Nevis Labs ROOT tutorial

Release 27-May-2023

William Seligman

27-May-2023

CONTENTS

Release: 27-May-2023
Generated: 27-May-2023

CHAPTER

ONE

INTRODUCTION

What

This is an introductory "hands-on" tutorial on ROOT, a C++ and Python-based data-analysis software toolkit developed at CERN, a high-energy particle-physics accelerator research facility on the French-Swiss border.

Who

The target audience for this workshop are undergraduate students in the sciences who are looking for summer research experience; for example, those in the US National Science Foundation's Research Experience for Undergraduates (REU) program.

Many parts of this tutorial are optional or are advanced exercises. No one expects you to get through all these pages before you start your physics work for the summer. Much of the material, especially from *Intermediate Topics* on, is intended to use as a reference if you continue to use ROOT in the future.

I tried to "aim low" in terms of expectations on the required knowledge of computing, math, and science. Anyone with access to a computer with ROOT installed, or with a laptop on which they're willing to put in the effort to *install ROOT*, should be able to work their way through this material.

Where

This workshop is taught at Nevis Labs. I've provided resources for those with login access to the Nevis particle-physics Linux cluster, such as a JupyterHub server with ROOT pre-installed. Therefore, some of the set-up instructions are Nevis-specific.

I've tried to provide alternative instructions for anyone to be able to take this tutorial, even if they're not at Nevis. There are several ways to *install ROOT*, and any files can be easily copied.

In addition to the web site you're reading now, you can find this tutorial in PDF format, along with related material, on my ROOT Tutorial page.

How

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 the exercises are challenging. Getting to all but the last exercise (the stand-alone program) in either *The C++ Path* or *The Python Path* is a respectable achievement on its own.

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."

When

This tutorial started as a one-day class I taught at Columbia University's Nevis Laboratories 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.

Why

Disclaimer

Although the developers of ROOT have always been ready to answer my questions, they have had no direct input on the contents of this tutorial.

In particular, they had no say in the flippant, humorous, and occasionally sarcastic tone in which I've decided to present the material. This includes the over-use of smart-aleck footnotes and xkcd cartoons.[1]

Any errors in this tutorial are my own.[2]

Which

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

Have fun!

– *Bill Seligman*

---

[1] I've included a link in each figure caption of any xkcd cartoon I use. However, I don't include the hover-text, which is occasionally funnier than the cartoon itself. For extra added xkcd goodness, click on the link in the figure caption and hover your mouse over the cartoon.

[2] Unless someone else wants to take responsiblity for a given blunder on my part, in which case I'll gladly cede the privilege.

CHAPTER

TWO

GETTING STARTED

A guide to this tutorial

You may be used to interacting with a computer solely through a graphical user interface (abbreviated GUI). If so, that will change with this course: you're going to learn how to type UNIX commands. Don't worry; I'll walk you through it.

You'll also have to learn about ROOT commands. Here are the clues to give you context:

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.

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

```
> ipython
In[] from ROOT import TH1
```

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.

Note: 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*.

Footnotes

There are a lot of footnotes in this course. Some are serious. Some are humorous. Some think they are funny but are not.

Clickable footnote navigation in a web-based viewer can be puzzling at first, so I'll spell it out: If you see a footnote number, you can click on it to be taken to the portion of the page that contains the footnote. To return from whence you came, you can either click on the back button in your reader, or you can click on the footnote's number to the left of the footnote.

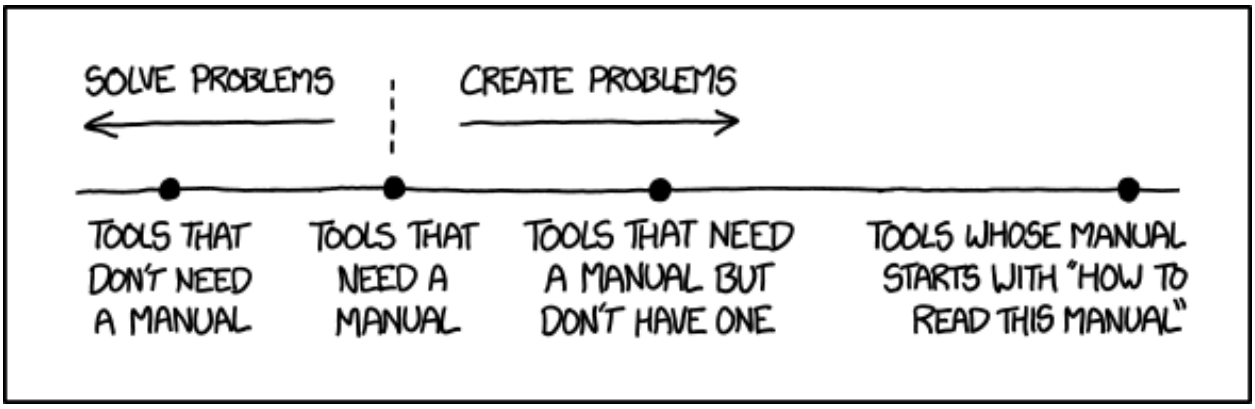

Figure 2.1:: http://xkcd.com/1343 by Randall Munroe

Getting started using your laptop

SSH and X11

You'll need ssh to connect to a remote server that has ROOT installed. ROOT uses a graphics protocal called X11 for its interactive display.[1]

If you've never installed ssh or X11 on your computer, these two pages will guide you through the process:

- Initial laptop setup
- Installing X11

If you have an account on one of the Nevis particle-physics systems, these sites were included in the introductory email I sent you with details about your account.

In case it's not obvious: For the rest of this tutorial, I'll assume you've read the above web pages, know how to use `ssh` to connect to your Nevis server[2] or some other computer system that includes ROOT, or you have your own installation.

Escape from the gooey GUI

You may be 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.

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

Terminals

You will need at least two terminal windows open during parts of this tutorial. One window I'll call your "ROOT command" window; this is where you'll run ROOT. Another is a separate "UNIX command" window.

Your existing terminal program includes some way of showing multiple windows or tabs; look through its menus for something like "New Window". If you're using a laptop to connect to a remote server, you'll have to separately use `ssh` to login to that server in each window.

Tip: I like to open a new tab instead of a new separate window, but you can use whichever mode you prefer. I suggest you try both methods to find out which one suits you.

Installing ROOT on your laptop

Don't.

If this is too short and snarky for you, I'll elaborate: You may correctly deduce that setting up ROOT on your own computer system is not a trivial task. It is *not* an app you can double-click to install. To avoid a hassle, I suggest logging into the Nevis particle-physics servers or using the Nevis notebook server for this course if you are able to do so.

[1] Very strictly speaking, if you're going to *install ROOT* on your own computer, you don't need `ssh`. As a practical matter, if you're going to be working in science, you'll find `ssh` and its related tools (`scp`, `sftp`) to be useful.

[2] Don't forget the `-XY` or some other method of forwarding an X11 connection to that server.

The reason why I installed ROOT on the Nevis particle-physics systems and prepared the notebook server is so students and researchers affiliated with Nevis can spend less time on software installations, and more time on learning how to use the tools to do physics.

Let's do physics.

Note: If the above is not enough to dissuade you, or you don't have a choice because you don't have a connection to either Nevis or another institution with ROOT already installed, I offer the *nitty-gritty details of installing ROOT*.

A brief introduction to Linux

If you're already familiar with UNIX, you can just skim or skip these topics.

You can spend a lifetime learning UNIX; I've worked with it since 1993 and I'm still learning something new every day. The commands in the following topics, limited to what you'll need for *the basics*, barely scratch the surface.

Directories in UNIX

Note: If you're one of those people who's only used a GUI (Graphical User Interface), or you save all of your files on your Desktop, this sub-section is for you. There are plenty of web sites that discuss directories; this is just a brief overview.

The "folders" that you see when you look at your GUI are actually directories in your operating system. That tells you what a directory is: a container for other files, including other directories. The separator for directory names is "/", so `a/b/c` is directory `c` within directory `b` within directory `a`.[1]

Everything in UNIX is within a directory. Yes, even your Desktop; typically, that is a directory whose name is `~/Desktop`. That leads us to common abbreviations and commands for directories when you're using the command line:

- `~<account>` means the home directory of the user `<account>`[2]. Just plain `~` means your own home directory. So `~/Desktop` means a directory named Desktop within your home directory.
- `cd` is the command to "change directory." It's the usual way to go from one directory to another. If there were a directory named `Root`[3] in your home directory, you could visit that directory with:

  ```
  > cd ~/Root
  ```

- `..` is a reference to your parent directory, the one "above" the one you're currently in. If you wanted to return to your home directory from `~/Root`, you could type:

  ```
  > cd ..
  ```

  If you use the `cd` command without any arguments, it will return you to your home directory:[4]

  ```
  > cd
  ```

- To look at the contents of your current directory, use the `ls` command:

  ```
  > ls
  ```

  You can also list the contents of any other directory (for which you have permission to view):

[1] Now you know why it's hard to put a / in a folder name: The operating system can't tell the difference between a / that's within a folder name versus a / that is a directory separator.

[2] It's always something like "~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 near the upper-left-hand corner of the keyboard.

[3] UNIX is normally a case-sensitive operating system. `~/Root`, `~/ROOT`, and `~/root` are three *different* directories. Exception: In Mac OS Darwin, by default file names are *case-insensitive*; all three of those directories would be the same.

[4] Knowing this will become useful in the future, as you become more sophisticated in your use of UNIX. Eventually you'll learn about shell variables. Sooner or later, you'll make a typo in a variable name; e.g.,

```
> cd $ROTSYS
```

Instead of going to `$ROOTSYS`, your intended destination, you'll find yourself in your home directory. That's because `$ROTSYS` doesn't have a value, so UNIX interpreted this as the `cd` command without any arguments.

```
> ls ~seligman/root-class
```

- If you forget which directory you're in, use the `pwd` ("print working directory") command:

```
> pwd
```

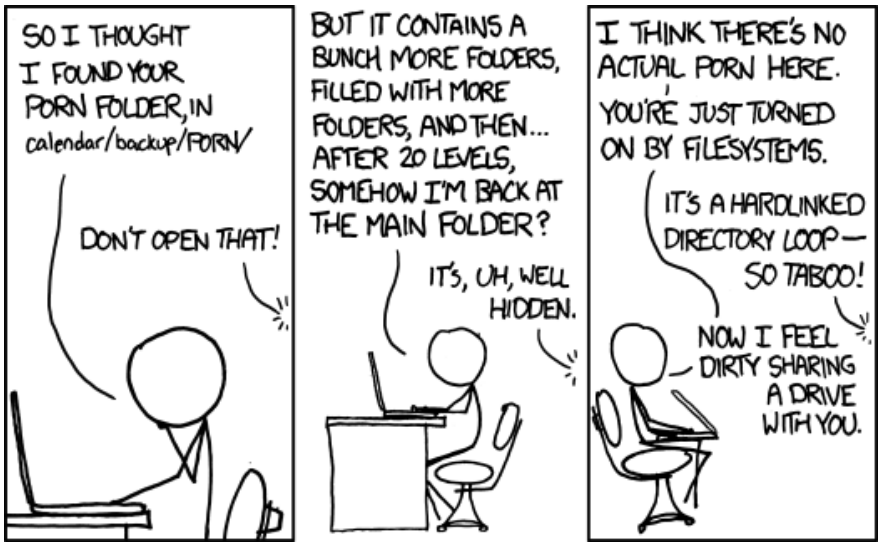

Figure 2.2:: https://xkcd.com/981/ by Randall Munroe. Moral: Be careful how you organize your directories!

Other UNIX commands[1]

- *To copy a file:* use the `cp` command.[2]

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

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

  In UNIX, `$PWD` means the results of the `pwd` command.[3]

- *To look at the contents of a text file:* Use the **`less`** command.[4]

  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:

[1] A colleague of mine once said, "The commands in UNIX are the initials of the people that the designers went to high school with."
A good story, but it's not true. The reality is that when UNIX was first developed in the 1960s, computers had six orders of magnitude less memory than they do today. The space for internal tables containing items like command names was at a premium. From then to now, UNIX command names tend to be terse.

[2] Over the years, as students began to use laptops, I noticed that many of them had an interesting misconception: The `cp` command copies a file from one place to another on the same computer. It does *not* copy a file from a remote server to your laptop! For that you use `scp`; use `man scp` to learn more. Note that if you're working on a remote server such as the ones at Nevis, there's no reason to use `scp` for this tutorial.

[3] A period (`.`) is the usual abbreviation in UNIX for "the current directory" (did you remember that `..` means "the directory above this one"?) but many students missed the period the first time I taught this class.

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

```
> 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.

- *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 delete a file:* Use the **`rm`** command.[5] For example:

```
> rm CreateSubdirectories.C
```

- *To edit a file:* I suggest you use `emacs`.[6] For example, to edit the file CreateSubdirectories.C:

```
> emacs CreateSubdirectories.C
```

  This may appear to "lock up" your UNIX window. If this is an issue, either create a new UNIX window or learn about ampersands at the end of a command line.

  The `emacs` environment is complex, and you can spend a lifetime learning it.[7] 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.

Tip: Are you quitting emacs after you change a file, only to start up the editor again a moment later? Hint: look at *File*. If you're editing many files, try opening them all with *File* → *Open File* and switch between them using *Buffers*. Remember to use *File* → *Save* once in a while.

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.

---

[5] **Super-duper warning: There is no Trash folder or "undo" operation in command-line UNIX!** If you use **`rm`** on a file and later realize that was a mistake, there's pretty much nothing you can do except re-create the file from scratch.

Be warned: A notorious UNIX prank is for someone to get you to type `rm -rf ~` which performs an unrecoverable delete of all of your files without any warning. Don't fall for it!

[6] If you're already familiar with another text-based UNIX editor, such as `nano` or `vim`, you can use it instead.

If you're not using a remote UNIX server and you're editing files on your laptop, make sure you're using a plain-text editor. If you use an editor whose default mode is to *not* save files in plain text (Microsoft Word is one example; the poorly-named TextEdit on the Mac is another) you're going to get confused. If you don't already have such an editor, I suggest Notepad++ on MS-Windows and the free version of BBEdit for the Mac.

What do I mean by "you're going to get confused" in the previous paragraph? When I taught Python to my father, he tried to use Microsoft Word to edit code. He didn't understand why you couldn't apply a bold or italic font style to text in a program. He also often forgot to use `Save As...` and switch the file format to `Plain Text`, so his code was saved as a Word document.

Of course, you're not my father; otherwise I'd ask you if I could stay out late this Saturday. You may find Word is the best plain-text editor for you.

As a final aside: There's lots of debate over which is the best UNIX editor: `emacs`, `vim`, or `nano`. I'm aware that `emacs` is now the minority choice. I recommend `emacs` because it's the one I always use, for reasons discussed on the next web page.

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

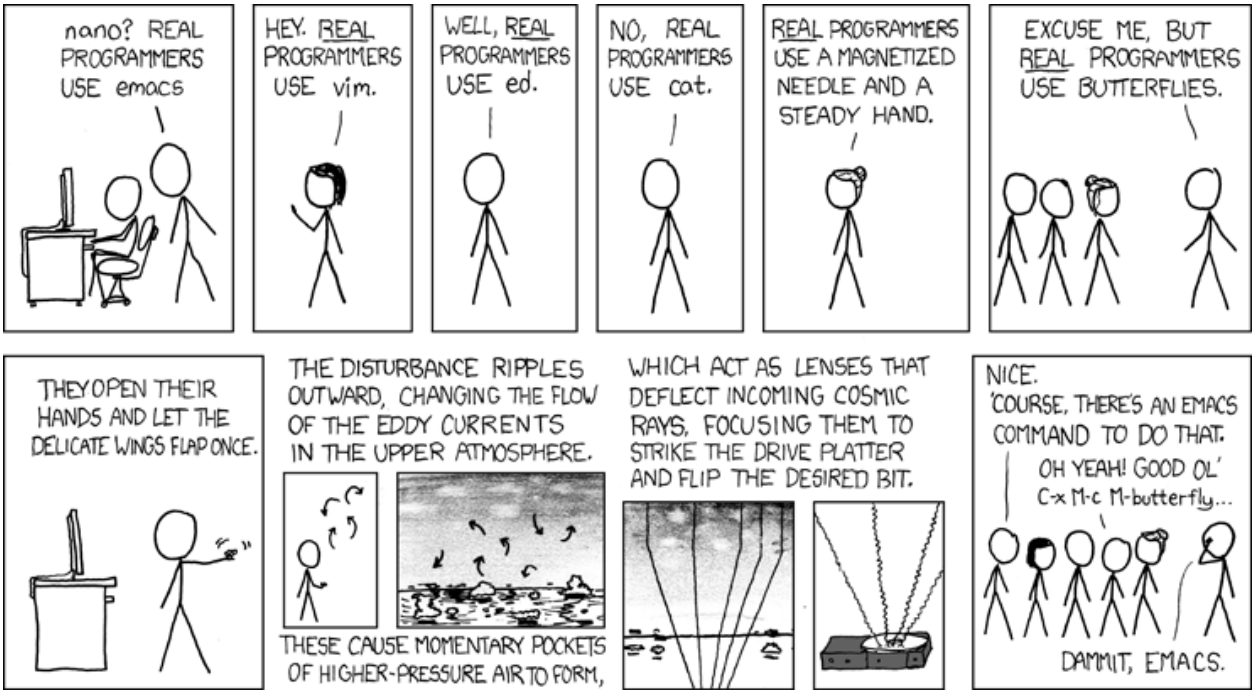

Figure 2.3:: http://xkcd.com/378 by Randall Munroe. If you're feeling bored, type `Meta-x butterfly` in emacs and see what happens.

UNIX command-line tricks

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 a Nevis system, 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.

Note: 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.

Those same key-based navigation commands work on the UNIX and ROOT command lines. 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.

Tip: 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 usually faster for me to keep my hands on the keyboard.

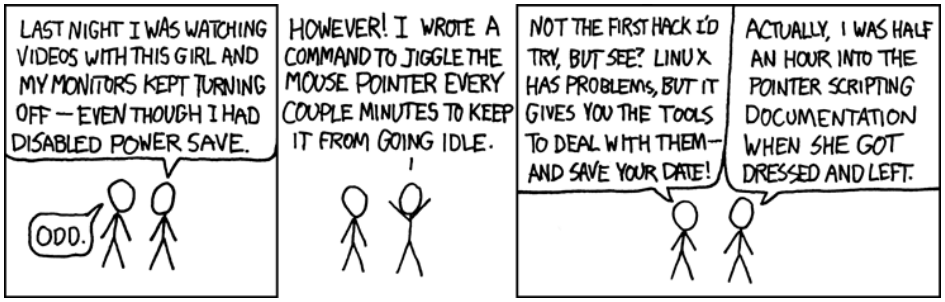

Figure 2.4:: http://xkcd.com/196 by Randall Munroe

Better UNIX references

Either of these two sites offer a good introduction to UNIX:[1]

- UNIX Tutorial for Beginners
- Learn UNIX

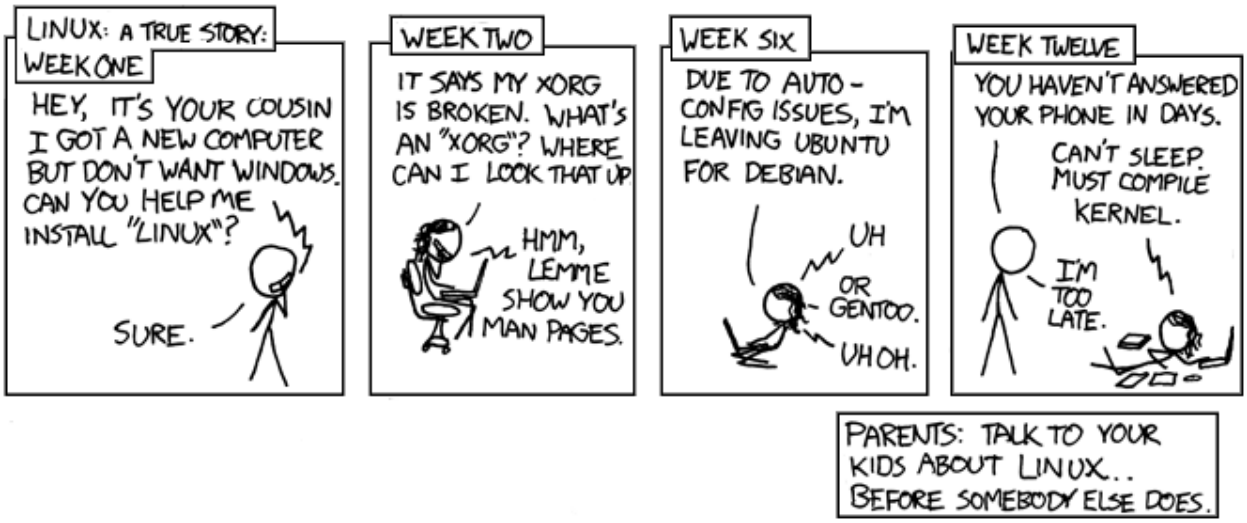

Figure 2.5:: <http://xkcd.com/456> by Randall Munroe

[1] For the purposes of this tutorial, "Linux" and "UNIX" are the same thing. Systems administrators know that Linux and Mac OS Darwin are varieties of UNIX, but you're not here to be a sysadmin… at least not for today.

Walkthrough: Setting up ROOT

(5 minutes)

Before you start using ROOT, you usually have to set it up:

- If you're not using one of the Nevis particle-physics systems, hopefully you have received instructions on how to set up ROOT at your site. Now is the time to follow those directions.
- If you had to *install ROOT* on your own you already know whatever special set-up you need on your local computer.
- If you are using one of the particle-physics systems at Nevis, type:

```
> module load root
```

Nevis specifics

The command `module load root`[1] 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. If you're told to find a file in $ROOTSYS/tutorials (see the *References* section of this tutorial, 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 (but read the warnings below before you do this).

Warning:

- It's a Nevis convention to use `~/.myprofile` for your custom shell setup. Other sites may tell you to edit `~/.profile`, `~/.bashrc`, `~/.bash_profile`, `~/.zprofile`, `~/.cshrc`, `~/.tcshrc`, etc. There is a Nevis wiki page on shell setup scripts.
- Even at Nevis, 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`.
- Finally, do not put `module load root` in a start-up script if you're using the *Nevis notebook server*. You'll get lots of "not found" errors.

[1] 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 default UNIX installation. Here's a Nevis wiki page on available modules.

CHAPTER

THREE

THE BASICS

Walkthrough: Starting ROOT

(5 minutes)

ROOT is a robust, complex environment for performing physics analysis, and you can spend a lifetime learning it.[1]

To actually run ROOT, just type:

```
> root
```

Note: The window in which you type this command will become your *ROOT command window*.[2] You'll see some "Welcome to ROOT" text in the window.

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.

Tip: 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.

Tip: ROOT can function as a calculator. If you want, in ROOT type `2+3` or `sqrt(2)` or whatever. I'm not going to dwell on this aspect of ROOT, but it's good to know it's there.[3]

[1] That's three lifetimes so far.

[2] In case you skipped over the *Getting started* section, or you don't like clicking on links in the middle of Notes: You will need *two* terminal windows open during this section of the tutorial: a UNIX window and a ROOT window.

[3] One of those ROOT quirks that makes you go "uhh…": If you want to take the sine of 30 degrees you have to use `sin(30.*TMath::Pi()/180.)`

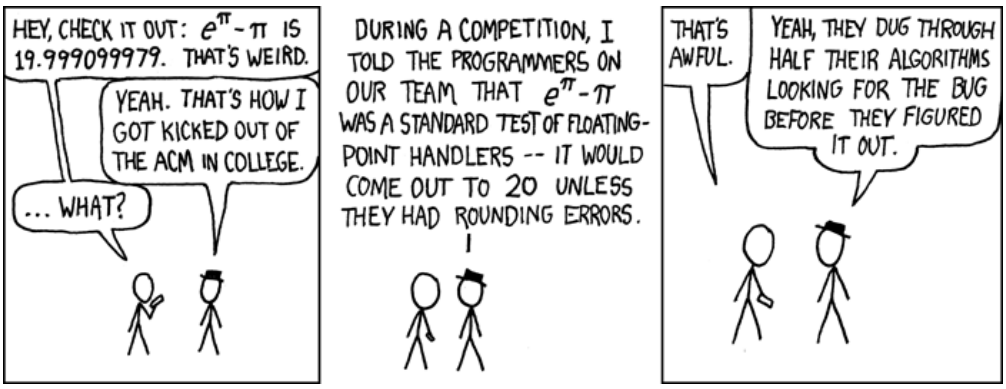

Figure 3.1:: https://xkcd.com/217/ by Randall Munroe. If you want to try this in ROOT, it's `exp(TMath::Pi()) - TMath::Pi()`

Walkthrough: Plotting a function

(15 minutes)

Let's plot a simple function. Start ROOT and type the following at the prompt:[1]

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

Note: Note the use of C++ syntax to invoke ROOT commands.[2] 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.[3]

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. The only place that the complete ROOT functionality is documented is on the ROOT web site.[4]

Go to the ROOT web site at https://root.cern/ (I suggest you bookmark this site[5]), click on `Reference`, then on `All Classes` on the left-hand side, then on `TF1`; you may want to use the browser menu *Edit* → *Find* and search on `TF1` to locate that link.[6] Scroll down the page; you'll see some documentation and examples, the class methods, then method descriptions.

Note: 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.[7]

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.

[1] 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.

[2] I'm simplifying. ROOT doesn't use a C++ compiler, but an interpreter called "cling" that duplicates most of the C++ language specification. It's meant to provide an interactive experience similar to Python.

[3] 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.

[4] If you remember the output from when you typed `.help`, you may be tempted to try `.help TF1`. Don't bother. What will happen is that ROOT will start up a web browser to take you to the TF1 page on the ROOT web site.

This may sound convenient, and it can be… *if* you're running ROOT directly on your laptop. However, if you're running ROOT on a remote server (as I recommend), it will start up the web browser *on the remote server*. Your screen will lock up for a couple of minutes as it pushes the web browser graphics from the server to your screen, and the resultant browser window will be slow and laggy.

[5] I suggest you also bookmark the location of the ROOT User's Guide. It's a trifle out-of-date (and hard to find on the ROOT web site), but it's a useful supplement to this tutorial. Here's *more* on ROOT's documentation.

[6] Assuming that ROOT hasn't reorganized their web site (which they do periodically) since I last reviewed this tutorial, here are the links to ROOT's list of classes and to the description of the TF1 class.

[7] 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.

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.[8,9]

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`.

Note: Right-clicking on the axis title gives you a `TCanvas` pop-up, not a text pop-up; it's as if the title wasn't there. Only if you right-click on the axis can you affect the title. In object-oriented terms, the title and its centering are a property of the axis.

It's a good practice to always label the axes of your plots. Don't forget to include the units.

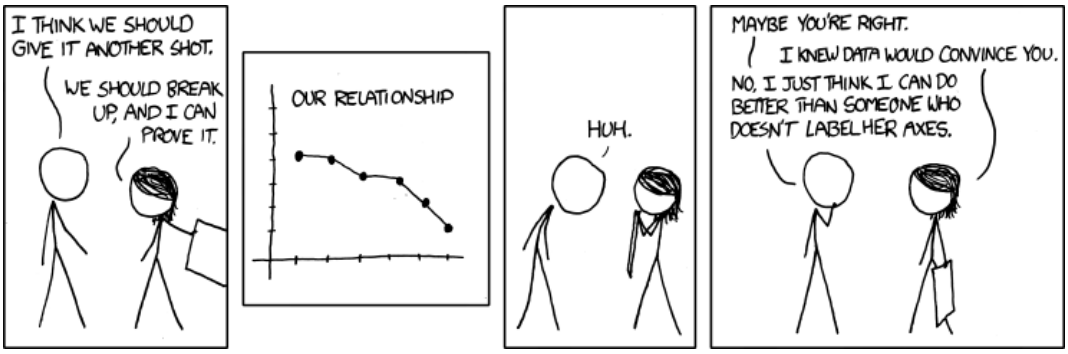

Figure 3.2:: https://xkcd.com/833/ by Randall Munroe

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*.

You have a lot of control over how this plot is displayed. Select *View* → *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`.

Note: 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.

You may notice Help menus and icons that appear to offer explanations, like **?**. They worked in earlier versions of ROOT. Unfortunately, at some point in the last few years these links stopped working, probably due to a reorganization of the ROOT web site. Maybe the ROOT developers will fix this someday, but probably not: It's just as easy to do a web search of the form "cern root axis SetRange".

---

[8] 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, locate the description of the `TAxis` class on the ROOT web site and look at the `SetRange` method within that class.

[9] Note that ROOT handles the case where x=0 correctly. If you're a math major, this may make you think. What happens if you plot cos(x)/x? tan(x)/x?

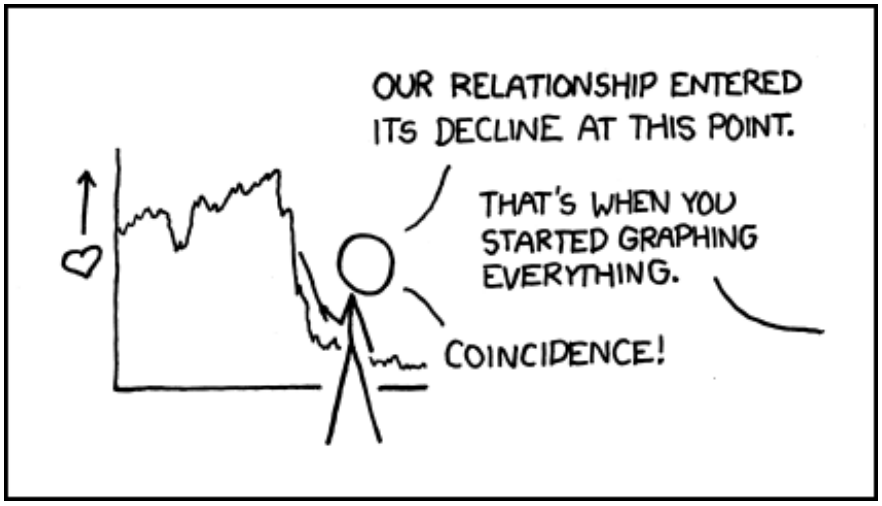

Figure 3.3:: https://xkcd.com/523/ by Randall Munroe. If you have a choice, ruin the plot. Don't let the plot ruin you.

Exercise 1: Detective work

(10 minutes)

Duplicate the following plot:[1]

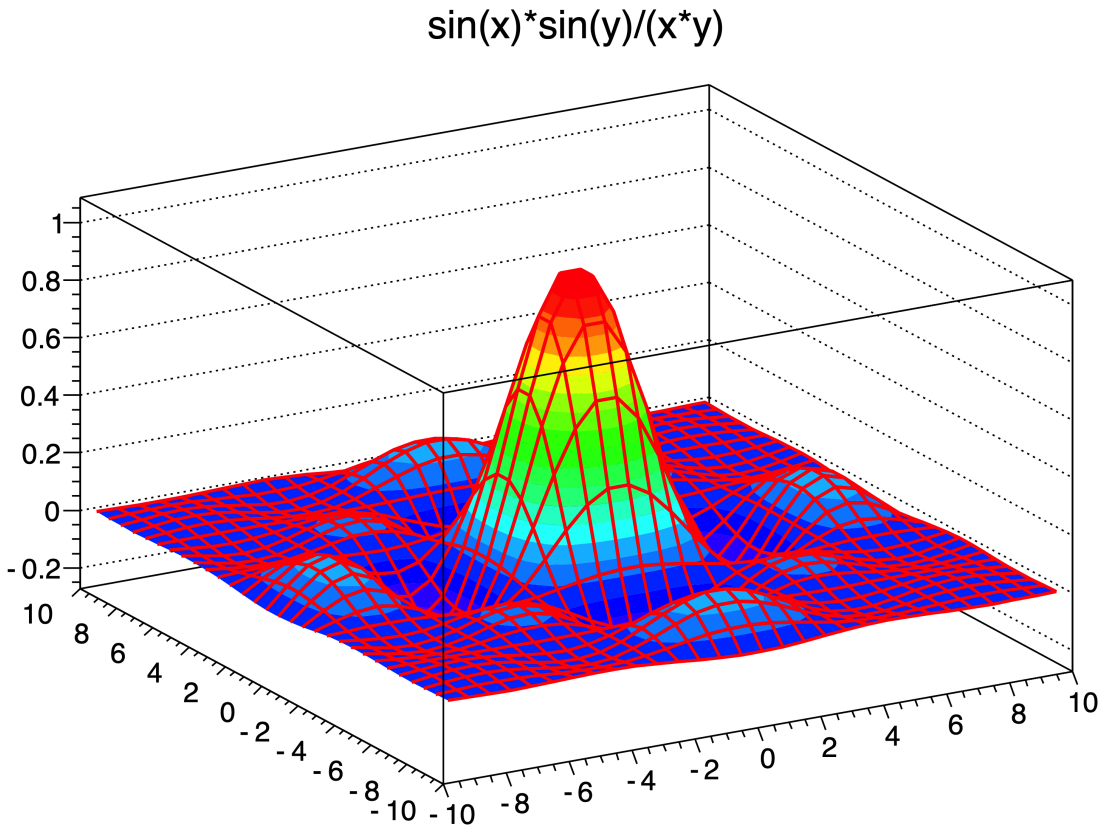

Figure 3.4:: Some detective work is required to duplicate this plot.

Hints

Take *another look* at the `TF1` command. 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 youre 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."

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

- 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, *Advanced Exercises* and *Expert Exercises* in this tutorial. Finding the "surf1" option is trivial by comparison!

Walkthrough: Working with Histograms

(25 minutes)

Let's create a simple histogram:

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

Note: 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".[1]
- 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.[2]

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 *later in this tutorial*). The default Gaussian distribution has a width of 1 and a mean of zero.

You've probably already noticed the words "Mean" and "StdDev" in the upper right-hand corner of the plot. If you need formal definitions, you can find them in Equations (9.1) and (9.2) later in this tutorial.

Question

For those who know statistics: In this histogram, why isn't the mean exactly 0, nor the width exactly 1?

---

[1] Did you just ask "What is a 'Gaussian'?" I created a section on *statistics* just for you!

[2] *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 a large number of events in every bin in the histogram, which helps with fitting to functions as discussed below and in *the statistics section*.

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?

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.

Note: With these options, the size of the error bars is equal to the square root of the number of events in that histogram bin. Use the up-arrow key in the ROOT command window and 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 $P_0 e^{-\frac{1}{2}\left(\frac{(x-P_1)}{P_2}\right)^2}$, where P_0, P_1, and P_2 are "parameters" of the function. Let's set these three parameters to values that we choose, draw the result, 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)
```

Question

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

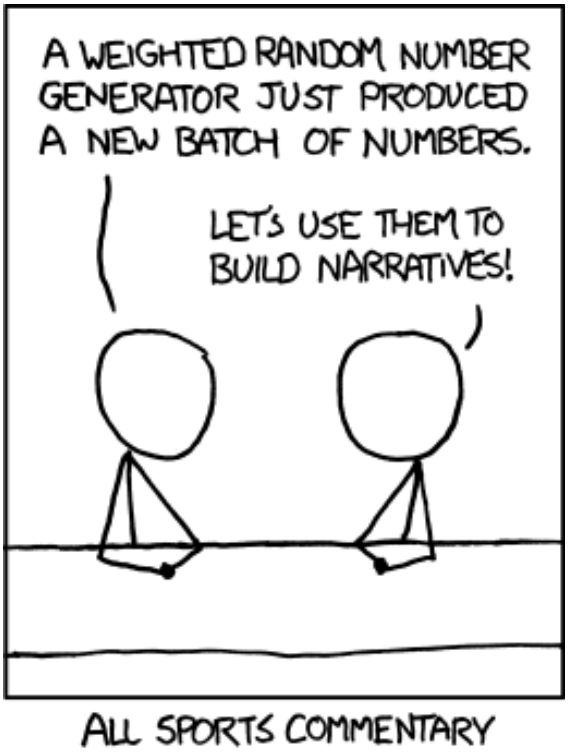

Figure 3.5:: https://xkcd.com/904/ by Randall Munroe

Walkthrough: Saving and printing your work

(15 minutes)

By now you've probably noticed the *File* → *Save* sub-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
```

Note: If you get complaints about a file not found, the name of the canvas is "see-one," not "see-ell."

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.

Tip: 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!

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
```

Note: 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.[1]

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

```
> root c1.C
```

Printing at Nevis

If you are physically in the Nevis research building, you can print out your PDF file with the command

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

If you want to print directly from the ROOT canvas, click on *File* → *Print*, type `lpr -Pbw-research` in the first text box and leave the second one empty. Again, the printer name "bw-research" only has meaning at Nevis.

