ROOT Workshop 2017

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.

Many parts of this tutorial are optional, or are advanced exercises. No one expects you to get through all 95 pages before you start your physics work for the summer.

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.

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++ or Python might barely get through all six 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

2017

We now have a Jupyterhub-based notebook installation available to Nevis students. I've incorporated this into the lessons. It's now a six-part course, but the part introducing the notebook is quite short.

2016

I've edited the Python portion to use IPython instead of the "vanilla" Python console.

The ROOT web site has changed, and its class documentation is now even worse than it was before. (Yay!) I've done my best to revise this course for those changes.

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 shell 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 Python part of this tutorial, the prompt is "In []". For example:

```
> ipython
In[] 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**.
>
> ROOT, Python, and Jupyter will put a session line number in brackets; e.g., [0], [1], [2]; In [0], In [1], In [2]. I'll omit the line numbers from this tutorial.
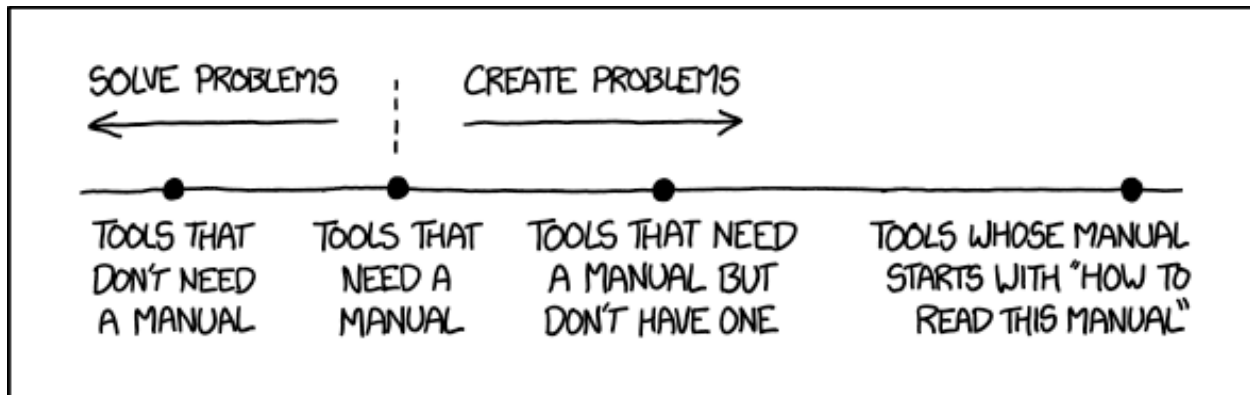
Figure 1: http://xkcd.com/1343 by Randall Munroe

Part One – The Basics

Getting started with Linux on the Nevis particle-physics systems

If you're sitting in front of a desktop 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 bookmark 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**

If you see a prompt about initializing zsh, just hit "q".

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/.

Connecting to the notebook server

This will be described in detail in Part Two. In short: go to https://notebook.nevis.columbia.edu (note that it's "https", not just "http") and enter your Nevis Linux cluster account name and password.

Installing ROOT on your laptop

Don't.

Here's the longer response, in case the above paragraph is too short and snarky for you: ROOT is not a pre-packaged app. You need to have familiarity with your OS command line and a formal understanding of shell scripts within that OS to install even the pre-compiled versions. If you want to compile ROOT from source on your laptop, you may experience difficulties integrating it with your Python distribution.

Obviously it's possible to install ROOT on a laptop (I do it all the time), but it can take a while to learn how to do it. You'd learn a lot about UNIX, but you'll be learning neither ROOT nor physics. Don't expect me to break from teaching other students about ROOT to teach you about your C++ compiler and the location of your Python distribution in your directory hierarchy.

---

[1] That's dash capital-X, dash capital-Y. 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
```

Don't bother including them if you're **ssh**'ing into your workgroup's server from the desktop systems in the Nevis research building; they're the default.

Still not convinced? Consider these installation tips:

ROOT is available for Mac OS X, Linux, and Windows at https://root.cern.ch/downloading-root. Download the "Pro" binaries for your operating system. Read all the installation directions.

In Mac OS X 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:[2]

```
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 valid for UNIX-style systems running sh-type shells. If this is not the case for you, you have some detective work ahead! If you're using bash (default on Mac OS X), you'll want to add these lines to `~/.bash_profile`; on Linux add them to `~/.bashrc`.

You also need to install Python if you're working in that language. 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.

Did you understand the above instructions? Did you immediately grasp that you have to replace the <text in angle brackets> with a directory path? If you didn't answer "yes" to both those questions, please don't try to install ROOT on your laptop during this ROOT course.

Installing Jupyter on your laptop

Don't.

This is a different snark that the previous one. If you have Python, then the command

```
> pip install jupyter
```

might install Jupyter for you.[3] However, it won't integrate ROOT into the Jupyter environment; that requires extra steps that depend on how you've installed both Python and ROOT.[4]

With all the hassle, I suggest logging into the desktop systems or using the notebook server for this course.

Or to put it in a non-snarky way: The reason why I installed ROOT on our desktop systems and prepared the notebook server is so you can spend less time on software installations, and more time on learning how to use the tools to do physics. Let's do physics.

---

[2] On Nevis particle-physics systems, these environment variables are set when you type **module load root**.

[3] Maybe. Or maybe not. For example, sometimes it takes a separate procedure to install **pip**. Or you may need administrative privileges to modify your Python installation. Or the jupyter program will be installed, but with the name **jupyter-2.7**.

[4] If you install ROOT with Python libraries, you can type **root --notebook** to get a ROOT C++ notebook. If you've set up ROOT and you've installed Jupyter, you can type **jupyter notebook** and be able to **import ROOT** within the Python environment (see page 61). Both of these commands require a correctly configured ROOT installation and shell environment set-up.

A Brief Intro to Linux

If you're already familiar with Linux, skip or skim 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 2: 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:[5]

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

In UNIX, the $PWD means the results of the **pwd** ("print working directory") command.[6]

To look at the contents of a text file: Use the **less** command.[7]

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.

---

[5] It's always "~seligman" (tilde-seligman), never "–seligman" (dash-seligman). Depending on the exact font used to print or display this tutorial, sometimes tildes look like dashes. On most keyboards, tilde is typed with SHIFT-` where ` (backtick) is on the upper-left-hand corner of the keyboard. In UNIX ~jsmith means "the home directory of the jsmith account." Just plain ~ means your own home directory.

[6] 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.

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

A Brief Intro to Linux (continued)

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`**.[8]

> 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.[9] 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.
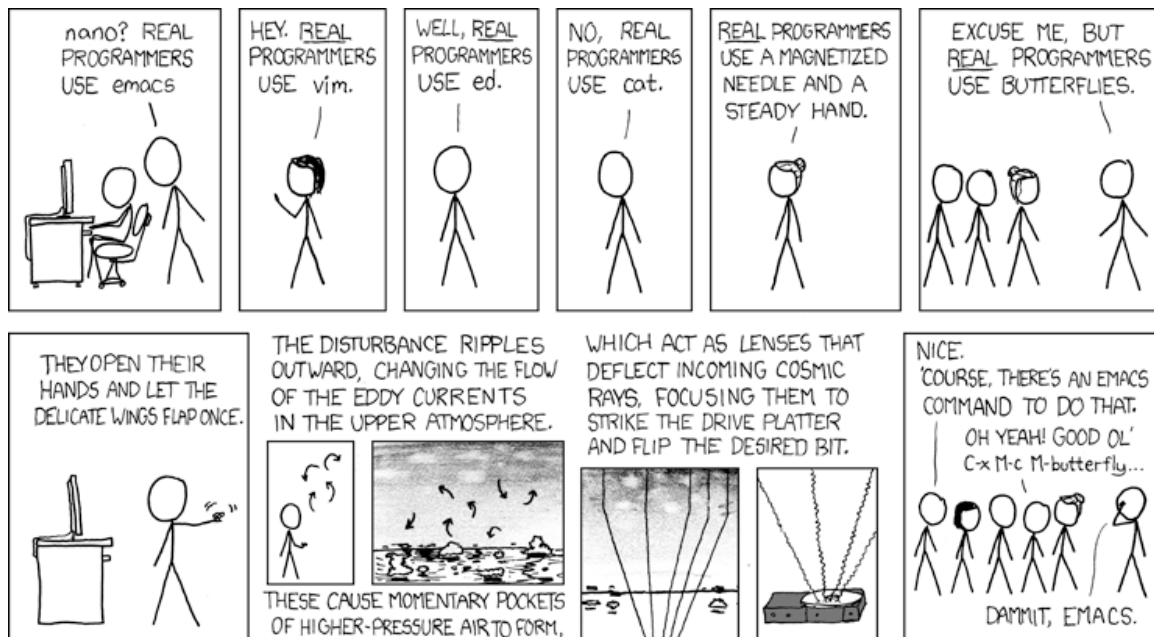
Figure 3: http://xkcd.com/378 by Randall Munroe
If you're feeling bored, type Meta-x butterfly in emacs and see what happens.

---

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

[9]   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 in ROOT, IPython, 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 were known outside of Xerox PARC.

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.[10]

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, navigation keypresses, and cut-and-paste, you can edit your previous commands for new tasks.

Don't get too 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 file-manager window and let UNIX pick an application for you.

This GUI advice won't apply if you start using ROOT notebooks. We'll get to that later.

---

[10] 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.[11] 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 78, for example) you'll be able to do this only after you've typed **module load 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).

Warnings:

- 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.[12] 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.

- Finally, do not put **module load root** in a start-up script if you're using the notebook server. You'll get lots of "not found" errors.

---

[11] That's three lifetimes so far.

[12] The **module load** UNIX command is part of a package called "environment modules." Though it's a standard package, environment modules are not normally included in a UNIX installation. You can read more about this at https://twiki.nevis.columbia.edu/twiki/bin/view/Main/EnvironmentModules.

Walkthrough: Starting ROOT (5 minutes)

You are going to need at least two windows open during this part of the 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:[13]

```
> 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" text will appear 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 **.help** 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.

---

[13] I'm starting with "basic" ROOT, which has a command syntax based on C++. For Python users, we'll explore pyroot later. For these simple examples, the ROOT commands are almost the same in both languages anyway.

Walkthrough: Plotting a function (15 minutes)

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.[14] ROOT may help you out with context-based colors for the keywords it recognizes. In C++ notation, the first command says: Create an object ("f1") that is a TF1 (we'll get to what that is in a moment) with some properties (name, function, low range, high range). The second command tells f1 to draw itself.
>
> When you type in the first command, you may see something like

```
(TF1 &) Name: func1 Title: sin(x)/x
```

> Don't worry about this. It's not an error.[15]
>
> If you have a keen memory (or you type **.help** 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 **6.08 - browse,** then on **All Classes**, then on **TF1**; you may want to use the browser menu **Edit->Find** and search on **TF1** to locate that link.[16] 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 c1. "Canvas" is ROOT's term for a window that contains ROOT graphics; everything ROOT draws must be inside a canvas.[17]

(continued on the next page)

---

[14] 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.

[15] If it's not an error, what is it? ROOT is printing out the type (TF1 &) and information about the object you've just created. When you become more familiar with programming, you'll see that ROOT is printing out the result of creating the TF1 object, in the same way it would print the result if you typed 2+3.

[16] Assuming that ROOT hasn't reorganized their web site (which they do periodically) since I last reviewed this tutorial, the link to the list of classes is https://root.cern.ch/doc/v608/annotated.html and the link to TF1 is https://root.cern.ch/doc/v608/classTF1.html.

