to do in Parts Four and Five, and probably for the rest of this summer, there's only one STL class you'll have to understand: vectors. Here are the basics:

If you want to use vectors in a program, or even a ROOT macro, you have to put the following near the top of your code:

```
#include <vector>
```

To create a vector that will contain a certain type, e.g., double-precision values:

```
std::vector<Double_t> myVector;
```

If you want to create a vector with a fixed number of elements, e.g., 8:

```
std::vector<Double_t> myOtherVector(8);
```

To refer to a specific element of a vector, use the same notation that you use for C++ arrays:

```
myOtherVector[2] = 0.05;
```

To append a value to the end of the vector, which will make the vector one element longer, use the push_back() method:

```
myVector.push_back( 0.015 );
```

To find out the current length of a vector, use the size() method:

```
Int_t length = myVector.size();
```

Here's a simple code fragment that loops over the elements of a vector and prints them out.

```
for ( Int_t i = 0; i != someVector.size(); ++i  )
{
   std::cout << "The value of element " << i
             << " is " << someVector[i] << std::endl;
}
```

You have a vector, but TGraphErrors wants a C++ array name. Here's the trick:

```
// Define four vectors.
std::vector<Double_t> x,y,ex,ey;
// Put values in the vectors (omitted so you can do it!)
Int_t n = x.size();
TGraphErrors* plot = new TGraphErrors(n, x.data(),  y.data(),
                                      ex.data(), ey.data());
```

In other words, if `v` has the type `std::vector<Double_t>`, then `v.data()` is equivalent to the underlying array.

---

[57] I've lost track of the number of your lifetimes I've spent. You're probably tired of the joke anyway.

Exercise 12: Create a basic x-y plot (1-2.5 hours)

You're going to re-create that "pun plot" that I showed during my initial talk:

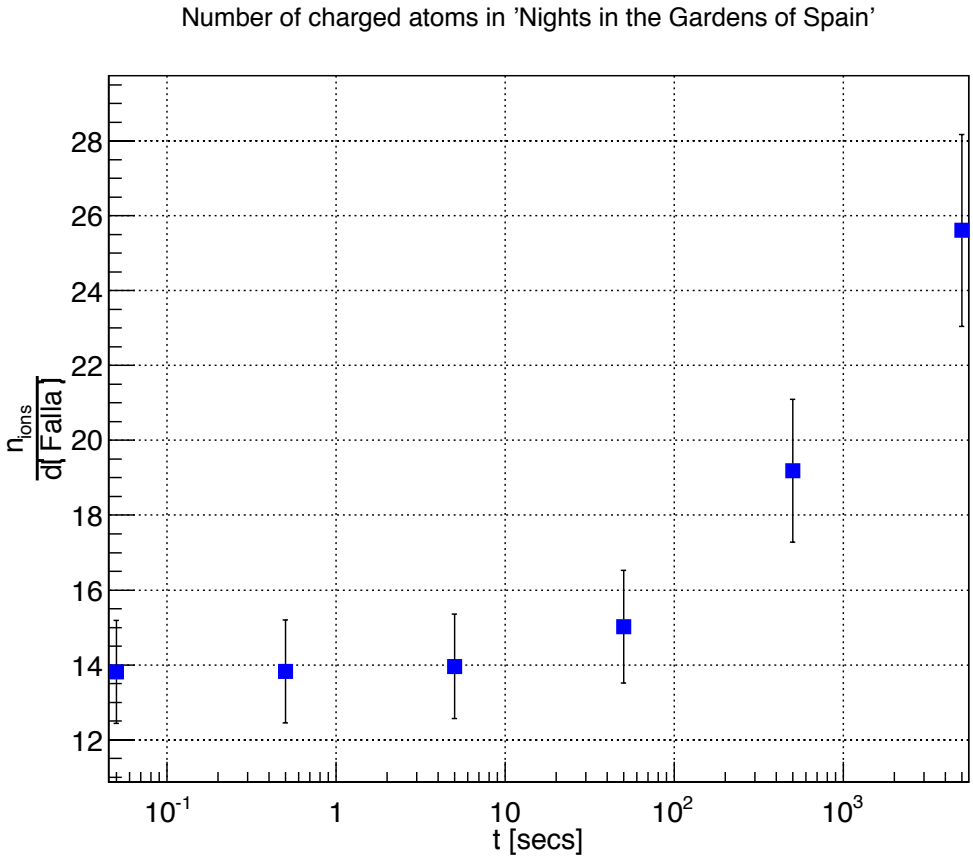

Figure 23: Can you spot the pun in this plot?
Hint: It involves the composer of a piece of music for piano and orchestra written in the early 20th century.