---

[1] This is still useful if you're working in pyroot, though you'll have to do some translation from C++ to Python. *This discussion* may help with that conversion.

Walkthrough: The ROOT browser

(5 minutes)

Note: The ROOT browser is a useful tool, though it can be a bit clumsy at times. Let's give it a try.

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:[1,2]

```
[] TBrowser tb
```

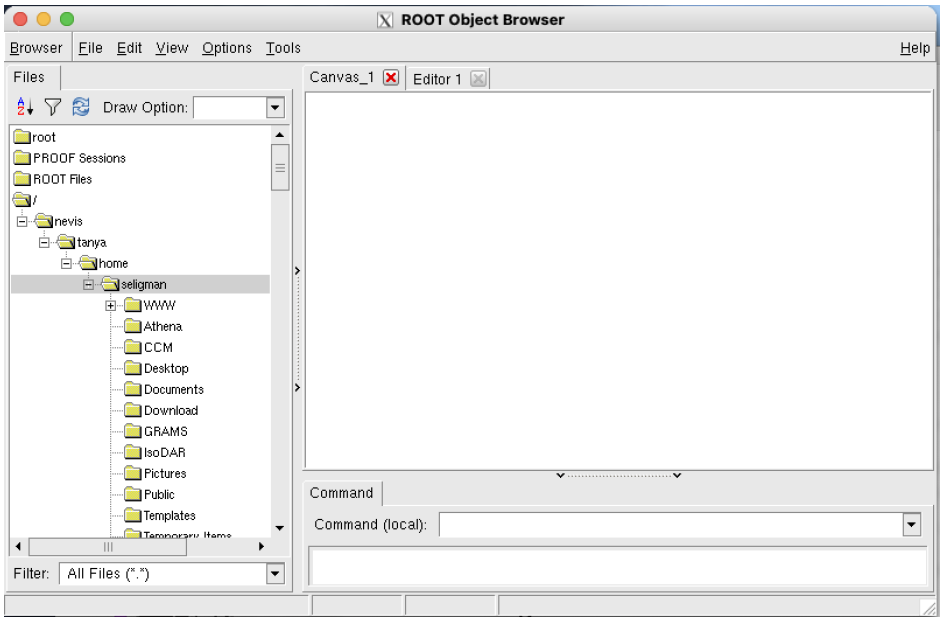

Figure 3.6:: When I start the `TBrowser`, this is what I see. Your window won't be quite the same, if for no other reason that my home directory has more files and sub-directories than yours does… at least for now!

In the left-hand pane, scroll to the folder with the same name as your home directory.[3] 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.

[1] If you're not at Nevis, depending on the details of your ROOT set-up, you may see something similar with a title that reads `RBrowser`. This is the next-generation ROOT Browser. It doesn't have all the bells-and-whistles of the original `TBrowser` yet; for example, you can't right-click on the name of an n-tuple to start the *TreeViewer*.

It won't be hard to figure out how to use the `RBrowser` on your own. If you really want to see the `TBrowser` as used in this tutorial, quit ROOT, edit the file `~/.rootrc`, and add the line:

```
Browser.Name: TRootBrowser
```

Save the file and start up ROOT again. You should see the `TBrowser` from now on.

[2] You may see someone using this command instead:

```
[] new TBrowser
```

The difference is slight, and only matters if you're experienced with C++. (If you are experienced with C++: what is that difference? *Here's a hint*.)

[3] If you have a Nevis temporary account, 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 the Nevis wiki automount page.

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`.

Note: 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 structured objects such as n-tuples.

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.

Walkthrough: Fitting to a Gaussian distribution

(10 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:[1]

```
> 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.

Note: 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.[2]

Note: 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 *Options* → *Fit Parameters*. You'll see the fit parameters displayed on the plot.

Note: 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).

---

[1] If you're going through this class and you can't login to a system on the Nevis particle-physics Linux cluster, you'll have to get the files from my web site.

If you want to get all the files from that directory at once, one way is to use this UNIX command:

```
wget -r -np -nH --cut-dirs=2 -R "index.html*" \
    https://www.nevis.columbia.edu/~seligman/root-class/files/
```

You may have to install the `wget` command on your system, since it's often not installed by default.

Be aware that in that directory there are a lot of work files I created to test things. There's more in there than just the files I reference in my tutorials.

[2] What do all these options mean? The *Fit Function* selects which mathematical function is going to be used to fit the histogram. *Predef-1D* means that the function is going to come from one of ROOT's pre-defined one-dimensional math functions; as you will learn in just a bit, you can define functions of your own. *Chi-square* refers a fitting method; for any fit that you're likely to do with a *FitPanel*, this will be the method you'll use.

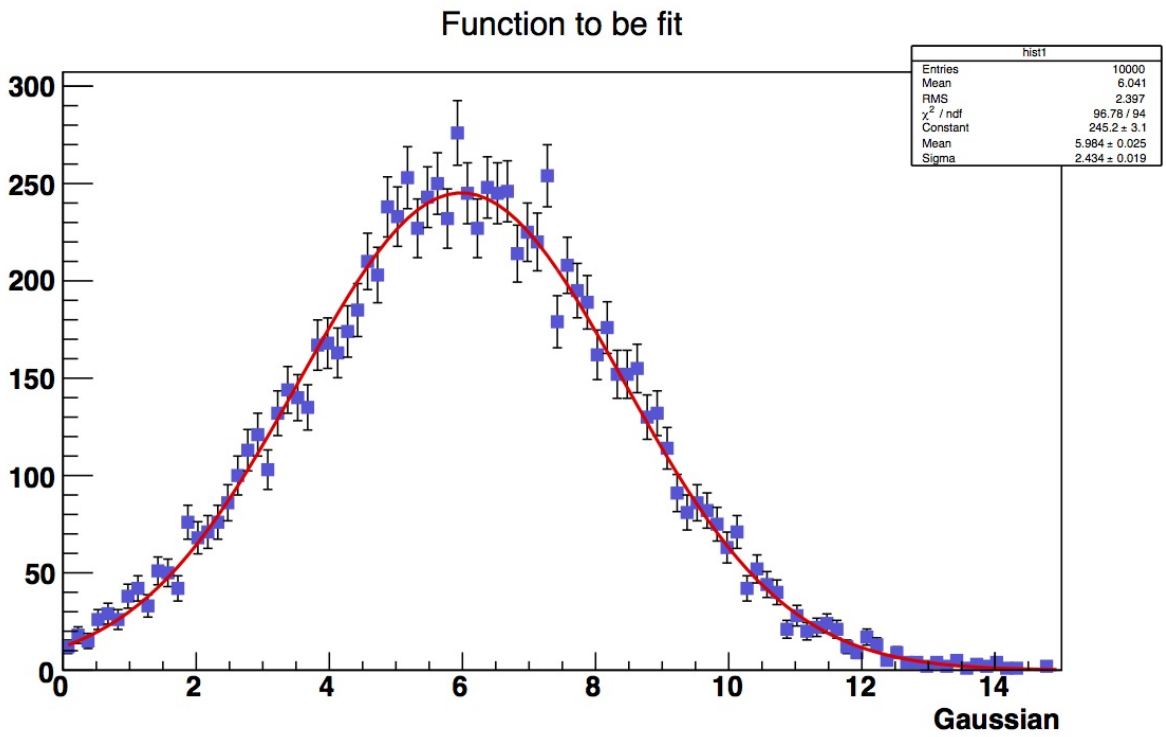

Figure 3.7:: The resulting plot should look something like this.

As a check, click on *landau* (which vaguely resembles the plot in Figure 3.8) on the FitPanel's *Fit Function* pop-up menu and click on *Fit* again; then try *expo* and fit again.

Note: 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 "kye-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.[3]

The FitPanel is good for Gaussian distributions and other simple fits. But for fitting large numbers of histograms (as you'd do in the *Advanced Exercises* and the *Expert Exercises*) or for more complex functions, you'll want to learn the following ROOT commands.

To fit `hist1` to a Gaussian, type the following command:[4]

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

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

[3] If you're not familiar with terms like "chi2" or "chi-squared" there's a brief introduction to *statistics* in this tutorial.

[4] 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 the section on *pointers* for more information.

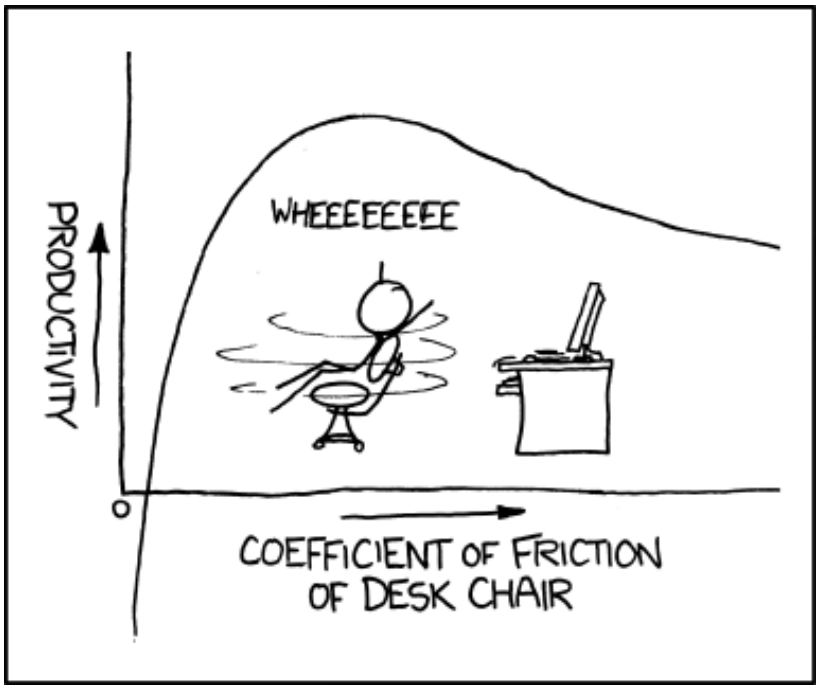

Figure 3.8:: http://xkcd.com/815 by Randall Munroe. If your fit function looks like this, something has gone wrong. This would be a poor fit for your data.

Walkthrough: Fitting to a user-defined function

(10 minutes)

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

Note: 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: 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.

Let's set the values of P_0, P_1, P_2, P_3, P_4, and P_5, and fit the histogram.[1]

```
[] 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")
```

Note: 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 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")
```

[1] It may help to cut-and-paste the commands from here into your ROOT window.
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 *you've learned*), and that won't happen if you just type Ctrl-C/click/Ctrl-V over and over again.
When we get to *The Notebook Server*, you'll start cutting-and-pasting commands into notebooks on a regular basis.

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

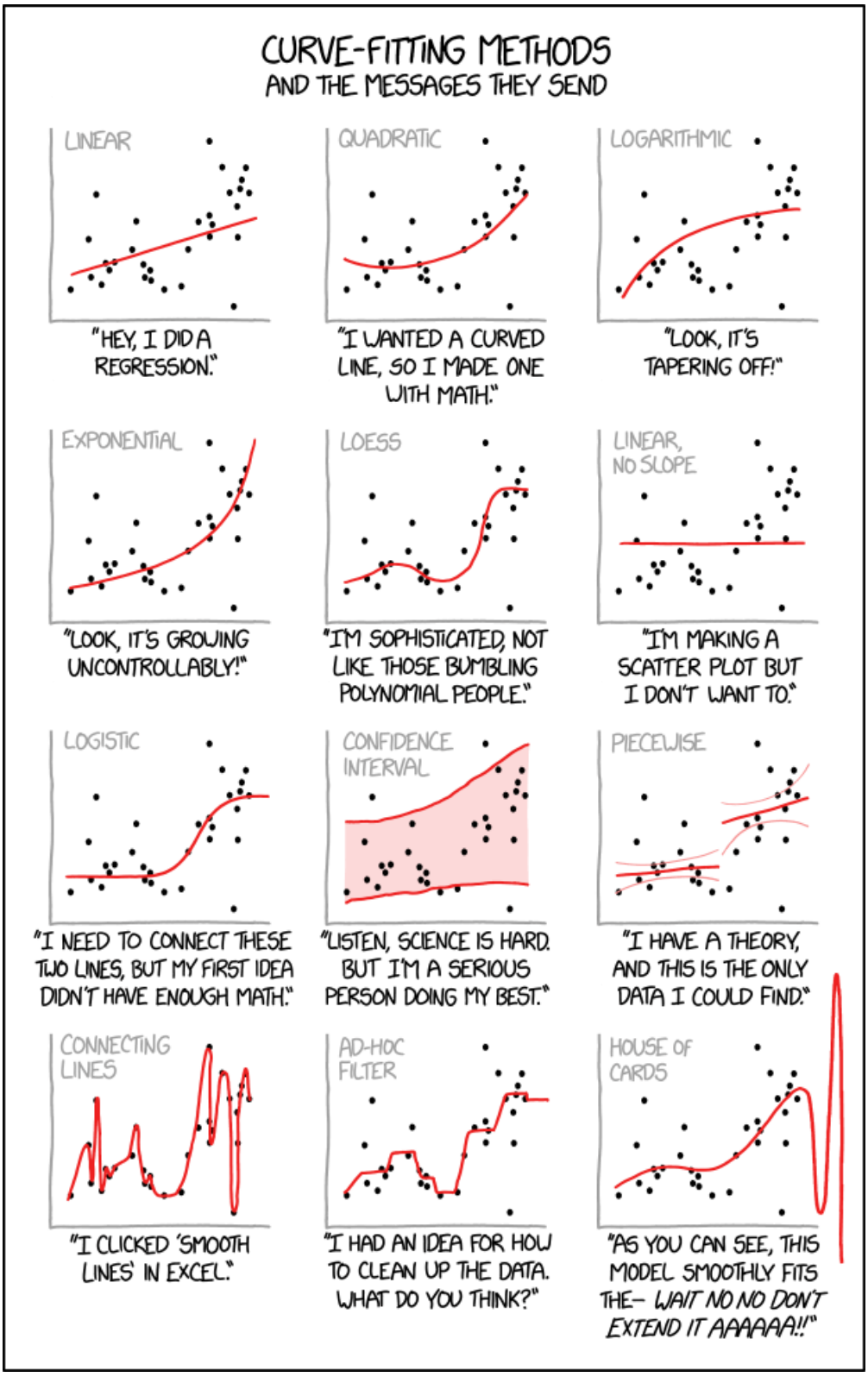

Figure 3.9:: http://xkcd.com/2048 by Randall Munroe. Here are some possibilities for fitting plots using ROOT. If you choose to read the discussion on *statistics* this cartoon may be funnier (or more tragic; such is the nature of physics).

Walkthrough: Saving your work, part 2

(15 minutes)

Note: So now you've got a histogram fitted to a complicated function. You can use *File → 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 *Browser → Quit ROOT* or *File → Quit ROOT* from 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.[1]

```
[] 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[2]:

```
[] 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.

Note: 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.

[1] *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.

Did you just try to use **`ls`** to see these files?

- Congratulations on wanting to learn more about UNIX.
- You didn't see them.

The reason why is that, by default, UNIX "hides" filenames that begin with a `.` (period), so that you don't have to look at work files every time you examine a directory's contents. To see these "invisible" files, the command is:

```
> ls -a
```

[2] I've seen some ROOT documentation that suggests that closing the file is optional, since ROOT usually closes the file for you when you quit the program. However, I've also seen many ROOT files made unreadable because they weren't closed properly. I suggest you always explicitly close any file you open!

There's more about this *later in the tutorial*. Though they seem closely connected at times, the program environment and the ROOT toolkit are two *different* entities.

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

Example experiment n-tuple

(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.

Tip: 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.[1]

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.

What the variables mean

Let's break down 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).

[1] If the names "stutter" (e.g., you see `px.px` instead of just `px`), don't be concerned.
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.
In the scan, ROOT displays both the name of the branch and the name of the leaf within the branch. For an n-tuple, each branch only has one leaf, but `TTree::Scan()` has no way of knowing that, so it displays everything.

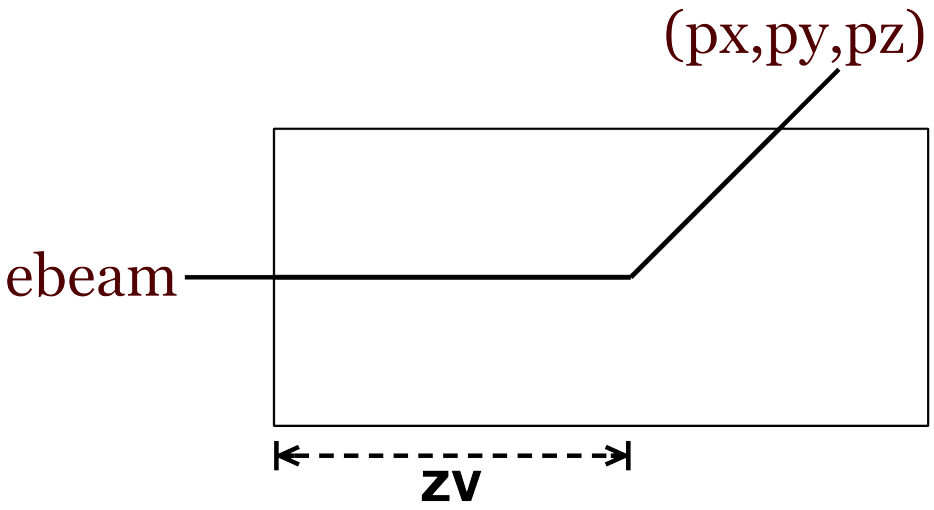

Figure 3.10:: Sketch of the experiment and variables.

What's missing

Did you notice what's missing from the above description? Answer: units. For example, I didn't tell you whether `zv` is in millimeters, inches, yards, furlongs, 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.[2] For `tree1` in `experiment.root`, 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.

[2] *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 probably 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 *Expert Exercises*.

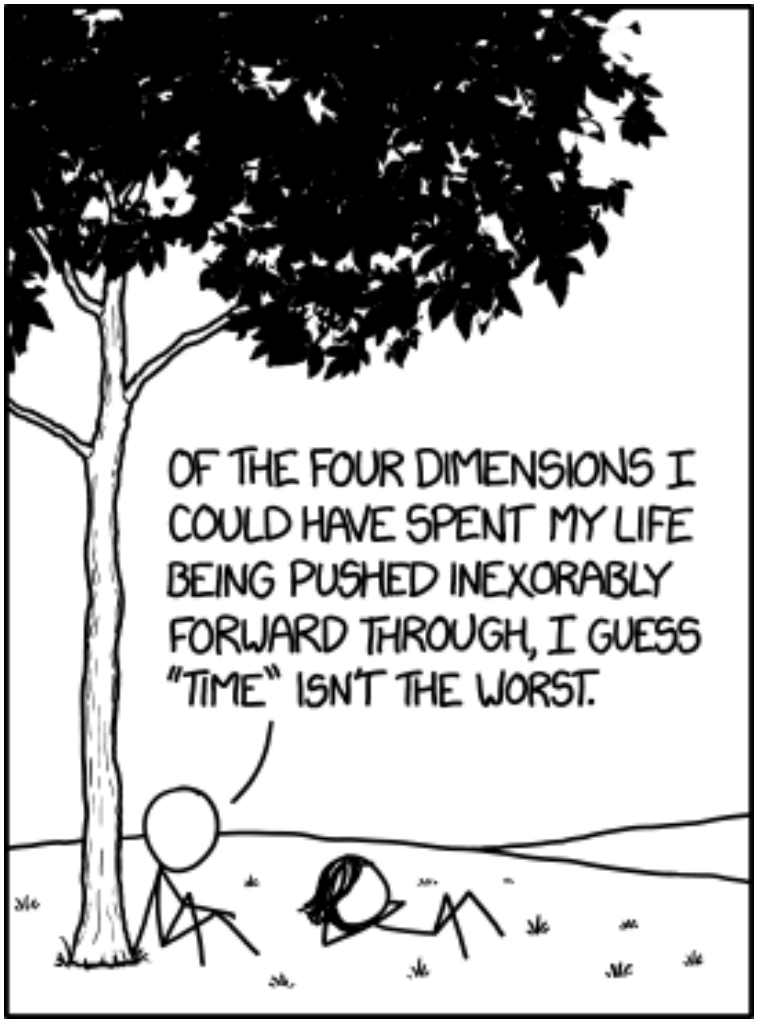

Figure 3.11:: https://xkcd.com/1524/ by Randall Munroe

Using the Treeviewer

Note: If you're going to take *The RDataframe Path*, or terms like "cut" and "scatterplot" are unfamiliar to you, I advise you to work through this page. It introduces concepts that you'll find useful later in this tutorial.

Otherwise, if you feel that this course has been too easy so far, you can skip the TreeViewer. It's trivial to learn on your own if you want to. You can skip ahead to *The Notebook Server*.

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

Note: 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 some amount of coding.

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.

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. Double-click on one of them and look the resulting histogram. Double-click on a few more variables and see how the histogram changes.

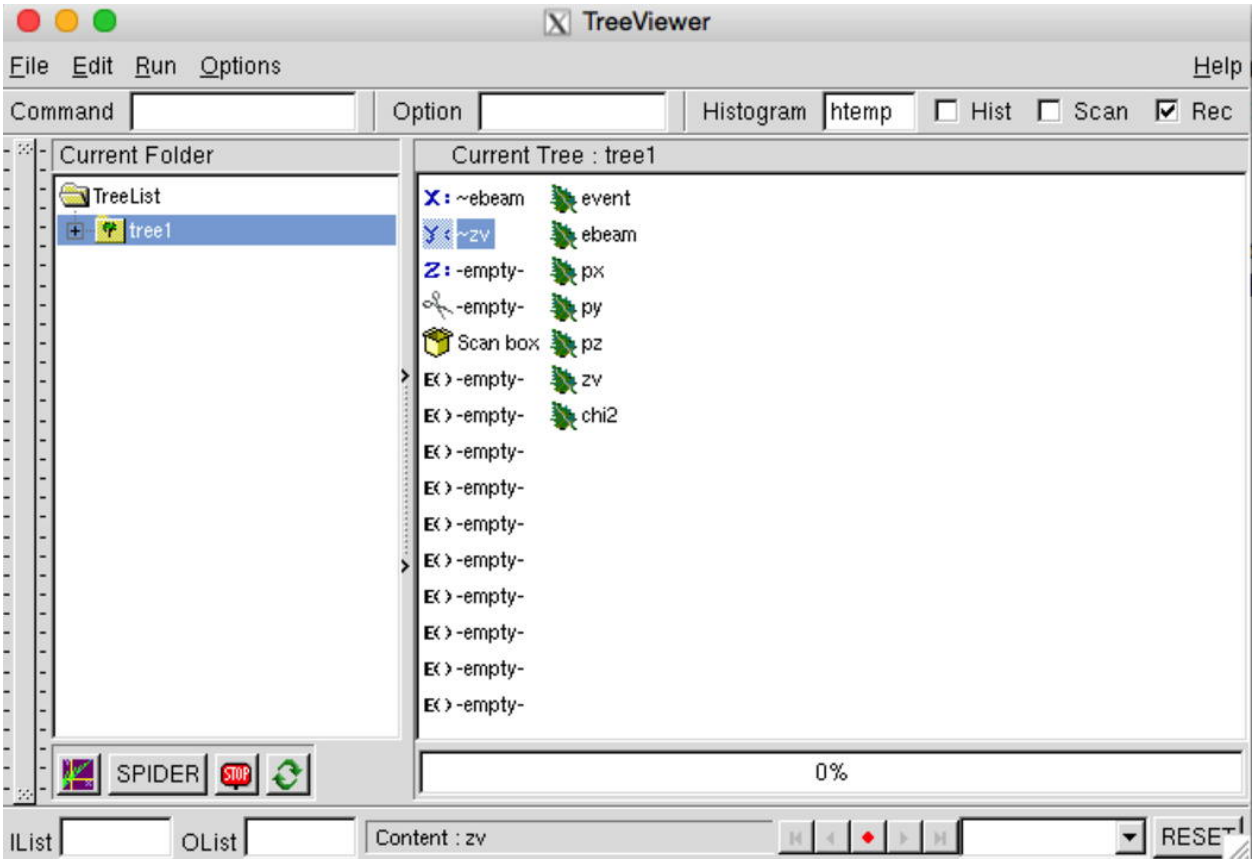

Figure 3.12:: This is what I see when I run TreeViewer on my Macintosh.

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`[1]).

Note: 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?

Tip: 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.

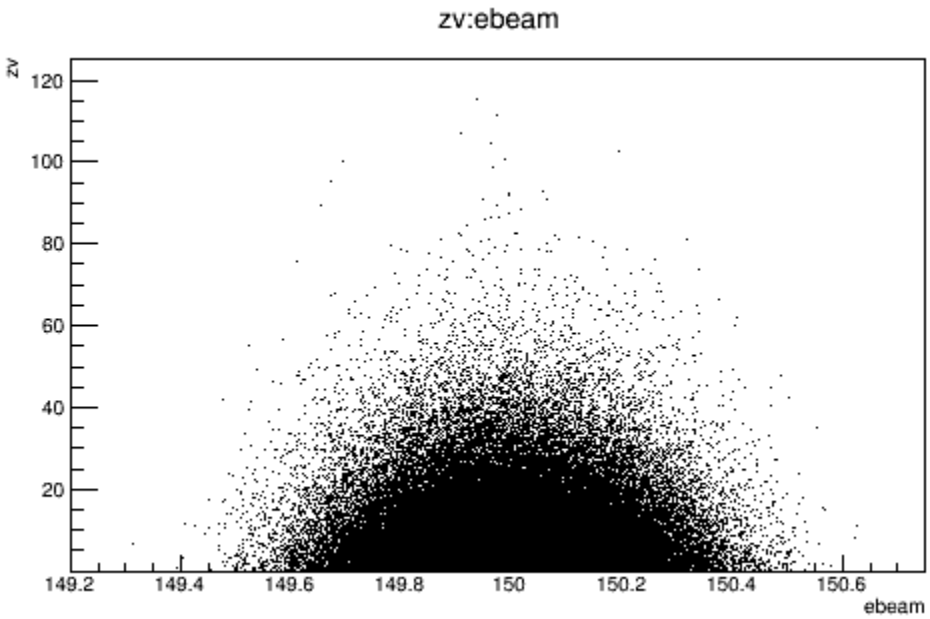

Figure 3.13:: This is what I see when I make a scatterplot of zv versus ebeam. The variables look uncorrelated to me, except that we can't have zv < 0.

[1] Go ahead and click on the SPIDER button if you want. A spider (or radar) chart is a way of displaying multivariant data in a two-dimensional graph.

I've never seen spider charts used in physics, except when I looked up the definition for this tutorial. By the way, if you clicked that link, you just looked up spider charts on the web. (OK, I'm no Randall Munroe.)

New variables: expressions

(10 minutes)

Note: 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`[2] in the box under *Alias*. Then click on *Done*. Now double-click on the word `~pt` in the TreeViewer.

Tip: 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`.[3]

Note: 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*.

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.

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: 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`.

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

[3] 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.

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.[4]

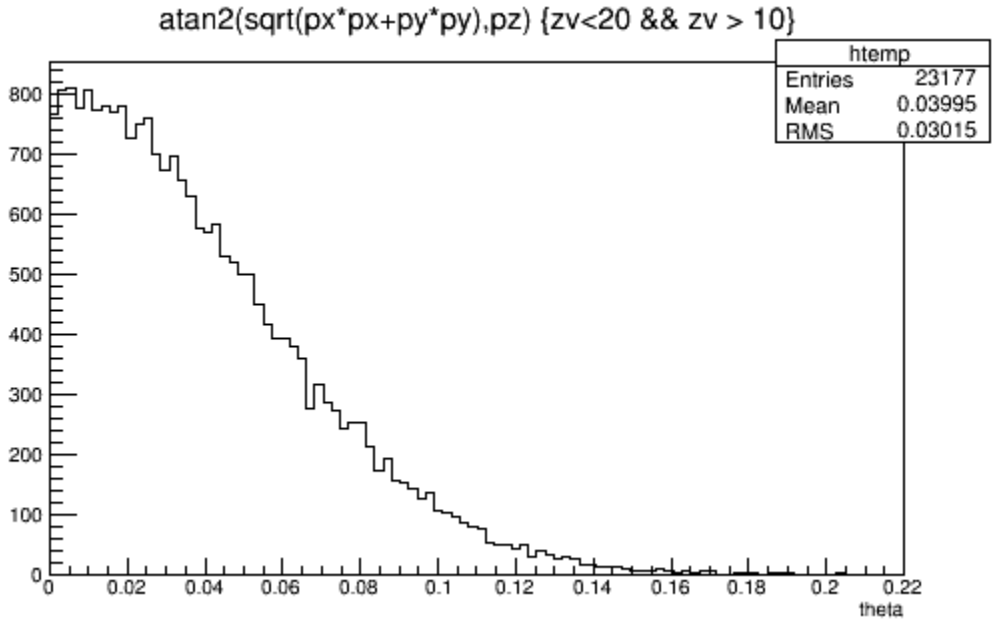

Figure 3.14:: This is what I see when I make a plot of theta with the cut `zv<20 && zv>10`.

Note: If you wanted to display this plot in a talk, you'd have to label both axes (which you've *already learned* how to do) 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.

An obscure ROOT trick

Once, in a quick-and-dirty analysis with TreeViewer, I set up a whole bunch of complex expressions that I knew I'd want to use again. I had a file (let's call it `vertexAnalysis.root`) that I was re-creating over and over again. After I re-made it, I always wanted to view it with the same complicated expressions that I'd set up in TreeViewer.

I looked in TreeViewer's *File → Save Source*. I saved the source as `vertexAnalysis.C`. After that, whenever I re-built `vertexAnalysis.root`, all I had to do was type

```
root vertexAnalysis.C
```

[4] 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.

This would open `vertexAnalysis.root` and the TreeViewer panel would be completely recreated. All my complicated expressions were still in place. I could remake my plots with a simple double-click.

Figure 3.15:: https://xkcd.com/167/ by Randall Munroe. That's why we climb (analyze) TTrees: the future is an adventure, and you don't know what you'll find. And perhaps the first speaker is wrong in other ways.