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

Position the mouse over the function itself (it will turn into 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.[18]

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**.

> 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.
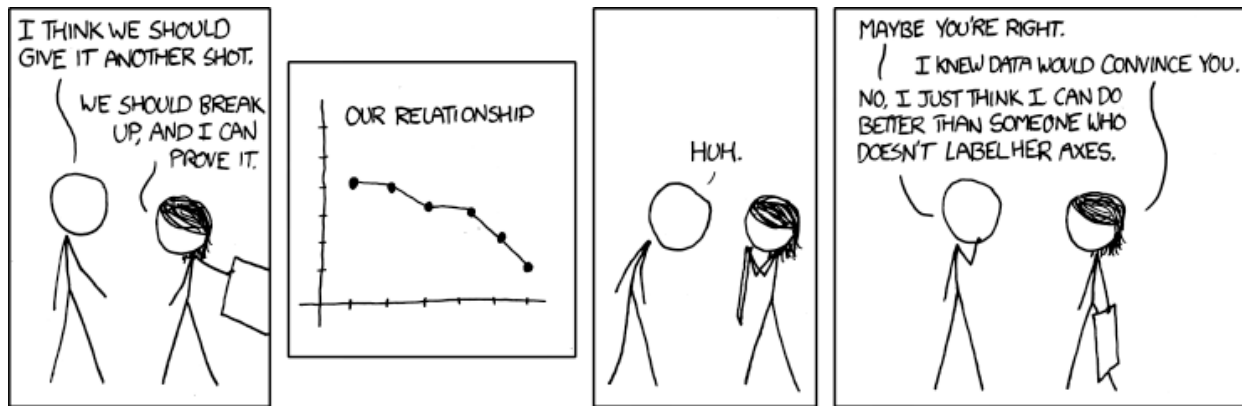
Figure 4: http://xkcd.com/833/ by Randall Munroe
Alt-text: "And if you labeled your axes, I could tell you exactly how MUCH better."

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**.

---

[18] Did you get something funky instead? You probably right-clicked on the axis, not the function. Quit ROOT and start from the beginning. Question: Why did the graph change in such an unexpected way? For the answer, click on the question mark in the axis SetRange item.

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.

If you want to change the color of the function, right-click on the function and select **SetLineAttributes**.

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:[19]

sin(x)*sin(y)/(x*y)
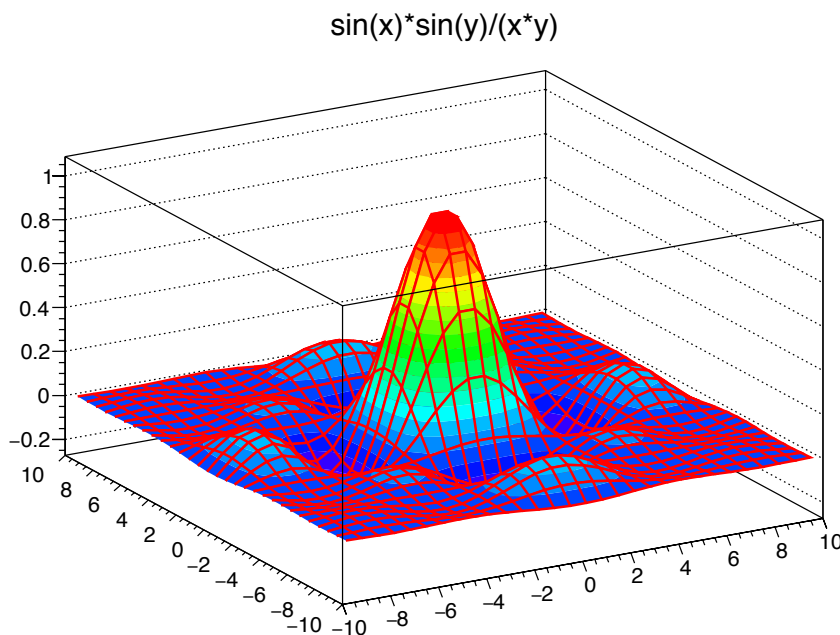
Figure 5: Some detective work is required to duplicate this plot.

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?

With your first try, you probably 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` and `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 Five and Six of this tutorial. Finding the "surf1" option is trivial by comparison!

---

[19]   The colors don't have to be the same, since the default colors change in different ROOT versions.

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.[20]

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?

---

[20] *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.[21] 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.

---

[21] *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: 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 `c1`, so the file "c1.C" is created. In your UNIX window, type

```
> less c1.C
```

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

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

Finally, select **Save->c1.root** from the same canvas to create the file "c1.root". Quit ROOT with the **.q** command, and start it again.

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

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

> This is your first experience with a ROOT "macro," a stored sequence of ROOT commands that you can execute at a later time. One advantage of the ".C method" is that you can edit the macro file, or cut-and-paste useful command sequences into macro files of your own.[22]

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

```
> root c1.C
```

Quit ROOT and print out your Postscript file with the command

```
> lpr –Pbw–research c1.pdf
```

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

> Not only is the PDF format useful if you want to print something, but it's usually simple to embed a PDF file in a paper or a presentation. You can't embed a ROOT macro in a Powerpoint document and expect to see its graph!

---

[22] 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.

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

```
[] TBrowser tb
```

In the left-hand pane, scroll to the folder with the same name as your home directory.[23] 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 **c1.C** to see what happens.

Now double-click on **c1.root**, then double-click on **c1;1**.

> Don't see anything? Click on the **Canvas 1** tab in the browser window.

> What does "c1;1" mean? You're allowed to write more than one object with the same name to a ROOT file (this topic is part of a 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; I sometime find that going through the nested web pages on the ROOT web site via Firefox to be too much of a hassle.

> As nifty as the ROOT browser is, for 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.

---

[23] The folder hierarchy may be puzzling to you; your home directory will be in `/nevis/milne/files/<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:[24]

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

Go back to your TBrowser 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 `c1` 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)

---

[24] 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/

Function to be fit

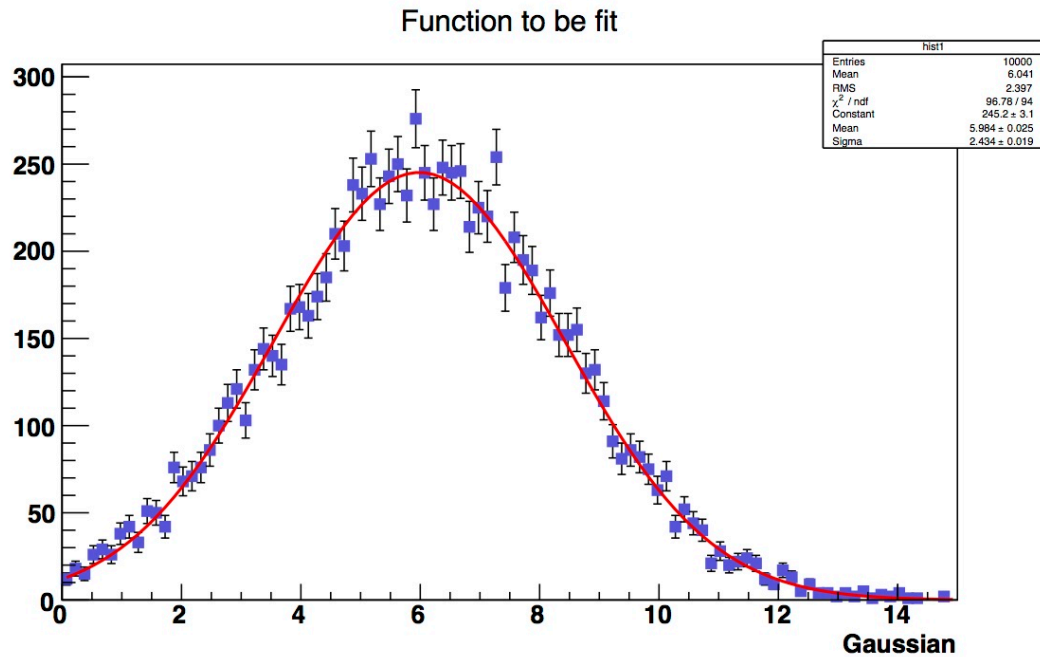| hist1	
Entries	10000
Mean	6.041
RMS	2.397
χ^2 / ndf	96.78 / 94
Constant	245.2 ± 3.1
Mean	5.984 ± 0.025
Sigma	2.434 ± 0.019

Gaussian

Figure 6: The resulting plot should look something like this.
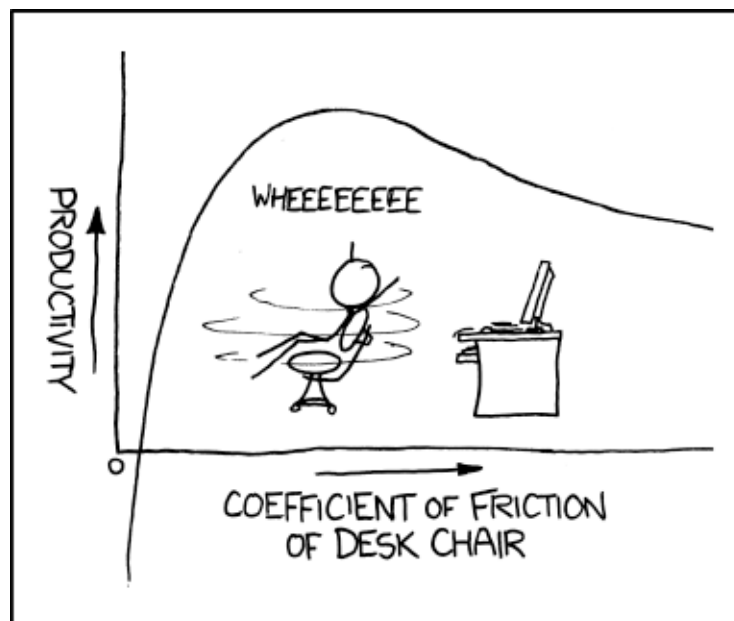
Figure 7: It will look nothing like this. This would be a poor fit for your function.
http://xkcd.com/815 by Randall Munroe
Alt-text: "As the CoKF approaches 0, productivity goes negative as you pull OTHER people into chair-spinning contests."

Walkthrough: Fitting a histogram (continued)

As a check, click on **landau** (which vaguely resembles the plot in Figure 7) 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 Five and Six) or more complex functions, you want to learn the ROOT commands.

To fit hist1 to a gaussian, type the following command:[25]

```
[] 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)

---

[25] 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 Python. See page 36 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.[26]

```
[] 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.

---

[26] 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: For now, 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/AnalyzeVariables.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.

When we get to Part Two, you'll start cutting-and-pasting commands into notebooks on a regular basis.

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 c1.root,** quit ROOT, restart it, then load canvas "c1;1" from the file. You'd get your histogram back with the function superimposed… but it's not obvious where the function is or how to access it now.