Use the histograms in folder example1 file folders.root. The y-values and error bars will come from fitting each histogram to a gaussian distribution; the y-value is the mean of the gaussian, and the y-error is the width of the gaussian.

> You've spent five pages reading about abstract concepts and are probably eager to do some work, but there's still a couple of things you'll have to figure out.

(continued on next page)

Exercise 12: Create a basic x-y plot (continued)

First of all, there's no n-tuple in this exercise. You'll have to create a ROOT or pyroot macro to create the graph on your own.[58] You've seen some macros before (remember Canvas_1.C?), and you'll find many more in the ROOT tutorials.

Want to see more examples of using TGraphErrors? Look at the ROOT tutorials directory. The problem is that there are lots of examples; how do you find those that use TGraphErrors? I copied the ROOT tutorials directory (see page 52), and then I used the UNIX grep command:

```
> cd tutorials
> grep –rl TGraphErrors *
```

This will list the names of the files that contain the text "TGraphErrors". That's how I found out how to draw a TGraphErrors plot inside a ROOT canvas.

The UNIX grep command is very useful; type **`man grep`** to learn about it.[59]

You need to figure out how to get the x-values. In this case, it's relatively simple. There are only six histograms in the example1 folder. In TBrowser, double-click on the histograms and read the titles. The histograms are numbered from hist0 to hist5; so you can derive a formula to go from the histogram index to the value of x.

You already know how to open a ROOT file within a macro (it was part of exercise 10 on page 49), but it's not obvious how to "navigate" to a particular folder within a file. Look at the description of the TFile class on the ROOT web site. Is there a method that looks like it might get a directory? No? TFile inherits from TDirectoryFile; is there a method in that class that might get a directory?

(continued on next page)

---

[58] You could try typing the commands on the ROOT command line one-by-one. Unless you have a shining grasp of ROOT concepts and perfect typing skills, you're going to make mistakes that will involve many quit-and-restarts of ROOT. It's much easier to write and edit a macro.

[59] Optional tangent:

`grep` is a program that implements "regular expressions," a powerful method for searching, replacing, and processing text. More sophisticated programs that use regular expressions include **`sed`**, **`awk`**, and **`perl`**. Regular expressions are used in manipulating text, not numerical calculations, so their deep nitty-gritty is rarely relevant in physics.

Regular expressions are a complex topic, and it can take a lifetime to learn about them. (You may be tired of the joke, but I'm not!)

There's a cool xkcd cartoon about regular expressions. It's too big to put into a footnote, so you'll have to click on the link yourself: https://xkcd.com/208/

Exercise 12: Create a basic x-y plot (continued)

By now, you've probably learned that for ROOT to know where to look to plot, read, or write something, it has to know where to "focus." If an object requires focus in some way, it will have a cd() method (short for "change directory"). Based on that hint, and what you can see on the TFile web page, something like this might work:

```
TDirectory* example1 = inputFile->GetDirectory("example1");
example1->cd();
```

The histograms are numbered 0 to 5 consecutively. It would be nice to write a loop to read in "hist0", "hist1", … "hist5" and fit each one. But to do that, you have to somehow convert a numeric value to a text string.

If you know C or C++, you already know ways to do this (and in python it's trivial). If all this is new to you, here's one way to do it:

```
#include <sstream> // put this near the top of your macro
for ( Int_t i = 0; i != 6; ++i )
{
   std::ostringstream os;
   os << "hist" << i;
   TString histogramName = os.str();
   // … do what you need to with histogramName
}
```

There are other problems you'll have to solve:

- *How do you read a histogram from a file? Or the more general question is: How do you get a ROOT object from a file?*

  Hint: How do you "find" an object in a TFile? (Once you've figured this out, look through the tutorial files for more clues.)

- *Once you fit a histogram to a gaussian distribution, how do you get the mean and width of the gaussian from the fit?*

  Hint: Look through the histogram methods; click through the links to other classes. The TH1D page doesn't list the method you'll need, but the TH1 page does.

(hints continued on next page)

Exercise 12: Create a basic x-y plot (continued)

- *In Figure 23, the x-axis is logarithmic. How do you make that change?*

  Hint: Remember how you found out how to label an axis?