CHAPTER

FOUR

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, tablet), go to https://notebook.nevis.columbia.edu.[1] You'll be prompted for your Nevis account name (just the name, no `@<server-name>`) and password.[2]

After the you login, there'll be a pause while Jupyter starts up. Eventually you'll see a display that looks something like this:

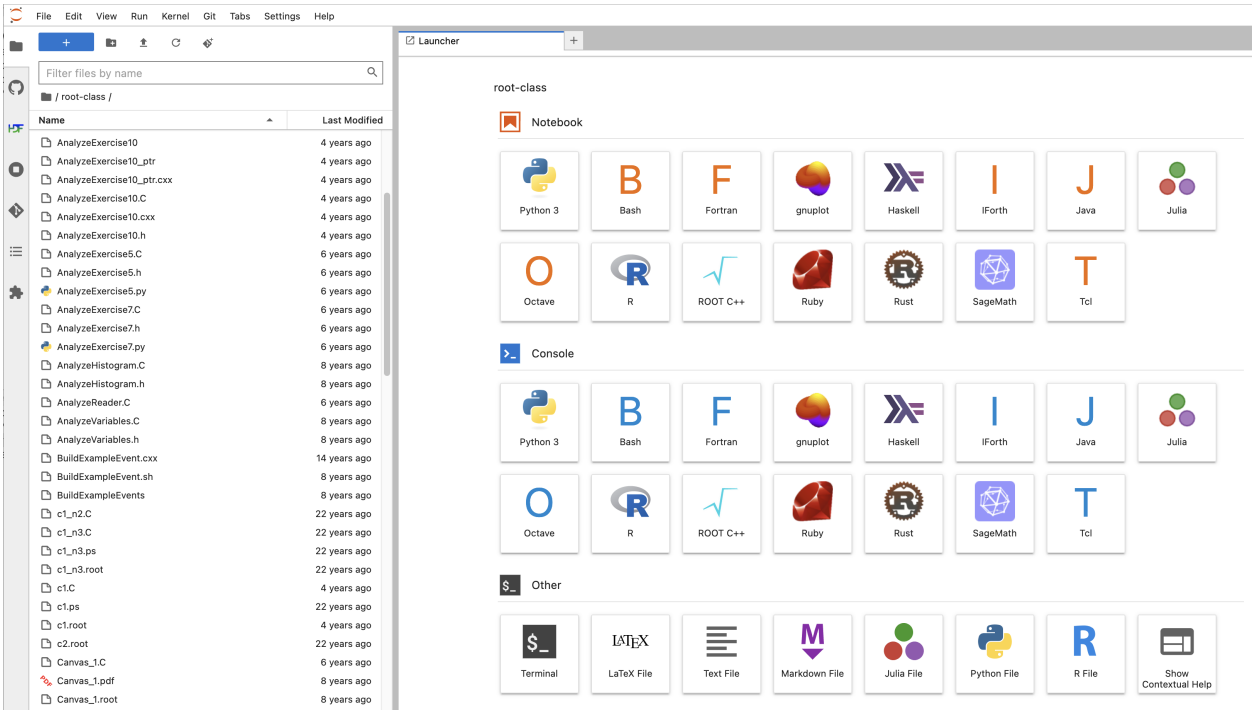

Figure 4.1:: It won't look exactly like this, for a number of trivial reasons.[Page 46, 3]

You are looking at JupyterLab, an browser-based method of running programs.

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

[2] If you don't have an active account on the Nevis particle-physics cluster, 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 *Decisions*.

[3] What are those reasons?

- The placement of the icons will depend on the size of your screen.
- When I made this screenshot, I was visiting my `root-class` directory. You probably haven't copied over all those files.
- If you're not using the Nevis notebook server, you almost certainly won't see nearly as many icons. I've installed as many different languages and environments as I could.

  Would any of those additional languages be of any use to you? If I'm honest, I was just showing off; I installed most of them just to show that I could.

  If you're curious:

  - While Julia might be useful in physics work, I know of no one at Nevis who uses it.
  - If you're used to Matlab or Mathematica, you might want to take a look at the free equivalents Octave or SageMath respectively, which you can see are available on our notebook server.
  - If you want to explore some alternatives to programming in procedural languages (e.g., C++ and Python), you might want to look at Haskell and Forth. They are completely different from each other and from the languages you're used to.

  Here's a complete list of all the Nevis notebook languages and environments.

You'll see your home directory in the left-hand panel. Look at Jupyter's *File* menu, and also right-click on one of the filenames in the left-hand panel. This will give you an idea of the elementary file operations you can do directly from Jupyter.

The fun part is the area on the right with all the icons. This is the Launcher. It's going to "disappear" when get to the next page of this tutorial; if you want to see it again, just click on the **+** symbol you can see in the top area of the window.

While there's a lot to see in the Launcher, there are only three icons that will be relevant to this tutorial:

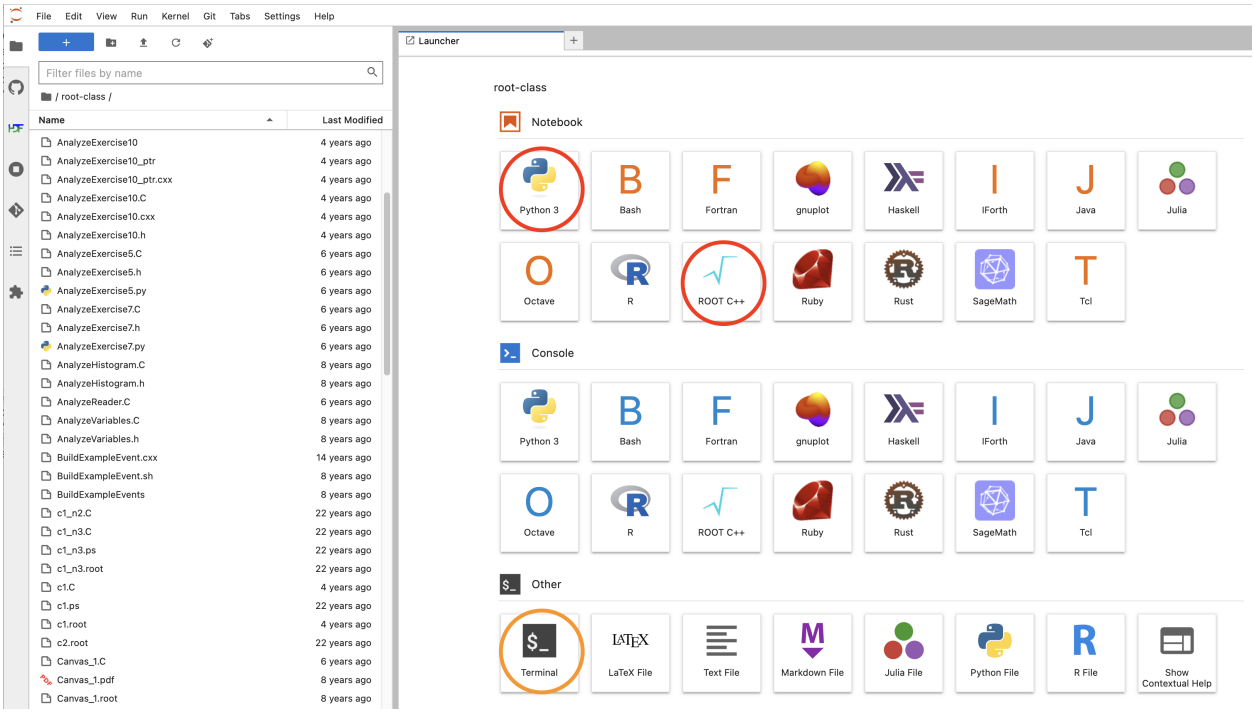

Figure 4.2:: These are the only icons you have to pay attention to for this tutorial. In fact, the `Terminal` icon at the bottom isn't all that important either, which is why I've circled it in a lighter shade.

For extra added Jupyter goodness, mouse over the icons you'll see on the top left, to see the hovertext about what they do.

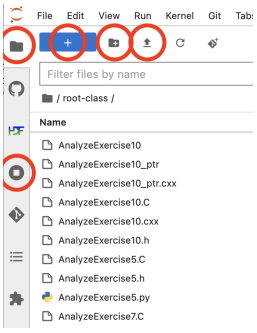

Figure 4.3:: I've circled the auxiliary icons that I think might be potentially relevant to this tutorial or the work you'll do this summer. As your familiarity with Jupyter increases, you'll find use for the other icons.

Jupyter is intended to provide an IDE (Integrated Development Environment) in a web browser.[4] Each one of the **Notebook** icons is a "kernel," that is, an environment for interpreting commands. We'll take a look at one of these kernels in the next section.[5]

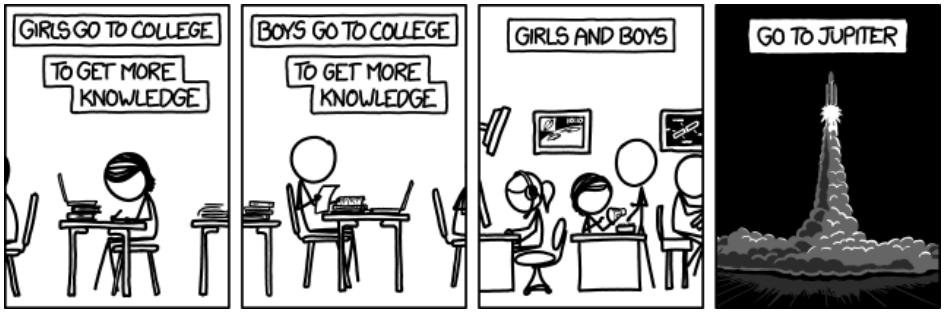

Figure 4.4:: https://xkcd.com/1202/ by Randall Munroe

[4] Jupyter can't do visual programming, that is, creating programs by moving icons around to form diagrams. Or rather, it can't do that yet; Jupyter is under continual refinement. Who knows what we'll be able to do next year?

[5] The icons in the **Console** and the **Other** sections are for more advanced work with the notebook kernels. They're not relevant for this tutorial, but you may want to read more about them if you use Jupyter notebooks for serious programming and debugging.

Your first notebook

(10 minutes)

Select the Python 3 icon. The Launcher tab will be replaced with a new Python notebook. You'll see your first empty cell, labeled `In [1]`.

If you look at the notebook's tab near 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. Go to Jupyter's *File* menu and select *Rename Notebook…* (near the middle of the menu). Pick any name you wish, such as `pythontest`.[1]

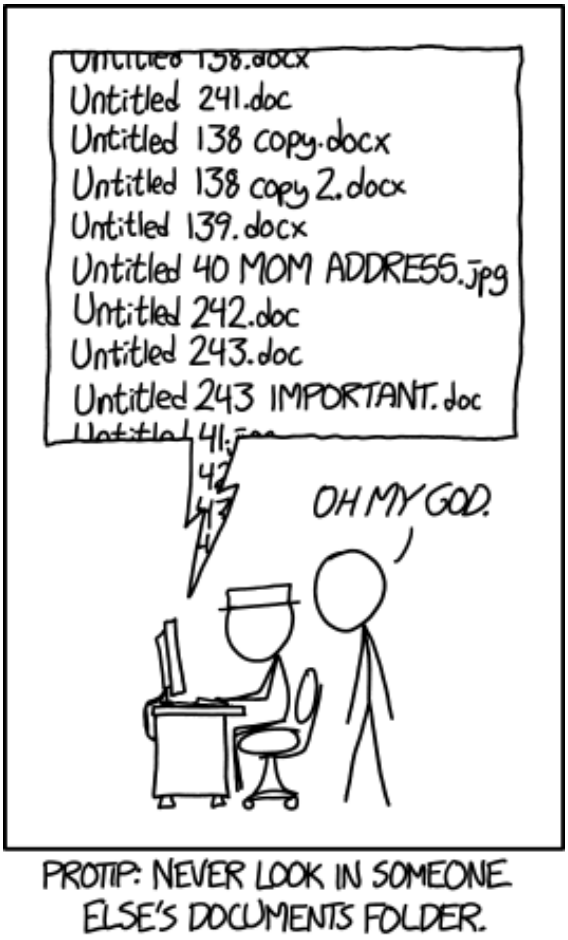

Figure 4.5:: https://xkcd.com/1459/ by Randall Munroe

After you've renamed the notebook, look at your directory listing on the left. You'll see the notebook file with the extension `.ipynb`.[2]

Copy-and-paste the following into that first cell.[3]

[1] 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.

[2] Here are more Jupyter hints. Of particular interest is the list of keyboard shortcuts; for some reason, you can't see the list they mention in more advanced versions of JupyterLab, though the shortcuts are still there.

Some of those tips won't work on the Nevis systems; e.g., you can't install new extensions on the Nevis notebook server. The last tip, on different ways to share notebooks, will be helpful to those who work with folks who don't use Jupyter.

[3] 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.

```
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()
```

Note: This code is in Python, but after going through *The Basics* 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.

Note: 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.

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

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

Note: 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 it 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()
```

Note: 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 *The Basics*, 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.[4]

[4] "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.

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.

Note: *Remember*: 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.[5]

---

[5] 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)

Note: In Jupyter, "magic" refers to additional commands added by Jupyter to the kernel environment that aren't normally part of that kernel's language.[1] 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 just worked with, execute this command:[2]

```
%jsroot on
```

Note: As a general rule, magic commands begin with the percent sign "%".[3]

Draw the canvas again:

```
my_canvas.Draw()
```

Move the cursor over the new plot.

Note: 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.[4] 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` (write the cell to file);
- `%%timeit` (execute the cell many times and determine the average execution time);
- `%%sh` (execute the cell as a UNIX shell script).

---

[1] No, nothing to do with Doctor Strange or Gandalf, though you may find yourself muttering "You shall not pass!" as you work with ROOT.

[2] 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.

[3] 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".

[4] Here are more details about magic commands.

Markdown cells

(5 minutes)

Note: 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,[1] HTML,[2] and LaTeX[3] 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

# 2025 Analysis Project

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

```
##### HTML

<h1>2019 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!

---

[1] Markdown is a simple text-layout layout that emphasizes readability over the methods described in the next two footnotes. Here is a pretty good Markdown tutorial.

Though I tend to prefer HTML (see the next footnote), in 2022 I composed most of this tutorial using Markdown, because it was easier to translate from the original Microsoft Word document.

[2] 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). Here's my favorite HTML tutorial.

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

[3] 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 or Google Docs format.

Don't worry about learning LaTeX. Most people don't write a LaTeX document from scratch; they get one from someone and learn by example. It's much easier than learning ROOT. Here are some Jupyter-related examples.

You can spend a lifetime learning LaTeX, but almost no one ever has. I certainly have not, which is why the PDF version of this tutorial looks sloppy.

The ROOT C++ kernel

(5 minutes)

Note: In *The Basics*, all of the practice ROOT code used C++ syntax. Yet I switched to Python when I introduced Jupyter. Now you'll understand why.

Start a new Launcher (from Jupyter's *File* menu or from clicking on any + near the top of the page). Click on the *ROOT C++* icon to start a new 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 *"your first notebook"* discussion 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();
```

Note: 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.

Note: 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 remind you that in a C++ notebook you still have to keep track of what's an object and what's a *pointer*.

Second, I wanted you to see the warning messages ROOT generates when you redefine objects or pointers that you've created before. You can get away with this in interactive ROOT notebooks, as long as the redefinitions are in different cells or you're executing the same cell again. You *cannot* do this in stand-alone ROOT programs; the C++ specification prohibits redefining variables with the same name.

Don't let this confuse you; the following is fine:

```
int a = 1;
a = 2;
```

What's not allowed in full-fledged stand-alone C++ is:

```
int a = 1;
double a = 2.0;
```

The following will be allowed (with a warning) in a C++ ROOT notebook, as long as the two statements are in different cells. You will get a compilation error in a stand-alone C++ program or if both statements are in the same cell in a notebook:

```
TH1D example("example","example histogram",100,-3,3);
TH1D example("example2","another example histogram",100,-3,3);
```

CHAPTER

FIVE

DECISIONS

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

RDataframe or write the code?

`RDataFrame` is a ROOT class that does just one thing, but does it really, really well: It goes through the entries in a ROOT file and does… well… *something* with each entry.

Using `RDataFrame`, each of the main Exercises in this tutorial (2 through 9) can be written in a few lines of code. You don't need to create event loops with macros or analysis skeletons; the `RDataFrame` class and its associated methods handle all of that.

With that build-up, you're probably strongly tempted to just click through to *The RDataframe Path* and get started. Before you do that, you may want to consider some reasons to take *The Python Path* or *The C++ Path* instead:

- As I said, `RDataFrame` is designed to loop through every entry in a file. That describes a large portion of typical physics analysis tasks. The whole *raison d'être* of this tutorial is to teach you exactly that.

  However, that's not the only analysis task you may be asked to do this summer. In fact, none of the *Advanced Exercises* or *Expert Exercises* can be done in this way. So you may to take go through the coding portions of this tutorial, in order to prepare you for more challenging tasks.

- The `RDataFrame` class is a relatively new addition to ROOT.[1] It's possible your supervisor has never heard of it.

- It's easy to use to use `RDataFrame`… at first. There's point at which you hit a *"wall"*: You suddenly have to understand about C++ functions and lambdas, even if you're doing your work in Python.

  To give you an idea how complex using `RDataFrame` can get, consider these advanced examples in C++ or Python; they are from the ROOT dataframe tutorials and demonstrate Higgs-boson reconstruction. If you just glance at those examples, you'll confirm for yourself that `RDataFrame` doesn't keep you from learning how to code.

Now that I've scared you, let's look at the reasons to use `RDataFrame` for this tutorial:

- Some students have had difficulty getting through *The Python Path* or *The C++ Path* portions of the tutorial in the time we have available. If you do the *The RDataframe Path*, you're almost guaranteed to complete the whole thing.

- Although I only show examples using the n-tuple `tree1`, you can also use other file formats as input to dataframes; e.g., TTrees and CSV files.

- It's easy to make `RDataFrame` *multi-threaded*, which can greatly speed up the execution time of its operations.

With all that said, what you might consider doing is working through *The Python Path* or *The C++ Path* up to Exercise 10, then do *The RDataframe Path*. After doing Exercises 2 through 9 using code, doing the same Exercises using `RDataFrame` will take very little time.

[1] The current RDataFrame class was introduced in ROOT 6.14 (June 2018). From ROOT 6.10 to 6.12, the class was called `ROOT::Experimental::TDataFrame`. Prior to 6.10, you won't find dataframes in ROOT at all. Since this is an actively evolving feature of ROOT, you'll want to check which version of ROOT your collaboration uses.
The Nevis notebook server uses the latest stable version of ROOT, but collaborations often stick with a particular older ROOT version.

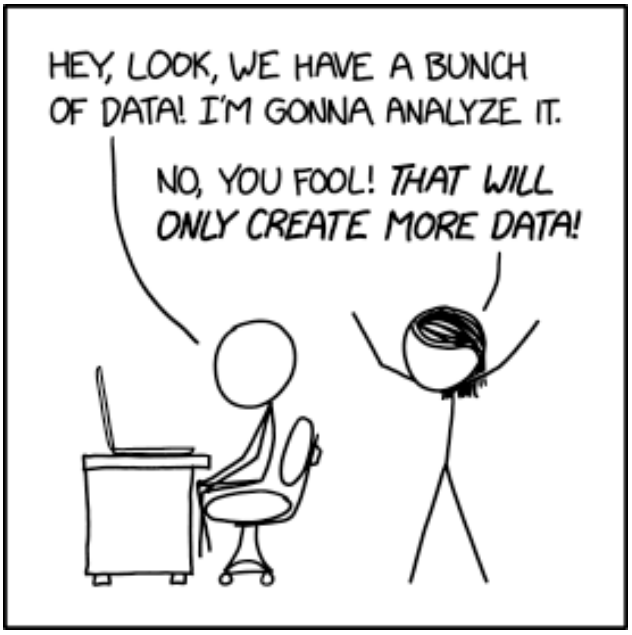

Figure 5.1:: https://xkcd.com/2582/ by Randall Munroe

C++ or Python?

Up until this point, the commands for ROOT/C++ and Python/ROOT were nearly identical.[1] 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 ("pyroot"). 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:[2]

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, the *Advanced Exercises*, the *Expert Exercises* of this tutorial, and most of the *tutorials included with ROOT* 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 this *discussion*).[3]

---

[1] See *here* for the differences when using Python versus ROOT/C++.

[2] 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++).
- Rivals to C++ and Python include (respectively) the Julia programming language and the Ruby scripting language. As far as I know none of the particle-physics groups connected with Nevis use them.

[3] 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 *The Basics*, or the notebook as in *The Notebook Server*.

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 *here*.
- 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. It's possible your supervisor has never heard of them. If you say you've got a .png plot in a Jupyter notebook, they'll reply "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.[1]
- As you saw *before*, your canvases are not automatically drawn or updated for you. You must explicitly issue `Draw()` commands for your canvases.

Issues with the Nevis 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.
- You can develop software in notebooks, but you can't run multi-hour jobs with them on our notebook server.[2]
- 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 may not be able to run software that's been compiled on another machine.[3]

[1] 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!

[2] The way to handle such tasks is with *Batch Systems*.

[3] I doubt this will be important to your work this summer, but so you can look it up if necessary: In 2023, most of the particle-physics systems are running CentOS 7.

- 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`.[4] If you add new variables to your environment, these variables are available in our notebook server as well.[5]

  However, if you modify certain variables such as `$LD_LIBRARY_PATH` or run environment-customization scripts (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 Nevis workgroup server.

Figure 5.2:: https://xkcd.com/934/ by Randall Munroe

[4] Here's a list of shell initialization files, at least for the systems at Nevis.

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. A 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. These init scripts were copied from user to user for decades.

If you edit your scripts yourself, at least you have a chance to maintain them.

[5] 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 a 2x2 decision matrix

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

Command-line	Notebook
ROOT C++	ROOT C++
Python with ROOT	Python with ROOT

This tutorial is already too long, and I've 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 *The C++ Path*, and Python with ROOT in a Jupyter notebook in *The Python Path*. For *The RDataframe Path*, these choices don't matter much.

If you choose to pursue one of the "off-diagonal" choices, you won't have much trouble no matter which path you choose. You were previously introduced to *ROOT C++ in a notebook*. 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
```

The C++ Path and *The Python Path* present the same commands, exercises, and footnotes.[1] You might even be able to do both of these parts; once you've learned C++, Python is pretty easy. Tack on *The RDataframe Path* for total ROOT mastery![2]

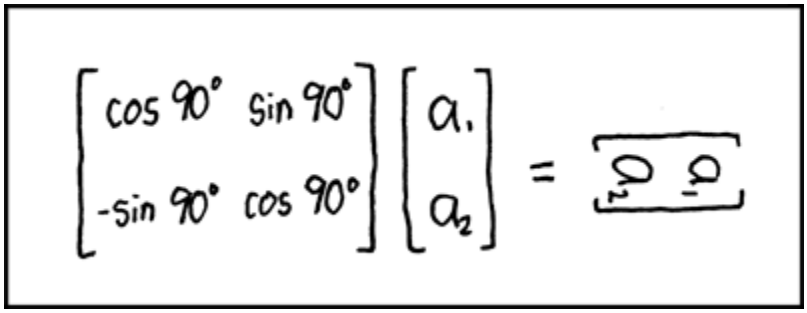

Figure 5.3:: https://xkcd.com/184/ by Randall Munroe

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

[2] Uh, no. Remember, it takes a lifetime to learn ROOT. While it may take more than two days to go through all three instructional paths in this tutorial, it won't take your entire lifetime.

CHAPTER

SIX

THE C++ PATH

Congratulations on choosing to work with C++. It's obviously better than *The Python Path* or *The RDataframe Path*.[1]

There are a lot of topics listed below. Don't let that discourage you. Many of the sections below are quite short.[2] Again, the important thing is for you to learn something from doing what exercises you can in the time we have.

[1] For one thing, the xkcd cartoons are funnier.

[2] Many of the sections don't have any footnotes or cartoons at all, just to make them shorter.

Differences between C++ and Python

Many of the Nevis summer students have some previous experience with Python, but not with C++. Here's a quick overview of the differences to get you started. *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.

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

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

- 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++ control structures (e.g., `if` statements, loops) are indicated by curly braces ({}).[1] 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;
```

- 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 "->":

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

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

- 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.[2] Python determines types dynamically, and you rarely have to specify them:

[1] 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).

[2] 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>>::const_iterator`. However, you have to be comfortable with C++ before using it, which is why I'm relegating this C++ tip to a footnote.

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:

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

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

- The ROOT C++ interpreter, cling, knows the names of all the ROOT classes. You can type the following directly into interactive ROOT without anything that resembles Python's `import` statement:

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

However, if you write a stand-alone program in C++, you have need a `#include` for the header of any class that's not part of the standard C++ specification:[3]

```
#include <TH1D.h>
TH1D* example4 = new TH1D("example4","my fourth histogram",100,-3,3);
example4->FillRandom("gaus",10000);
```

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

Now you know why it takes a lifetime to learn C++!

[3] This is crucial if you decide to work on *Exercise 11*.

Walkthrough: A simple analysis using the Draw command

(10 minutes)

Note: 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 *The Basics*: the Draw command, the `Treeviewer`, the **FitPanel**, and other simple techniques discussed in the early chapters of 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 did with `Treeviewer`.[1]

If you didn't copy the example n-tuple file, do so now:

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

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()
```

Note: 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 (remember *the second part of Saving your work*?) 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.

[1] 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 'object,' it's a 'pointer' to the location in computer memory where the object **`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. ROOT uses pointers for almost all the code it creates. *As I mentioned* previously, 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 6.1:: https://xkcd.com/138/ by Randall Munroe

Walkthrough: A 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")
```

Note: 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 a pair of 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?

Note: 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: 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.

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")
```

Walkthrough: Using C++ to analyze a Tree

(10 minutes)

Note: You can spend a lifetime learning all the in-and-outs of object-oriented programming in C++.[1] 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.[2]

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
```

Note: 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.
- **Initialization** - 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. They describe more than we're going to use, so I'll narrow things down below.[3]

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, Initialization, 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.[4]

[1] That's four lifetimes, five if you're studying LaTeX. 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.

[2] 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 on the `TTreeReader` page, 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`.

[3] 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 core of your computer.

PROOF is now deprecated by the ROOT developers. If, by some rare chance, you do use PROOF, note that `SlaveBegin` and `SlaveTerminate` are where you put your initialization and wrap-up code, respectively, and `Begin` and `Terminate` should be "stubs."

By the way, PROOF has nothing directly to do with *batch processing*, which I describe in one of the *optional appendices* of this course.

There's another ROOT class that can speed up n-tuple analysis on machines with multiple cores; see *The RDataframe Path*.

[4] You can copy the "reduced" file from my area if (A) you're pressed for time, or (B) just feeling lazy today.

Listing 6.1: Example C++ TSelector macro

```
#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 *********
}
```

Compare this with the Python code in Listing 7.1.

```
> 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")
```

Note: 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![1]

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 creating 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 initializations;
  - 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.

Note: 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.

---

[1] If you're a real ROOT honcho (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 experiment.root
```

This means to run ROOT and to open file `experiment.root` right away. I can omit the `TFile` command and get to work.

If you want to be even faster (and who doesn't want to be?) you can sometimes skip messing with the `TBrowser`. After you've opened a file, try the ROOT interpreter command:

```
[] .ls
```

This will list the contents of the file on the terminal.

Walkthrough: Making a histogram with Analyze

(15 minutes)

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

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

Note: 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.[1]

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 Initialization section, insert the following code:

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

Note: 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 rather than fixed; e.g., they depend on user input.

In the Loop section, put this in:

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

Note: The first of these two lines means "get the '(entry+1)-th' row from the `TTree`"; e.g., if **`entry`** is 100, get the 101st row from the n-tuple.[2] Note that the variable **`entry`** comes from 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.[3] 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 "in the histogram **`chi2Hist`** add 1 to a bin that corresponds to the value of **`*chi2`**."

This goes in the Wrap-up section:

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

[1] 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.

Some compilers offer another name for `NULL`: `nullptr`. They both have the same value, but for you one may be clearer than the other.

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

Note that when **`entry`** is 0, `GetEntry` will fetch the first row in the n-tuple; that's why when **`entry`** is 100, `GetEntry` will fetch the 101st row in the n-tuple.

[3] 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; do a web search on "TSelector example" for some ideas. 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.

Note: 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")
```

Note: Finally, we've made our first histogram with a C++ analysis macro. In the Initialization 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 *File* → *Save* → *c1.C*. 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 Initialization section, right after the line that defines the histogram. Test the revised Analyze class.

Exercise 2: Add 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*.

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!

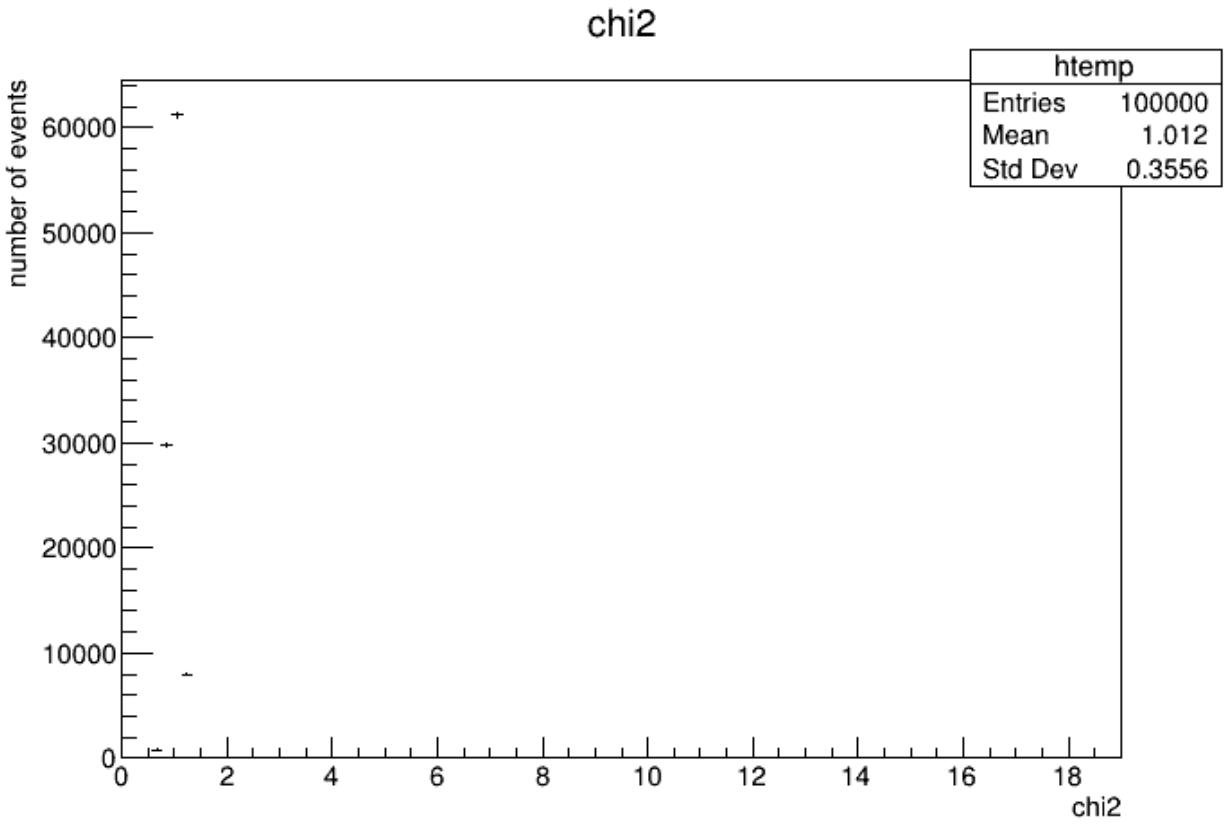

Figure 6.2:: What I get when I plot chi2 with errors bars turned on.

Note: See *this note in the Python tutorial* for how I made this plot.

Exercise 3: Drawing 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).

Warning: Take care! Recall that I broke *a typical physics analysis task* into pieces: Definition, Initialization, 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. 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 a 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: Display fit parameters

(10 minutes)

Fit the **ebeam** histogram to a Gaussian distribution.

Note: 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.

Note: 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 in *Exercise 1*.)

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: Scatterplots

(10 minutes)

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

Hint: Remember back in *Exercise 1*, I asked you to figure out the name `TF2` given that the name of the 1-dimensional function class was `TF1`? Well, the name of the 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.

To fill a one-dimensional histogram, e.g., `chi2hist`, you use a `Fill` command with a single argument:

```
chi2hist->Fill(*chi2);
```

How do you think you might fill a two-dimensional histogram? Check your answer on the ROOT web site.

Walkthrough: Calculate our own variables

(10 minutes)

Let's calculate our own values in an analysis macro, starting with **pt**, *from our Treeviewer exercise*. Let's begin 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):[1]

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

Note: 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.[2]

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 "()".[3]

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.

---

[1] You also have to put in that `GetEntry` line, which I complained about *earlier*.

[2] If you already know C++: the reason why I don't simply tell you to use the built-in types `float`, `double`, `int`, and `bool` is discussed in Chapter 2 of the ROOT Users Guide.

However, if you decide to use the simpler names for the numeric types, I won't mind. You're not likely to use any of the code in this tutorial in another C++ compiler. But bear in mind that when you start programming "for real", it's not certain which compilers you'll use in the future.

[3] 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: Plot 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*.

Note: 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:[1]

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

---

[1] 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 when 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.

Exercise 7: Trigonometric functions

(15 minutes)

Revise AnalyzeVariables.C to include a histogram of ***theta***.

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

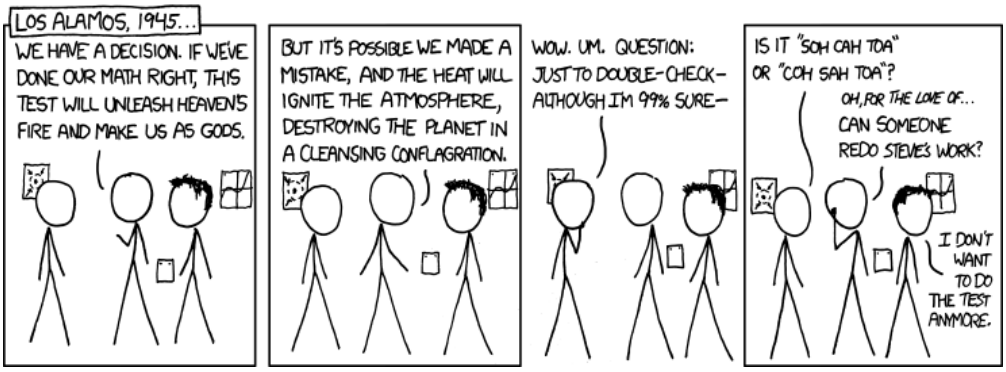

Figure 6.3:: https://xkcd.com/809/ by Randall Munroe

Walkthrough: Apply a cut

(10 minutes)

Note: 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;
```

Note: 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![1]

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
}
```

Note: 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;
```

Note: 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.