What if you want to save your work in the same file as the histograms you just read in? You can do it, but not by using the ROOT browser. The browser will 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.[27]

```
[] 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 45. I wanted to point it out so that you were aware that, though they seem closely connected at times, C++ and ROOT are two *different* entities.

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

---

[27] *In case you care:* ROOT stores your 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).
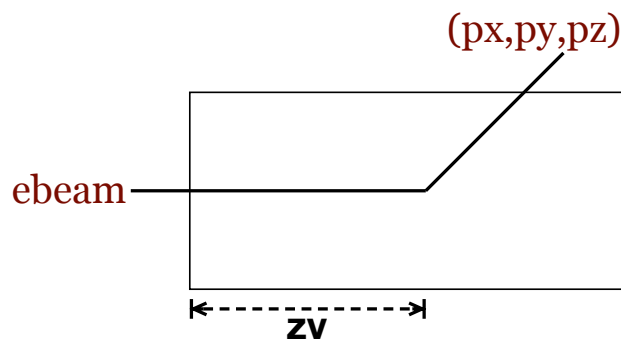
Figure 8: Sketch of the experiment and variables.

(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.[28] 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 tutorial.
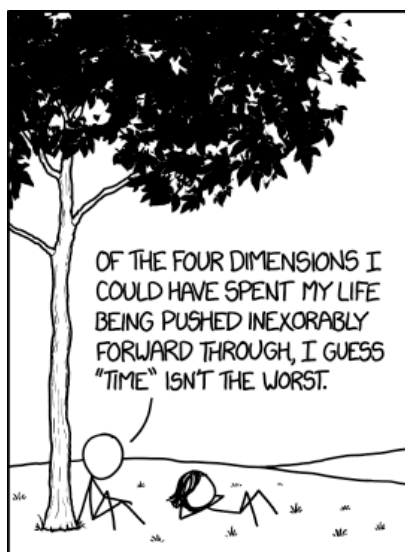
Figure 9: http://xkcd.com/1524 by Randall Munroe
Alt-text: "I would say that time is one of my top three favorite dimensions."

---

[28] *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 Six of this tutorial.

Using the Treeviewer [29]

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 Python 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.[30] Double-click on one of them and look the resulting histogram. Double-click on a few more variables and see how the histogram changes.
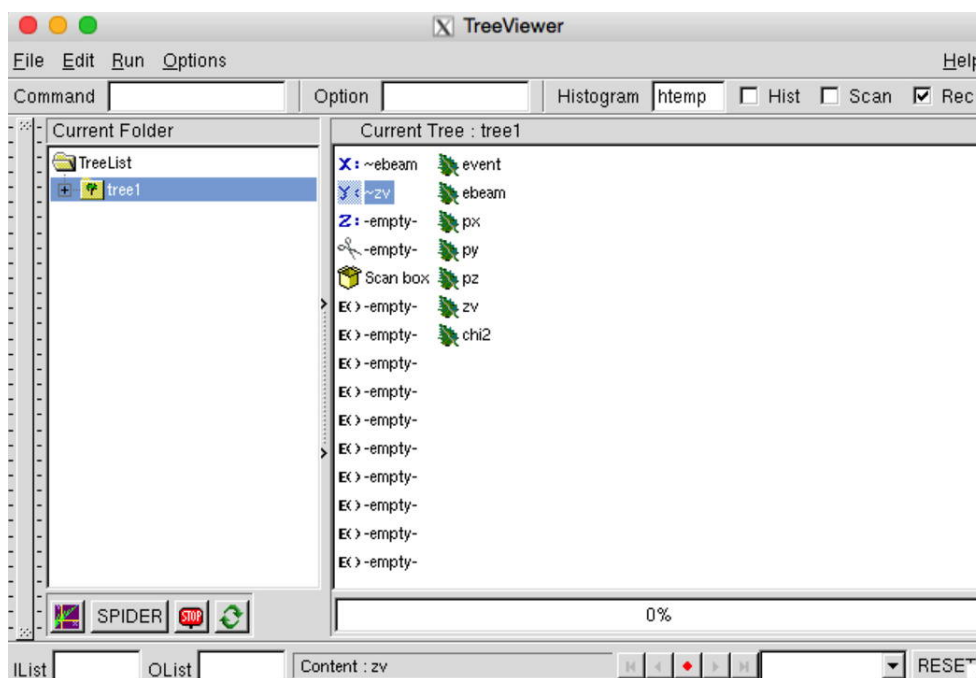
Figure 10: This is what I see when I run TreeViewer on my Macintosh.

---

[29] 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.

[30] 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"[31]).

> 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.
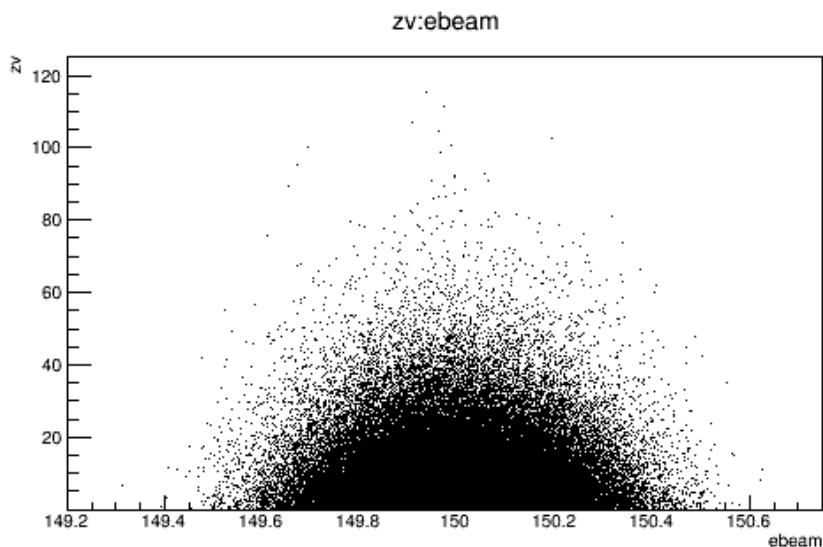
Figure 11: This is what I see when I make a scatterplot of zv versus ebeam. The variables look uncorrelated to me, subject to the restriction that we can't have zv < 0.

---

[31] 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"[32] 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". [33]

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 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.

---

[32] That first character is a tilde (~), not a dash.

[33] 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. [34]
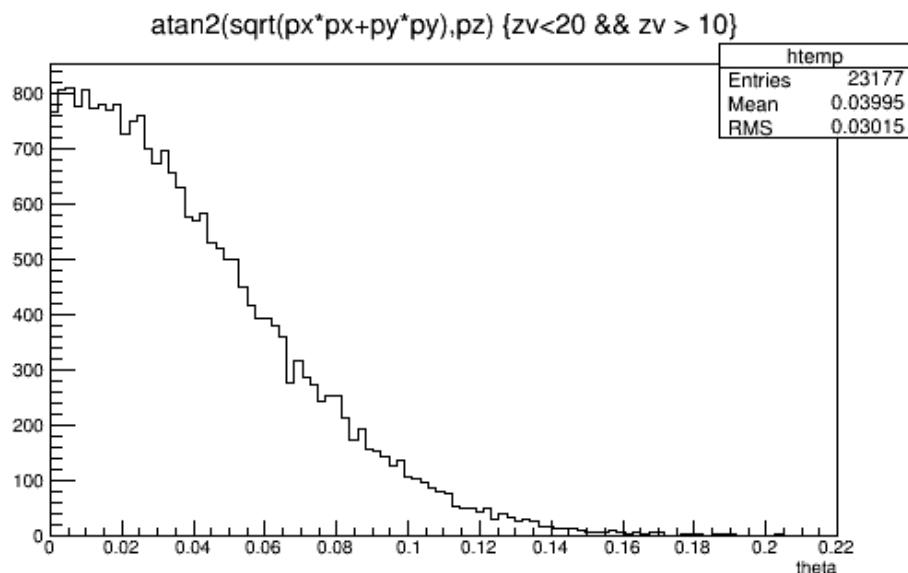
Figure 12: This is what I see when I make a plot of theta with the cut "zv<20 && zv>10".

(continued on next page)

---

[34] 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 16) 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 c1.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 **InsertText** and type in your new title.

Figure 13: 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.

Part Two – The Notebook Server

If you're familiar with Jupyter or IPython, you can skim or skip this part.

Now I'm going to introduce a different software development tool, the notebook. It's independent of ROOT, but it can be handy for creating ROOT programs.

Starting with Jupyter (5 minutes)

In any web browser (laptop, desktop, phone), go to https://notebook.nevis.columbia.edu.[35] You'll be prompted for your Nevis Linux cluster account name and password.[36]

When you visit `notebook` for the first time, you'll see your home directory. You can perform some elementary file operations from this screen: check the box next to a filename, and you'll see an option near the top of the screen to rename or delete the file. The "Upload" button near the top left allows you to copy files from the computer you're using to the Nevis cluster.

The fun part is in the pop-up menu you get from clicking the "New" button near the top left:

- "Text File" will give you a basic text editor. You will also get a text editor if you click on a text file on the home directory page. The "Edit" menu within the editor page will let you select which text editor you use; from page 11 you know my favorite editor is Emacs, but you can use whatever you wish.

- "Folder" lets you create a new sub-directory.

- "Terminal" will give you a "login not available" screen, since users are not allowed to login directly to `notebook`. For a UNIX shell, login to your workgroup's server.

- And then we have the notebook kernels…

  In Jupyter, a "kernel" is an environment for interpreting commands. I installed lots of kernels on the notebook server for users to explore,[37] but for this tutorial there are only two of interest: "Python 2" (which includes an interface to ROOT) and "ROOT C++".

Figure 14: https://xkcd.com/1202/ by Randall Munroe
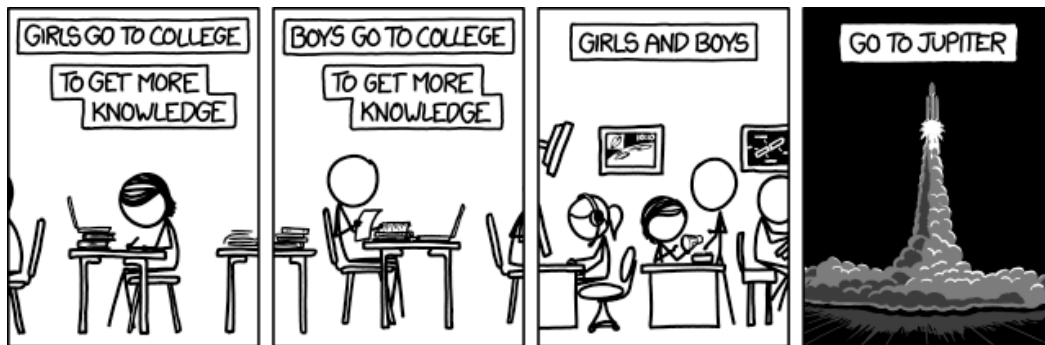
---

[35] Take care: it's "https", not just "http".

[36] If you don't have an active account at Nevis, then you won't be able to login. You'll have to install Jupyter on your own system, or proceed without it; go on to the next section.

[37] For a description of each of the kernels, see https://twiki.nevis.columbia.edu/twiki/bin/view/Main/IPython.

Your first notebook (10 minutes)

From the "New" menu, select Python 2.[38] You'll see your first empty cell, labeled **In [1]**. At the top of the page, you'll see that the name of the notebook is "Untitled". The first thing you should do when creating a new notebook is to give it a new name.[39] Go to the **File** menu within the Jupyter page and select **Rename…**. Pick any name you wish, such as "pythontest".

After you've renamed the notebook, go back to the Jupyter Home window. You'll see the notebook file with the extension ".ipynb".

Go back to the notebook window. Under the **Help** menu, take the **User Interface Tour** (it's about a minute long). Note the **Keyboard Shortcuts**.[40]

Cut-and-paste the following into that first cell.[41]