- *Speaking of axis labels, how do you put in* $\frac{n_{ions}}{d(Falla)}$ *?*

  Hint: look up TLatex in the ROOT web site. You don't have to declare a TLatex object; just put the text codes into the axis label and ROOT will interpret them.[60]

- *How do you get the marker shapes and colors as shown in the plot?*

  Some looking around the ROOT web site should give you the answer.

  Now you can get to work!

[60] Another tangent:

LaTeX is a document-preparation package that's often used in research. If you write a paper for publication this summer, you are going to use LaTeX; physics publications don't accept articles in MS-Office format. A real LaTeX document is much more complex than you'll read about in the TLatex documentation, but don't worry about that. No one writes a LaTeX document from scratch; they get one from someone and learn by example. It's much easier than learning ROOT. It's closer to other page mark-up languages, such as HTML or Markdown, which you've probably seen before.

You can spend a lifetime learning LaTeX, but no one ever has.

Exercise 13: A more realistic x-y plotting task (1-2 hours)

It took nine pages to set up the previous exercise. It only takes one page to describe this one. Don't be fooled: this exercise is harder!

Take a look at folder example2 in folders.root. You'll see a bunch of histograms, and an n-tuple named histogramList. Right-click on histogramList and Scan the n-tuple. On the ROOT text window, you'll see that the n-tuple is a list of histogram ID numbers and an associated value.

Once again, you're going to fit all those histograms to a gaussian and make an x-y plot. The y values and error bars will come from the fits, as in the previous exercise. The x values will come from the n-tuple; for example, the value of x for histogram ID 14 is 1.0122363.

I'll let you pick the axis labels for this graph; don't make the x-axis logarithmic.

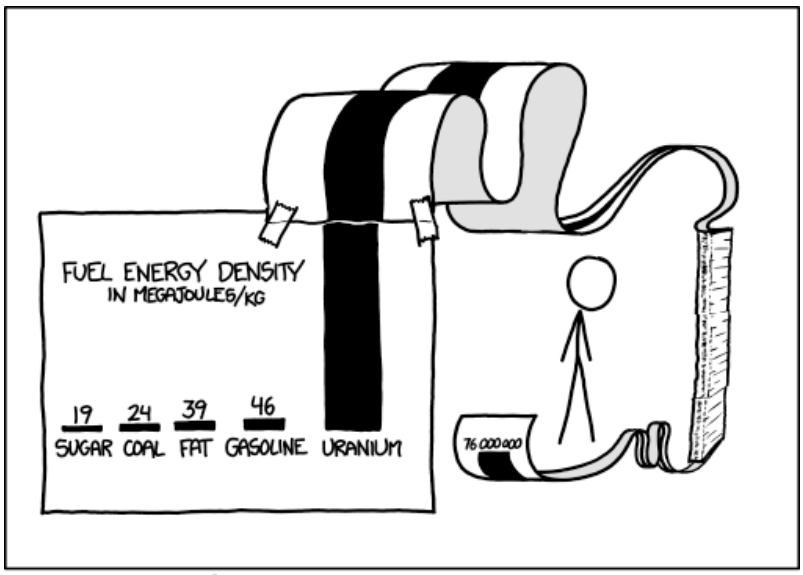

Figure 24: http://xkcd.com/1162 by Randall Munroe

You've probably already figured out that you can use MakeSelector on the histogramList n-tuple, just like you did on page 38. The challenge will be putting together the code inside the Process method of the new class with code from the previous exercise.

In the previous exercise, perhaps you hard-coded the number of histograms in the folder. Don't do that here. You could get the number of histograms from the number of entries in the n-tuple.

Or maybe that's not a good idea; what if there were an entry in the n-tuple but no corresponding histogram? Keep a separate count of the number of "valid" histograms you're able to read. This means you'll have to check if you've read each histogram correctly. Tip: If a ROOT operation to read a pointer fails, that pointer will be set to zero (see page 82).

Part Five – Expert Exercises

Exercise 14: A brutally realistic example of a plotting task (1-2 hours)

Now take a look at folder example3. You probably already looked in there, and were overwhelmed with the number of histograms.

Here's the task: it's another x-y plot, with the y values and error bars from fitting the histograms. You only want to include those histograms whose names begin with "plotAfterCuts"; the other histograms you can ignore.

The x values come from the histograms themselves. Double-click on a few histograms to plot them. You'll see that the x values are in the titles (not the names!) of the histograms.