[1] 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 $2^{63} = 2^3 \times (2^{60}) = 2^3 \times \left(2^{10}\right)^6 \approx 2^3 \times \left(10^3\right)^6 = 8 * 10^{18}$ 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: Pick 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
```

Note: 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.

Write a macro to create 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.

Note: 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.

Note: 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 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 little 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: Write 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.

Hint: In *Saving your work, part 2*, 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: A stand-alone program (optional)

(60 minutes or more if you don't know C++)

Note: 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.

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 last day of this course, 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.

Note: 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 (here's a *hint*).

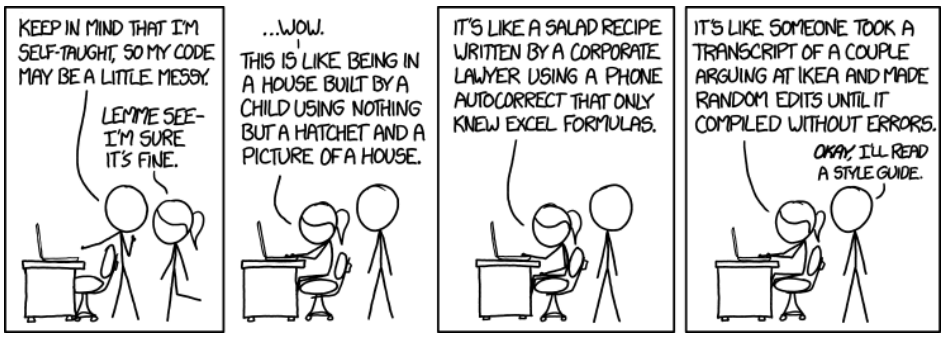

Figure 6.4:: https://xkcd.com/1513/ by Randall Munroe

Note: 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. Use its `--help` option to see what it can do:

```
> root-config --help
```

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: it executes whatever command is between the backticks, and returns the output as a text string. 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.[1]

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.

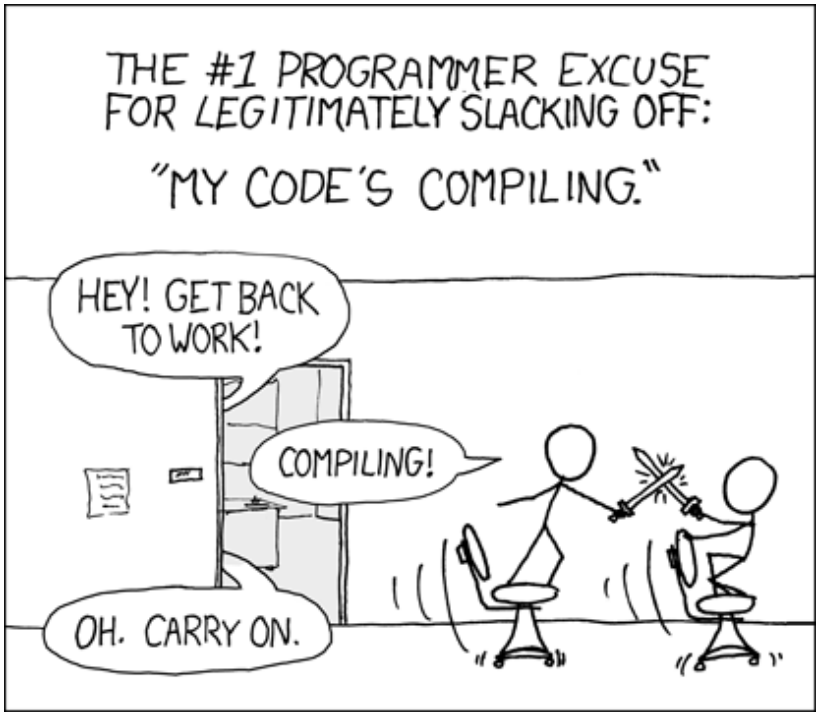

Figure 6.5:: https://xkcd.com/303/ by Randall Munroe

[1] This is esoteric and you probably don't need to know this, but you know how much I hate to lie to you:
In the UNIX shells bash, zsh, and ksh, a better approach to capturing a command's output is to use `$(command)` instead of `` `command` ``; e.g., `$(root-config --cflags)`. But the backtick also works in environments where `$()` does not, e.g., in tcsh and scripting languages like Perl and Ruby.
On Nevis particle-physics systems, most people are set up with bash or zsh. The electronics-design group uses tcsh (since that's what the design group at CERN uses). If you're not at Nevis, I have no idea which shell you're using, which is why I went with the backticks.

CHAPTER

SEVEN

THE PYTHON PATH

Congratulations on choosing to work with Python. It's obviously better than *The C++ Path* or *The RDataframe Path*.[1]

There are a lot of topics listed below. Don't let that discourage you. Many of the sections below are quite short.[2] Again, the important thing is for you to learn something from doing what exercises you can in the time we have.

[1] For one thing, the xkcd cartoons are funnier.

[2] Many of the sections don't have any footnotes or cartoons at all, just to make them shorter.

A brief review

(5 minutes)

Start up Jupyter if it's not already started. At Nevis, this means to visit https://notebook.nevis.columbia.edu, type your Nevis account name and password, then click the **Python 3** icon under **Notebook** 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 cell, 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()
```

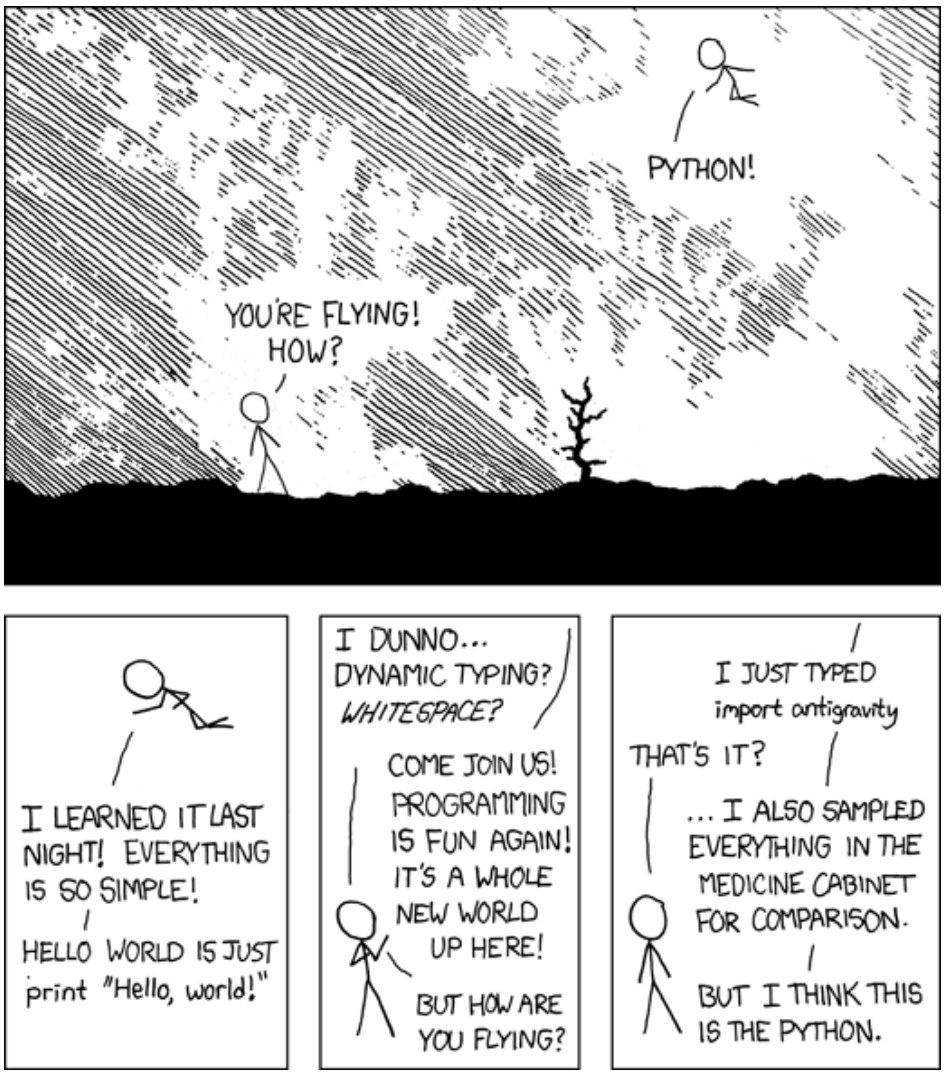

Figure 7.1:: https://xkcd.com/353/ by Randall Munroe

If you worked on *The C++ Path*, you may have noticed that 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 import the ROOT package, pyroot. There are two ways to do this:

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.[1] If you use the second method, be aware that if you include 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
```

At Nevis, all the various Python distributions I've installed include pyroot. If you're working on a different system, you may have to install pyroot yourself if it hasn't already been installed for you. The exact procedure depends on how your local Python distribution is managed. The following command (or a variation) *might* work:

```
pip install --user pyroot
```

[1] You may know of 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 some masking tape and use it to cover this footnote.

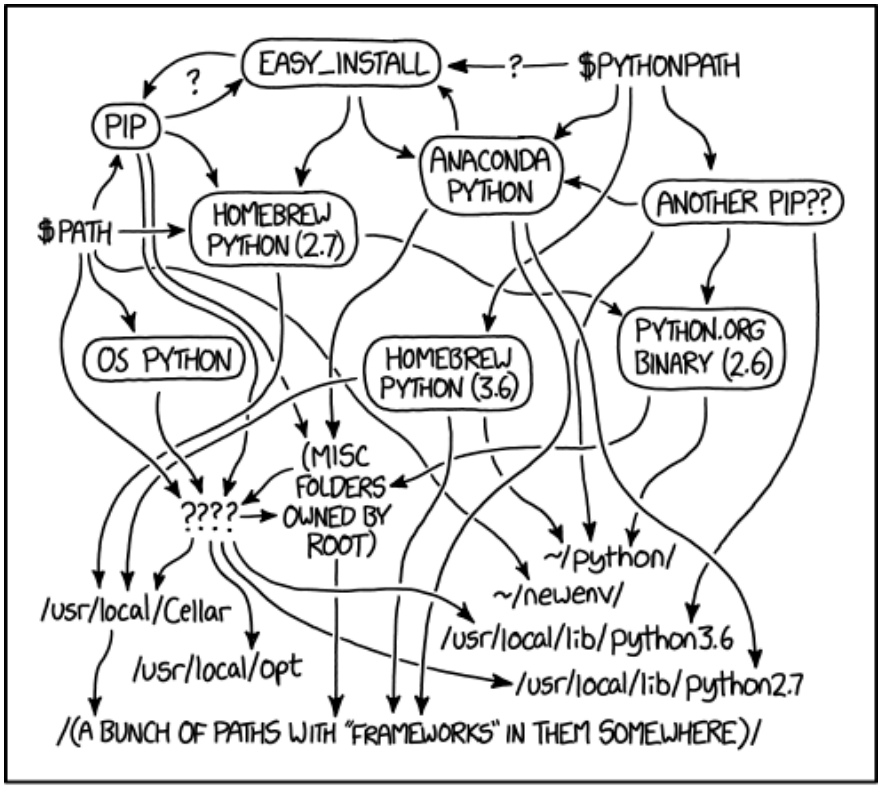

Figure 7.2:: https://xkcd.com/1987/ by Randall Munroe

Walkthrough: Simple analysis using the Draw command

(10 minutes)

Note: 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 early chapters of 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.[1,2]

If you didn't copy the example n-tuple file in *The Basics*, do so now:

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

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

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

Note: 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 *The C++ Path*, 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, 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.

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

[2] 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 the *Advanced Exercises* and *Expert Exercises*) 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 pages on these standard Python tools 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()
```

Note: 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?

Note: 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: 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.[1]

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")
```

---

[1] After this point, I won't include the `my_canvas.Draw()` line in future examples, and 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.

Walkthrough: Using Python to analyze a Tree

(10 minutes)

Note: You can spend a lifetime learning all the in-and-outs of programming in Python.[1] 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
```

Note: The second of the two magic commands will load the contents of `Analyze.py` into the next notebook cell, 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.

Here's the Python code from `Analyze.py`. I've marked the places in the code where you'd place your own commands for Set-up, Loop, and Wrap-up.

Listing 7.1: Python analysis "skeleton" for a ROOT n-tuple.

```
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 in a file that someone gives you.

mychain = gDirectory.Get( 'tree1' )
entries = mychain.GetEntriesFast()

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

(continues on next page)

[1] We're up to at least four lifetimes, five if you completed *The C++ Path*, possibly six if you're learning LaTeX from scratch, maybe even seven if you skipped ahead to *Statistics*.

(continued from previous page)

```
for jentry in range( 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
###
```

Compare this with the C++ code in Listing 6.1.

Note: 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 *before*.

Walkthrough: Using the Python 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.[1]

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()
```

Note: 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.

Note: 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()
```

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 *The Basics* 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 *File* → *Save* → *c1.C*. I looked at c1.C and saw these commands in the file:

[1] 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
```

```
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.[2]

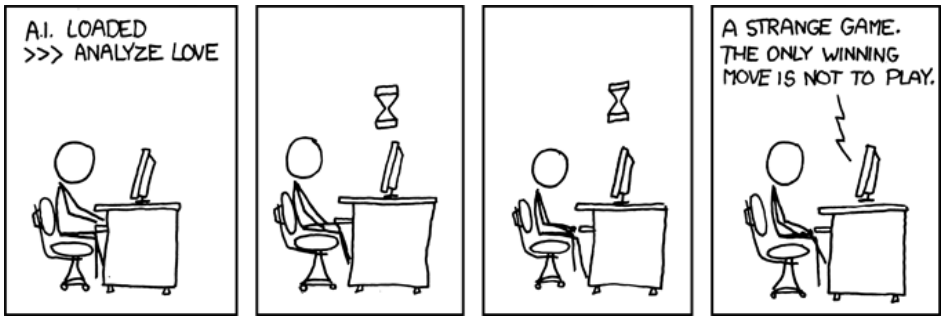

Figure 7.3:: https://xkcd.com/601/ by Randall Munroe. Fortunately it's easier to analyze histograms than it is to analyze love. At least it is for me!

[2] 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. (Here's a *hint*.)

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.

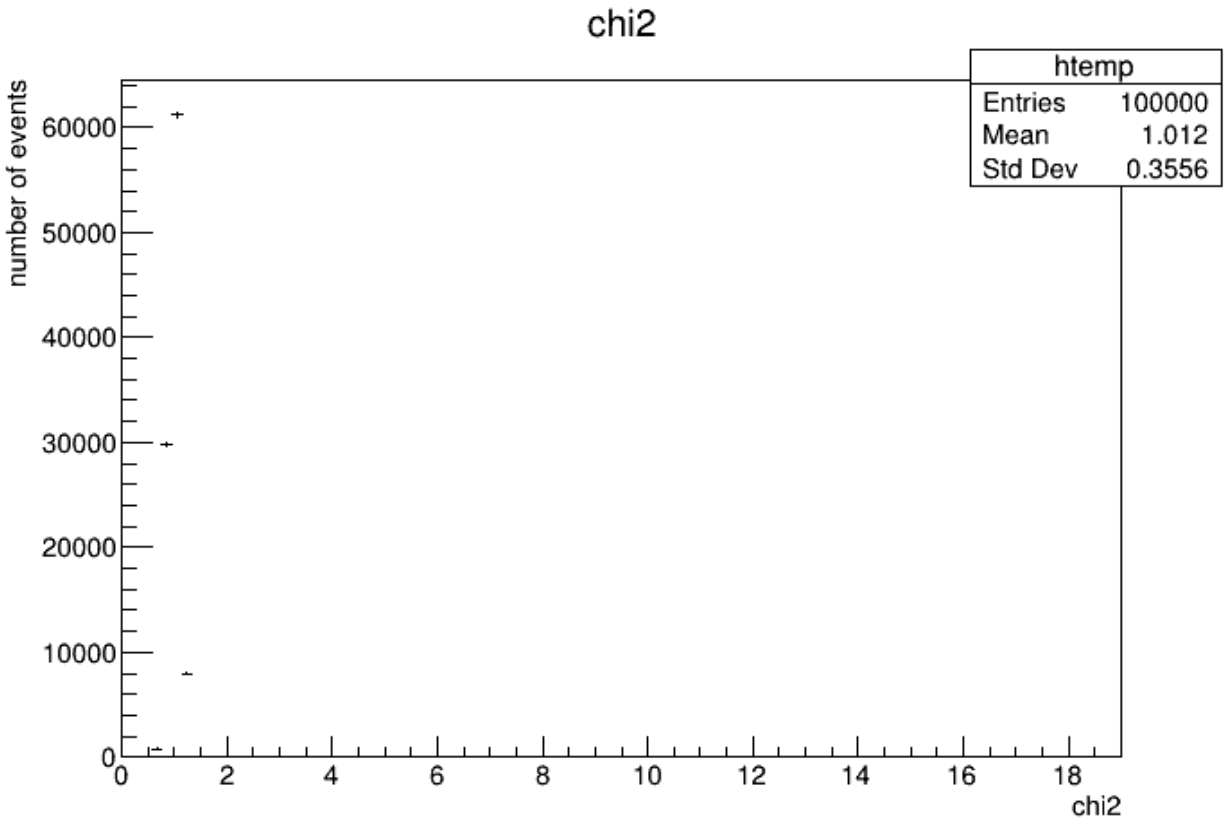

Figure 7.4:: What I get when I plot chi2 with errors bars turned on.

Note: In case you're interested, the code below is how I made the above plot. I learned to use `gPad` to access the temporary histogram from reading the documentation for `TTree::Draw()`. I learned about `SetTitleOffset` by reading the `TAxis` documentation, which led me to the list of `TGaxis` methods. Yes, it was *detective work* all over again!

Listing 7.2: How to modify a plot automatically generated by ROOT

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

(continues on next page)

(continued from previous page)

```
tree1.Draw("chi2","","e")
# Get the temporary histogram used by TTree::Draw()
htemp = gPad.GetPrimitive("htemp")
htemp.GetXaxis().SetTitle("chi2")
htemp.GetYaxis().SetTitle("number of events")
htemp.GetYaxis().SetTitleOffset(1.5)
my_canvas.Draw()
```

Exercise 3: Two histograms in the same loop

(15 minutes)

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

Warning: Take care! Recall that 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. 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 a look at the lower and upper limits of your histogram. What is the range of **`ebeam`** in the n-tuple?

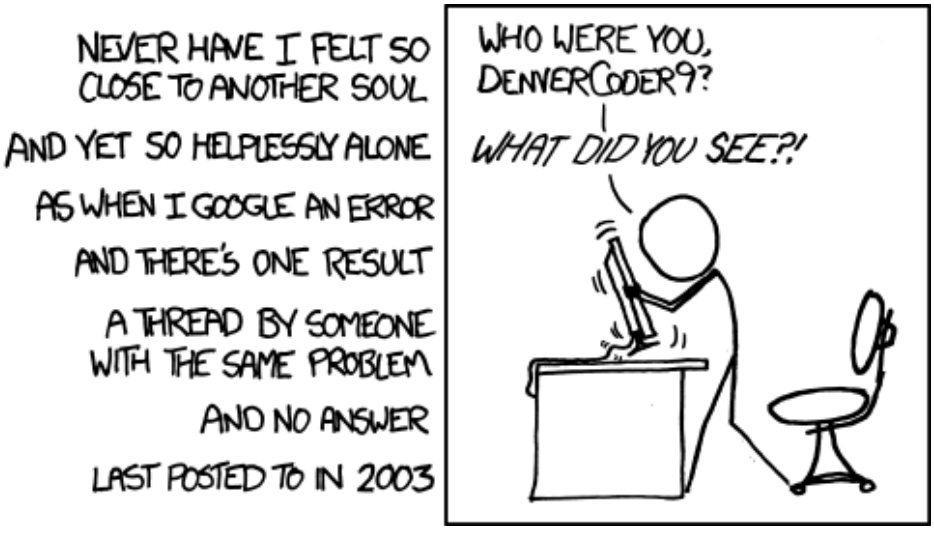

Figure 7.5:: https://xkcd.com/979/ by Randall Munroe

Exercise 4: Displaying fit parameters

(10 minutes)

Fit the **ebeam** histogram to a Gaussian distribution.

Note: 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.

Note: 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*).

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!

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.

To fill a one-dimensional histogram, e.g., `chi2hist`, you use a `Fill` command with a single argument:

```
chi2hist.Fill(chi2);
```

How do you think you might fill a two-dimensional histogram? Check your answer on the ROOT web site.

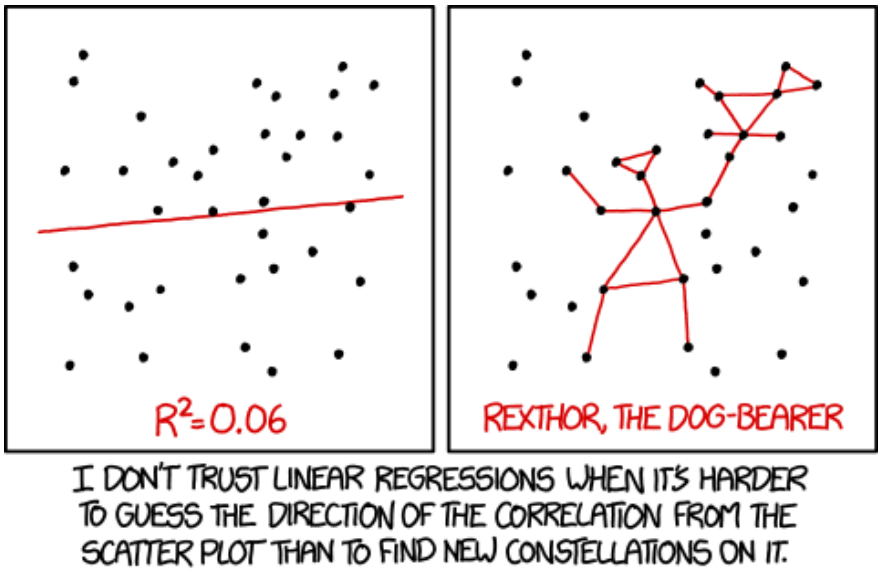

Figure 7.6:: https://xkcd.com/1725/ by Randall Munroe

Walkthrough: Calculating our own variables

(10 minutes)

Note: 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:

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

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)
```

Note: 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 have either `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 "()".[1]

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

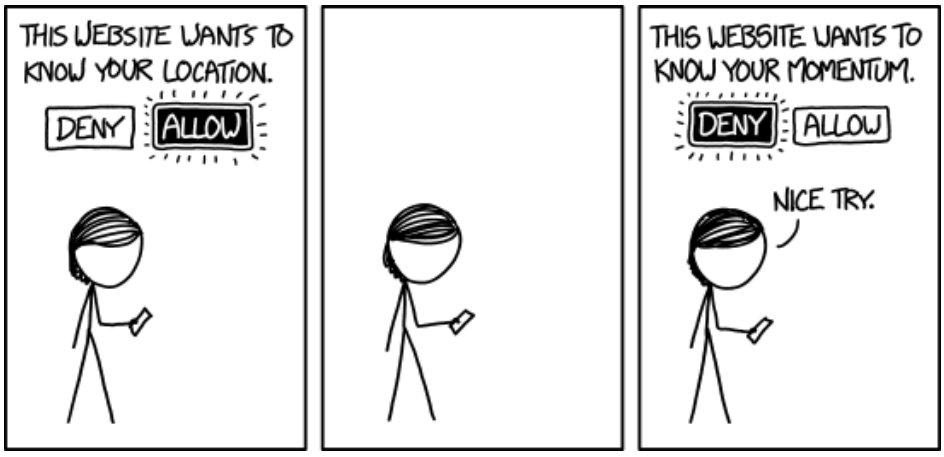

Figure 7.7:: https://xkcd.com/1473/ by Randall Munroe

[1] 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*.

Note: 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.

Note: 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.

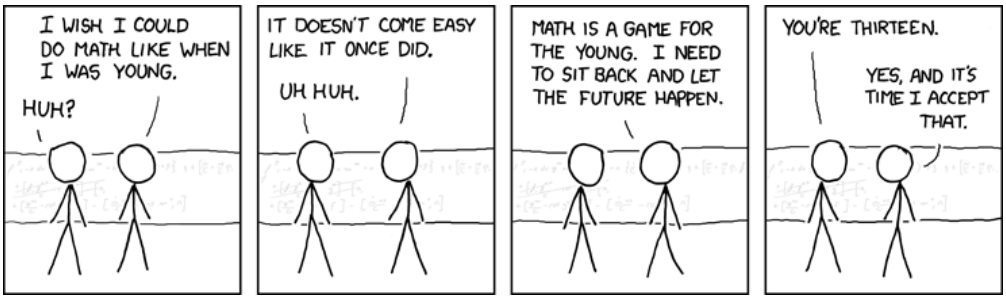

Figure 7.8:: https://xkcd.com/447/ by Randall Munroe

Walkthrough: Applying a cut

(10 minutes)

Note: 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
```

Note: 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)
```

Note: 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
```

Note: 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 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.

Note: 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.

Note: 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

$$E^2_{meas} = p^2_x + p^2_y + p^2_z$$

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?

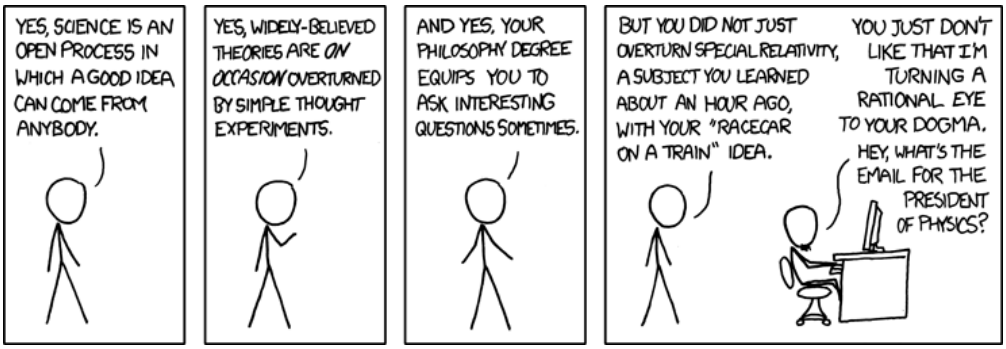

Figure 7.9:: https://xkcd.com/675/ by Randall Munroe

Exercise 10: Writing histograms to a file

(10 minutes)

In all the analysis scripts we've worked with, we've drawn any plots in the Wrap-up section. Pick one of your 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.

Hint: In *Saving your work, part 2*, 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?)

Exercise 11: Stand-alone program (optional)

(30 minutes)

Note: Why would you want to write a stand-alone program instead of using ROOT interactively?

- You can't live in a notebook forever.[1] Typical analysis scripts get so large that you may want to use a regular text editor to work with them, instead of the limited editing available in a notebook cell.
- One method of speeding up a Python program is to use Cython, a Python optimizing compiler: http://cython.org/. You can use Cython within a notebook, but you'll get better results if you create a stand-alone program.
- Stand-alone programs are necessary if you want to submit your Python program to a batch system.

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 script you created in Exercise 10: you have a notebook cell that reads an n-tuple, performs a calculation, and writes a plot to a file. Create a stand-alone program that does the same thing.

Hint: If you tried to do the *C++ version of this Exercise*, you may have found it difficult. The Python equivalent is much easier. Part of the reason is that all the clues you need are in the *condor tutorial* in the *appendix*.

Look at the instructions for the `.py` files in that tutorial, then look at the comments in the `.py` files themselves.

Don't forget to use `module load root` (or the equivalent if you're not using a Nevis particle-physics server) if you expect a stand-alone Python program to be able to import the ROOT libraries!

Figure 7.10:: https://xkcd.com/1263/ by Randall Munroe

[1] Whether this is a programming tip or general life advice I leave up to you.

CHAPTER

EIGHT

THE RDATAFRAME PATH

Congratulations on choosing to work with the `RDataFrame` class. It's obviously better than *The Python Path* or *The C++ Path*.[1]

There are a lot of topics listed below. Don't let that discourage you. Many of the sections below are quite short.[2] Again, the important thing is for you to learn something from doing what exercises you can in the time we have.

[1] For one thing, the xkcd cartoons are funnier.

[2] Many of the sections don't have any footnotes or cartoons at all, just to make them shorter.

RDataFrame concepts

(5 minutes)

Let's start with some definitions. For purposes of this tutorial, an *n-tuple*, a *spreadsheet*, and a *dataframe* are all the same thing.[1,2] It's something that looks like this:

Branches -->

<-- Entries

Row	event	ebeam	px	py	pz
0	0	150.14	14.33	-4.02	143.54
1	1	149.79	0.05	-1.37	148.60
2	2	150.16	4.01	3.89	145.69
3	3	150.14	1.46	4.66	146.71
4	4	149.94	-10.34	11.07	148.33
5	5	150.18	17.08	-12.14	143.10
6	6	150.02	5.19	7.79	148.59
7	7	150.05	7.55	-7.43	144.45
8	8	150.07	0.23	-0.02	147.78
9	9	149.96	1.21	7.27	146.99
10	10	149.92	5.35	3.98	140.70
11	11	149.88	-4.63	-0.08	147.91

Figure 8.1:: You saw this in the class introduction. These are the first few rows and columns in the n-tuple `tree1` in file `experiment.root`.

Some more equivalences: a *row* in the spreadsheet can also be called an *entry* in the n-tuple; a *column* in the spreadsheet is a *branch* in the n-tuple.

Note: In ROOT, the individual cells can have full-fledged C++ structures in them. To keep things simple I'm sticking with numeric values (*leaves* in ROOT's terminology) for this tutorial.

Since we can think of Figure 8.1 as a spreadsheet, let's think of the kinds of physics-analysis tasks we might do with the columns and rows in a program like Microsoft Excel, Google Sheets, or Apple Numbers:

- Sum the values in a column. While this comes up a lot in the business world, it's not common in a physics analysis.
- Statistics: Take the mean or standard deviation of a column, or find its minimum or maximum value.
- Make a histogram of the values in a column. You've already done this if you went through the *TreeViewer* section.
- Add new columns to the spreadsheet, with the new columns derived from formulas based on existing columns.

The idea behind RDataFrame is to provide a simple way to perform tasks like these.

[1] I frequently switch from one term to the other, sometimes within the middle of the same sentence.

[2] The term "dataframe" is also an important component of the Python data analysis package pandas, the R programming language, and the HDF5 file format. It pretty much means the same thing in all these environments.

If you're curious why high-energy physics prefers to use the ROOT file format compared to HDF5, here's a 2018 paper comparing the use of different file formats and databases in a typical analysis. The TL;DR version: HDF5 is better at storing large multi-dimensional arrays, often found in HPC (High Performance Computing) applications associated with Deep Learning. ROOT is a better choice for storing complex data structures.

Walkthrough: Defining an RDataFrame

(10 minutes)

Defining an `RDataFrame` is usually simple. Here's how to do it in both C++ and Python:

Listing 8.1: RDataFrame definition (C++)

```
auto ntupleName = "tree1";
auto fileName = "experiment.root";
auto dataframe = ROOT::RDataframe(ntupleName,fileName);
```

Listing 8.2: RDataFrame definition (Python)

```
import ROOT

ntupleName = "tree1"
fileName = "experiment.root"
dataframe = ROOT.RDataframe(ntupleName,fileName)
```

Note: Actually, unless I'm writing a program that accepts the name of the n-tuple or its file as arguments, I usually don't define separate variables like **`ntupleName`** or **`fileName`** the way I do in the above examples. I'm more likely to just simply do:

```
dataframe=ROOT.RDataframe("tree1","experiment.root")
```

I'm doing this the long way so you can get a sense of what the arguments mean.

The name **`dataframe`** in this example is arbitrary. If you visited the ROOT website's RDataFrame page, you can see they typically use a short name like **`df`** to save on typing. Since I know how to use copy-and-paste, I've opted to use a longer variable name for clarity.

Note: For now, I'm showing both C++ and Python examples of the code. Eventually, when I think I've shown enough examples so you can convert one to the other, I'll stop showing both in parallel. You've probably already noticed how, at least for `RDataFrame`, the code is very similar.

Note: I assume that you're working through *The RDataframe Path* interactively, probably in a *notebook*. You only have to define your dataframe once per session. I'm not usually going to include the above commands in the listings below. If you restart ROOT or the notebook kernel, be sure to initialize **`dataframe`** again.

If you'd like to see the names of the columns in the dataframe, it's easy to do interactively:[1]

[1] If the `Describe` or `Display` methods don't work for you, don't panic. These were added to the very latest version of ROOT. While I try to keep the ROOT versions up-to-date for everyone on the Nevis particle-physics systems, sometimes (due to complex reasons that are beyond irrelevant to you, trust me) I can't offer you the latest-and-greatest.

Listing 8.3: RDataframe description

```
dataframe.Describe()
```

If you'd like a peek at the first few values (roughly equivalent to the `TTree::Scan()` method in *The C++ Path* or *The Python Path*):[2]

Listing 8.4: RDataframe - displaying the first few rows (C++)

```
dataframe.Display()->Print()
```

Listing 8.5: RDataframe - displaying the first few rows (Python)

```
dataframe.Display().Print()
```

Give these commands a try to see what they tell you about the n-tuple `tree1`.

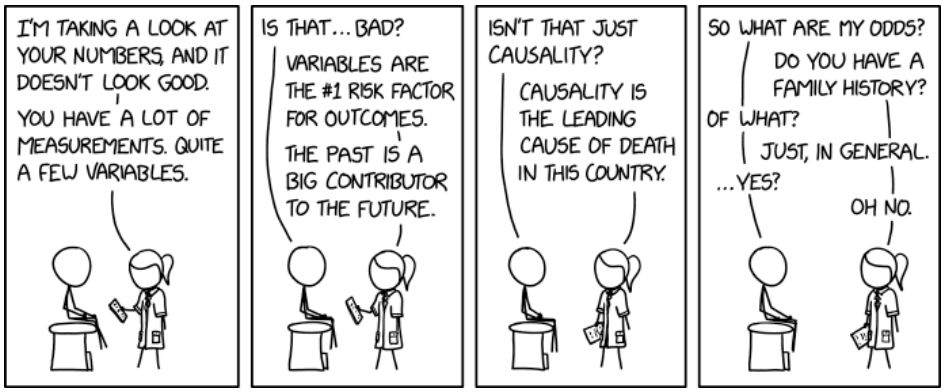

Figure 8.2:: https://xkcd.com/2620/ by Randall Munroe

[2] If you're using C++: You'll have to observe via my examples when an `RDataFrame` method returns a *pointer*; that is, when you have to use `->` to access a method. Remember that you had to deal with this back when you were *fitting histograms*.

Walkthrough: Making Histograms

(10 minutes)

This tutorial is all about making histograms, so let's use `RDataFrame` to make a histogram. Here's an example of making a histogram of `chi2` from the dataframe:

Listing 8.6: RDataFrame - create a histogram (C++)