```
from ROOT import TH1D, TCanvas
my_canvas = TCanvas("mycanvas","canvas title",800,600)
example = TH1D("example","example histogram",100,-3,3)
example.FillRandom("gaus",10000)
exomple.Draw("E")
my_canvas.Draw()
```

> This code is in Python, but after going through Part One of this tutorial you can probably figure out what most of these lines are supposed to do.

To "execute" the contents of a given cell, hit SHIFT-ENTER with your cursor in that cell. Do that now.

Oops! There's an error. Fix the error in the cell and hit SHIFT-ENTER again.

> Assuming there have been no mistakes, you should see a histogram embedded in the web page.

> There are also a couple of warning messages:

```
Warning in <TCanvas::Constructor>: Deleting canvas with same name:
mycanvas
Warning in <TROOT::Append>: Replacing existing TH1: example (Potential
memory leak).
```

> Let's think about what those messages mean. When you execute lines in a cell, your environment doesn't "start fresh"; everything you defined before is still there. ROOT is warning you that the TCanvas and the histogram are being overridden.

> In Python, you can usually ignore these warnings.

---

[38] Don't pick "Python 3". ROOT is not yet fully compatible with Python 3, and I didn't install ROOT in it.

[39] You don't *have* to do this. But if you don't, your home directory will soon be littered with notebooks named Untitled.ipynb, Untitled1.ipynb, Untitled2.ipynb, etc., and you won't know what's in any of them.

[40] There are more Jupyter hints at https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/. Some of those tips won't work on the Nevis systems; e.g., you can't install new packages on your own. The last tip, on different ways to share notebooks, will be helpful to those who work with folks who don't use Jupyter.

[41] You can type it in manually if you want, but (a) that's a lot of typing, and (b) you have to make sure you get each character correct for the sake of this example.

Click in the next cell and cut-and-paste this line, then hit SHIFT-ENTER:

```
exampleFit = example.Fit("gaus")
```

Note that this next cell recognizes the histogram object you defined in the previous cell. This gives you some idea of one feature of notebooks: You can fiddle with something in a given cell until you does what you want, then move on to the next phase of your task that depends on the previous cell.

Wait a moment… We just added a fit to the histogram, but the plot didn't change. Maybe we have to plot it again.

Enter this line after the one you just pasted, or into a subsequent cell, and hit SHIFT-ENTER:

```
example.Draw()
```

No new plot, and the plot above it still didn't change. What's wrong? Nothing. Jupyter runs in a web browser, and browsers behave differently than X-Windows (the underlying graphics protocol of UNIX). You may have noticed that, unlike the ROOT plots in Part One, the shape of the cursor doesn't change as you move it over the plot, and right-clicking on it brings up a browser menu, not a ROOT one. If you right-click on the plot and select **View Image**, you'll see that the plot is not a dynamic object, but a static .png file.[42]

How do we get a plot? You probably guessed the answer from what I had in the first cell. Paste the following line after that last Draw() command, or in a new cell:

```
my_canvas.Draw()
```

Finally, you see the histogram with the fit superimposed.

Remember page 15: ROOT plots everything in a canvas. In Jupyter, a TCanvas is not automatically drawn when its underlying plot updates. You have to explicitly draw the TCanvas yourself. That's why the first example contains the lines:

```
my_canvas = TCanvas("mycanvas","canvas title",800,600)
# … stuff …
my_canvas.Draw()
```

I had to define the TCanvas that would be used as the "target" of any Draw commands, then Draw that TCanvas in order for the plot to be displayed.[43]

---

[42] "PNG" stands for Portable Network Graphics. It's a standardized format for uncompressed images to be sent over the web. Jupyter uses that format instead of GIF because the GIF algorithm is patented.

[43] Since we haven't had to explicitly define our canvases before, I should mention: the canvas name and title are usually not important; the name only matters if you were to write the canvas to a file, and the canvas title is rarely displayed (as opposed to the histogram title, which appears at the top of the plot).

What matters is the size of the canvas. Here, I used 800 pixels wide and 600 pixels tall, which is the size of our old friend c1 that's automatically defined if you don't define a canvas yourself.

I could have defined the canvas using only defaults with

```
my_canvas = TCanvas()
```

but I thought that might be even more confusing to see for the first time.

Magic commands (5 minutes)

In Jupyter, "magic" refers to additional commands added by Jupyter to the kernel environment that aren't normally part of that kernel's language.[44] I'm going to start with a slightly exotic magic command because I think you'll find it useful.

In a new cell in the Python notebook we worked with above, execute this command:[45]

```
%jsroot on
```

As a general rule, magic commands begin with the percent sign "%".[46]

Draw the canvas again:

```
my_canvas.Draw()
```

Move the cursor over the new plot.

Ah, that's more like it! The plot is not interactive in the same way as in X-Windows ROOT, but you can get a lot done. Play around a bit, looking at tooltips and right-clicking. Note the faint icons below the lower left-hand corner of the plot.

If you execute **%lsmagic** you'll see a list of available magic commands. There's probably more here than you can absorb right now.[47] Here are examples of the magic commands I find to be the most useful:

```
%mkdir subdirectory
%cp ~seligman/root-class/jsroot-test.ipynb subdirectory
%ls subdirectory
%less c1.C
%man root
%cd subdirectory
```

The above commands are "line magics", which are executed line-by-line within a cell. There are also "cell magics" that affect the contents of the entire cell in which they appear; they must appear as the first line in a cell. They begin with a double "%". Examples: **%%writefile** <filename> (write the cell to file <filename>); **%%timeit** (execute the cell many times and determine the average execution time); **%%sh** (execute the cell as a UNIX shell script).

---

[44] No, nothing to do with Doctor Strange or Harry Potter, though you may find yourself muttering "Avada Kedavra" as you work with ROOT.

[45] Note that the **%jsroot** magic command is only available in Python-based notebooks after you've executed **import ROOT** or **from ROOT import**… In ROOT C++ notebooks it's built-in.

"JSROOT" is short for "Javascript ROOT"; it's an evolving project to bring more interactivity of ROOT graphics into web browsers. For more information, see https://github.com/root-project/jsroot/blob/master/docs/JSROOT.md.

[46] Well… not really. There's an option (%automagic on|off) that allows you to omit the leading %. In this tutorial I'll always include the % prefix to make it clear when a command is "magic".

[47] You can find a description of magic commands here: https://ipython.org/ipython-doc/3/interactive/magics.html. Ignore the "old version" warning, since you're using Python 2.7 with ROOT.

Markdown cells (5 minutes)

One of the hardest habits to get into is documenting your work. Jupyter makes it easy.

Click in an empty cell. Go to the pop-up menu near the top of the page that reads **Code**. Select **Markdown** from that menu. Now you can type plain text in that cell; e.g., "The following code sums all the histograms in the analysis." When you're done, hit SHIFT-ENTER to see the formatted result.

You can also include Markdown,[48] HTML,[49] and LaTeX[50] commands to format the text. Here are some examples: declare a cell to be Markdown, paste one of the following paragraphs into the cell, and hit SHIFT-ENTER:

Markdown

```
# 2017 Analysis Project
*Energy*, **time**, and `momentum` are all variables in this n-tuple.
```

HTML

```
<h1>2017 Analysis Project</h1>
<p><i>Energy</i>, <b>time</b>, and <tt>momentum</tt>.</p>
<p>The following code reads in an n-tuple.</p>
```

LaTeX

```
\begin{align}
\nabla \times \vec{\mathbf{B}} -\, \frac1c\,
\frac{\partial\vec{\mathbf{E}}}{\partial t} & =
\frac{4\pi}{c}\vec{\mathbf{j}} \\
\nabla \cdot \vec{\mathbf{E}} & = 4 \pi \rho \\
\nabla \times \vec{\mathbf{E}}\, +\, \frac1c\,
\frac{\partial\vec{\mathbf{B}}}{\partial t} & = \vec{\mathbf{0}} \\
\nabla \cdot \vec{\mathbf{B}} & = 0
\end{align}
```

Can you mix all of them in a single Markdown cell? Give it a try!

---

[48] Markdown is a simple text-layout layout that emphasizes readability over the methods described in the next two footnotes. There are a lot of tutorials on the web; here's one: https://help.github.com/articles/basic-writing-and-formatting-syntax/.

[49] HTML ("HyperText Markup Language") is the standard language for formatting content in web browsers. If you've never seen it before, it's because you've used some program that formats web pages for you into HTML (Markdown is one such program). My favorite HTML tutorial is at https://www.w3schools.com/html/.

If you want a couple of xkcd cartoons on HTML: https://xkcd.com/1341/ and https://xkcd.com/1144/.

[50] 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. Don't worry about learning LaTeX. No one writes a LaTeX document from scratch; they get one from someone and learn by example. It's much easier than learning ROOT. For some Jupyter-related examples, see http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Typesetting%20Equations.html.

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

The ROOT C++ kernel (5 minutes)

In Part One, all of the practice ROOT code used C++ syntax. Yet I switched to Python when I introduced Jupyter. Now you'll learn why.

Start a ROOT C++ notebook (either from your Jupyter home page, or from the **File** menu of your existing notebook). Rename it to "cplusplustest" or whatever you want. Paste the following into a cell and execute it.

```
TH1D example("example","example histogram",100,-3,3);
example.FillRandom("gaus",10000);
example.Draw();
```

You won't see anything, but after the explanation on page 36 you know why: you have to draw the canvas. The warning message says it drew the plot on TCanvas c1, so add the following line to the end of the above cell and hit SHIFT-ENTER:

```
c1.Draw();
```

Uh-oh. It's not just warning you that you're creating a new histogram with the same name. ROOT's C++ interpreter is treating it as an error and won't let you continue.

This is a general issue of Python vs. C++: Python is more forgiving. If you want to execute that cell, you'll have to restart ROOT. Fortunately, there's an easy way to do that.

Go to the **Kernel** menu on the page and select **Restart**. It will warn you that you're about lose all your variables, which in this case is exactly what you want. Click in the cell with your code and hit SHIFT-ENTER.

I'm being sneaky, aren't I? I knew **c1.Draw()** would not work. The error message tells you why: the automatically created **c1** is a pointer, and requires the -> symbol.

Edit the "." to the pointer symbol "->" and hit SHIFT-ENTER. You forgot to restart the kernel again, didn't you? Restart the kernel then hit SHIFT-ENTER in the cell.

If you think about it for a second, I could have given you a more complete example, the same way I did for the Python notebook:

```
TCanvas my_canvas();
TH1D example("example","example histogram",100,-3,3);
example.FillRandom("gaus",10000);
example.Draw();
my_canvas.Draw();
```

I presented it this way to make a couple of points. First, I wanted to show you how to restart a kernel within a notebook, which you may want to do even in Python.[51]

Second, I wanted you to learn that if you're working with C++ in ROOT, you'll have to be aware when you're redefining objects that ROOT thinks you've created before. You can work with ROOT C++ in Jupyter[52] but you have to be mindful of your environment.

---

[51] If you looked at the keyboard shortcuts, you know another way: Hit ESC to get into Command Mode, then hold down the 0 (zero) key.

[52] There are lots of examples at https://swan.web.cern.ch/notebook-galleries.

Decisions

If you've already made up your mind about the questions posed in the section headers, you can skip or skim this section.

C++ or Python?

Up until this point, the commands for ROOT/C++ and Python/ROOT were nearly identical.[53] 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: in which language do you want to use ROOT? My initial advice is to ask your supervisor. Their response, in ascending order of likelihood, will be:

- A clear decision (C++ or Python).
- "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:[54]

In favor of Python:

- Learning Python is easier and faster than learning C++.
- Python 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 Five and Six of this tutorial, and most of the tutorials included with ROOT (see page 78) 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 60).[55]

---

[53] See page 53 for the differences when using Python versus ROOT/C++.

[54] 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.

[55] There are various tricks for making Python run faster; e.g., the %pypy cell magic, the Cython extension, list comprehensions, clever use of NumPy. You'll learn about them if you choose to become a Python expert.

Command–line or notebook?

Once you've decided on the language, you next have to decide on your programming environment: the command line as in Part One, or the notebook as in Part Two.

In favor of notebooks

- They facilitate rapid code development. You fiddle inside a cell, hit SHIFT-ENTER to execute it, get it to do what you want. Then you move to the next cell.

- Documentation is easy, as shown on page 39.

- Notebooks are easy to share. For example, a colleague of yours can copy one of your notebooks to their own area to look at it:

```
%cp ~jsmith/energy-calibration/myanalysis.ipynb jsmith-analysis.ipynb
```

- The interface to a notebook is through a web browser. You don't need ssh or an X-Windows emulator.

Against notebooks

- They're relatively new in software development. Most likely your supervisor has never heard of them. If you say you've got a .png plot in a Jupyter notebook, they'll go "You've got a what in a where?"

- The `%jsroot on` magic command does not enable every X-Window feature available from within the ROOT command line. There's no TBrowser, TreeViewer, or FitPanel. You can't add new elements to a plot and then save the ROOT commands so you can examine how to use them in a .C file.[56]

- As you saw in the examples above, your canvases are not automatically drawn or updated for you. You must explicitly issue Draw() commands for your canvases.

Issues with our notebook server

Most of these points involve technical sysadmin issues. You may want to skip them, and come back later if you have a notebook problem.

- The Nevis particle-physics JupyterHub notebook server is not something you find at most institutions, at least not for now. Only the REU students and the particle-physics groups have access to it. Your astrophysics or RARAF colleagues won't be able to view your notebooks. You can install Jupyter on your laptop, but that won't help anyone else see your work.

---

[56] Oops, I just lied to you. If you've drawn something to `my_canvas`, you can write its associated ROOT commands to a file with

```
my_canvas.SaveSource("filename.C");
```

where `filename.C` can be any name you want. Don't forget to use `->` if your canvas variable is a C++ pointer instead of an object!

- You can develop software in notebooks, but you can't run multi-threaded or multi-hour jobs with them on our notebook server.[57]

- Some physics software is a "chimera", a blend of software compiled in two languages. For example, the Neutrino Deep Learning group uses Python to call pre-compiled C++ routines. Our notebook server can't run software that's been compiled on another machine.[58]

- As you get more familiar with the UNIX shell, you may start making changes to your standard shell setup. You do this by editing special shell initialization files such as `.profile`.[59] If you add new variables to your environment, these variables are available in our notebook server as well.[60]

  However, if you modify certain variables such as $LD\_LIBRARY\_PATH$ or run customization programs (such as **module load root**) in your default initialization, it can affect the execution of the notebook server. The typical symptoms are a notebook kernel that refuses to start or you get library load errors.

You can get around many of these issues by running Jupyter on your workgroup server; this is described at https://twiki.nevis.columbia.edu/twiki/bin/view/Main/IPython.

---

[57] The way to handle such tasks is with a batch system, which I discuss on the second day of this course.

[58] I doubt this will be important to your work this summer, but so you can look it up if necessary: The Nevis particle-physics systems are running Scientific Linux 6, while the notebook server is running CentOS 7. Sometimes code compiled on SL6 will run on CentOS 7, but more often you'll get crashes with an error message about a missing or incompatible library.

[59] You can find a list of which files you can change in https://twiki.nevis.columbia.edu/twiki/bin/view/Main/Shell.

A UNIX survival tip: Never let a well-meaning friend start editing your shell initialization scripts for you. I can't count the number of times I've looked at someone's shell init scripts and saw they were last edited in the 1990s. The user had either forgotten or never knew that their friend had put any such commands there, and so never kept them updated in the years since. If you edit your scripts yourself, at least you have a chance to maintain them.

[60] For reference:

If you define a variable in a shell, e.g.,
```
 export mywork=~/analysis_work_directory
```
then you can access the variable within your program in ROOT C++:
```
 TString my_location = gSystem->Getenv("mywork");
```
In Python:
```
 import os
 my_location = os.environ["mywork"]
```

Diagonalizing the 2x2 decision matrix

It's probably occurred to you that I've left you with four choices:

ROOT C++ on the command line	ROOT C++ in a Jupyter notebook
Python with ROOT on the command line	Python with ROOT in a Jupyter notebook

This tutorial is already 95 pages long, and I've probably taken longer than I should have to offer you too many options. For simplicity, I've chosen to present ROOT C++ on the command line in Part Three, and Python with ROOT in a Jupyter notebook in Part Four.

If you choose to pursue one of the "off-diagonal" choices, you won't have much trouble following Parts Three or Four. You were introduced to ROOT C++ in a notebook on page 40. To run Python with ROOT on the command line (including magic commands), the following will set you up on a Nevis particle-physics system:

```
module load root
ipython
```

Parts Three and Four of this tutorial present the same commands, exercises, and footnotes.[61] Pick which language you want to learn and go there; Part Three (ROOT C++) starts on the next page and Part Four (Python with ROOT) starts on page 62.

You might even be able to do both Parts Three *and* Four; once you've mastered C++, Python is pretty easy!
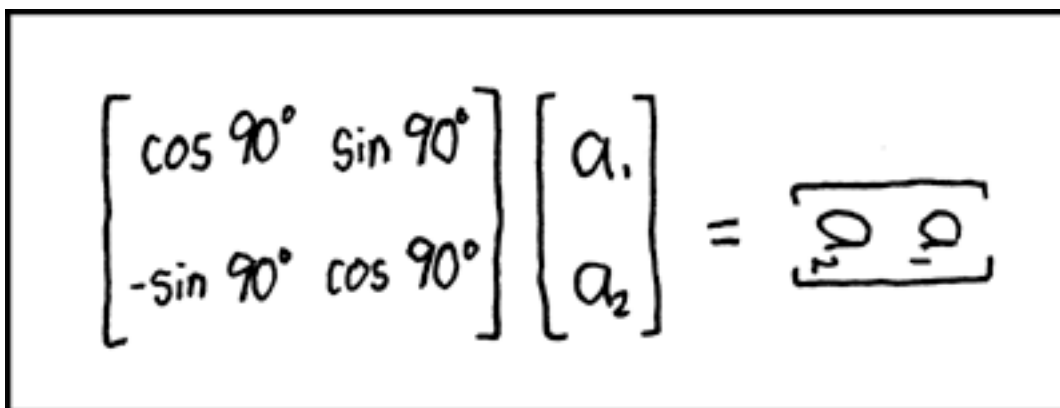
$$\begin{bmatrix} \cos 90° & \sin 90° \\ -\sin 90° & \cos 90° \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \boxed{a_2} \ \boxed{a_1}$$

Figure 15: https://xkcd.com/184/ by Randall Munroe
Alt-text: In fact, draw all your rotational matrices sideways. Your professors will love it!
And then they'll go home and shrink.

---

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

Part Three – 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 tools you learned about in Part One: 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.[62]

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 27) 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.

---

[62] 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 'pointer' to the location in computer memory where `hist1` is stored. I must use the pointer syntax to issue commands:

[] `hist1->Draw()`

Take another look at the file c1.C that you created in a previous example. Note that ROOT uses pointers for almost all the code it creates. 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 creates 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), and this is often reflected in the code it generates.

Figure 16: 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.[63]

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")
```

---

[63] *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->SetCanExtend(TH1::kXaxis); // 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++.[64] 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.[65]

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.[66]

---

[64] 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.

[65] If you're bolder or familiar with C++, you don't have to use MakeSelector to write an analysis class (specifically, a TSelector class) for you; look up TTreeReader on the ROOT web site. I use MakeSelector in this tutorial to spare you from having to define a `TTreeReaderValue` for every branch in the TTree. If you're willing to follow the directions for TTreeReader, you may get code that will be easier for you to revise in the long run. For an example, see `~seligman/root-class/AnalyzeReader.C`.

[66] 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 analyzing 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. If you do use PROOF, note that `SlaveBegin` and `SlaveTerminate` are where you put your set-up and wrap-up code, respectively, and `Begin` and `Terminate` should be "stubs."

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

Here's a simplified version of the C++ code from Analyze.C. I've removed the automatically generated comments created by ROOT, and minimized the routines `SlaveBegin` and `SlaveTerminate` which we won't use for this tutorial. 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.[67]

```
#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)
{
  // Don't delete this line! Without it the program will crash.
  fReader.SetEntry(entry);

  //******** Loop section ********
  // You probably want GetEntry(entry) here.
  return kTRUE;
}

void Analyze::SlaveTerminate() {}

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

Figure 17: Example C++ TSelector macro (Analyze.C). Compare with the code in Python (Figure 22, page 68).

---

[67] 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![68]

> 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")` – 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 second 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.

---

[68] 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. 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 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.[69]

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 fixed.

In the Loop section, put this in:

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

The first of these two lines means "get an entry from the TTree."[70] 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.[71] In code prepared by MakeSelector, the variables extracted from an n-tuple are pointers; they have to be prefixed with "*" to access their values.

The second line means "add 1 to a bin that corresponds to the value of `*chi2` in the histogram `chi2Hist`."

---

[69] Why are we defining a pointer then setting it equal to NULL? I'm teaching you to avoid a common problem in programming: uninitialized variables. 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.

[70] Actually, in the context of `MakeSelector` it means "get the data from the TTree pointed to by `fReader.SetEntry(entry)`".

[71] 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; there are examples on the ROOT web site. Since there are times when a simple line like `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.

Walkthrough: Making a histogram with Analyze (continued)

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->c1.C** from the **File** menu. I looked at c1.C and saw these commands in the file:

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

I scrolled up and saw that ROOT had used the variable `chi2` 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 20.

> 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.

> After you make a change to Analyze.C, you have to restart ROOT 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 48 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 `c1`) is created. So both histograms are being drawn to the same canvas.

> The easiest way to solve this problem is to create a new canvas for each histogram. Look at c1.C to see an example of how a canvas is created. Look up the `TCanvas` class on the ROOT web site to figure out what the commands do. To figure out how to switch between canvases, look at `TCanvas::cd()` (that is, the `cd()` method of the `TCanvas` class).

> Is the `ebeam` histogram empty? Take at look at the lower and upper limit of the x-axis 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 "Fit Statistics." (This is a repeat of how I found the "surf1" option for Exercise 1 on page 18).

> There was another way to figure this out, and maybe you tried it: Draw a histogram, select **Options->Fit Parameters**, fit a function to the histogram, save it as c1.C, and look at the file. OK, the command is there, mingled with the `TPaveStats` options... 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![72]

> 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.

---

[72] 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 31. Let's start with a fresh analysis skeleton:

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

In the `Process` section, put in the following line (remember: all the n-tuple variables are pointers):[73]

```
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 to use.

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 in Chapter 2 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 "()".[74]

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.

---

[73] You also have to put in that `GetEntry` line, which I complained about in Footnote 71.

[74] 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, finding polynomial roots).

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:[75]

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

Exercise 7: Trig functions (15 minutes)

Revise AnalyzeVariables.C to include a histogram of `theta` (recall page 32).