> You'll be able to re-use code you developed for the previous two exercises. There are some new problems to solve: how to get the list of all the histograms in the example3 folder, how to test if a histogram's name begins with "plotAfterCuts", and how to convert a histogram's title from string form to a number.
>
> Let's think about the easier problems first.
>
> If you're fairly familiar with C or C++, you probably already know how to convert strings into numbers. If you're not, then I suggest you take a look at the description of the TString class on the ROOT web site; the Atof() method looks interesting.
>
> The TString class is pretty good about converting string formats implicitly.[61] You probably already figured out how to look up getting the title from a histogram. The method returns "const char *" but something like this will work:

```
TString title = histogram->GetTitle();
```

> What about testing if the text in a TString begins with "plotAfterCuts"? Take another look at the TString web page. Is there a method that looks like it might help you with that test?

---

[61] Yet another digression: There are three main ways of handling strings in ROOT/C++:

- The original way from the older language C, as an array of char: `char oldStyleString[256];`
- A newer way, added to the C++ language: `std::string newStyleString;`
- The ROOT way: `TString rootStyleString;`

Which is better? My attitude is that none of them is best. In a ROOT program, I tend to use TString; if my program doesn't use ROOT, I use std::string for string variables and arrays of char for constant strings.

Until recently, C++ didn't have the built-in text manipulation facilities of languages like perl or python. This can be important in a physics analysis procedure; while your calculations are based on numbers, manipulating files or program arguments can be based on strings. The latest language update, C++11, has a "regex" library for handling regular expressions; this can also be found in ROOT's cling.

Exercise 14 (continued)

The next problem is trickier: How do you get a list of objects in a directory?

By now you've got the hang of the above hint: I want to "Get" a "List" of objects in a directory. When I worked on this problem, I went to the TFile web page and looked for methods with names that began with "GetList". Nothing there, so I went to the parent class TDirectoryFile, continuing to search for "GetList." I found something, clicked on the name of the method… then pounded my head against the desk.[62]

I finally got the answer by using the UNIX grep command to search through the ROOT tutorials directory for the text "GetList". There are many files there with a "GetList…" call, but one file name stood out for me. Since I had read the TList web page first, I could see that the answer was there. But it's sloppily written and you'll have to change it.

To understand what you'd have to change, consider a chain of class inheritance that's similar to what you looked at on page 69:

Go to the web page for class TH1D. It inherits from two classes, TH1 and TArrayD; ignore the latter for now. Click on the TH1 link to see from which classes TH1 inherits. Eventually you'll work your way up to the class TObject.

Go back to the TH1D page, and search for the "Class Charts" section. You'll see a diagram of the inheritance; you'll note that there's another class, TProfile, which inherits from TH1D.

In C++, the practical aspect of class inheritance is that you can use a pointer to a base class to refer to a derived class object; if class Derived inherits from class Base, you can do this:

```
Base* basePointer = new Derived();
```

If that's a little abstract for you, consider this in terms of the classes with which you've worked. Any of the following is correct in C++:

```
TH1D* doublePrecisionHistogram = new TH1D(…);
TH1* histogram = new TH1D(…);
TObject* genericRootObject = new TH1D(…);
```

Why does this matter? Because ROOT does not read or write histograms, functions, n-tuples, nor any other specific object. *ROOT reads and writes pointers to class TObject.* After you read in a TObject*, you'll probably want to convert it to a pointer to something useful.

---

[62] Optional editorializing again: If you followed the steps I just described, you saw what I did: there's nothing there. Maybe the routine's author planned to put in some documentation later.

Here's a tip for writing code that will make you a hero: "later" does not exist. (As of 2015, the ATLAS collaboration has collected 25 fb^{-1} of data, and they still haven't discovered evidence of "later"!) Treat the comments as part of the code-writing process. If you have to edit the code, edit the comments.

Yes, I know it's a pain. But pounding your head on a desk is a bigger pain. It's the biggest pain of all when you realize that you wrote the code six months ago, have completely forgotten what it means, and must now spend an hour figuring it out. It would have taken five seconds to write a comment.

Exercise 14 (continued)

In C++, the simplest way to attempt to convert a base class pointer to a derived class pointer something like this (assuming genericRootObject is a TObject*):

```
TH1* histogram = (TH1*) genericRootObject;
If ( histogram == 0 )
{
   // The genericRootObject was not a TH1*
}
else
{
   // The genericRootObject was a TH1*; you can use it for things like:
   histogram->FillRandom("gaus",10000);
   histogram->Draw();
}
```

If I didn't put that test in there and just tried histogram->FillRandom("gaus",10000), and histogram==0, then the program would crash with a segmentation fault.[63]

Figure 25: http://xkcd.com/371 by Randall Munroe

Why did I just take one-and-half pages to go over such a dry topic?