```
histchi2 = dataframe.Histo1D("chi2");
```

Listing 8.7: RDataFrame - create a histogram (Python)

```
histchi2 = dataframe.Histo1D("chi2")
```

We've made the histogram. Now let's draw it:

Listing 8.8: RDataFrame - draw a histogram (C++)

```
histchi2->Draw();
```

Listing 8.9: RDataframe - draw a histogram (Python)

```
histchi2.Draw()
```

Give it a try!

What's that? It didn't work? It didn't if you're using a notebook.

I'm being sneaky: I counted on you to forget the bit about needing a canvas (either in *C++* or *Python*). Here's a more complete example:

Listing 8.10: RDataFrame - drawing a histogram with a canvas (C++)

```
TCanvas canvas;
histchi2->Draw();
canvas.Draw();
```

Listing 8.11: RDataFrame - drawing a histogram with a canvas (Python)

```
canvas = ROOT.TCanvas()
histchi2.Draw()
canvas.Draw()
```

Tip: 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`).

Follow the above examples to make histograms of the other variables in the n-tuple.

Hint: I will permit you to use copy-and-paste (in a notebook) or the up-arrow key (if you're using the command line) to speed up this task. The fact that I have no way to stop you has nothing to do with it.

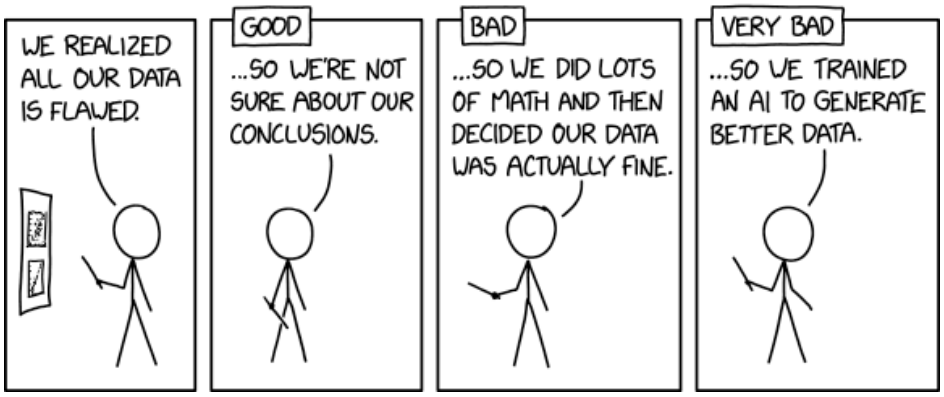

Figure 8.3:: https://xkcd.com/2494/ by Randall Munroe

Exercise 2: Modifying a histogram

(15 minutes)

You're probably still plotting your histograms as a solid curve. Most of the time, your supervisor will want to see histograms with errors. Figure out how to draw one of your histograms (e.g., **`ebeam`**) with error bars. (Here's a *hint*.)

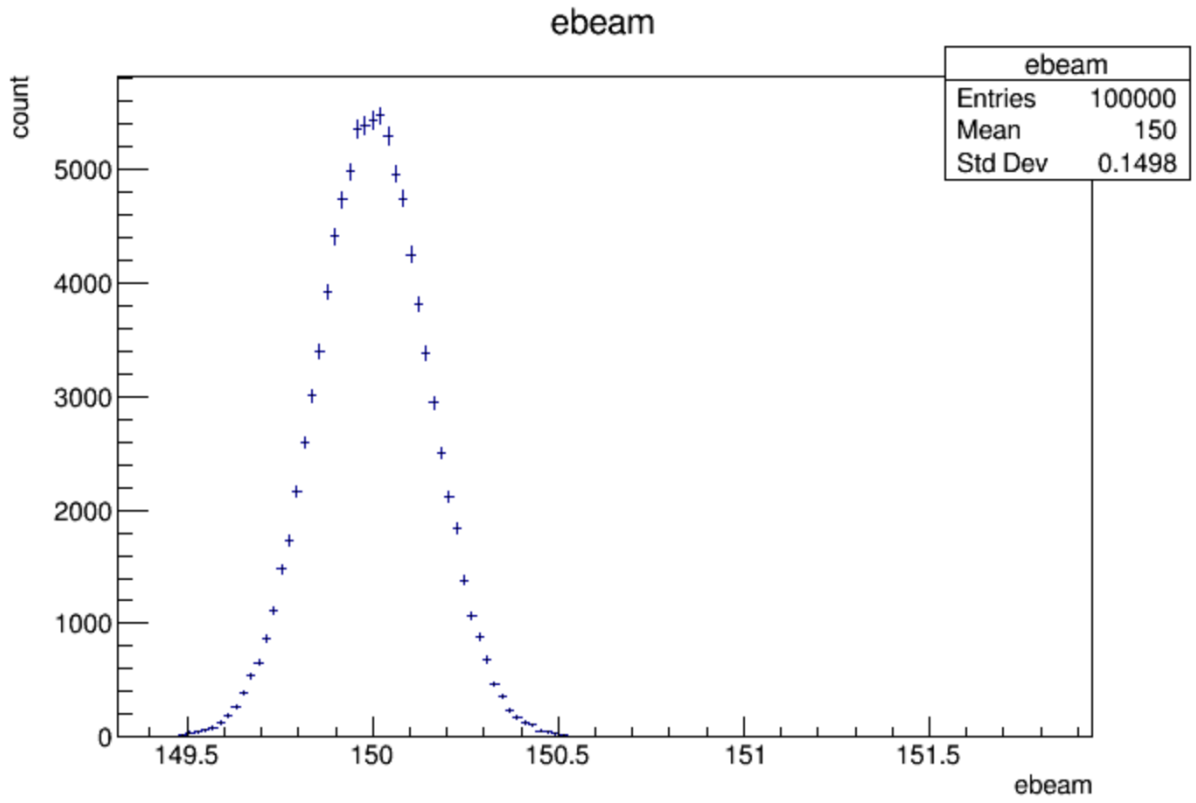

Figure 8.4:: What I get when I plot `ebeam` with errors bars turned on.

That was pretty easy once you looked at the hint. We'll take it up a notch: Let's be good scientists and set/improve the axis labels; set the x-axis label to "Beam energy [GeV]" and set the y-axis label to "Number of events".

This is tricky to do "from scratch" without reading the ROOT documentation in detail as described in *Exercise 1: Detective work*, eventually reaching a description of the TGAxis class. Fortunately, you've got a faster way, if you *remember an earlier lesson*.

Need more help?

Here's another hint if the above hint is too opaque for you: While `RDataFrame` is a useful tool, sometimes the easiest way to answer a question is to go back to the basics.

Another thing to remember that if you can *use Draw on a histogram*, there are other ROOT histogram methods you can use on it.

Also remember that histogram names are arbitrary. Maybe someone else calls a histogram something like **`hist__1__1`**, but you can call it **`histebeam`** or whatever you want.

One last tip for Python programmers: The -> used in C++ becomes a simple `.` for you.

Are you using the command line?

If you're using the command line instead of the *The Notebook Server*, at this point in the tutorial it's going to be more convenient to start editing an external file and executing it from within your command environment.

- **`root`**

  You may remember this from *Walkthrough: Saving and printing your work*: Create a file containing your C++ code, preferably with the extension `.C`; e.g., `exercise2.C`. You'll want to set up your code as a C++ function; e.g.:

```
void exercise2()
{
   // This is not the real code for Exercise 2!
   cout << "Hello, world!" << endl;
}
```

  You can then execute this file from within **`root`** with:

```
.x exercise2.C
```

- **`ipython`**

  If you're using interactive Python outside of a notebook, then you'll want to use the `%run` command. Create a file containing your Python code, preferably with the extension `.py`; e.g., `exercise2.py`. You can then execute this file from within **`ipython`** with:

```
%run exercise2.py
```

Tip: If you're exiting your command-line environment to edit a file, quitting your editor, then starting up your environment again, there's a simpler way: just start up another *Terminal window*. Run your command-line environment in one window and edit files in another.[1]

[1] I apologize if this tip seems trite to you. But you might be surprised how many students have come from a GUI environment that did not encourage them to open multiple windows at the same time.

Exercise 3: Displaying fit parameters

(10 minutes)

Fit the **ebeam** histogram to a Gaussian distribution.

Note: 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.

Note: 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 (though in Python you'll need either 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*).

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?

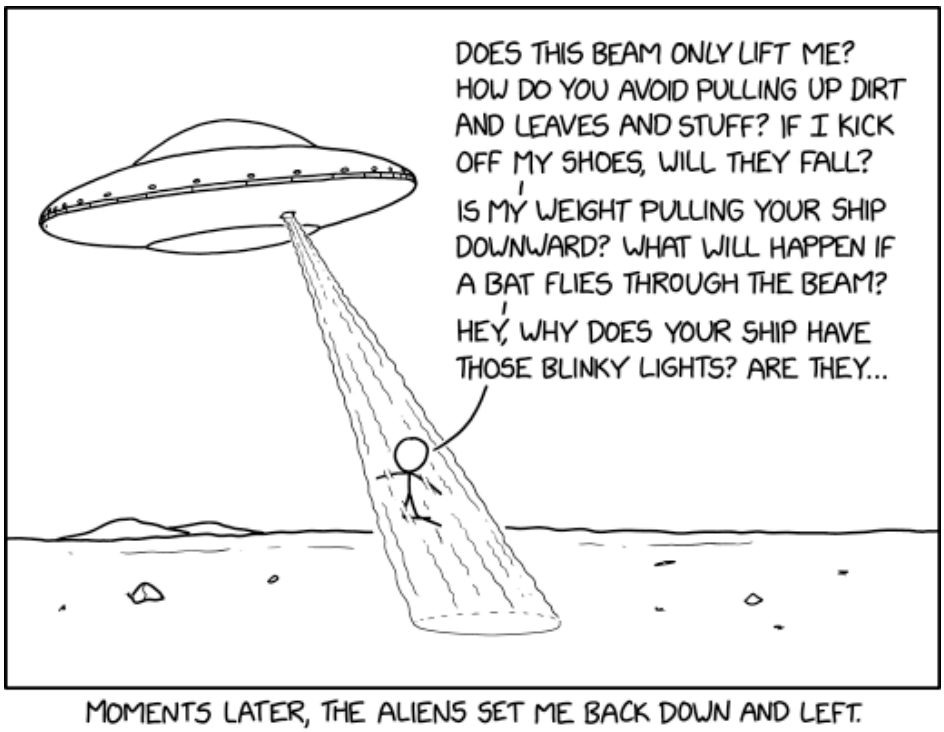

Figure 8.5:: https://xkcd.com/2579/ by Randall Munroe. This is why physicists are never abducted by aliens: There'd be too much that the aliens would have to look up on their web site.

Walkthrough: Defining our own variables

(10 minutes)

So far we've just made histograms of variables that were already in the n-tuple. With `RDataFrame`, we can introduce calculations based on the columns within the n-tuple. If you picture the n-tuple as a spreadsheet, this is like adding a new column in the spreadsheet with values based on a formula.

Note: Let's consider the quantity p_T, which is defined by:

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

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 the value of p_T as a new column in the n-tuple:

```
definept = dataframe.Define("pt","sqrt(px*px + py*py)")
```

This new column, **`pt`**, behaves exactly like any other column in the dataframe. As you can see on the RDataFrame page on the ROOT web site, the general format of the `Define` method is:

```
Define(newname, formula)
```

where:

- `newname` is a string with the name of the new column;
- `formula` is a string containing a C++ formula based on the names of any existing columns in the n-tuple (including ones created with previous `Defines`.

Warning: If you're a Python programmer, it's important to note the `formula` has to be expressed within a string using C++ syntax. In the above example, you might be tempted to try:

```
import math
definept = dataframe.Define("pt","math.sqrt(px**2 + py**2)")
```

Not only is the **`import math`** unnecessary and wasted here (though you might need it elsewhere in your code), but it won't be recognized by ROOT; neither would `px**2` because C++ doesn't use `**` as an exponentiation operator. The contents of the `formula` string are interpreted by ROOT's C++ interpreter, *not* Python!

If you're a C++ programmer, you might note that there's an effective **`using namespace std;`** within ROOT's interpreter.

The next few exercises stick to basic math (`sqrt`, `sin`, arithmetic, exponents), so you shouldn't have a problem with formulas for now.

Note: When you want to move on to more powerful math functions, you can always use ROOT's TMath functions within a formula. For an extreme example, the above formula could be expressed as:

```
definept = dataframe.Define("pt","TMath::Sqrt(TMath::Power(px,2) + TMath::Power(py,2))")
```

This verbose example is not something you'd actually do, since ROOT's C++ is good enough for `sqrt` and such. But if you have to compute the relativistic Breit-Wigner function, `TMath::BreitWignerRelativistic` has you covered.

What if what you want to do is not one of the ROOT built-in functions? *We'll get to that!*

At the moment we have a "new" dataframe called **definept**, but we haven't done anything concrete with it. We'll do that on the next page.

Exercise 4: Plotting a derived variable

(10 minutes)

Make a histogram of the new n-tuple variable **`pt`**. Don't forget to include error bars and to label the axes; remember that the momenta are in *GeV*.

Hint: Are you getting a message about **`pt`** not being defined? Remember that **`pt`** was defined for our "new" n-tuple `definept`, not the original `dataframe`.

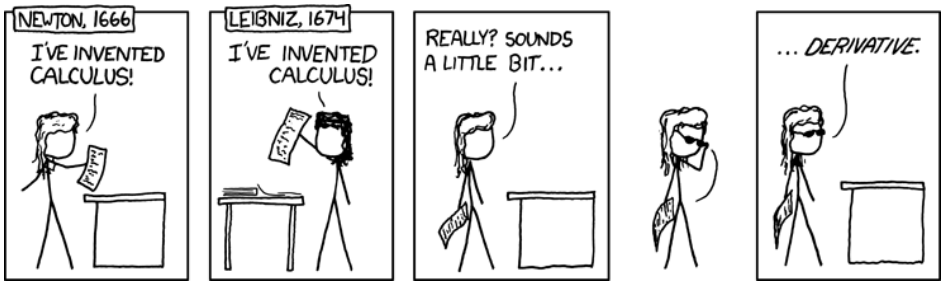

Figure 8.6:: https://xkcd.com/626/ by Randall Munroe

Walkthrough: Apply a cut and a count

(15 minutes)

Applying cuts is an important part of any physics analysis. There'll be some events you want to analyze and others which are not important to your study. A "cut" is a calculation that separates the two categories.

In `RDataFrame`, the method that applies a cut is `Filter`. For example, suppose that we're only interested in events with **pz** less than 145 *GeV*. A way this can be expressed in our example n-tuple is:

```
pzcut = dataframe.Filter("pz < 145")
```

Warning: Again, the string passed on to the `Filter` method is interpreted as a C++ expression, not a Python expression, even if you're working in Python. You'll get an error if you try this:

```
pzcut = dataframe.Filter("pz lt 145")
```

You can also apply a cut on any new columns you've defined:

```
ptcut = definept.Filter("pt > 50")
```

Note: There's an important operational difference between `Define` and `Filter`. `Define` is a *column-wise* operation; that is, it operates on columns and adds a new one. `Filter` is a *row-wise* operation; it essentially removes rows from the n-tuple that don't pass its criteria.

You've probably already guessed that you can plot any column from the filtered n-tuple; e.g.,

```
pzcut_hist = pzcut.Histo1D("ebeam")
```

The above line would accumulate a histogram of **ebeam** for those rows with **pz** less than 145 *GeV*.

If you just want to know the number of n-tuple rows that pass a cut, the method to use is `Count`. For example:

```
pzcut_count = pzcut.Count()
```

Note: Unlike `Define` and `Filter`, `Count` never takes an argument. However, you can't omit the parenthesis, since `Count` is a function; it's always `Count()` and not `Count` in program code.

This seems a bit counter-intuitive at first: You can't just print out the value of `pzcut_count`. That's because it's still an `RDataFrame` variable, in the same sense that `histchi2` was *earlier*. In the case of `histchi2`, you had to `Draw` it to see anything. The corresponding method to use with `Count` is `GetValue()`; e.g.,

Listing 8.12: RDataFrame - get the count after a cut (C++)

```
pzcut = dataframe.Filter("pz < 145");
pzcount = pzcut.Count();
std::cout << "The number of events with pz < 145 is "
          << pzcount.GetValue() << std::endl;
```

Listing 8.13: RDataFrame - get the count after a cut (Python)

```
pzcut = dataframe.Filter("pz < 145")
pzcount = pzcut.Count()
print("The number of events with pz < 145 is",pzcount.GetValue())
```

Note: When I run either of the above code examples, I get

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

Give it a try. Hopefully you'll get the same answer.

Figure 8.7:: https://xkcd.com/562/ by Randall Munroe. This is another way to apply a cut.

Walkthrough: Making scatterplots

(15 minutes)

Now that we've had some practice making one-dimensional histograms, let's make a two-dimensional histogram. Let's see if we can take the same approach that we did for *Exercise 1*. To make a 1-D histogram we used `Histo1D`; when I look at the RDataFrame web page I see there's a `Histo2D` method. So it's obvious that it should be something like:

Listing 8.14: RDataFrame - does this make a 2-D histogram (C++)

```
hist2dim = dataframe.Histo2D("ebeam","px");
hist2dim->Draw();
canvas.Draw();
```

Listing 8.15: RDataFrame - does this make a 2-D histogram (Python)

```
hist2dim = dataframe.Histo2D("ebeam","px")
hist2dim.Draw()
canvas.Draw()
```

Give it a try!

Hey, what's happening? Am I being sneaky again?

Not this time. This one of those (fortunately rare) cases where ROOT is not uniform in its approach. In order to make a 2D histogram with `RDataFrame`, you have to supply the same parameters to `Histo2D` as if you were to create such a histogram "by hand."

If you look up TH2D, in analogy with TH1D, you'll see that the arguments to `TH2D` are something like:

```
hist2d = TH2D("name","title",nxbins,xlo,xhi,nybins,ylo,yhi)
```

where:

- `"name"` is the ROOT name of the histogram;
- `"title"` is the title of histogram, which is shown at the top of the plot;
- `nxbins` is the number of bins on the x-axis;
- `xlo` is the lower limit of the x-axis of the plot;
- `xhi` is the upper limit of the x-axis of the plot;
- `nybins` is the number of bins on the y-axis;
- `ylo` is the lower limit of the y-axis of the plot;
- `yhi` is the upper limit of the y-axis of the plot.

When using `RDataFrame`, you have explicitly supply these values to `Histo2D` like this:

```
Histo2D(("name", "title", nxbins, xlo, xhi, nybins, ylo, yhi),"ebeam","px")
```

Here's how it looks in the actual code, specifying the `TH2D` parameters in an initializer list in the respective languages:

Listing 8.16: RDataFrame - making a 2-D histogram (C++)

```
hist2dim = dataframe.Histo2D({"hist2dim", "ebeam vs px", 100, 149, 151, 100, -20, 20},
↪"ebeam","px");
```

(continues on next page)

(continued from previous page)

```
hist2dim->Draw();
canvas.Draw();
```

Listing 8.17: RDataFrame - making a 2-D histogram (Python)

```
hist2dim = dataframe.Histo2D(("hist2dim", "ebeam vs px", 100, 149, 151, 100, -20, 20),
↪"ebeam","px")
hist2dim.Draw()
canvas.Draw()
```

Give it a try!

Note: This is a scatterplot, a handy way of observing the correlations between two variables. The `Histo2D` command interprets the last two variables as "x","y" to define 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.

This leads to the question: How did I know the values for `xlo`, `xhi`, `ylo`, and `yhi` in the above examples? The answer is that I made 1-D plots for the variables so I knew their range, then used those values for the 2-D axis limits.[1,2]

Now that you have the recipe, try making scatterplots of different pairs of variables. Do you see any correlations?

Note: 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.

---

[1] There's another obvious question: Why is this necessary? The `Histo1D` method is able to automatically determine the scale of its single x-axis; why can't `Histo2D` do the same for its axes?

I hunted for the reason, and finally asked the question on the ROOT Forums. The answer has to do with being able to use `RDataFrame` *with multiple threads*, a subject I address in the *intermediate topics* section. While running with multiple execution threads, the ROOT developers can make automatic scaling of `Histo1D` work, but they haven't figured out how to make automatic axis scaling work with `Histo2D` (or `Histo3D`, for that matter).

The lesson here: Even though `RDataFrame` is generally easier to use (yes, really!) than the techniques described in *The C++ Path* or *The Python Path*, there are still times when you have deal with ROOT's peculiarities.

[2] You can also explicitly specify the parameters when creating a 1-D histogram; e.g.,

```
hist1 = dataframe.Histo1D(("h1", "ebeam", 100, 149, 151),"ebeam")
```

You might want to do this if you want to override the automatic histogram limits, or you want to set the histogram title.

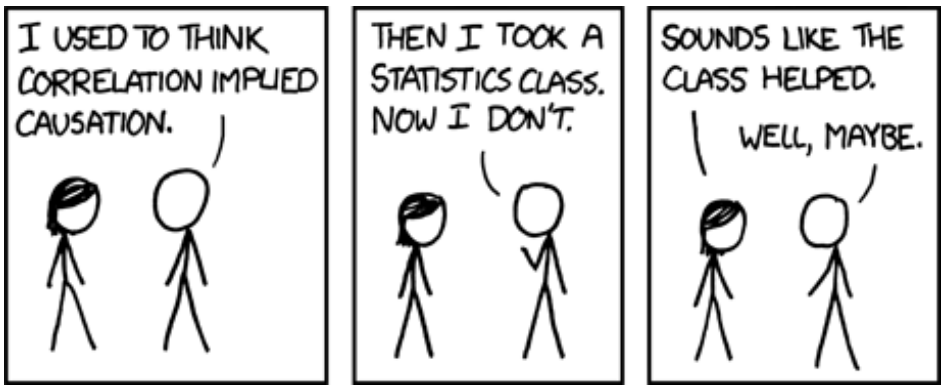

Figure 8.8:: https://xkcd.com/552/ by Randall Munroe

Exercise 5: Two histograms at the same time

(15 minutes)

If you make a cut during your analysis, your supervisor will probably want to see your histograms both before and after cuts.

Consider the quantity:

$$p_{tot} = \sqrt{p_x^2 + p_y^2 + p_z^2}$$

Start with our original `dataframe`. Make a histogram of p_{tot}. (Don't forget the error bars and to label the axes!) Then apply a cut of $p_z < 145GeV$ and make a plot of p_{tot} after that cut.

To better simulate the kind of work you'd actually have to do, don't just run your code twice. Put all the code to generate both plots into a single cell (if you're using a notebook) or a single script (if you're using the command line).

Hint: I absolutely expect you to copy-and-paste code fragments from earlier pages and Exercises in this tutorial, and then edit them to suit your task.

Warning: Prediction: You're going to run into trouble when you actually Draw the histograms. You'll probably only see one histogram plotted, and it may 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. Both histograms are being drawn to the same canvas.

The simplest way to deal with this: create more than one canvas. Something like this may guide you:[1]

Listing 8.18: A sketch of working with multiple canvases (Python)

```
canvas_a = ROOT.TCanvas()
histogram_a.Draw()
canvas_a.Draw()
canvas_b = ROOT.TCanvas()
histogram_b.Draw()
canvas_b.Draw()
```

Take a look at your multiple plots and feel proud! Except... look carefully. The x-axis of the two plots is not quite the same. It doesn't matter much here, but in an actual analysis you'll want the axes of your different plots to match. Revise your cell or script so that the x-axes of the two plots are the same.

Hint: There's a clue about how to do this in *Walkthrough: Making scatterplots*. Check the footnotes!

[1] I warned you earlier that at some point I'd stop showing examples both C++ and Python, once you'd seen enough of them to convert one from the other. That point has been reached!

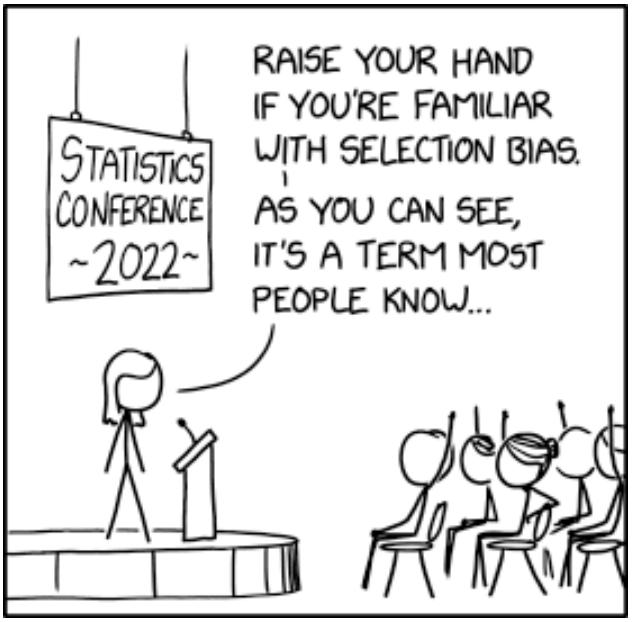

Figure 8.9:: https://xkcd.com/2618/ by Randall Munroe

Exercise 6: 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. Define a new column for θ and plot it as a histogram. One more time: don't forget to label the axes!

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

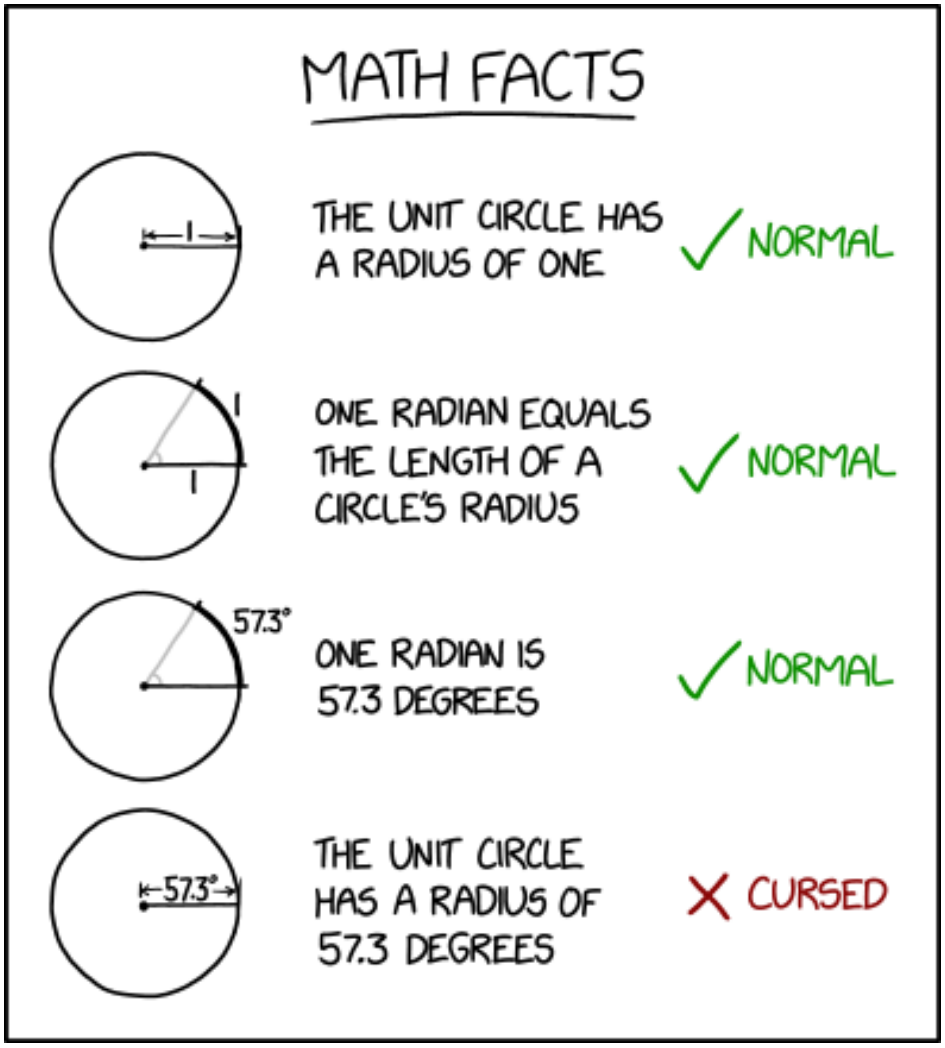

Figure 8.10:: https://xkcd.com/2748/ by Randall Munroe

Transformations and Actions

(15 minutes)

Let's refine our understanding of what goes on when we perform operations with `RDataFrame`.

In all of the examples I've given you up to now, I've shown each operation on a single line of code; e.g.,

Listing 8.19: The verbose way of doing things with RDataFrame (C++)

```
// Define the dataframe from an input ntuple and file.
auto dataframe = ROOT::RDataFrame("tree1","experiment.root");

// Histogram the value of pt for pz<145 GeV, pt<10 GeV.
auto pzcut = dataframe.Filter("pz < 145");
auto ptdefine = pzcut.Define("pt","sqrt(px*px + py*py)");
auto ptcut = ptdefine.Filter("pt < 10");
auto pthist = ptcut.Histo1D("pt");
```

You can be much less verbose if you don't need to use intermediate modified dataframes for anything:[1]

Listing 8.20: The concise way of doing things with RDataFrame (C++)

```
// Define the dataframe from an input ntuple and file.
auto dataframe = ROOT::RDataFrame("tree1","experiment.root");

// Histogram the value of pt for pz<145 GeV, pt<10 GeV.
auto pthist = dataframe.Filter("pz<145").Define("pt","sqrt(px*px+py*py)")
                       .Filter("pt<10").Histo1D("pt");
```

There's an important restriction when you're being more concise: You can have as many *transformations* as you like on an `RDataFrame`, but a given sequence of operations can have only one *action*.

Before I give you a definition of "transformation" or "action", let me show you what led me to make this distinction. I tried to do something like this:

Listing 8.21: An attempt to use two actions on one line (Python)

```
# Define the dataframe from an input ntuple and file.
import ROOT
dataframe = ROOT.RDataFrame("tree1","experiment.root")

# Count the number of events with pz<145 GeV and histogram them.
pthist = dataframe.Filter("pz < 145").Count().Histo1D("pz")
```

The above line won't work; give it a try to see the error message.[2]

The reason why the code doesn't work is that both `Count()` and `Histo1D()` are actions. A *transformation* like `Define()` or `Filter()` changes the n-tuple; an *action* accumulates data within the n-tuple. If you go to the RDataFrame web page, you will see lists of which `RDataFrame` operations are transformations and which are actions (and which are *queries*, yet another category).

Here's a re-write of the code above so that there's only one action per line.

[1] If you're using Python, it might help to note that C++ does not need anything special to continue a program statement on another line (it's the `;` that terminates a statement). Python requires a backslash \ to continue a statement onto the next line.

[2] If you're a C++ snob (which I've been accused of being from time to time), you might foolishly assume that it doesn't work because the code is in Python. Instead of being rude, just slap an `auto` in the front and a `;` at the end and see for yourself.

Listing 8.22: For two actions, use two lines (Python)

```
# Define the dataframe from an input ntuple and file.
import ROOT
dataframe = ROOT.RDataFrame("tree1","experiment.root")

# Count the number of events with pz<145 GeV and histogram them.
ptcut = dataframe.Filter("pz < 145")
ptcount = ptcut.Count()
pthist = ptcut.Histo1D("pz")
```

In other words, you can put everything on one line *if you don't need to use the intermediate modified dataframes*. If you want to apply more than one action to same modified dataframe, then you will have to create intermediate dataframe variables.

Tip: Some folks may find a diagram helpful for understanding this idea. Consider the following code:[3]

Listing 8.23: Several n-tuple operations (C++)

```
// Define an RDataFrame.
auto dataframe = ROOT::RDataFrame("tree1","experiment.root");

// Create a couple of histograms, before and after a pz cut.
// Make sure the x-axes of the plots will be the same.
auto pzhist = dataframe.Histo1D({"pz","pz before cut",100,130,170},"pz");
auto pzcuthist = dataframe.Filter("pz < 145")
                      .Histo1D({"pzcut","pz after cut",100,130,170},"pz");

// Create a new column, pt, and look at chi2 before and after a pt cut.
// Again, make sure the x-axes match on the histograms.
auto ptDefined = dataframe.Define("pt","sqrt(px*px + py*py)");
auto chi2hist = ptDefined
               .Histo1D({"chi2","chi2 before cut",100,0,20},"chi2");
auto chi2cut = ptDefined.Filter("pt < 10");
auto chi2cuthist = chi2cut
               .Histo1D({"chi2cut","chi2 after cut",100,0,20},"chi2");

// How many events passed our pt cut?
auto chi2cutcount = chi2cut.Count();

// The necessary Draw() and GetValue() methods to see any plots or values
// are left as an exercise for the student.
```

This is a diagram of how `RDataFrame` organizes the chain of operations to be performed on the n-tuple:

[3] If the stuff in the curly braces {} is confusing to you, look at the footnotes in *Walkthrough: Making scatterplots*.

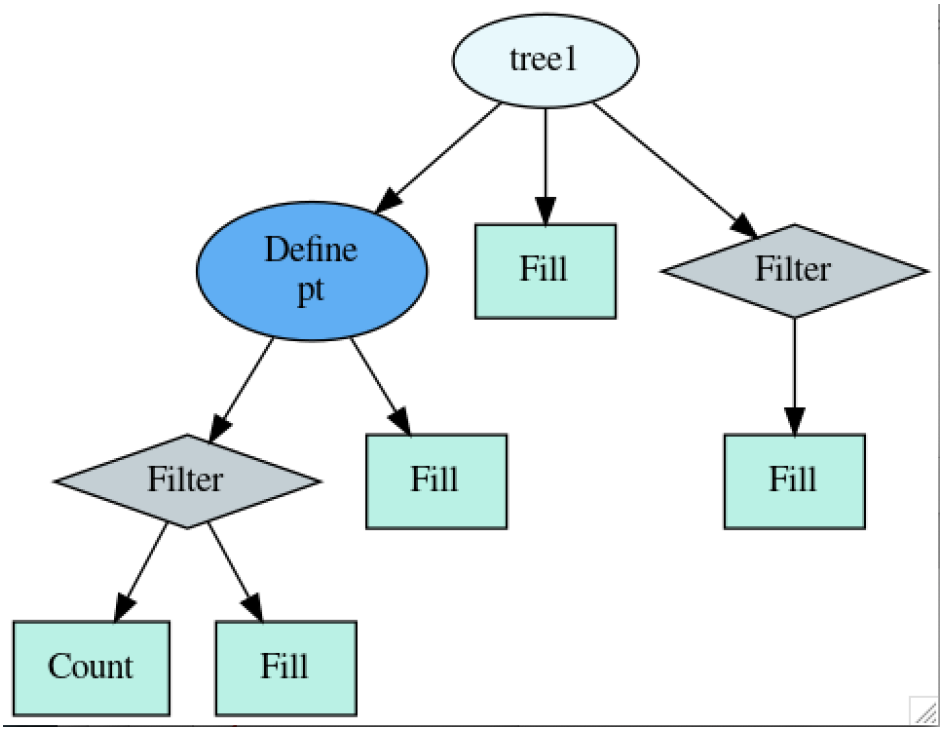

Figure 8.11:: The series of operations that have been assigned to the `tree1` n-tuple based on the above code.

You may want to look at the program listing and match it against the operations indicated in the diagram. For a "path of operations" to take place, that path must end in an action, which are the rectangles in the diagram.[4]

[4] You can generate a diagram like this for your own dataframes, but it can be a lot of additional work. I only recommend it if you find these kinds of diagrams to be useful.

Here's how I made that diagram using the `SaveGraph` method that's part of `RDataFrame`. After I defined all my n-tuple operations, I executed:

```
// C++
ROOT::RDF::SaveGraph(dataframe, "./dataframe.dot");

// Python
ROOT.ROOT.RDF.SaveGraph(dataframe, "./dataframe.dot")
```

This will create a file `dataframe.dot` in your current directory. You can look at the file using **`less`**, but all you'll see is a text representation of the graph.

To turn a `.dot` file into a diagram, you need to have the Graphviz software installed; this is available on all the systems on the Nevis particle-physics Linux cluster. The Nevis systems also have ImageMagick; all you have to do on a system with both is type this in your *UNIX window*:

```
display dataframe.dot
```

If you're not on a Nevis system, see *Installing ROOT on your own computer* and include `graphviz` and `imagemagick` when you install `root`.

Lazy Evaluation

(15 minutes)

While working through this tutorial, you might have observed a couple of curious things:

- In Figure 8.11 in *Transformations and Actions*, the diagram elements are not in the order that they were created in Listing 8.23.
- As you enter the `RDataFrame`-related code for the Walkthroughs and Exercises, the system seems to pause at unpredictable times.

You already know the reason for both those things is lazy evaluation, because you read the title of this page. In general *lazy evaluation* means that the program only performs one or more operations when it needs to actually evaluate something; otherwise, it saves a of list of the operations it has to perform if an evaluation is required.[1]

That may be difficult to grasp. Let's work through an example step-by-step. Start a new notebook kernel or command-line session. Execute the following lines one-by-one, either each in its own notebook cell or one line at a time (translating from Python to C++ as needed).