> 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 18: http://xkcd.com/809 by Randall Munroe
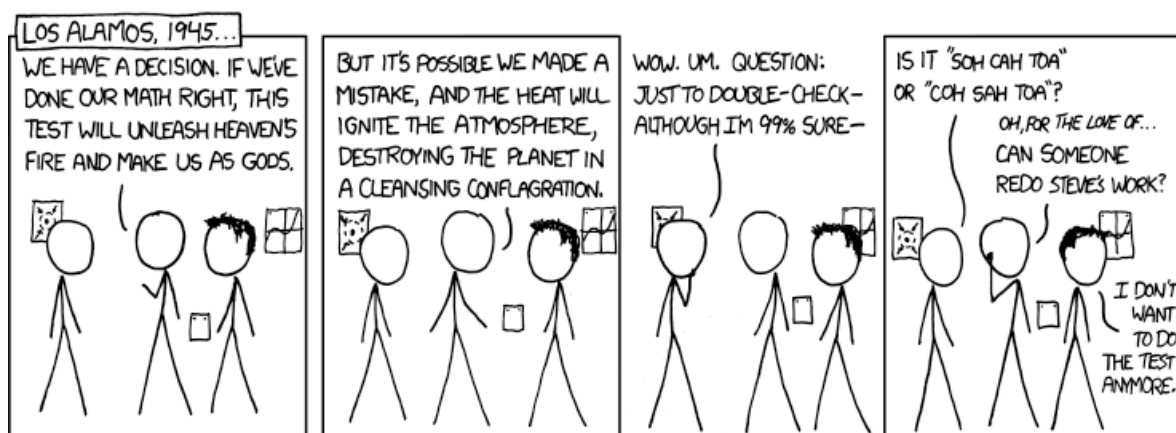
---

[75] If you compare this command with the computation of `pt` on the previous page, you may be either confused or irritated: When using C++ you have to access the n-tuple variables using pointer notation like "(*px)", while using ROOT directly you can get away with just using the variable names like "px". This is one of the reasons many folks prefer Python.

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![76]

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.

---

[76] 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
> cp ~seligman/root-class/AnalyzeExercise5.h $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 are files AnalyzeExercise7.C and .h 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 27, 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) (60 minutes or more 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 in chapter 7 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 14 on page 8.

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:

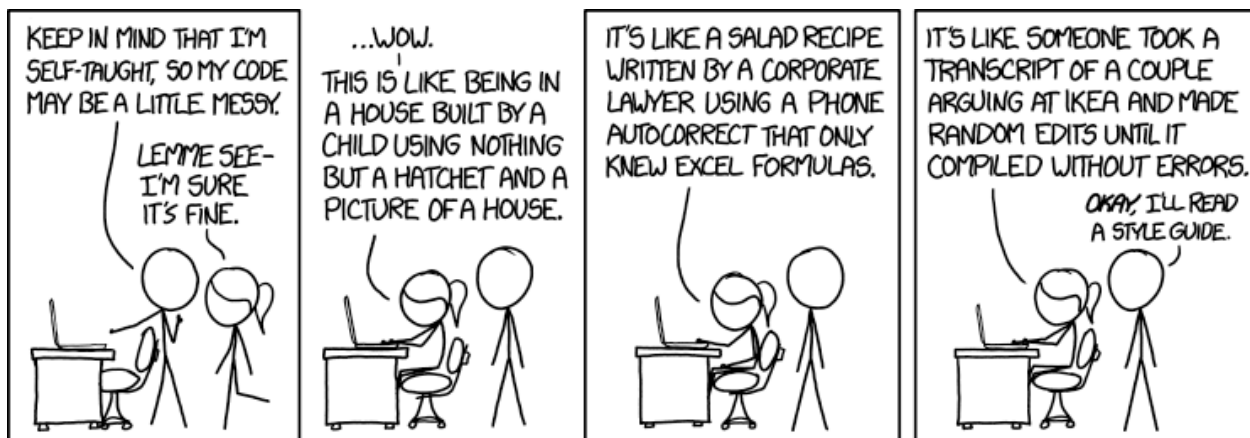http://www.nevis.columbia.edu/~seligman/root-class/links.html

Figure 19: https://xkcd.com/1513/ by Randall Munroe
Alt-text: "I honestly didn't think you could even USE emoji in variable names.
Or that there so many different crying ones."

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.
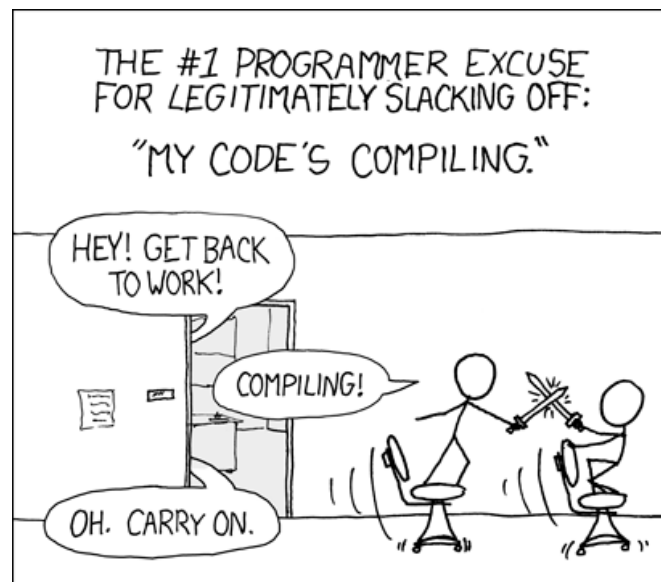
Figure 20: http://xkcd.com/303 by Randall Munroe
Alt-text: "Are you stealing those LCDs?" "Yeah, but I'm doing it while my code compiles."

Part Four – The Python with pyroot Path

If you're not interested in pyroot or Python, skip or skim this part. Go to page 78.

A brief review (5 minutes)

Skip this page if you've gone through the examples in Part Two.

Visit https://notebook.nevis.columbia.edu, type your Nevis account name and password, then select **Python 2** from the **New** pop-up on the upper left. Use **File->Rename...** to change the name from "Untitled" to anything you want; e.g., "Basic Test".

What turns Python into pyroot is the inclusion of the ROOT libraries. That's done with the **import** command. Cut-and-paste the following into the first, then press SHIFT-ENTER.

```
from ROOT import TH1D, TCanvas
my_canvas = TCanvas()
example = TH1D("example","example histogram",100,-3,3)
example.FillRandom("gaus",10000)
example.Fit("gaus")
example.Draw()
my_canvas.Draw()
```

Differences between C++ and Python

If you already know C++, or you've already done Part Three, 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.
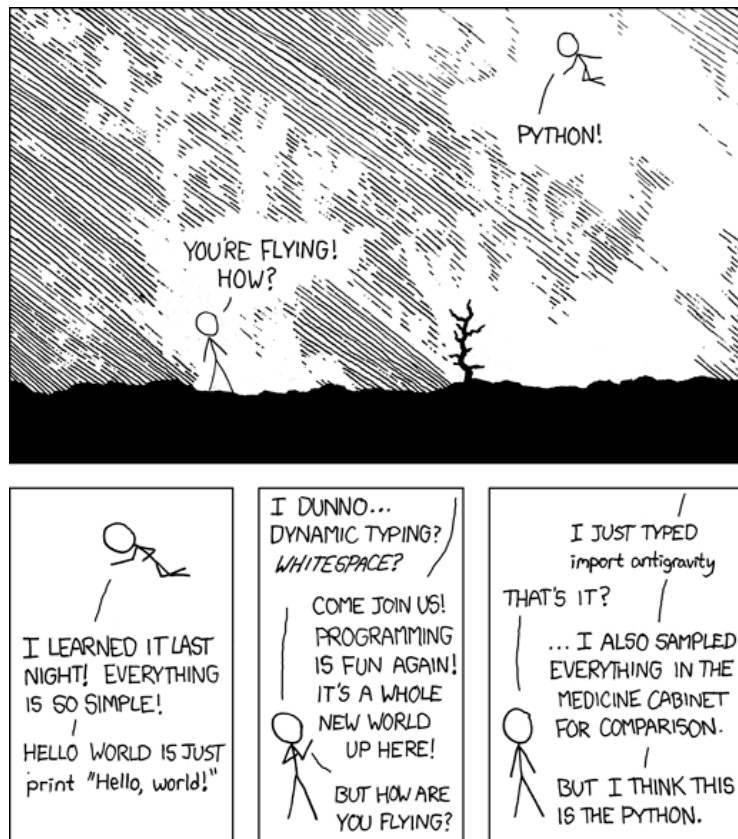
- C++ statements end with a semi-colon. Python statements end with a RETURN; no semi-colons.

```
[] myhist.FillRandom("gaus",10000); myhist.Fit("gaus");

In [] myhist.FillRandom("gaus",10000)
In [] myhist.Fit("gaus")
```

- C++ control structures (e.g., `if` statements, loops) are indicated by curly braces ({}).[77] 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. 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");

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

---

[77] 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.[78] Python determines types dynamically, and you don't have to specify them:[79]

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

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

- Finally,[80] the ROOT C++ interpreter, cling, knows the names of all the ROOT classes.

```
[] TH1D* example4 = new TH1D("example4","my fourth histogram",100,-3,3);
[] 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 part of ROOT with a prefix:

```
In [] import ROOT
In [] example4 = ROOT.TH1D("example4","my fourth histogram",100,-3,3)
In [] example4.Draw()
```

Method 2: Import the classes you'll need explicitly so you can omit the prefix:

```
In [] from ROOT import TH1D
In [] example4 = TH1D("example4","my fourth histogram",100,-3,3)
In [] example4.Draw()
```

I'm typically going to use the second method in this tutorial, but you can use either one.[81] 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:

```
In [] from ROOT import TH1D, TCanvas
```

---

[78] Since I hate to lie to you, I should mention the C++ `auto` keyword, which lets C++ determine the type for you. Both of the following are correct:

```
 TH1D* hist = new TH1D("hist","title",100,-3,3);
 auto hist = new TH1D("hist","title",100,-3,3);
```

This can be a great timesaver if a C++ function returns something with a type like `std::vector<std::pair<int,double>>::iterator`. However, you have to be comfortable with C++ before using it, which is why I'm relegating this C++ tip in a footnote in the Python section.

How comfortable with C++ do you have to be before you can use `auto`? Enough so that you understand why both of the above lines are not the best choice. A better choice would be:

```
 std::uniq_ptr<TH1D> = new TH1D("hist","title",100,-3,3);
```

Aren't you glad you're learning Python?

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

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

[81] 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.[82, 83]

Open the sample ROOT TTree in the notebook with the following:

```
from ROOT import TFile, gROOT
myFile = TFile("experiment.root")
tree1 = gROOT.FindObject("tree1")
```

The first command imports specific ROOT classes into Python (see the previous page).

That third 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 Three, note that in Python we have to read in the n-tuple explicitly.

In a notebook, you can't use the Scan method to look at the contents of the Tree (see page 28), but you can display the names of the variables and the size of the TTree:

```
tree1.Print()
```

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:

```
from ROOT import TCanvas
my_canvas = TCanvas()
tree1.Draw("ebeam")
my_canvas.Draw()
```

While we have to explicitly Draw a canvas, we can re-use a previously-defined canvas (the same way command-line ROOT keeps re-using c1).

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

---

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

[83] If you're experienced with Python, you may ask why I'm not including NumPy, SciPy, and matplotlib in this tutorial. I want to focus on the ROOT toolkit, even though many tasks (especially in Parts Five and Six) can be more easily accomplished using those additional packages. I wrestled with this issue for a while, before deciding that there are hundreds of web sites on matplotlib but few sites on ROOT. But I may change my mind next year!

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")
my_canvas.Draw()
```

> 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")
my_canvas.Draw()
```

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

```
tree1.Draw("zv")
my_canvas.Draw()
```

> 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.[84,85]

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")
```

---

[84] After this point, I won't include the `my_canvas.Draw()` line in future examples, but you'll have to remember to execute that line. I assume you've gotten into the habit of re-using or cut-and-pasting lines between cells.