- Understanding object inheritance makes it clear why the macros that ROOT automatically creates for you use pointers, why those container classes you saw on page 73 only contain TObject*, and other aspects of ROOT.
- It's so when you see a line like this in the ROOT tutorials, you have an idea of what it's doing: using a TKey to read in a TObject*, then converting it to a TH1F*:

```
h = (TH1F*)key->ReadObj();
```

Now you should have an idea of how to edit this line to do what you want to do… and how to check if what you've read is actually a histogram, or is some other object that was placed inside that folder.

---

[63] If you haven't encountered a segmentation fault yet in this tutorial, you're either very lucky or very good at managing your pointers. Now you know why it happens: someone tried to call a method for an object that wasn't there.

Exercise 15: Data reduction (1–2 hours)

Up until now, we've considered n-tuples that someone else created for you. The process by which a file that contains complex data structures is converted into a relatively simple n-tuple is part of a larger process called "data reduction." It's a typical step in the overall physics analysis chain.

As I implied in the first day of this tutorial, perhaps you'll be given an n-tuple and told to work with it. However, it's possible you'll be given a file containing the next-to-last step in the analysis chain: a file of C++ objects with data structures. You'd want to extract data from those structures to create your own n-tuples.

Copy files whose names contain "Example" from my `root-class` directory:

```
> cp ~seligman/root-class/*Example* $PWD
```

The file exampleEvents.root contains a ROOT tree of C++ objects. The task is to take the event information in those C++ objects and reduce it to a relatively simple n-tuple.

First, take a look at ExampleEvent.h. You're not going to edit this file. It's the file that someone else used to create the events in the ROOT tree. If you're given an ExampleEvents object, you can use any of the methods you see to access information in that object; for example:

```
ExampleEvent* exampleEvent;
// Assume we fill in exampleEvent somehow.
Int_t numberLeptons = exampleEvent->GetNumberLeptons();
```

For this hypothetical analysis, you've been told that the following information is to be put into the n-tuple you're going to create:

- the run number;
- the event number;
- the total energy of all the particles in the event;
- the total number of particles in the event.
- a boolean indicator: does the event have only one muon?
- the total energy of all the muons in the event;
- the number of muons in the event;

The task is to write the code to read the events in exampleEvents.root and write an n-tuple to a different file, exampleNtuple.root.

(continued on the next page)

Exercise 15 (continued)

After what you've done before, your first inclination may be to open exampleEvents.root directly in ROOT and look at it with the TBrowser. Try it.

It doesn't fail, but you'll get an error message about not being able to find a dictionary for some portions of the ExampleEvent class.[64] I mentioned this earlier in footnote 21 on page 28: it's possible to extend ROOT's list of classes with your own by creating a custom dictionary. Only classes that have a dictionary can be fully displayed using the ROOT browser.

Try to see how much of the ExampleEvent tree you can see without the dictionary. Then restart ROOT and type the following ROOT command:

```
[] gSystem->Load("libExampleEvent.so");
```

This causes ROOT to load in the code for a dictionary that I've pre-compiled for you.[65] Now you can open the exampleEvents.root using a TFile object and use the ROOT browser to navigate through the ExampleEvent objects stored in the tree.

As you look at the file, you'll see that there's a hierarchy of objects. There's only one object in the file, exampleEventsTree. Inside that tree, there is only one "branch", exampleEventsBranch.

That's a bit of a clue: a ROOT n-tuple is actually a TTree object with one Branch for every simple variable.

(continued on the next page)

---

[64] If you didn't get such a message, then you probably copied my entire `root-class` directory to your working directory. That's OK, but you might want to temporarily create a new directory, go into it, start ROOT, and open the file just so you can see the error message. That way you'll know how it looks if you have a missing-dictionary problem.

[65] This library may not work if you're on a different kind of system than the one on which I created the library. If you get some kind of load error, here's what to do:

Copy the following additional files from my `root-class` directory if you haven't already done so:

LinkDef.h
ExampleEvent.cxx
BuildExampleEvent.cxx
BuildExampleEvent.sh

Run the UNIX command script with:

> sh BuildExampleEvent.sh

This will (re-)create the libExampleEvent shared library for your system. It will also create the program BuildExampleEvent, which I used to create the file exampleEvent.root.

If you're running this on a Macintosh, the name of the library will be libExampleEvent.dylib; that's the name to use in the gSystem->Load() command in the Mac version of ROOT.