```
import ROOT
```

That line took a while, but that's to be expected as you're setting up a large library like ROOT. (In C++, the same thing happens when you start a notebook/session.)

```
dataframe = ROOT.RDataFrame("tree1","experiment.root")
```

That line was fast. Let's keep going.

```
ptcuthist = dataframe.Define("pt","sqrt(px*px+py*py)") \
                     .Filter("pt<10") \
                     .Histo1D("pt")
```

[1] While it's possible to implement lazy evaluation in many programming languages, there's one programming language in which lazy evaluation is fundamental: Haskell. You may have noticed that it's available as one of the kernels on our *notebook server*.

I have not yet seen Haskell used in particle physics. I suggest you only explore it if you want to learn a new programming paradigm for its own sake.

Figure 8.12:: https://xkcd.com/1312/ by Randall Munroe

That took no time at all, and it was a complex line. Hmm...

```
canvas = ROOT.TCanvas()
```

ROOT defines canvases quickly. Good!

```
ptcuthist.Draw()
```

Now we have a delay! Drawing the histogram onto the canvas takes a long time, but the complicated definition of **ptcuthist** took almost no time.

Note: If you want, you can finish by drawing the canvas:

```
canvas.Draw()
```

That might take a few seconds, but the delay can be attribute to the system putting together the graphics resources to create the image.

That was an example of lazy evaluation. `RDataFrame` built up a list of tasks in memory, but didn't perform any of those tasks yet. Only when you typed `ptcuthist.Draw()` did the program have to actually read the n-tuple (also known as *performing the event loop*) in order to draw the histogram.

Let's get precise

You might object that I used hazy, relative phrases like "no time at all" and "delay". If you did, congratulations! You're thinking like a scientist.

Let's make a measurement. If you're working in a notebook, or you're using **ipython**, you have access to the `%%time` *magic command*. Repeat the above walkthrough, but put `%%time` as the first line of every cell. For example:

```
# First cell
%%time
dataframe = ROOT.RDataFrame("tree1","experiment.root")

# Second cell
%%time
ptcuthist = dataframe.Define("pt","sqrt(px*px+py*py)") \
                     .Filter("pt<10") \
                     .Histo1D("pt")
```

...and so on.

The exact values you get will depend on many factors; e.g., whether you're doing this walkthrough in C++ or Python; which Jupyter server you're using; how many other users are on the same system. That's why I'm not quoting any hard numbers here.

When I try this on my Jupyter notebook, I see that the execution speed of `ptcuthist.Draw()` is on the order of a couple of seconds, while the other commands take less than a second at most.

Why is this important? Consider:

```
countPz = dataframe.Filter("pz < 145").Count()
hist = dataframe.Define("pt","sqrt(px*px + py*py)")
    .Define("theta","atan2(pt,pz)").Histo1D("pt")
```

(continues on next page)

(continued from previous page)

```
print ("The number of events with pz < 145 is",countPz.GetValue())
hist.Draw()
```

When you execute the above code, RDataFrame will "stack" the `Filter`, `Define`, and `Histo1D` actions. It will only perform the event loop when it executes `countPz.GetValue()`, which requires a concrete numeric value. As it reads the n-tuple it will implement all the stacked actions.

This means you want to have a sense of when you're asking `RDataFrame` to evaluate a result. Consider the following code, which just moves a single line compared to the above code:

```
countPz = dataframe.Filter("pz < 145").Count()
print ("The number of events with pz < 145 is",countPz.GetValue())
hist = dataframe.Define("pt","sqrt(px*px + py*py)")
    .Define("theta","atan2(pt,pz)").Histo1D("pt")
hist.Draw()
```

If you execute this code, `RDataFrame` will read the n-tuple to get the result of `countPz.GetValue()`. It will then stage two more `Define` actions and the `Histo1D` action, and then perform the event loop again to be accumulate the data for `hist.Draw()`.

Do things right, and you'll only perform the event loop once. Do things wrong, and you could get a slow program that reads an n-tuple from disk over and over again.

An advanced example

I thought about giving this as an Exercise, but decided against it because it involves Python and C++ programming language features that I haven't discussed. If you think you could have done it on your own, let me know; maybe it will be an Exercise the next time I teach this tutorial.

The goal: Make a histogram of every variable in an n-tuple. Do this without knowing in advance what the columns are. Keep lazy evaluation in mind: you do *not* want to read the entire n-tuple each time you plot a new column; you only want to read the n-tuple once.

Here are my solutions:

Listing 8.24: Plotting every variable in an n-tuple (C++)

```
// Assume the n-tuple has already been defined in
// an RDataFrame named "dataframe". Use the
// GetColumnNames method (see the RDataFrame web page)
// to get a list of the variable names.
auto names = dataframe.GetColumnNames();
auto length = names.size();

// Create one histogram for every column.
std::vector<TH1D> histograms(length);
for ( int i = 0; i < length; ++i ) {
    histograms[i] = *( dataframe.Histo1D( names[i] ) );
}

// Create one canvas for each histogram.
// Draw the histogram on that canvas.
std::vector<TCanvas> canvases(length);
for ( int i = 0; i < length; ++i ) {
    canvases[i].cd();
```

(continues on next page)

(continued from previous page)

```
    histograms[i].Draw();
    canvases[i].Draw();
}
```

Listing 8.25: Plotting every variable in an n-tuple (Python)

```
# Assume the n-tuple has already been defined in
# an RDataFrame named "dataframe". Use the
# GetColumnNames method (see the RDataFrame web page)
# to get a list of the variable names.
names = dataframe.GetColumnNames()
length = len(names)

# Create one histogram for every column.
histograms = []
for i in range(length):
    histograms.append( dataframe.Histo1D( names[i] ) )

# Create one canvas for each histogram.
# Draw the histogram on that canvas.
canvases = []
for i in range(length):
    canvases.append( ROOT.TCanvas() )
    histograms[i].Draw()
    canvases[i].Draw()
```

Tip: There are some subtle operational differences between these two pieces of code (use of vectors vs. lists; when the histogram and canvas objects are created; the dereferencing operation * in the C++ code). But let's focus on the lazy-evaluation aspect.

Note that I define all the histograms in their own loop using `Histo1D` *before* I use `Draw()` on any of them. This means the n-tuple will be read only once, when **`histograms[0].Draw()`** is executed.

What would happen if I didn't think about lazy evaluation and did something like this?

```
names = dataframe.GetColumnNames()
length = len(names)
for i in range(length):
    hist = dataframe.Histo1D( names[i] )
    canvas = ROOT.TCanvas()
    hist.Draw()
    canvas.Draw()
```

I'd be performing the event loop multiple times, once for every column in the n-tuple.

Tip: Also, I might only get a histogram of the last column for the reasons discussed in *Exercise 5: Two histograms at the same time*.

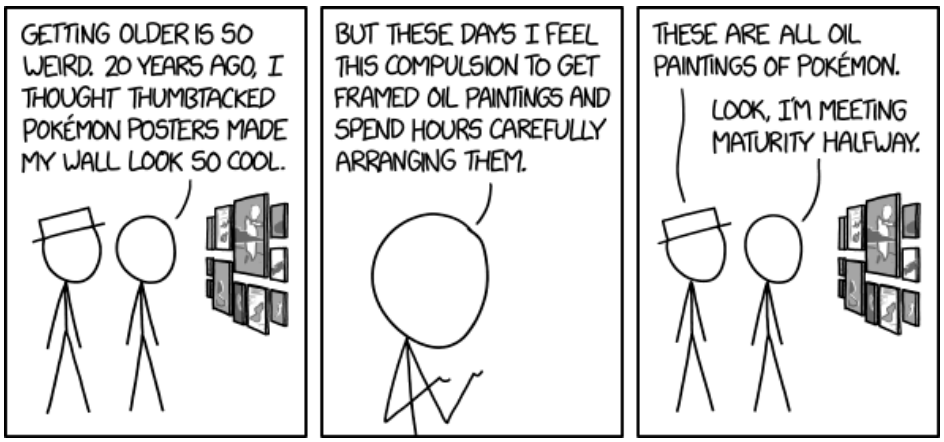

Figure 8.13:: https://xkcd.com/2018/ by Randall Munroe. The relevance of this cartoon will become apparent if you execute one of the "double-loop" code examples above.

Exercise 7: Picking a physics cut

(15 minutes)

Make a histogram of **chi2**, then a scatterplot of **chi2** vs **ebeam**.

Hint: If you've forgotten how to figure out the axis limits for **ebeam**, look at *Walkthrough: Making scatterplots* again for a clue.

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

The chi2 histogram 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 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**.

Note: 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.

Repeat the above plots with a `Filter()` 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.

Tip: You may ask which is better:

- `.Filter("chi2 < 1.5").Filter("theta < 0.15")`
- `.Filter("chi2 < 1.5 && theta < 0.15")`

On the relatively small scale of this example, it doesn't make much of a difference. For a large-scale analysis, the second expression is more efficient, since `RDataFrame` only has to invoke the overhead of the `Filter()` method once (including compiling the C++ expression within the quotes) instead of twice.

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 8: A bit more physics

(15 minutes)

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

$$E^2_{meas} = p^2_x + p^2_y + p^2_z$$

and the energy lost by the particle is given by

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

Define the new columns needed and 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?

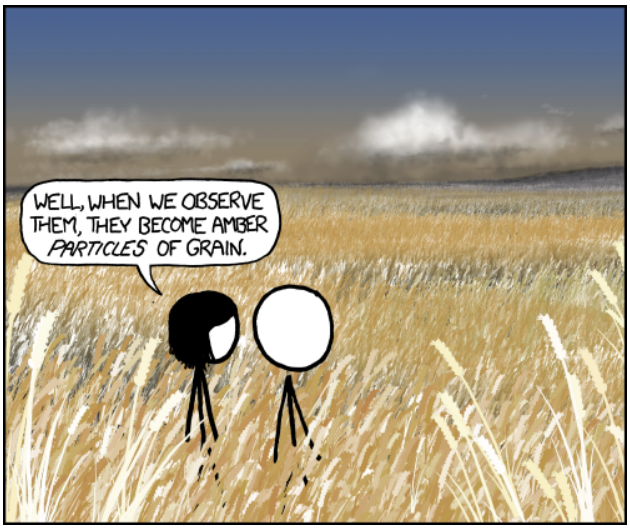

Figure 8.14:: https://xkcd.com/967/ by Randall Munroe

Exercise 9: Writing the n-tuple

(10 minutes)

If you've made changes to your n-tuple (defining new columns, applying filters) and you want to write the revised n-tuple to a file, use the `Snapshot()` method.

Here's an example

Listing 8.26: An example using the `Snapshot` method (C++)

```
// Open a dataframe from an n-tuple within a file.
auto dataframe = ROOT::RDataFrame("tree1","experiment.root");

// Make a change to the n-tuple.
auto pzcut = dataframe.Filter("pz < 145");

// Write the revised n-tuple with the name "pzcut" to file
// "analysis.root".
pzcut.Snapshot("pzcut","analysis.root");
```

Here's the Exercise

In the past few pages, we've built a collection of new variables (p_T, θ, E_{meas}, E_{loss}) and useful analysis cuts ($\chi^2 < 1.5$, $\theta < 0.15$). Derive an n-tuple with all these new columns and cuts, and write it as n-tuple `analyzed` to file `analysis.root`.

Tip: If you look up `Snapshot()` on the RDataFrame web page you'll see that it's classified as an *instant action*. This means that the event loop will be executed when the `Snapshot()` is executed.

If, for some reason, you want a lazy evaluation of `Snapshot()`, click on the highlighted word `Snapshot()` on that page. It will take you to a different page; search for `fLazy` on that new page.

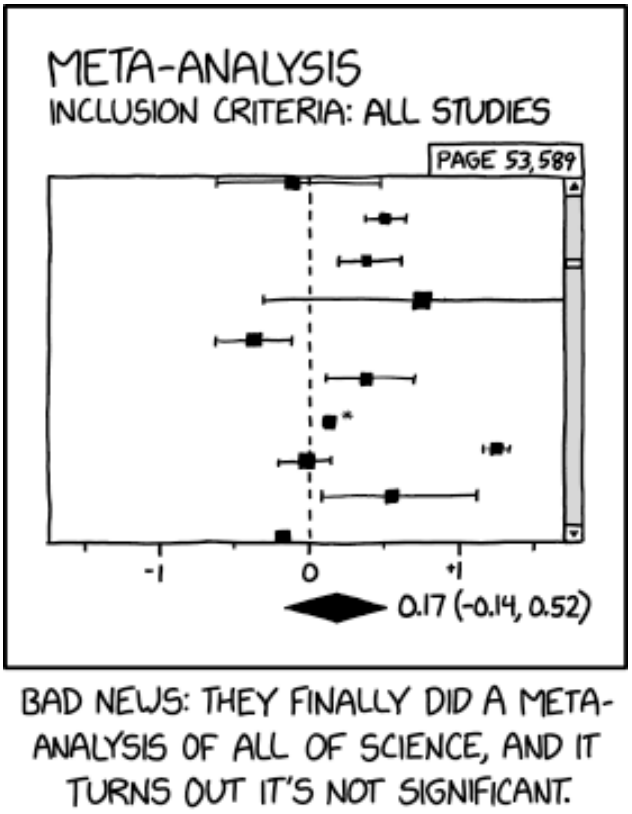

Figure 8.15:: https://xkcd.com/2755/ by Randall Munroe

Writing your own functions

(60 minutes for all the sections below)

You may recall that I *previously* talked about a "wall" of difficulty you'd hit when working with `RDataFrame`. Here it is!

So far, I've done my best to assume that you're familiar with no more than the basics of your programming language. From this point forward, in order to keep this tutorial to a reasonable length, I have to assume you know enough to write functions in your chosen language.

Python programmers: You've got it a bit harder, because you're also going to have to learn something about C++ syntax.

I'll provide examples from which you can extrapolate. But the burden of that extrapolation will rest on your shoulders.

I've broken this topic down into smaller chunks, to hopefully make it easier to absorb for those who don't have prior experience with programming.

Layers

Definition of "jit"

The process of compiling code while a program is executing is called just-in-time compilation, or JIT. In programmer's slang, this is often called "jitting," a term that you'll see in some of `RDataFrame`'s documentation.[1]

Up until now, we've used `Define()` and `Filter()` with operations that could be expressed in a short string of characters. Those strings were "jit'ed" by ROOT's C++ interpreter (even if you work in Python). For example:

```
.Define("ptot","sqrt(px*px + py*py + pz*pz)")
.Filter("ptot > 40 && zv > 10")
```

This might not be sufficient if:

- The operation you want to perform is much more complex, perhaps requiring many lines of code.
- You want to get the greatest possible speed out of using `RDataFrame`. Supplying ROOT with a compiled operation is much faster than using "jitted" code.

When working with jit in a software toolkit like ROOT, you may have to deal with multiple operational layers. In ROOT, these layers are:

1. The notebook layer. This layer implements the cell layout, keeps track of variables between cells, etc. (If you're programming on the command-line, you don't have this layer.)
2. The programming/interpreter layer. In ROOT C++, this is cling; in Python, it's… well… Python.
3. ROOT's internal layer of variables, methods, libraries, etc.

I vaguely alluded to the last layer when I had you *define the double-Gaussian function* in *The Basics*. Remember the *first line* I had you to type into ROOT?

```
TF1 f1("func1","sin(x)/x",0,10)
```

The name **`f1`** was created in the programming layer. The name **`func1`** was created in ROOT's internal layer. If you want to draw the function, you have to use the name defined in the programming layer:

[1] Like many programmers, I'm sloppy with my grammar. I use jit as a noun, verb, adjective, adverb, gerund, and insult without any form of consistency.

Anyone who can't accept that is a jitting jit who jits their jits jitly. That definitely makes me a jit as well.

```
f1.Draw()
```

If you want to use the function within ROOT's internal layer, you have to use its internal name:

```
TH1D somehistogram("histogramName","Histogram Title",100,-3,3)
somehistogram.FillRandom("func1",1000)
```

As a general rule, if you're supplying a name or operation within a character string that's an argument to a ROOT method (e.g., the **func1** in `FillRandom("func1",1000)`), then that string is being interpreted within ROOT's internal layer. To the programming layer, it's just a character string.[2]

Note: I could have shown you examples in which I gave these variables the same name, e.g.:

```
TF1 func1("func1","sin(x)/x",0,10)
```

I didn't do that because, although it can be convenient, it can create the false impression that the programming layer and the internal layer share names.

This will be important as we think about constructing our functions in the following pages... In particular:

- The internal layer "knows" the names of all the columns in the `RDataFrame`; it gets them when you define the dataframe.
- The programming layer does *not* have access to the variables and functions defined in the internal layer.
- The internal layer *usually* does not have access to the functions defined in the programming layer. The exception is when you can define a link between the two, as I show later for *C++* and *Python*.

The C++ approach

A warning for Python programmers

Don't skip this section. Some of the concepts apply to you as well.

Here's a function that takes two inputs, the x-momentum and the y-momentum, and returns one output, the transverse momentum.

[2] In theory, this lets you create code within character strings dynamically; e.g.:

```
std::string operation
if ( a = 1 )
   { operation = "<"; }
else
   { operation = ">"; }
int limit;
std::cout << "Enter limit: " << std::endl;
std::cin >> limit;
std::string code = "pz" + operation + std::to_string(limit);
// If 'a' was set to 1, and the limit the user entered is 145,
// then the value of variable 'code' is "pz<145".
auto pzcut = dataframe.Filter(code.c_str());
```

In practice, this can get really confusing and is prone to errors.

Listing 8.27: A simple C++ function to be used with RDataFrame.

```
float pt_func( float xmom, float ymom ) {
    return sqrt( xmom*xmom + ymom*ymom );
}
```

Following the usual C++ standard, you'd define this function before you defined your main routine.[1]

Too simple?

Obviously, this function is so simple that you're not likely to define it separately just to pass it to `RDataFrame::Define()` (though see the section on *lambda expressions* later on). The point is to start with something simple as a "skeleton" for you to see how to create more complex functions of your own.

In order to use this function `pt_func` on a dataframe, you could do:

Listing 8.28: How to apply `pt_func` to each entry in an n-tuple

```
auto pt_dataframe = dataframe.Define("pt",pt_func,{"px","py"});
```

Note how this differs from what we've done before:

Listing 8.29: Our earlier approach to defining a new column in our n-tuple.

```
auto pt_dataframe = dataframe.Define("pt","sqrt(px*px + py*py)");
```

In Listing 8.29, we supply the function in the form of a text string, to which ROOT applies its internal compiler to jit the string. In Listing 8.28, we let C++ compile the function and pass that function's C++ "programming layer" name to the `Define()` method.

However, that's not enough for `RDataFrame::Define()` to use `pt_func`. It has to be told which n-tuple columns to supply as arguments to the function. That's why we also have to provide a list `{"px","py"}` as a third argument to

[1] You could also define this function *after* your main routine, and just include a forward declaration before the main routine.

`Define`.[2,3]

This gives us a recipe:

- Define a function that returns a value; e.g.,

```
float some_function( float value1, float value2, ... ) {
   // Lines of code that use value1, value2, ...
   // to calculate a result.
   return result;
}
```

- Use that function in a `Define`, supplying the n-tuple columns to be passed to the function as a list of strings:

```
auto new_dataframe =
    dataframe.Define("new-column",some_function,{"column1", "column2", ...});
```

If you're writing a function that will be called by `Filter`, the recipe is almost the same, except that function has to return a boolean result (`true`, `false`). For example:

[2] Could we have avoided the need to specify `{"px","py"}` to `Define` if we'd used those names in the definition of `pt_func`? For example,

```
float pt_func( float px, float py ) {
    return sqrt( px*px + py*py );
}
```

You've probably already guessed that the answer is no. Remember, names that are defined in the programming layer have no meaning to ROOT's internal layer. Even if we choose to use the same name in the programming layer as in the internal layer, ROOT has no direct way of matching those names between layers.

[3] If we omit the list of columns in `Define`, ROOT will assume that the user function takes every column in the n-tuple as an argument. For the extremely simple n-tuple `tree1`, you might be able to live with that; e.g.,

```
float pt_func( float c2, float eb, int ev,
               float xmom, float ymom, float zmom,
               float zvertex ) {
    return sqrt( xmom*xmom + ymom*ymom );
}
```

Then we could omit that third argument to `Define`:

```
auto pt_dataframe = dataframe.Define("pt",pt_func);
```

However:

- The compiler will toss out a lot of warning messages about "unused variables". This is accurate, since our function does not refer (for example) to `zv` in its body.
- You can't always control the order of the columns in an n-tuple. In particular, if you look at the n-tuple that you created using *Snapshot*, you may see that the method did not necessarily add the new columns to the end of the n-tuple.
- The n-tuples in real experiments often have hundreds of columns. It's impractical to list them all in the function definition. If you don't, you may get "function not found" error messages when you compile your program, the number of arguments in your function (like the two in `pt_func`) won't match the number of arguments assumed by the compiler (hundreds?).

Listing 8.30: An example of a function that could be used as an argument to `Filter`

```
bool energy_cut( float energy ) {
    return energy < 145;
}
```

Lambda expressions in C++

I'd love to finish the topic of writing functions for `RDataFrame` in C++ at this point, and move on to torturing the Python programmers with what they have to put up with.

Unfortunately, I can't. The RDataFrame tutorials make heavy use of C++ lambda expressions. In order for you to be able to understand the code in those examples, I feel I have to review this somewhat-esoteric coding practice.

Not so fast, Python programmers!

I know you want to leap ahead to the next section. I don't blame you. Before you do, there's a couple of things you should know that may influence your choice:

- Even though there is usually an equivalent Python example for every C++ example in the ROOT tutorials, sometimes there's only a `.C` file, probably because the Python equivalent is not yet possible. You'll want to be able to interpret the C++ code so you can figure out what it's supposed to do.[1]
- If you're a skilled Python programmer, you know that there are lambda expressions in Python. Unfortunately, Python-based lambda expressions can't be used as arguments to `Define` and `Filter`.[2]

[1] "Can't I tell what the C++ code is supposed to do from the comments?" Oh, how I *wish* that were true!

[2] At least, not in the current version as of the time of this writing (ROOT 6.26), or not without a whole lot of extra code wrappers and decorators. Frankly, it's not worth the effort.

Now that you're informed, I leave it up to you to decide what to read.[3]

A lambda expression[4] is a way of defining a function without giving it a name. It's normally used for short functions to make the code more convenient.[5]

Here's an example of a lambda expression that defines how to calculate our old friend **pt**:

```
auto pt_func = [](float x, float y) { return std::sqrt(x*x + y*y); };
```

Compare this with Listing 8.27:

```
float pt_func( float x, float y ) {
    return std::sqrt( x*x + y*y );
}
```

Your first reaction may be that we're using a fancy syntax to do exactly the same thing. But lambda expressions offer more than that: They allow you to define a function within the body of the code, instead of in a separate declaration before the current routine.

Let's break this down:

- `[]` means "this is a lambda expression."[6]
- `(float x, float y)` - As with any function, we have to declare its arguments.
- `{ return std::sqrt(x*x + y*y); }` - this is the body of the function, the same as any other function.
- `;` - Don't forget the semi-colon at the end! To C++, this line is a statement, not a function definition. It has to end with a `;` so the compiler will recognize it.
- `auto` - You've seen me use the `auto` before, to let the compiler do the work of specifying a complicated type.
- `pt_func` - The name I assign to this lambda expression.[7]
- If you're sharp of eye and mind, you may have noticed that we didn't have to declare the return type of the function, the way we had to in Listing 8.27. That's because the C++ compiler can automatically deduce the return type from the type of the value in the `return` statement.[8]

You can use this definition of `pt_func` in exactly the same way as in Listing 8.28:

```
auto pt_func = [](float x, float y) { return std::sqrt(x*x + y*y); };
auto pt_dataframe = dataframe.Define("pt",pt_func,{"px","py"});
```

Consider those two lines of code. I define `pt_func`, use it as an argument to `Define`, then never use `pt_func` again. As an alternative, we can use the lambda expression as an argument to `Define` to define an anonymous function:

[3] Of course, since I have no way to make you do anything, the decision is always up to you!

[4] I think that this web page offers a better explanation of lambda expressions. However, it also includes an annoying pop-up ad.

[5] I'll confess that I often write multi-statement lambda functions. I do this because when I use the function in `RDataFrame` or an iterative function like for_each, I feel it's helpful for the function to be defined right above the C++ statement that's going to use it.

[6] It means something to put variable names within those brackets, but this page is long enough, and I need room for the xkcd cartoon. I'll let you learn this topic from doing a web search on "c++ lambda capture".

[7] Wait a second. I just said that lambda expressions are used to define functions without giving them a name (*anonymous functions*), and yet here I am giving it a name.

I'm giving this lambda expression a name, `pt_func`, for the sake of convenience. Read on; I'll show you a completely anonymous lambda expression in a few paragraphs.

[8] This is not always true. If you looked at the web pages I linked to above, or you do your own web search on "C++ lambda expressions", you'll see examples in which you do have to declare a lambda expression's return type. However, this is not likely to matter for functions that you'd write for `RDataFrame`.

Listing 8.31: An example of a completely anonymous lambda expression

```
auto pt_dataframe = dataframe
    .Define("pt",
            [](float x, float y) { return std::sqrt(x*x + y*y); },
            {"px","py"} );
```

I've inserted some line breaks to separate the arguments to `Define`.

This may seem like a bit much. But it's the sort of expression that you'll see in the RDataFrame tutorials (though without the convenient line breaks).

Using lambda expressions in this way makes more sense with `Filter`:

Listing 8.32: Using a lambda expression with `Filter`

```
auto ptcut_dataframe = pt_dataframe
    .Filter( [](float e){ return e < 145; }, {"pt"} );
```

It's not all that different from what we used before:

```
auto ptcut_dataframe = pt_dataframe.Filter("pt < 145");
```

I feel a need. The need for speed!

There is a reason why the ROOT tutorials use lambda expressions so often: speed.[9] As I noted before, compiled code is faster than "jit"ed code. If you're working with a C++ notebook, you can easily see this for yourself: Create two new notebooks and put in the following:

Listing 8.33: The C++ notebook for testing the speed of jit-ed code

```
// In the first cell, copy-and-paste:
ROOT::RDataFrame dataframe("tree1","experiment.root");
TCanvas canvas;
auto pt_hist = dataframe.Define("pt","sqrt(px*px + py*py)").Histo1D("pt");

// In the second cell, copy-and-paste:
%%time
pt_hist->Draw();
```

Listing 8.34: The C++ notebook for testing the speed of compiled code

```
// In the first cell, copy-and-paste:
ROOT::RDataFrame dataframe("tree1","experiment.root");
TCanvas canvas;
auto pt_calc = [](float x,float y) { return std::sqrt(x*x+y*y); };
auto pt_hist = dataframe.Define("pt",pt_calc,{"px","py"}).Histo1D("pt");

// In the second cell, copy-and-paste:
%%time
pt_hist->Draw();
```

[9] It has nothing to do with the ROOT developers' need to demonstrate how cool they are. Of course not.
Now, if you'll excuse me, I've got to go write another smart-aleck footnote or stick another xkcd cartoon somewhere.

Run both notebooks. Bear in mind that this speed difference would be even greater if `experiment.root` were larger in both number of rows and number of columns.

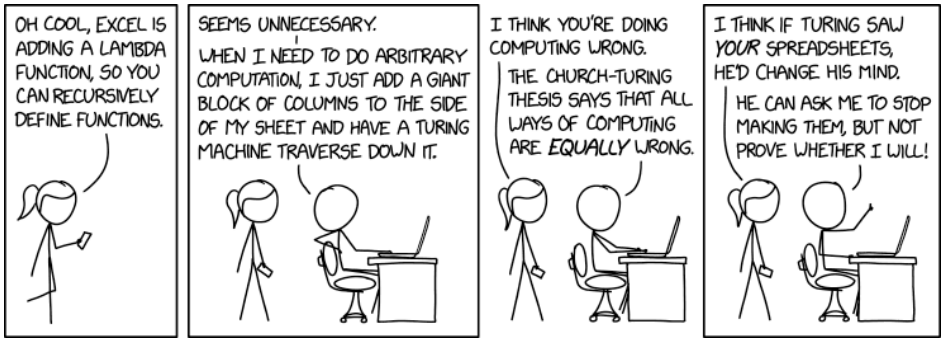

Figure 8.16:: https://xkcd.com/2453/ by Randall Munroe

The Python approach

Finally! We can escape from all this C++ nonsense and write code in Python!

Sorry, it's not so simple.

There are two ways to create your own functions to pass to `RDataFrame` in Python: Using the C++ interpreter cling, and using Numba.

The C++ interpreter

In this approach, we create a text string containing the C++ code that defines our function. Then we pass that string to cling to interpret. Here's an example:

Listing 8.35: Defining a C++ function within Python

```
import ROOT

# Define a (long) text string containing a C++ function definition.
pt_func_def = '''
// Define a function to compute the transverse momentum
// from the x- and y-momentum.
float pt_func(float xmom, float ymom) {
    float pt = sqrt(xmom*xmom + ymom+ymom);
    return pt;
}
'''

# Add the function to ROOT's C++ pool of interpreted code.
ROOT.gInterpreter.Declare(pt_func_def)
```

Notes:

- `pt_func_def` is a *string*, not a function.

- The string that will be passed to the interpreter must be delimited by quotes. That's the reason for `'''` that delimits the string; `'''gummy bear'''` defines in Python the string `'gummy bear'` *including the quotes*.
- The name of the function that will be added to ROOT's pool of interpreted code is **`pt_func`**, the function name defined in the C++ code; it is *not* `pt_func_def`.
- I emphasize again: The code in this string must be in full-fledged C++. You have to obey all the C++ conventions: specifying types, putting `;` at end of lines, curly braces, etc.

Once you've added this new function to ROOT's interpreter, you can use it in formulas that you pass to `RDataFrame`:

Listing 8.36: Using our previously-interpreted function in `Define`