[85] *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 = ROOT.TH1D(...) # define your histogram
hist.SetCanExtend(ROOT.TH1.kXaxis) # 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.[86] 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
%load Analyze.py
```

The second of the two magic commands will load the contents of Analyze.py into the next notebook cell, all ready for you to play with it.

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 65.

---

[86] We're up to at least four lifetimes, five if you completed Part Three, possibly six if you're learning LaTeX from scratch.

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

# 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 22: Python analysis "skeleton" for a ROOT n-tuple.
Compare with the same code in C++ (Figure 17, page 49).

Walkthrough: Using the Analyze script (10 minutes)

As it stands, the Analyze script does nothing, but let's learn how to run it anyway. Hit SHIFT-ENTER in the cell to run the script.[87]

Python 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.

Let's start making histograms. 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:

```
canvas = TCanvas()
chi2Hist.Draw()
canvas.Draw()
```

> Don't forget about the indentation. The lines in the Loop section must be indented to show they're part of the loop.

Execute your revised script.

> 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.

> 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")
acanvas = TCanvas()
tree1.Draw("chi2")
acanvas.Draw()
```

(continued on next page)

---

[87] You may want to organize your scripts in files outside of notebook cells. This lets you keep track of different versions of your scripts, and allows you to use your favorite text editor. To run an external script from within a notebook cell, use the %run magic command; e.g.,

```
%run Analyze.py
```

You can save the contents of notebook cells by putting the %%writefile cell magic command at the top of the cell and hitting SHIFT-ENTER; e.g.,

```
%%writefile AnalyzeChi2.py
```

Walkthrough: Using the Analyze script (continued)

Hmm, the histogram's axes aren't labeled. How do I put the labels in the script? Here's how I figured it out: I went back to command-line ROOT from Part One and plotted `chi2` with the TreeViewer. I labeled the axes on my test histogram by right-clicking on them and selecting **SetTitle**. I saved the canvas by selecting **Save->c1.C** from the **File** menu. I looked at c1.C and saw these commands in the file:

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

I scrolled up and saw that ROOT had used the variable `chi2` for the name of the histogram pointer. I copied the lines into my Python script, 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 script.[88]
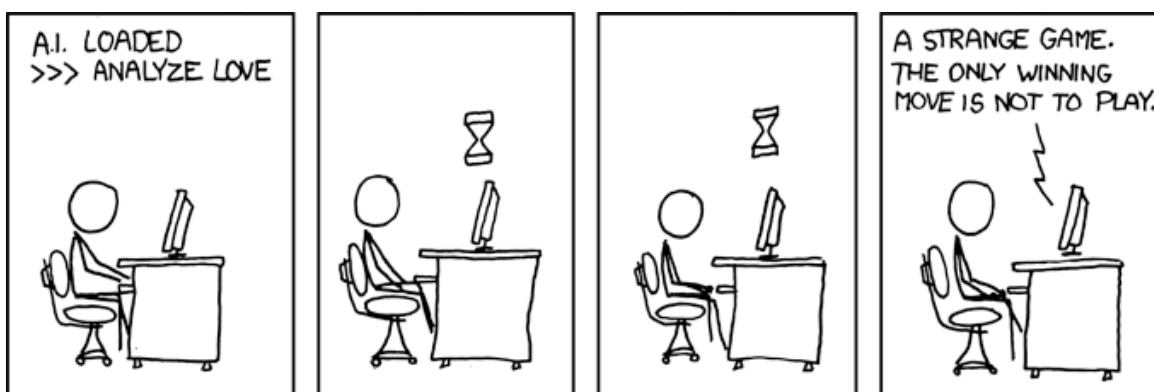
Figure 23: http://xkcd.com/601 by Randall Munroe.
Alt-text: "Wait, no, that one also loses. How about a nice game of chess?"
Fortunately it's easier to analyze histograms than it is to analyze love. At least it is for me!

---

[88] There's another way to do this in the notebook. Plot the graph with JSROOT:
```
%jsroot on
acanvas.Draw()
```

You can then right-click on the axes, select **Title->SetTitle**, and enter the axis label you want. However, this solution can't be automated; if you have to generate a hundred histograms each with different axis labels, you'll want a method you can put into a script.

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 script to draw the histograms with error bars.

> Hint: Look back at "Working with Histograms" on page 20.

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

> Take care! On page 67 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 execute, 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 most recent canvas you created. So both histograms are being drawn to the same canvas.

> Some clues to solve this problem: Look at the examples above to see 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 `c1.cd()`. Or you might define a canvas, draw in it, define a new canvas, then draw in the newer one.

> 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 what's on the web site (with a prefix of ROOT or including it on the **from import** line).

> 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 "Fit Statistics." (This is the same procedure for figuring out the "surf1" option for Exercise 1 on page 18).

> There was another way to figure this out, and maybe you tried it: Draw a histogram in command-line ROOT, select **Options->Fit Parameters**, fit a function to the histogram, save it as c1.C, and look at the file. The command is there, but would you have been able guessed how to apply it outside `TPaveText` if you without 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![89]

> 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.

---

[89] 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. Load a fresh copy of that script into your notebook:

```
%load Analyze.py
```

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

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

Did that not work? To get at the variables `px` and `py`, you have fetch them from the n-tuple with something like `mychain.px`. You also have to either have `import ROOT` or `from ROOT import TMath`.

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 "()".[90]

Test the script to make sure it runs. You won't see any output, so we'll fix that in the next exercise.
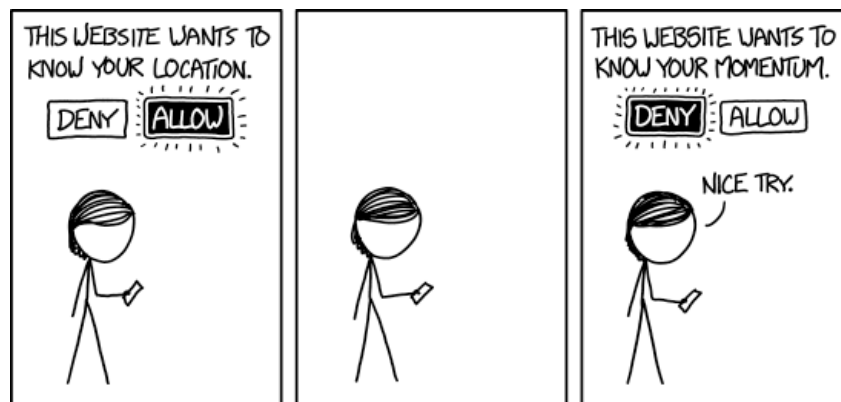
Figure 24: http://xkcd.com/1473 by Randall Munroe

---

[90] 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 your script 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.
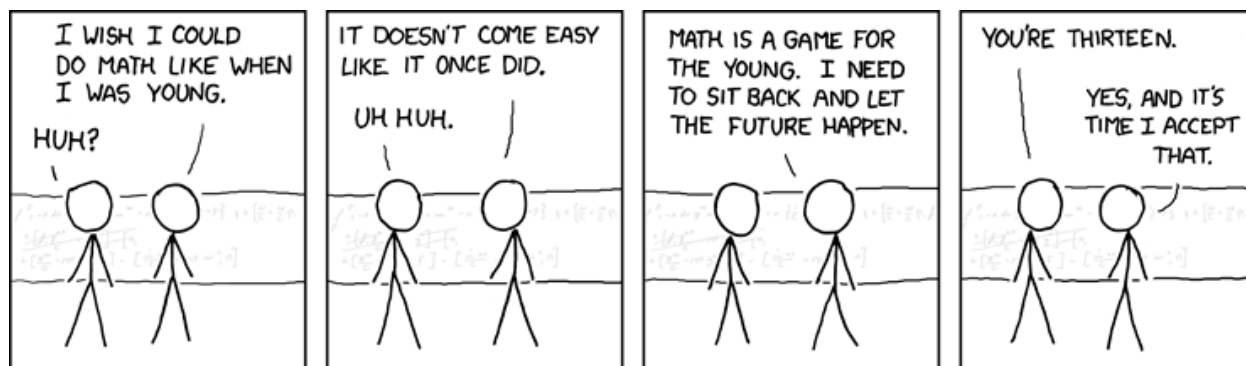

Figure 25: http://xkcd.com/447 by Randall Munroe
Alt-text: "They say that if a mathematician doesn't do their great work by age eleven, they never will."

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 script:

```
%load Analyze.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:

```
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 script you created in Exercise 5. If you've overwritten it, you can copy my version:

```
%cp ~seligman/root-class/AnalyzeExercise5.py $PWD
%load AnalyzeExercise5.py
```

> 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?
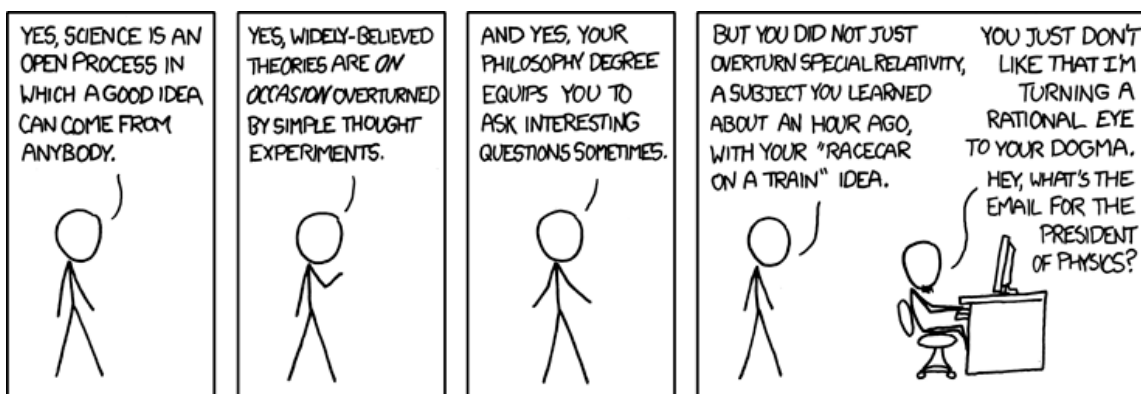

Figure 26: http://xkcd.com/675 by Randall Munroe
Alt-text: "I mean, what's more likely – that I have uncovered fundamental flaws in this field that no one in it has ever thought about, or that I need to read a little more? Hint: it's the one that involves less work."

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" with the TBrowser in command-line ROOT and check that your plots are what you expect.