Exercise 15 (continued)

At this point, you could use MakeSelector() to create a ROOT macro for you, but I suggest that you only do this to get some useful code fragments to copy into your own macro.[66]

I'll offer you the following additional hints:

- The first line of your ROOT macro for this exercise is likely to be the library load command on the previous page.
- If you're writing a stand-alone program, instead of loading the library you'll have the line:

```
#include "ExampleEvent.h"
```

and include libExampleEvent.so on the line you use to compile your code.

- Look at the examples in the tutorials/tree directory, on the TTree web page, and in the macro you created with MakeSelector (if you chose to make one).
- Yes, the ampersands are important!

One more hint:

How do you tell if a lepton is a muon or an electron? I'm not talking about their track length in the detector, at least not for this example. I'm talking about what indicator is being used inside this example TTree.

There's a standard identification code used for particles. The Particle Data Group developed it, so it's called the "PDG code". There are methods in the TTree that return this value (e.g., `LeptonPDG`). You can find a complete list of codes at http://pdg.lbl.gov/2002/montecarlorpp.pdf. For this exercise, these will do:

Particle	PDG Code
e^-	11
e^+	−11
μ^-	13
μ^+	−13

If the sign of the PDG codes for leptons seems puzzling to you, recall that under the usual particle-physics nomenclature, electrons are assigned a lepton number L of +1, positrons are assigned L=-1, and so on.

Get to work!

---

[66] Why don't I want to you use MakeSelector here? The answer is that some physics experiments only use ROOT to make n-tuples; they don't use it for their more complex C++ classes. In that case, you won't be able to use MakeSelector because you won't have a ROOT dictionary. It's likely that such a physics experiment would have its own I/O methods that you'd use to read its physics classes, but you'd still use a ROOT TTree and branches to write your n-tuple.

Wrap-up

The last four exercises that make up Parts Four and Five are difficult. I chose those tasks because they represent the typical kind of work that I find myself doing whenever I use ROOT: pulling together documentation from different places, translating the examples into the work I'm actually doing… and pounding my head against the desk whenever there are no comments, or I get yet another segmentation fault.[67]

If you'd like to see how I solved those same exercises, you'll find my code in PlotGraphs.C (for exercises 13-15) and MakeNtuple.C (for exercise 16).[68]

Good luck![69]

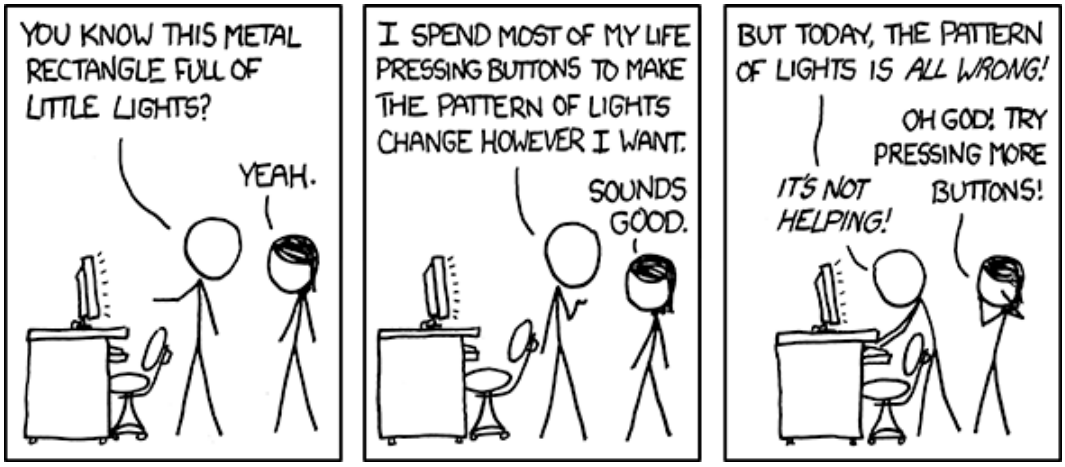

Figure 26: http://xkcd.com/722 by Randall Munroe
Alt-text: "This is how I explain computer problems to my cat. My cat usually seems happier than me."

---

[67] Now you know the reason for my going bald!

[68] Maybe you're thinking, "Wow! It's lucky I turned to the last page before I actually started doing any of the work!" Take my word for it: reading my solutions is not a substitute for working through the problem yourself.

[69] Total lifetimes used up: up to nine, depending on you chose to learn both ROOT/C++ and pyroot, which tangents you took, how much LaTeX you study, and whether you devote yourself to physics. I generously give any remaining lives back to you.