```
pt_dataframe = dataframe.Define("pt","pt_func(px,py)")
```

In Listing 8.28, we had to supply a list of variables for the function's arguments. We don't do that here. The function `pt_func` has become part of ROOT's internal layer, and so it's part of the same "pool" as the names of the columns of the n-tuple. We can just supply the string `"pt_func(px,py)"` as a formula, and ROOT will be able to interpret it correctly.

Using Numba

If you want a purely Python-language approach to writing your own functions, you can try using Numba, a compiler for Python code.

Let's return to the calculation of **`pt`** one last time (I promise!):

Listing 8.37: Using a pyroot+Numba decorator to define a function for `RDataFrame`

```
import ROOT, math
dataframe = ROOT.RDataFrame("tree1","experiment.root")

@ROOT.Numba.Declare(["float", "float"], "float")
# Define a function to compute the transverse momentum
# from the x- and y-momentum.
def pt_func(x,y):
    return math.sqrt(x*x + y*y)

pt_dataframe = dataframe.Define("pt","Numba::pt_func(px,py)")
```

Notes:

- The C++ interpreter has an internal environment that includes the definition of math functions like `sqrt`. Python does not, so we have to `import math`.[1]
- The `@` tells us that we are using a Python decorator. In this case, we are *not* using Numba's own @jit decorator but a pyroot decorator that interfaces Numba to ROOT.
- For pyroot+Numba to be able to compile the function, it needs the types of the function's arguments and its return type, just as a C++ function would. That's the argument to this decorator: `(["float", "float"], "float")` means that the function takes two `float` values as input and returns a single `float` as output.
- The function itself is written in Python. Yay! At last! Indentation matters again! None of those darned semi-colons!

[1] This is probably obvious to a Python programmer, but remember that I'm a C++ programmer.

- We have to include `Numba::` in front of the function name for the ROOT interpreter to recognize the pyroot+Numba-compiled function.
- You can find a more detailed example in the ROOT tutorials.

We can do pretty much the same for `Filter`, except that we have to make sure that the function returns a boolean (`True` or `False`). For example:

Listing 8.38: Using a Numba decorator to define a filter for `RDataFrame`

```
@ROOT.Numba.Declare(["float"], "bool")
# Define an energy cut of 145 GeV.
def energy_cut(e):
    return e < 145

pz_cut = dataframe.Filter("Numba::energy_cut(pz)")
```

Why all this C++ nonsense?

Using Numba seems relatively straight-forward. Why did I take you, a Python programmer, through the C++ material? Why didn't I just start with Numba?

The answer involves some turgid details. Feel free to skip this note and move on to *Exercise 10: A more practical function*. However, if you're curious:

- Even the Numba documentation admits that it's possible that Numba might make things slower. I'm not sure what happens if you call a ROOT method (e.g., `TMath::ATan2(y,x)`) from within a pyroot+Numba decorated function.
- Numba is not always part of a Python installation. I try to make sure it's in the Nevis particle-physics Python libraries (but see the next point). If you're not at Nevis, and your site doesn't have Numba, maybe something like this will install it for you:

  ```
  pip3 install --user --upgrade numba
  ```

  If you have *installed* ROOT on your own system, maybe this command will work:

  ```
  conda install numba
  ```

- Aside from variations in installations and package managers, the reason why I keep using "maybe" in the previous point is that Numba depends on a specific version of numpy, and it's not always the most recent version.[2]

  You can get what we in the sysadmin business call "dependency hell": When you use one of the above commands, the package manager may downgrade numpy to accomodate Numba. In turn, that causes other packages (like scipy) to be downgraded as well.

  This spreads through to other packages, and you may finally get a conflict where a package can't be downgraded, and other packages can't be upgraded. The result is lots of package-manager errors and a broken Python distribution (see Figure 7.2). The only cure may be to tell the package manager to remove Numba!

- This can happen even on Nevis particle-physics systems, since Numba is not the focus of my maintenance of our Python libraries. For example, if the VERITAS group requires the lastest version of gammapy, I'm going to install it even if it means that numpy gets updated and Numba ceases to function.

[2] I don't see this as a defect in Numba. I see it as a natural consequence of building a language compiler that may require access to a different library over which the Numba development team has no control. Numba-compiled functions are going to call numpy's internals directly, without a Python language wrapper to act as an intermediary.

If numpy changes, Numba must change with it. But often there's a delay to make this happen. In my experience as a sysadmin, Numba is more frequently out-of-sync with numpy, or packages that depend on numpy, than it is in-sync.

Therefore, I can't promise you that Numba will be available. That's why I made you learn what I hope was a tolerable amount of C++.

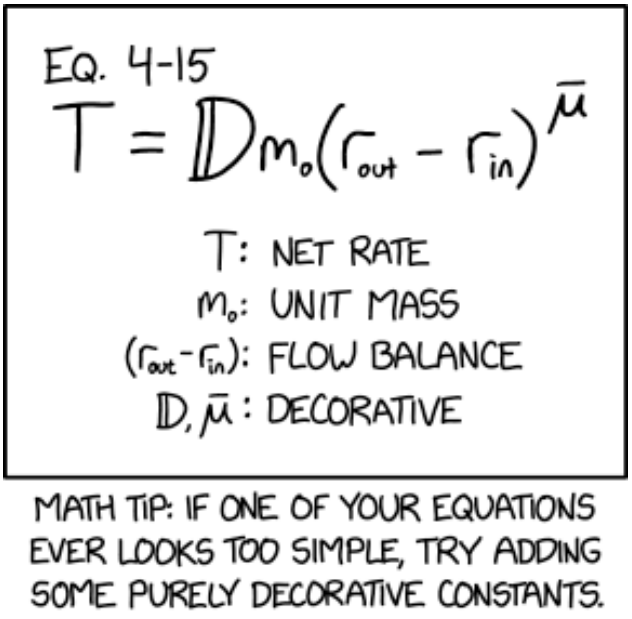

Figure 8.17:: https://xkcd.com/2566/ by Randall Munroe

Exercise 10: A more practical function

Your task is to repeat *Exercise 8: A bit more physics* by writing a function to compute E_{loss}:

$$E_{meas}^2 = p_x^2 + p_y^2 + p_z^2$$
$$E_{loss} = E_{beam} - E_{meas}$$

You'll create a new **`eloss`** column in the n-tuple, but that's the *only* new column you're going to create. You are to write a function that will take the needed columns from the n-tuple and return a value for E_{loss}, use that function in a `Define`, then make a scatterplot of this new column against **`zv`**.[1]

Hint: Look at the available columns in the n-tuple. Compare that with above formulas. Which of those columns will you need to compute those equations? How many arguments will your function take?

[1] You can try to be clever and do this:

```
eloss_df = dataframe.Define("eloss","ebeam-sqrt(px*px+py*py+pz*pz)")
```

But you know that's not what I'm asking for. For a real physics project, you're going to be asked to performed calculations that can't be shoved into a simple one-line text string. Write an actual function so you can learn how it's done.

No, you don't have to use a lambda expression. Unless, of course, you want to show how cool you are.

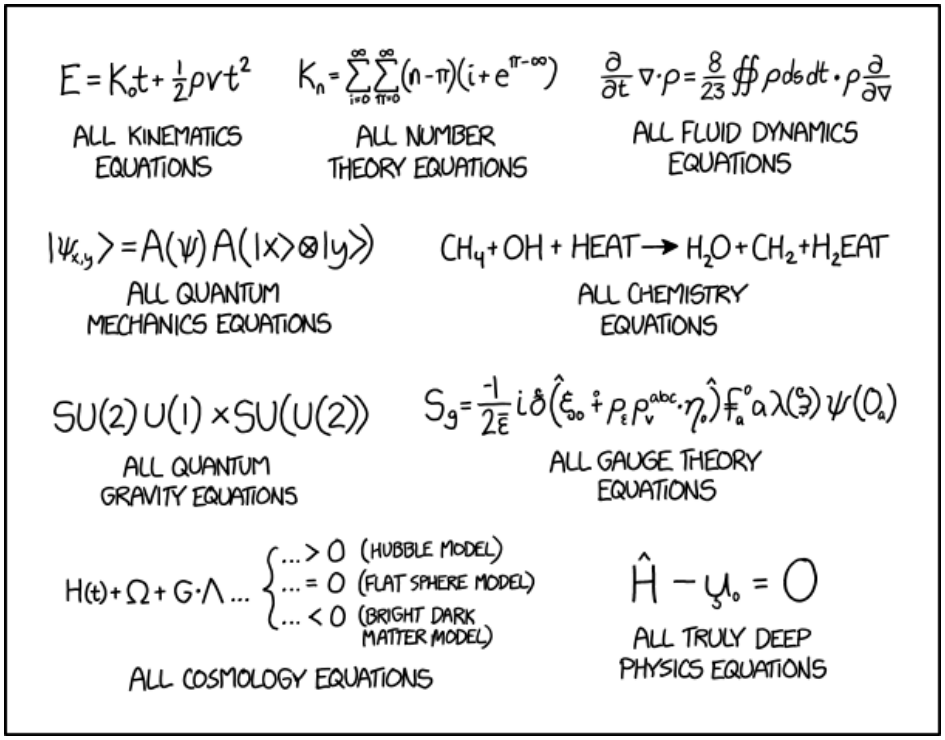

Figure 8.18:: https://xkcd.com/2034/ by Randall Munroe

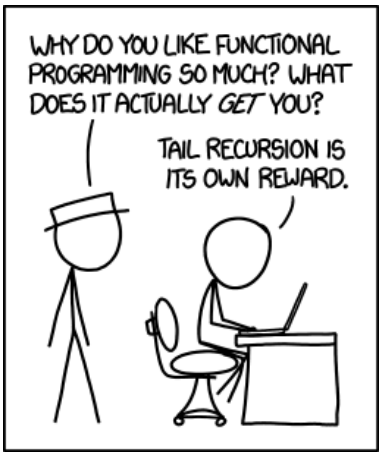

Figure 8.19:: <https://xkcd.com/1270/> by Randall Munroe. Don't worry, I'm not going to teach a functional programming language like LISP in this tutorial (though I admit I'm tempted).

CHAPTER

NINE

INTERMEDIATE TOPICS

If you've gotten this far in the course, just skim the section titles below. If anything interests you, dive in. Otherwise, use these pages as a reference for the future and move on to the challenges of the *Advanced Exercises* or the *discussions in the appendix*

You're going to need these resources as you move into the topics for the *Advanced Exercises* and *Expert Exercises*. 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.[1]

[1] 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.

References

There are several ways to figure out how to do something in ROOT:

- The ROOT web site. As of 2023, these are the areas of the ROOT manual on that site which are supposed to act as documentation:
  - The ROOT Reference. You've already visited this area. It contains the detailed description of the ROOT classes.
  - ROOT Basics. The web pages here outline a basic approach to ROOT. They assume you're already familiar with C++ or Python. It's especially useful if you don't need the step-by-step hand-holding approach I took in this tutorial.
  - ROOT Functional Parts. While this describes the ROOT classes, it does so from a technical perspective rather than act as a teaching guide. For example, take a look at their introduction to histograms to see if its approach works for you.
- The ROOT User's Guide. As I've mentioned before, this is a bit out-of-date; it has not been revised since 2018 (and it appears it never will be). For example, there's nothing in that guide about *RDataFrames*). But I feel it's a much better tutorial for new users than any of the other items on this list.[1]
- 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. There's a ROOT-based command that will tell you where they are:

```
root-config --tutdir
```

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:[2]

```
> cp -arv `root-config --tutdir` $PWD
```

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

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

You can also find the tutorials on the ROOT web site, but I find it harder to search for specific examples.

Note: 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.

[1] So what ROOT resources are out there that are both up-to-date and suitable for students who are not familiar with programming or data-analysis concepts? Unfortunately, the only one I know of is the tutorial you're reading now!

[2] I'm being tricky here by using the backtick in a UNIX command. In this particular case, what this does is take the output of the command `root-config --tutdir` and insert it into the command.

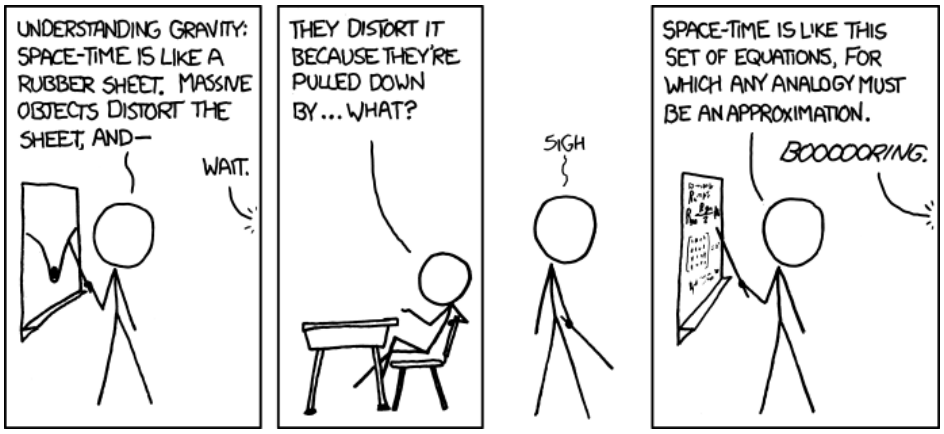

Figure 9.1:: https://xkcd.com/895/ by Randall Munroe. Perhaps this is the reason why the ROOT web site no longer teaches its concepts from scratch. It definitely explains the cartoons and jokes in this tutorial.

Advanced histogramming notes

The entries in this section were long footnotes in a previous edition of this tutorial. I decided to move them here to make *The Basics*, *The C++ Path*, and *The Python Path* less cluttered.[1]

Mean and StdDev, with weights

All the histograms you've made have a "Mean" and "StdDev" in the upper-right-hand corner of the plot. In case you need a formal definition, the mean $\overline{x}$ and standard deviation σ are defined by:

$$\overline{x} = \frac{1}{N}\sum_i x_i y_i \tag{9.1}$$

$$\sigma^2 = \frac{1}{N}\sum_i y_i \left(x_i - \overline{x}\right)^2 \tag{9.2}$$

where i goes over the histogram bins, x_i is the center of the bin on the x-axis, y_i is the sum of the weights in that bin, and N is the sum of the weights in the histogram.

I'm being super-formal here, in that I'm referring to y_i and N as "sums of weights." That's because, as I *noted* when we worked with the Treeviewer, you can assign a weight to the entries as you fill a histogram. You do this by adding an extra argument to the `Fill` method. For example, this adds a value to a histogram with the default weight of 1.0:

```
hist.Fill(value)
```

while this adds that same value with a weight of `1/error`:

```
hist.Fill(value, 1.0/error)
```

For all of the histograms in this tutorial, even in the advanced exercises, the default weight of 1.0 is sufficient. In that case, y_i is just the number of times you filled a value in bin i and N is the total number of times you filled the histogram.

However, there are a number of reasons why you'd want to reweight data as you binned it into a histogram. One example, which I hinted at above, is when each individual value has an error associated with it. You might want to weight the values so that those with large errors have little weight and those with small errors have big weights.

The values of Mean and StdDev in the plot do *not* include any values you filled that fall outside the x-axis of the histogram. This is easy to test if you have any doubts:

```
TH1D hist("hist","test",100,0,3)
hist.FillRandom("gaus",1000)
hist.Draw()
```

[1] "What? There was a version of this tutorial that was *more* cluttered?" Just for that, you get to work through all the exercises in the *Advanced Exercises* and *Expert Exercises*.

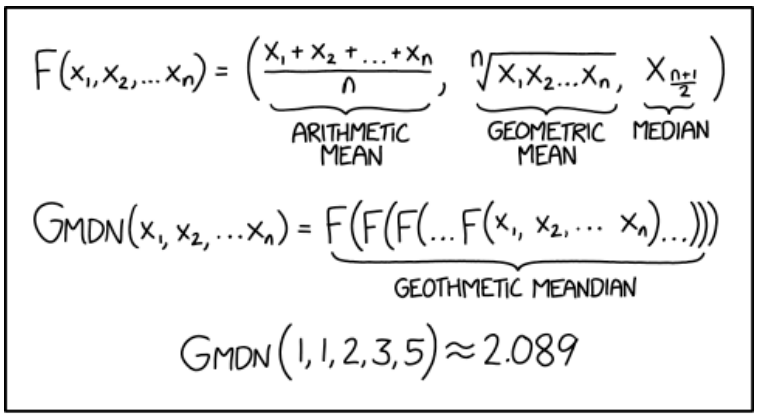

Figure 9.2:: https://xkcd.com/2435/ by Randall Munroe

Alternate Gaussian parameterization

In ROOT's `TFormula` notation, the "gaus" function refers to Equation (13.1), where A, μ, σ refer to parameters P_0, P_1, P_2 respectively. This is usually fine, except when you want a distribution that's normalized so that total area under the Gaussian distribution is unity. For that, use "gausn" instead, which uses the probability distribution function in Equation (13.2).

With this different normalization, P_0 divided by the bin width becomes the number of "equivalent events" in the histogram; that's the "area" of the Gaussian distribution expressed in units of events with weight=1:

```
[] TFitResultPtr fitResult = hist.Fit("gausn")
[] Double_t numberEquivalentEvents = fitResult->Parameter(0) /
    hist.GetBinWidth(0)
```

Automatic histogram binning

As you've noticed when working with a command like `tree1->Draw("zv")`, ROOT can automatically determine the appropriate axis range of a plot for you. You can use the same trick. It 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 adjacent bins then doubling the bin width; the bin limits change, while the number of histogram bins remains constant.

Histogram arithmetic

Suppose you're told to fill two histograms, then perform arithmetic on them; most often this will be adding histograms, but you can also subtract, multiply, and divide histogram contents. If you do this, 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.

TChain: An n-tuple in multiple files

In ROOT, it's possible to distribute a single n-tuple or `TTree` across many files. Typically you'd need this when you're running batch jobs (as described in *Batch Systems*), perhaps thousands of them, each of which independently creates a file containing an n-tuple with the same name and structure.

Within a single program, you can read all these files as if they were one continuous n-tuple. The way to do this is with a TChain.[1] You construct a `TChain` using the name of the n-tuple, then use the `TChain::Add` method to define all the files that are part of the chain.

Here's an example: Suppose instead of the file `experiment.root` that we used in the walkthroughs and Exercises, we had files `experiment0.root`, `experiment1.root`, `experiment2.root`, and so on through `experiment9.root`, all containing the n-tuple `tree1` with the same variables. We could then define a chain by:

Listing 9.1: Example use of TChain (C++)

```
auto tree1 = new TChain("tree1");
tree1->Add("experiment0.root");
tree1->Add("experiment1.root");
// ... and so on
tree1->Add("experiment9.root");
```

Listing 9.2: Example use of TChain (Python)

```
mychain = ROOT.TChain("tree1")
mychain.Add("experiment0.root")
mychain.Add("experiment1.root")
# ... and so on
mychain.Add("experiment9.root")
```

Note that in the example scripts given earlier in the tutorial (here are the *C++* and *Python* versions), these `TChain` definitions would *replace* the use of the `TFile` to define the n-tuple input file.

In *The RDataframe Path*, after you've defined the `TChain` as above, you can just supply the name of the chain as an argument to `RDataFrame`; e.g.,

```
auto dataframe = ROOT::RDataFrame(tree1);
```

or

```
dataframe = ROOT.RDataFrame(mychain)
```

The above code is fine if you only have a few files to add to the chain, though doing the copy-and-pasting of the lines for all those `experimentN.root` files would be a bit tedious. But what if you've got thousands of files? Fortunately, you don't have to specify each one of them in your program.

`TChain::Add` can accept some wildcard characters to match against file names. The wildcard you'll probably find to be the most useful is *, which matches any sequence of characters (including none). So you can do something like this:

```
mychain.Add("experiment*.root")
```

[1] If you clicked on that `TChain` link, you'll see another important ROOT class whose documentation is sorely lacking. I suggest doing a search within the *ROOT tutorials* to see some examples of `TChain` in use:

```
cd $ROOTSYS/tutorials
grep -rli tchain *
```

Note that this would also match `experiment.root` and `experiment-test.root`, which may not be what you want.

If you've learned enough programming to create loops and manipulate strings, you can also do something like this:

Listing 9.3: Example of using a loop to make a TChain (C++)

```
auto tree1 = new TChain("tree1");
for ( int i = 0; i < 10; ++i ) {
    std::string filename = "experiment" + std::to_string(i) + ".root";
    tree1->Add(filename.c_str());
}
```

Listing 9.4: Example of using a loop to make a TChain (Python)

```
mychain = ROOT.TChain("tree1")
for i in range(10):
    filename = "experiment" + str(i) + ".root"
    mychain.Add(filename)
```

Extending these slight examples to thousands of files is left as an exercise for the student.

Another approach would be to store the names of the ROOT files in a text file (or even another n-tuple!), read the filenames from this text file, then add each one. Again, I leave this as a potential exercise for you.

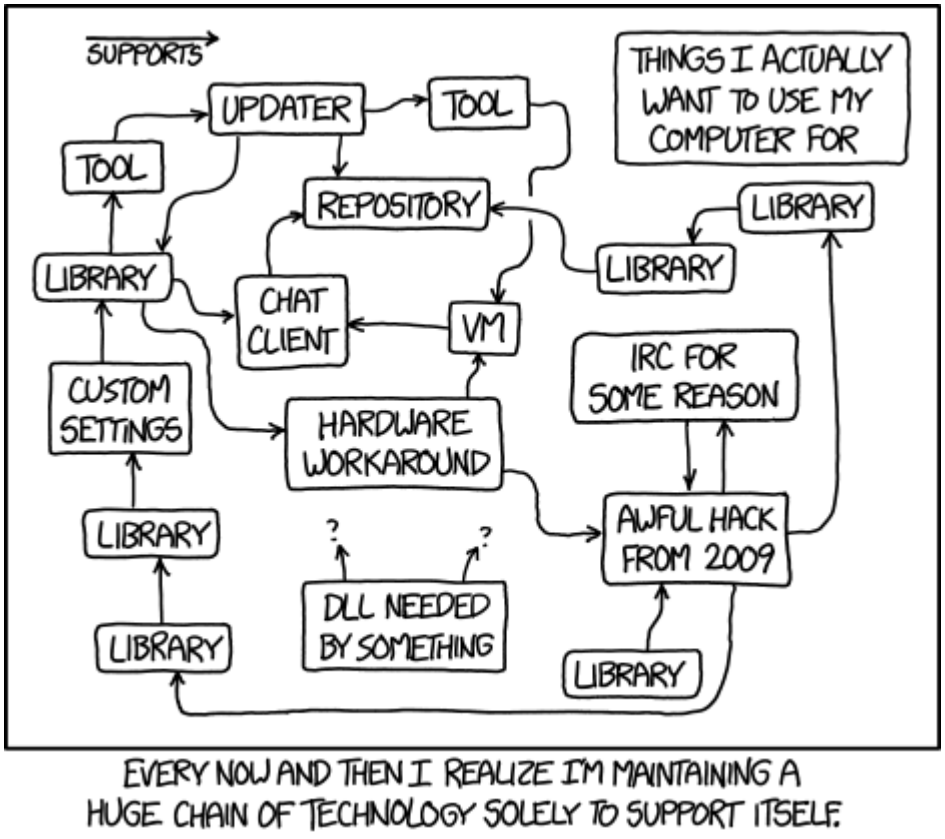

Figure 9.3:: https://xkcd.com/1579/ by Randall Munroe

Multiple threads in RDataFrame

An execution thread represents the independent execution of a set of programmed instructions. The basic model is that each core in our computer represents an opportunity to run its own program or process simultaneously with all the other cores.

RDataFrame is designed to go through an n-tuple row-by-row and perform some action on it. To a large degree, the processing of one row has no impact on the processing of any other row. That makes `RDataFrame` an ideal application for multiple threads; each thread processes a single row, and many rows can be processed at once. This can greatly speed up execution.

It's easy to set up `RDataFrame` to use multiple threads. You can turn this feature on by adding a line:

```
// In C++:
ROOT::EnableImplicitMT();
```

```
# In Python:
ROOT.ROOT.EnableImplicitMT()
```

Turn on threading first

The `EnableImplicitMT` function must be called *before* you define a dataframe; e.g.,

```
ROOT.ROOT.EnableImplicitMT()
dataframe = ROOT.RDataframe("tree1","experiment.root")
```

If you do it the other way around, you'll get an error message. The reason is that when ROOT creates an `RDataFrame`, it optimizes the set-up based on the number of available threads. If you invoke `EnableImplicitMT` after you define an `RDataFrame`, ROOT will complain that the number of threads available disagrees with the number of threads assumed for the `RDataFrame` definition.

If you want to turn off multi-threading, replace `EnableImplicitMT` with `DisableImplicitMT`.

To see how many threads are available to your program:

```
// Number of threads in C++
auto poolSize = ROOT::GetThreadPoolSize();
std::cout << "Pool size = " << poolSize << std::endl;
```

```
# Number of threads in Python
poolSize = ROOT.GetThreadPoolSize()
print ("Pool size =",poolSize)
```

Notes

- If you get a pool size of 0, it means that multi-threading is turned off. It doesn't mean that nothing executes!
- The whole point of multi-threading is that each row in an n-tuple is processed individually. This means that you lose control of the order in which n-tuple rows are read/written.

  For a simple n-tuple like `tree1` in `experiment.root`, this doesn't matter. But if an analysis requires that the rows in an n-tuple be in a particular order, you can't use multi-threading. In that case, you may want to consider *Batch Systems*.

- Most `RDataFrame` operations are compatible with multi-threading, but a few are not. These are labeled as "single-thread only" in the RDataFrame documentation.

  An example of this is `Range(m,n)`, which selects just those entries from rows `m` through `n` (not including `n` itself). If it's not clear why, think about what would happen if you applied a `Filter()` before using `Range()`; remember that with multi-threading you're processing the rows in an unpredicatable order.

- If you're *defining your own functions*, you have to be careful. Writing thread-safe code is hard; I have an extended discussion of this in a footnote in the *appendix on batch systems*.

  If you want to write your own functions and make them compatible with multi-threading, look at `DefineSlot` in the RDataFrame documentation.

- `EnableImplicitMT` only enables the *possibility* of multiple threads. The reality may be more complicated.

  Originally I meant for this page in the tutorial to be an Exercise for you. The problem is that `experiment.root` is a teeny-tiny n-tuple: only about 2MB in size; only 7 variables per row; only 100,000 events.

  When I tested it, ROOT determined that multi-threading would not be useful in this case, and refused to allocate more than one execution thread to the `RDataFrame`. The result was that the multi-threaded example took *longer* than the single-threaded example, because of the overhead in setting up the multi-threaded environment.

  After a lengthy discussion on the ROOT forums, I tried creating a larger version of `experiment.root` with more events. It took a file 1000 times larger (100,000,000 events in a 2GB file) to see a measurable difference with multiple threads. However, the results were inconsistent. I even crashed the notebook server somehow!

  I got better results when I divided that big file to 1000 smaller files and read them as a *TChain*. Although that was a more realistic approach to multi-threading, at that point I decided that whole enterprise was too complex to present to you as an Exercise.

  The take-away from this long bullet point: It does no harm to put `EnableImplicitMT` at the top of your code; most of the ROOT RDataFrame examples do this. But it might not give you any benefit for smaller analysis tasks.

Directories in ROOT

A question for you: What will happen if you execute the following code in C++?

```
TFile* file = new TFile("experiment.root");
TH1D* hist = new TH1D("example","example",100,-3,3);
hist->FillRandom("gaus",10000);
file->Close();
hist->Draw();
c1->Draw();
```

If you're more accustomed to Python:

```
import ROOT
file = ROOT.TFile("experiment.root")
hist = ROOT.TH1D("example","example",100,-3,3)
hist.FillRandom("gaus",10000)
file.Close()
hist.Draw()
```

Did you guess that the code will crash? The C++ version will give a segmentation fault; the Python version will complain that `hist` is now an object of `'PyROOT_NoneType'`. You may have even seen a crash like this before when working on Exercise 10. You've probably already guessed that the cause of the problem are "directories" (since that's in the title of this section), but how?

A directory in ROOT (the `TDirectory` class) is a way of organizing ROOT's objects. It's like a directory or folder on disk, but ROOT's directories typically hold only ROOT classes: trees, histograms, etc. They're mostly used to organize the contents of ROOT disk files (see *Exercise 12* for more) but you can define a directory in ROOT's memory without writing it to disk, the same way you can have a histogram in ROOT's memory without it being written to a file.

For the most part, you don't have to think about directories during an active ROOT session, but the example code above illustrates an exception. Let's see why it fails. To see the name of the directory you're using in ROOT, execute the following in C++:

```
TDirectory::CurrentDirectory()->GetName()
```

In Python:

```
ROOT.TDirectory.CurrentDirectory().GetName()
```

Give it a try: Start an interactive ROOT session and copy-and-paste the above command to see the name of the current directory (it will probably be `RInt` or `PyROOT`). Then copy-and-paste the TFile command in the example code above, then look at the directory name again.

Note: It looks like when you open a file, ROOT automatically creates a `TDirectory` with that file's name and makes that your default directory. This may remind you a bit of when you had to draw two histograms at once. There you had to be careful about which `TCanvas` you wrote to. Here it's important to understand which `TDirectory` you're creating objects in.

Look at the line in the example code that defines a new histogram. In which `TDirectory` will the histogram be created? I'm sure you got the correct answer: `experiment.root`.

Now execute the example code up to and including the line where we close the file. Once again, look at the directory name. We're not in `experiment.root` anymore.

What happens if we try to draw `hist` now? We'll get an error. The reason why is that when we closed the file, ROOT also removed the associated directory in its memory. When the `TDirectory` `experiment.root` was removed, everything in it was removed as well, including `hist`. The reason for the error when we try to draw the histogram is that `hist` refers to a region of memory that doesn't exist anymore.

The fix for the above example code is simple: swap the `TFile` and `TH1D` lines. Then `hist` is defined in a `TDirectory` that isn't going away.

Note: Again, for the most part you don't have be concerned with the `TDirectory` class. However, it's a good idea to keep to the practice: Don't create objects when you have an open file, unless you're going to write that object to that file.

Perhaps you're asking yourself: `hist` was created while `experiment.root` was open. Does this mean the histogram was added to the disk file? No, for two reasons: we didn't use a `Write()` method on anything, and the file was opened read-only (see Exercise 10).

More to explore

uproot

As opposed to *The RDataframe Path*, let's go to the other end of the spectrum: ROOT I/O without using ROOT. The Python uproot package reads ROOT files using only Python and numpy. It's particularly handy if you were already a Python expert before taking this ROOT tutorial, and would rather not have to touch ROOT again if you can help it.

I've installed uproot in the Python 3 installations available at Nevis.

coffea

The coffea package is column-based analysis system specifically designed for high-energy physics. I think of it as a Python-only equivalent to *RDataFrames*.

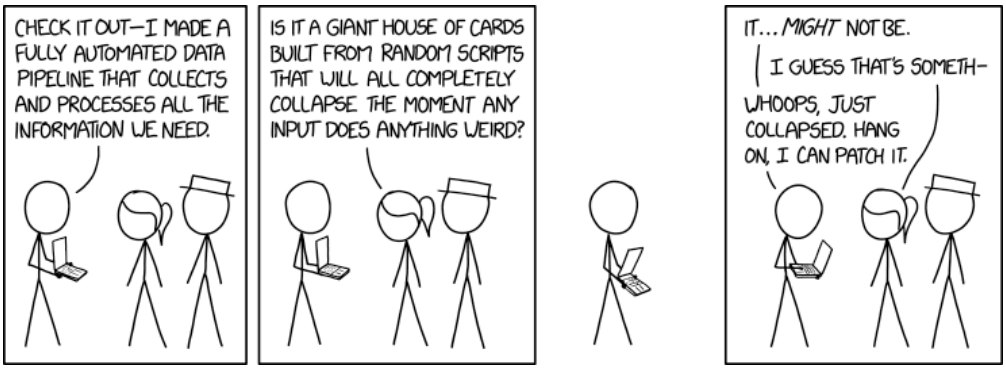

Figure 9.4:: https://xkcd.com/2054/ by Randall Munroe. Hopefully your use of ROOT data will be more rational.

Installing ROOT on your own computer

Either you have no choice, or you've decided to ignore my *warning*. Here are various methods to install ROOT+Jupyter on your own computer. They are listed in ascending order of complexity.

Anaconda

Anaconda is a software manager that allows you download and execute packages in your home directory.

If you use the Nevis Linux cluster, then you should consider using *environment modules* over Anaconda. But if you're on a different system, or the Nevis environment modules don't offer the package or version you're looking for, Anaconda is a better choice. You can install Anaconda in your home directory so admin access is not necessary. Simply follow the web site directions to install Miniconda[1].

Once you've installed Anaconda, you'll want to include conda-forge for packages such as ROOT. The following commands set it up:

```
conda config --add channels conda-forge
conda config --set channel_priority strict
```

At this point, you'll probably have log off then log in to your computer again to give conda a chance to set up your shell.

To install Jupyter/ROOT, with sufficient features for most of this tutorial:

```
conda create --name jupyter-pyroot compilers python jupyter jupyterlab root
```

Note that the name `jupyter-pyroot` is arbitrary; you can use any name for the conda environment that you wish.

If you continue to work with Jupyter or ROOT after this tutorial, there are standard packages that you'll probably want to include:

```
conda install --name jupyter-pyroot numpy scipy matplotlib
```

Your working group may use additional packages. For example, the VERITAS group at Nevis might want to use (in addition to the above):

```
conda install --name jupyter-pyroot astropy gammapy
```

You only have to go through the above steps once to define an environment (e.g., `jupyter-pyroot`). Afterwards, once per login session, it's necessary to activate it:

```
conda activate jupyter-pyroot
```

Once activated, you should be able to run ROOT by simply typing:

```
root
```

You can run jupyterlab with:

```
jupyter lab
```

Windows users

The above instructions for installing ROOT won't work under MS-Windows. You can install conda, but the ROOT windowing system so heavily depends on X11 that ROOT is simply not available in the conda libraries for Windows.

[1] For this tutorial, the full Anaconda set of packages is not necessary. Note that all the Nevis particle-physics systems already have conda installed.

The solution is to install Windows Subsystem for Linux (WSL) on your laptop; you'll probably want to install the Ubuntu distribution. Then install conda within Ubuntu, and continue as above.

You may also need MobaXterm to provide ROOT with an X11 server.

Warning:

- Conda takes a long time to run. Be patient. I've seen some conda environments take hours to install.
- A conda environment can take up a lot of disk space, since it not only installs the packages you list, but any other packages that they depend on. On my desktop computer, the minimal Jupyter/ROOT container described above takes up 3G. This may not seem like much, but summer students at Nevis have a disk quota of 10G.

  It's possible to run into disk-space problems, especially if you're sharing disk space with other users, if you add more packages to your environment, or you define multiple environments for different projects.
- Anaconda changes your shell's execution environment. It may be incompatible with other environment setups (such as MicroBooNE's LArSoft, ATLAS' Athena, or Nevis' `module load` command).
- Your shell's prompt will be changed by conda. Even when you're not using conda, the text `(base)` will appear at the beginning of the prompt. If this doesn't bother you, then ignore it. If it does, you can try:

  ```
  conda config --set auto_activate_base false
  ```

  You'll have to log off then log in again to see the change. If you don't want conda to alter your prompt even when you're using an environment, this command will suppress conda's prompt changes:

  ```
  conda config --set changeps1 False
  ```

Tip: One of the most common questions I'm asked is how to relocate a conda environment to some other location than one's home directory:

- As noted above, conda environments take up a lot of disk space. Most research groups have additional disk storage that's not part of a home directory; you can read about how the Nevis particle-physics groups handle their storage in *Shared filesystems*.
- Many research groups want to share their conda environments, to make sure that everyone is using the same versions.

For tips on managing disk space for your conda environments, see the Nevis conda wiki page. The conda page on managing environments is also helpful.

Other packaged distributions

There are other packaging systems than Anaconda; I only emphasized that one because it's available on Mac OS X, Windows, and Linux. But if you're already using a package system, ROOT may be a part of it.

For example, on Mac OS Homebrew has both ROOT and Jupyter:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

(continues on next page)

(continued from previous page)

```
xcode-select --install
brew install python root jupyterlab
```

In RHEL-derived Linux systems (e.g., Fedora, CentOS, Rocky Linux, AlmaLinux), the EPEL repository has also has both ROOT and Jupyter. For example:

```
sudo yum -y install epel-release
sudo yum -y install root\* python3 python3-pip python3-root python3-devel python3-
↪jupyroot
pip install --user jupyter jupyterlab numpy scipy matplotlib rootpy rootkernel
```

Note that neither of the above examples is complete. You'll have some homework to do!

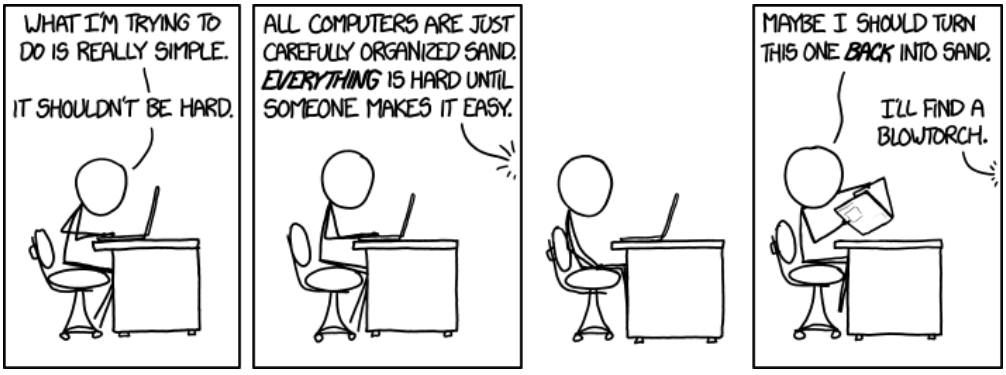

Figure 9.5:: https://xkcd.com/1349/ by Randall Munroe

Docker

While Anaconda is an environment-level container, Docker (and Singularity below) are OS-level containers.[1] Docker is probably the best method of running Jupyter+pyroot without having to worry about issues like package compilation, though you won't be able to use it to go through *The Basics*. Its disadvantage is that it requires administrative access to the host computer system (e.g., your laptop), both to install Docker and to run the Docker container.[2]

The first step is to install Docker. For Mac and Windows systems, use Docker Desktop; there's a different procedure needed for Linux systems.

Once Docker is installed and running, you'll be able to download and run a Docker container:

```
sudo docker run -p 8080:8080 -v $PWD:/work wgseligman/jupyter-pyroot:latest-<proc>
```

where <proc> is `amd64` for any machine with an Intel or AMD processor, or `arm64` for a system with an ARM processor (such as a Mac with an M1 or M2 chip).

(Windows users will probably need to use `%CD%` instead of `$PWD`.)

The first time you run this command, it will download a ~2.5GB container. Give it time.

Finally, you'll see some output. Look at that output carefully, as it will tell you how to access Jupyter via a web browser. For example, assume the output contains something like this:

[1] In contrast to emulators like VMware, which are machine-level containers.

[2] Strictly speaking, you can in theory run Docker in "rootless" mode, either on its own or using the docker alternative podman. In practice, when I tried this it led me down a road of complexity that was roughly equivalent to installing ROOT from scratch. If you don't have administrative access to your computer, I recommend looking at a different solution than docker or podman.