In "Saving your work, part 2" on page 27, 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 user's guide

- The **Help** menu located in the upper right-hand corner of most command-line 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:[91]

```
> 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 Five and Six 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.[92]

> 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.

---

[91] 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.

[92] 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 in chapter 2 of the ROOT Users Guide.

Part Five – Advanced Exercises

If you still haven't finished the exercises for Parts One, Three, or Four, 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 Python. 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. 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.[93]
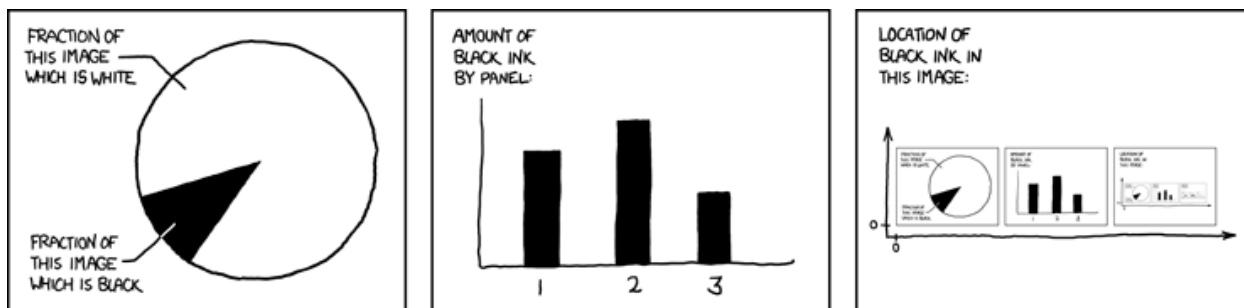

Figure 27: http://xkcd.com/688 by Randall Munroe

---

[93] For Python programmers: Because I have a generous soul, I'll permit you to use matplotlib instead of TGraphErrors for your x-y plots. The fact that I have no way to stop you has nothing to do with it.

C++ 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 given you a guide to four ways to construct a TGraphErrors object, you can probably figure out what the others 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.[94] 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 46).

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);
```

---

[94] If you're doing these exercises in Python: You'll want to read up on numpy arrays instead. Fortunately, numpy arrays will automatically be converted to C++ arrays when they're passed as arguments to ROOT methods.

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

ROOT's container classes are described in chapter 16 of the ROOT Users Guide.

In the TGraphErrrors constructors, the TVectorF and TVectorD classes are containers for single- and double-precision real numbers, respectively. Click on the class names in the ROOT web site to see the clear and detailed explanation of how to use them.

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.[95]

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:

C++ 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++?

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.[96]

---

[95] In previous versions of this tutorial, I spent a couple of pages discussing object inheritance, and what it means to, e.g., "inherit from TObject." The new ROOT web documentation makes it harder to determine object inheritance; you have to actually look at ROOT's C++ source code. I decided to spare you that as much as possible.

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

Vectors

For the work that you'll be asked to do in Parts Five and Six, 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.

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:[97]

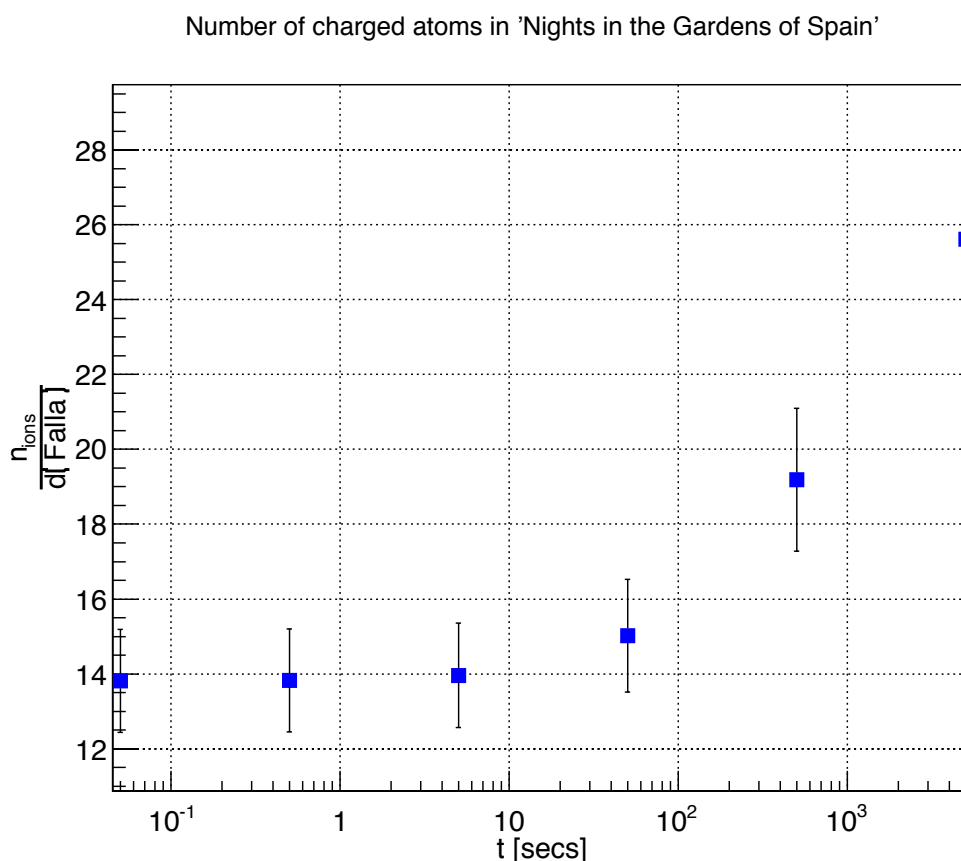Number of charged atoms in 'Nights in the Gardens of Spain'

Figure 28: 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)

---

[97] For Python programmers: if you use an appropriate routine from matplotlib, you'll have to figure out how to get that mathematical formula to label the y-axis of the plot. This page may help you get LaTeX expressions (see footnote 50) into your axes labels: http://matplotlib.org/users/usetex.html

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.[98] You've seen some macros before (remember c1.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 62), 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.[99]

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 59), 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?

---

[98] 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 (or use a Jupyter notebook).

[99] Another tangent:

grep is a program that interprets "regular expressions" (also known as "regexes"), a powerful method for searching, replacing, and processing text. More sophisticated programs that use regular expressions include **sed**, **awk**, and **perl**; there are also regex libraries in Python and C++. Regexes 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: The TH1 page lists the method you'll need.

(hints continued on next page)

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

- *In Figure 28, 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 $\dfrac{n_{ions}}{d\left(Falla\right)}$?*

  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.

- *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.
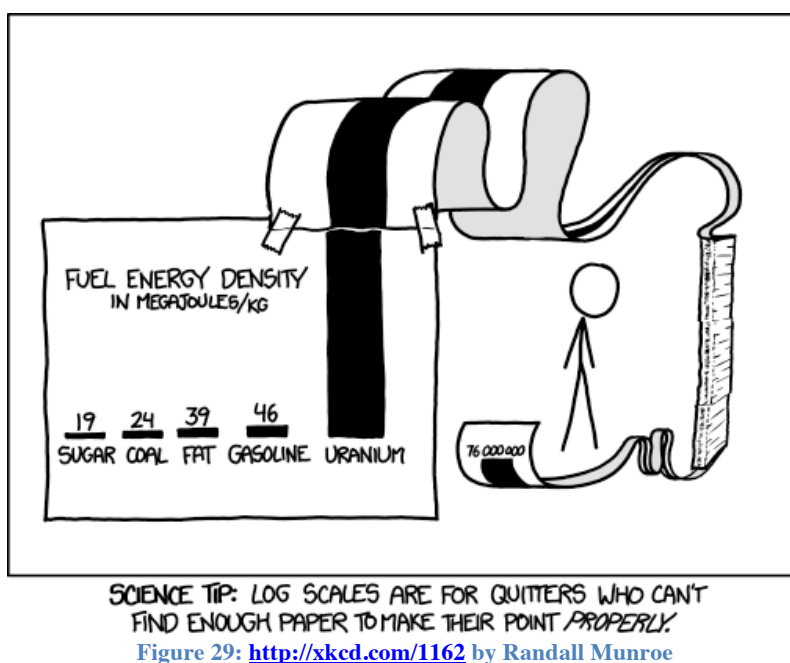

  Now you can get to work!

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 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.

SCIENCE TIP: LOG SCALES ARE FOR QUITTERS WHO CAN'T FIND ENOUGH PAPER TO MAKE THEIR POINT *PROPERLY.*
Figure 29: 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 48. 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. C++ tip: If a ROOT operation to read a pointer fails, that pointer will be set to zero (see page 91).

Part Six – 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.[100] 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?

---

[100] 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. A recent 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?[101]

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." Nothing there, so I went to its parent[102] TDirectory. I found something, clicked on the name of the method… then pounded my head against the desk.[103]

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, you'll have know a little bit about class inheritance. 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++:[104]

```
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.

---

[101] For Python programmers: This discussion of object inheritance is relevant to you as well, but you deal with it in a different way. Look up the Python **type()** and **isinstance()** functions.

[102] TFile's "grandparent."

[103] I suppose the programmer thought that they would write the documentation for GetList later.

Here's a tip for writing code that will make you a hero: "later" does not exist. (As of 2016, the ATLAS collaboration has collected over 35 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 yourself 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.

[104] How do you figure out which classes derive from where? As I noted in footnote 95 on page 72, the only way to find out in the current ROOT documentation is to search ROOT's C++ source code, which you can browse via the links to the .h files in the class' web page. Welcome to the wild adventure hunt that is ROOT!

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.[105]

Figure 30: http://xkcd.com/371 by Randall Munroe

Why did I just take two 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 ROOT's container classes only contain TObject*, why the default canvas is a `TCanvas* c1`, etc.

- 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.

---

[105] 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.[106]

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.

---

[106] If you've been trying to get through the advanced exercises using Python, this one may stump you; it certainly stumps me. I know of no simple way of loading a C++-based ROOT dictionary using Python. Something like this may be a start:

```
ROOT.gInterpreter.ProcessLine('#include "ExampleEvent.h"')
ROOT.gSystem.Load("./libExampleEvent.so")
```

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.[107] I mentioned this earlier in footnote 28 on page 29: it's possible to extend ROOT's list of classes with your own by creating a custom dictionary. Only classes that have a dictionary defined 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.[108] 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)

---

[107] 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.

[108] 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.[109] Some 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

```
#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!

---

[109] 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 Five and Six 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.[110]

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).[111]
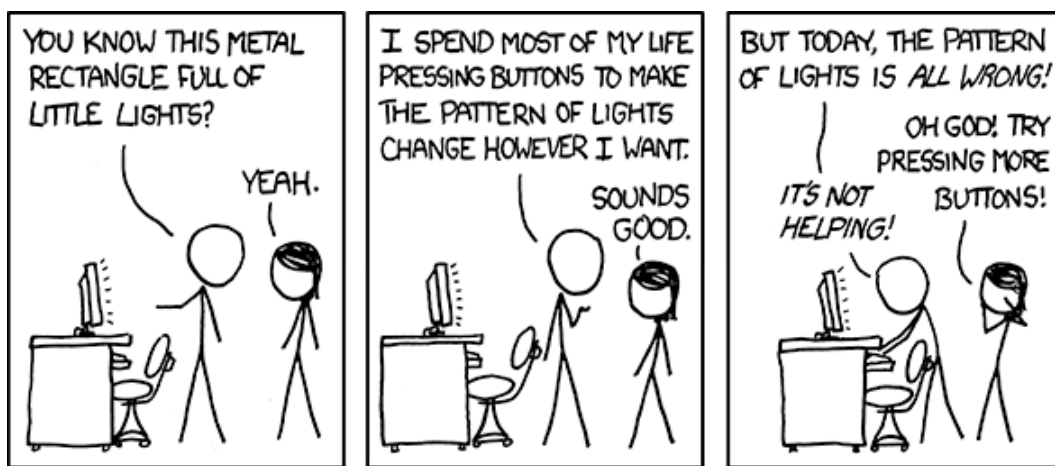
Good luck![112]

Figure 31: 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."

---

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

[111] 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.

[112] Total lifetimes used up: up to nine, depending on if you chose to learn both ROOT/C++ and pyroot, which tangents you took, how much LaTeX you learn, and whether you choose a career in physics. I generously give any remaining lives back to you.