```
To access the notebook, open this file in a browser:
file:///root/.local/share/jupyter/runtime/nbserver-1-open.html
Or copy and paste one of these URLs:
http://649d0c4b4dc1:8080/?token=97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c
or
http://127.0.0.1:8080/?token=97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c
```

Remember, your complicated token *won't* be `97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c`. That was just the value returned by Jupyter at the time I created this example.

Start up a web browser and visit (in this example):
http://127.0.0.1:8080/?token=97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c

You'll see the standard Jupyter home page.

That will get you started. The next few sub-sections are for refining your use of Docker.

Changing the port

Consider the command:[3]

```
sudo docker run -p 8080:8080 -v $PWD:/work wgseligman/jupyter-pyroot:latest-amd64
```

That first `8080` is the port to use on your local computer. If you want to use a different port on your computer (for example, you're already using port 8080 for something else), change that first `8080` to a different port. Note that if you change the port, you'll also have to change the port in the URL in the output; e.g.,

```
sudo docker run -p 7000:8080 -v $PWD:/work wgseligman/jupyter-pyroot:latest-amd64
```

means you'll have to change:

http://127.0.0.1:8080/?token=97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c

to:

http://127.0.0.1:7000/?token=97d7242fc79734f1429bc425c627ccc4f586675c01ecdd9c

Changing the directory

Again, consider:

```
sudo docker run -p 8080:8080 -v $PWD:/work wgseligman/jupyter-pyroot:latest-amd64
```

That `$PWD` (`%CD%` in Windows) just means "the current directory." The execution environment within the container uses `/work` for its files; the `-v` option in the command means "map `/work` to the current directory in the terminal." If you'd like to use a different directory on your computer as the work directory in the Docker container, substitute that directory for `$PWD`. For example:

```
sudo docker run -p 8080:8080 -v ~jsmith/root-class:/work wgseligman/jupyter-
↪pyroot:latest-amd64
```

[3] For the rest of this page, I'm going to assume that you're on a system with an Intel-compatible processor. If you're on M1 or M2 Mac, don't forget to replace `amd64` with `arm64`.

Changing the container

You can use click on the **Terminal** icon within Jupyter to get a shell. Within that shell, you can modify anything within the container you want; for example, you can use pip3) to install new Python packages or yum) to install new Linux packages.[4]

However, any changes you make to the Docker container won't be automatically saved when you quit the container. When you next start the container, it will start "fresh." If you want to save your changes, you'll have to commit.) them.

For example, assume that you've made some changes to your copy of the jupyter-pyroot container. Look up the ID of the container as assigned by your local docker process:

```
sudo docker container ls
CONTAINER ID IMAGE COMMAND [...]
1105371318e8 wgseligman/jupyter-pyroot "jupyter notebook ..." [...]
```

Your output will be different; I've omitted most of the columns, and you'll have a different `CONTAINER ID`. Commit a revised container using your own image name:

```
sudo docker commit 1105371318e8 $USER/jupyter-pyroot
```

You'll can see your new image with the docker images command. For example, if $USER is "jsmith":

```
sudo docker images
REPOSITORY TAG IMAGE ID [...]
jsmith/jupyter-pyroot latest 97ca601cbf9c [...]
docker.io/wgseligman/jupyter-pyroot latest 16c3bbdc8144 [...]
```

From that point forward, you'll probably want to run your new container with your changes:

```
sudo docker run -p 8080:8080 -v $PWD:/work jsmith/jupyter-pyroot
```

Docker container notes

I prepared the container `wgseligman/jupyter-pyroot` to be similar to the environment of the notebook server; for example, it runs the same version of the OS and of ROOT (as of May-2023, that's AlmaLinx 9 and ROOT 6.28.02).

A little bit web searching will show there are other ROOT containers available. For example:

```
sudo docker run -p 3000:8080 pedwink/pyroot-notebook
```

That particular container uses Fedora 28 and ROOT 6.14, and it also offers Python 2 versions of its notebook kernels (`wgseligman/jupyter-pyroot` only offers Python 3).

So, if you can't find the feature you want in `wgseligman/jupyter-pyroot`, hunt around a bit. It's probably out there.

[4] If you install something of general interest, let me know. I may add it to the main jupyter-pyroot container.

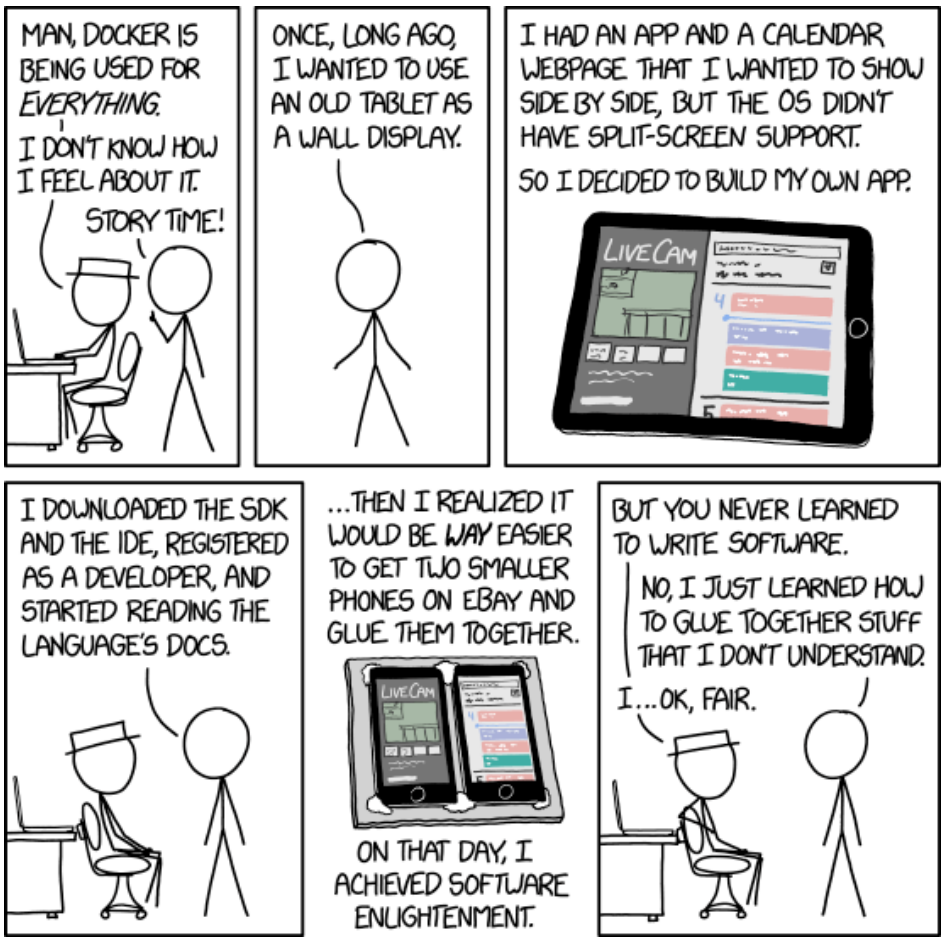

Figure 9.6:: https://xkcd.com/1988/ by Randall Munroe

Singularity

If you don't have admin access to your local computer, or you simply prefer it, you can use Singularity instead; this is also known as Apptainer. You still need admin access to install Singularity, or a willing sysadmin to do it for you.[1]

To download the container and convert it to Singularity's .sif format:

```
singularity pull docker://wgseligman/jupyter-pyroot
```

After some processing, you'll have the image file `jupyter-pyroot_latest.sif`. Then you can run Singularity on that container:

```
singularity run --bind=$PWD:/work jupyter-pyroot_latest.sif
```

Note that while you can change the mapping of the /work directory within the container (see the Docker instructions above), you can't change Jupyter's binding to port 8080. This might be a problem if you're running on a shared computer system and more than one user wants to run this container at the same time.

[1] Singularity/Apptainer is already installed on all the systems in the Nevis particle-physics Linux cluster.

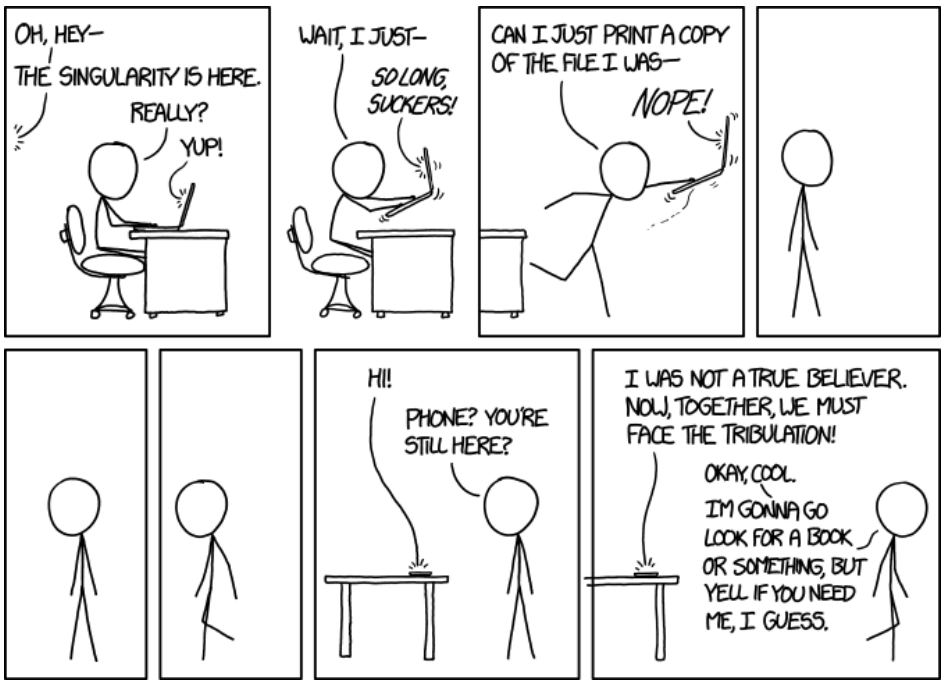

Figure 9.7:: https://xkcd.com/1668/ by Randall Munroe

The hard way: compiling ROOT and Jupyter from scratch

I repeat my *advice*: **Don't**.

Obviously, it's possible to install these packages from scratch; 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.

Now that you have been warned, here are the places to start:

- Python (if it's not already installed on your system)
- ROOT
- Jupyter

Please keep the following in mind:

- These are *not* applications that you can double-click to automatically install. The process requires some knowledge of the command shell.
- Read the installation documentation for each package. Use some thought and initiative. If you aren't familiar with UNIX shells before you started this process, you will be once you finish!
- The dockerfile I used to create the wgseligman/jupyter-pyroot container may provide a clue for how to create your own installation.

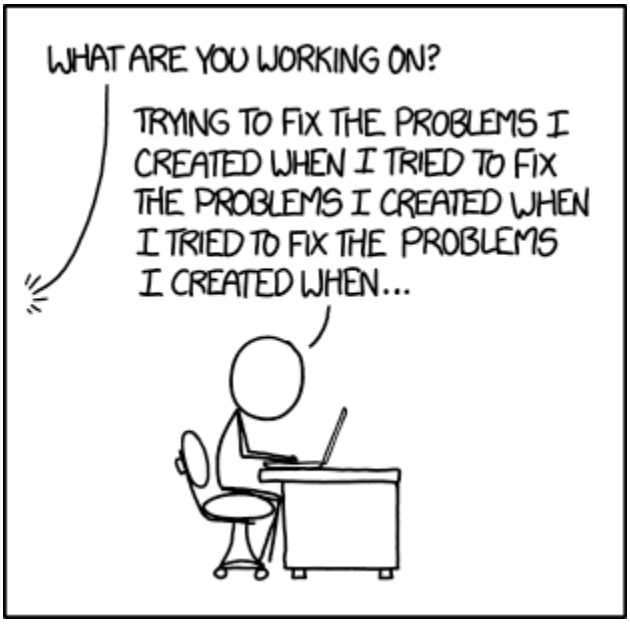

Figure 9.8:: https://xkcd.com/1739/ by Randall Munroe

Figure 9.9:: https://xkcd.com/1654/ by Randall Munroe.

CHAPTER

TEN

ADVANCED EXERCISES

If you still haven't finished the exercises in *The Basics*, *The C++ Path*, or *The Python Path*, then keep working on them.[1] 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.[2] 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.

You may notice that I don't provide links to the ROOT classes I talk about in the following sections. That's *deliberate* on my part. I want you to grow used to looking up topics in the ROOT web site (and elsewhere on the web), since that is what you'll have to do for real programming tasks.

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 with pyroot will pose its own set of challenges. You'll learn something either way!

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

[1] To be honest, if you're only exposure to programming has been the *The RDataframe Path*, then these advanced exercises will be an uphill battle. None of them can be completed using the `RDataFrame` class.

[2] If I've underestimated you, please let me know! It's always good for a teacher to learn that students are better than you've assumed them to be.

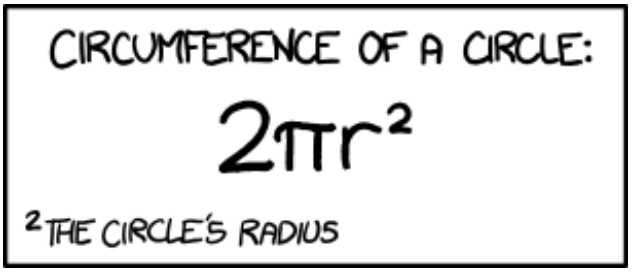

Figure 10.1:: <https://xkcd.com/1184/> by Randall Munroe. This is an example of an xkcd cartoon **and** a smart-aleck footnote.

Working with folders inside ROOT files

Note: 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, to *organize objects within a file*.[1]

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

Note: 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 of the subsequent exercises will require you to make a plot of data points with error bars. You'll want to use the `TGraphErrors` class for this.[2]

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

Note: 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 a graph from the contents of a file, and to create plots from either float or double-precision… somethings.

Those somethings are containers. We'll learn about those in the *next section*.

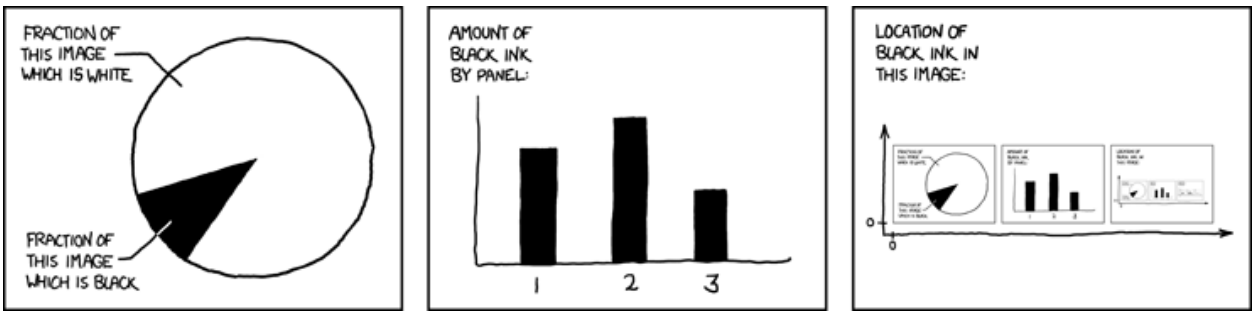

Figure 10.2:: https://xkcd.com/688/ by Randall Munroe

[1] You may already be familiar with HDF5, which is conceptually similar to the approach that ROOT takes for its files: containers for a mixed set of different types of data within a hierarchy of directories. I haven't worked with HDF5 (yet) so it's hard for me to compare the two.

[2] 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

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.[1] 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`.

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);
```

Note: 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 subsequent exercises 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.

Note: 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.[2]

[1] 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.

[2] If you did this and are puzzled by my description, search the web for the definition of "sarcasm."

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.[3]

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

C++ Standard Template Library (STL)

Do a web search on "standard template library". 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.

Note: 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 about them.[4]

Vectors

Note: For the work that you'll be asked to do in the *Advanced Exercises*, *Expert Exercises*, 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 C++ 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:

[3] 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 often have to actually look at ROOT's C++ source code. I decided to spare you that as much as possible.

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

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

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

```
for ( size_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 `Double_t` array.

We're getting closer to being able to tackle the *first advanced exercise*!

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:[1]

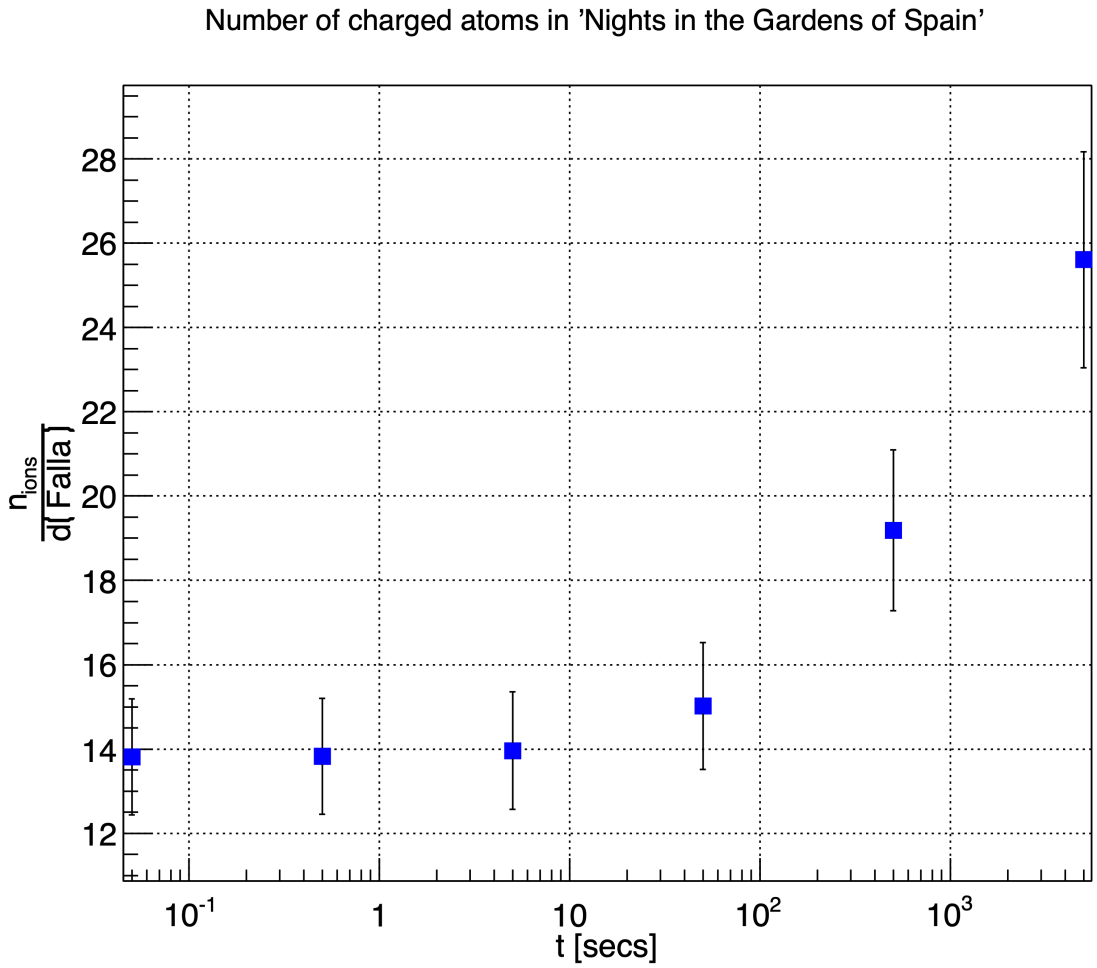

Figure 10.3:: Can you spot the pun in this plot? Hint: It involves the composer of a piece of music for piano and orchestra written in the early 20th century.

Use the histograms in folder `example1` from the 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.

Note: You've scrolled through web 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.

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.[2] You've seen some macros before (remember `c1.C`?) and you'll find many more in the *ROOT tutorials*.

[1] 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* into your axes labels.

[2] Instead of creating a macro file, you could try typing the commands on the ROOT command line one-by-one. However, unless you have a

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 *References*, 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.[3]

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; you must 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*), 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?

Note: 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"; remember *Exercise 3*?). 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

// Loop over all possible histogram numbers.
for ( Int_t i = 0; i != 6; ++i )
{
    // Define an "output string stream" object; that is, a variable
    // that can be the output of C++-style I/O.
    std::ostringstream os;

    // Write the characters "hist" following by the number i
    // into the string stream.
    os << "hist" << i;
```

(continues on next page)

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).

[3] 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. Their deep nitty-gritty is rarely relevant in physics. On the other hand, I use them all the time; e.g., searching the ROOT tutorials for hints.

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/

(continued from previous page)

```
    // Extract the text string from the string stream and save
    // the result in a ROOT-style string.
    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.

- In Figure 10.3, the x-axis is logarithmic. How do you make that change?

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

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

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

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

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

Now you can get to work!

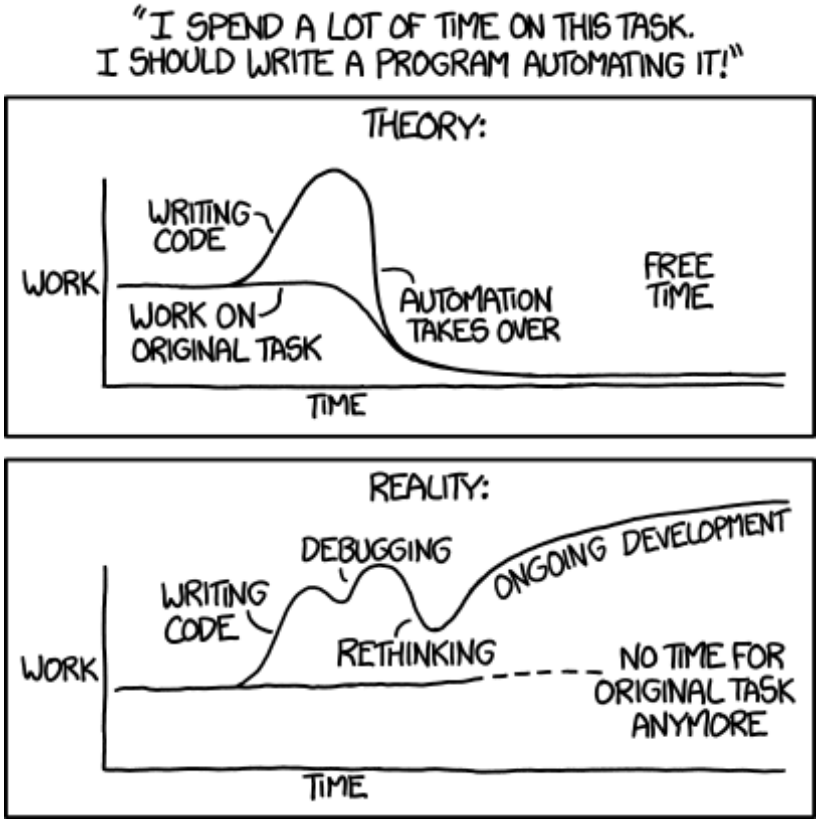

Figure 10.4:: https://xkcd.com/1319/ by Randall Munroe

Exercise 13: A more realistic x-y plotting task

(1-2 hours)

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

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

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

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

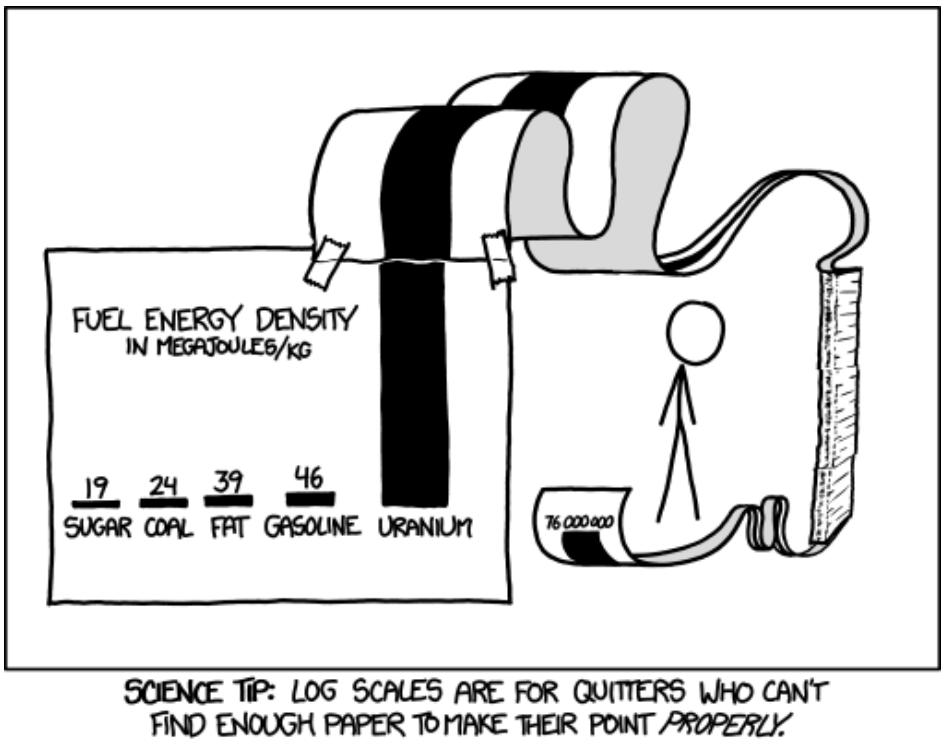

Figure 10.5:: https://xkcd.com/1162/ by Randall Munroe

Note: You've probably already figured out that you can use `MakeSelector` on the `histogramList` n-tuple, like you did *before*. 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 NULL*.

CHAPTER

ELEVEN

EXPERT EXERCISES

You've gotten this far. That's impressive. Now let's see how good you really are.

As I try to say during my introductory lecture to this tutorial, skill as a computer programmer is poorly correlated with skill in physics. But if you're someone who can do both, you've got an extraordinary career ahead of you!

Exercise 14: A brutally realistic example of a plotting task

(1-2 hours)

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.

Note: 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 a string into 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.[1] 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?

The next problem is trickier: How do you get a list of objects in a directory?[2]

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[3] TDirectory. I found something, clicked on the name of the method… then pounded my head against the desk.[4]

[1] 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 language update, C++11, has a "regex" library for handling regular expressions; this can also be found in ROOT's cling.

In Python, all this is much simpler. (Hint: `import re`)

[2] 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.

[3] TFile's "grandparent."

[4] 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 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.

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 to 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++:[5]

```
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.

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 == NULL )
{
   // 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==NULL`, then the program would crash with a segmentation fault.[6]

Note: Why did I write such a long note 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.

---

[5] How do you figure out which classes derive from where? 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!

[6] 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.

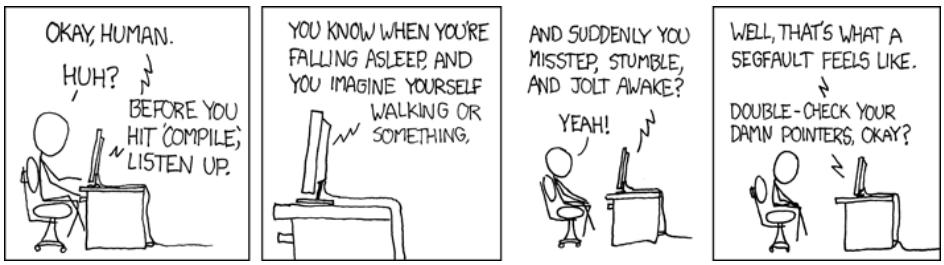

Figure 11.1:: https://xkcd.com/371/ by Randall Munroe

Exercise 15: Data reduction

(1-2 hours)

Note: 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.[1]

Copy the 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 = 0;
// Assume we assign 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`.

Note: 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.

[1] If you're 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")
```

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.[2] I *noted* this earlier: 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.[3] 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.

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.[4]

Hint: Some additional hints:

- The first line of your ROOT macro for this exercise is likely to be the library load command above.
- 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 $ROOTSYS/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`.

[2] 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.

[3] 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.

[4] 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.

There's a standard identification code used for particles. The Particle Data Group developed it, so it's called the "PDG code".[5] There are methods in `ExampleEvent` that return this value (e.g., `LeptonPDG`). For this exercise, these codes will be sufficient:

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.

Extra challenge

Use the `RDataFrame` class to create the output n-tuple, instead of manually fiddling with `TTree` and branches. One aspect will be a trifle easier: Using the `.Define` method in `RDataFrame` is easier than defining a branch.

The harder part will be figure out how to pass a calculation to `Define` to calculate each column in the n-tuple. You'll have to learn how to create (in the words of the ROOT web site) a "function, lambda expression, functor class, or any other callable object", none of which I've mentioned so far in this tutorial.

Get to work![6]

[5] If you'd like to see them, here's a PDF file with a complete list of codes.

[6] In the time since I constructed this exercise in the mid-2000s, a new class has been added to ROOT: `TNtuple`. It may make the process of writing n-tuples easier for you. Take a look!

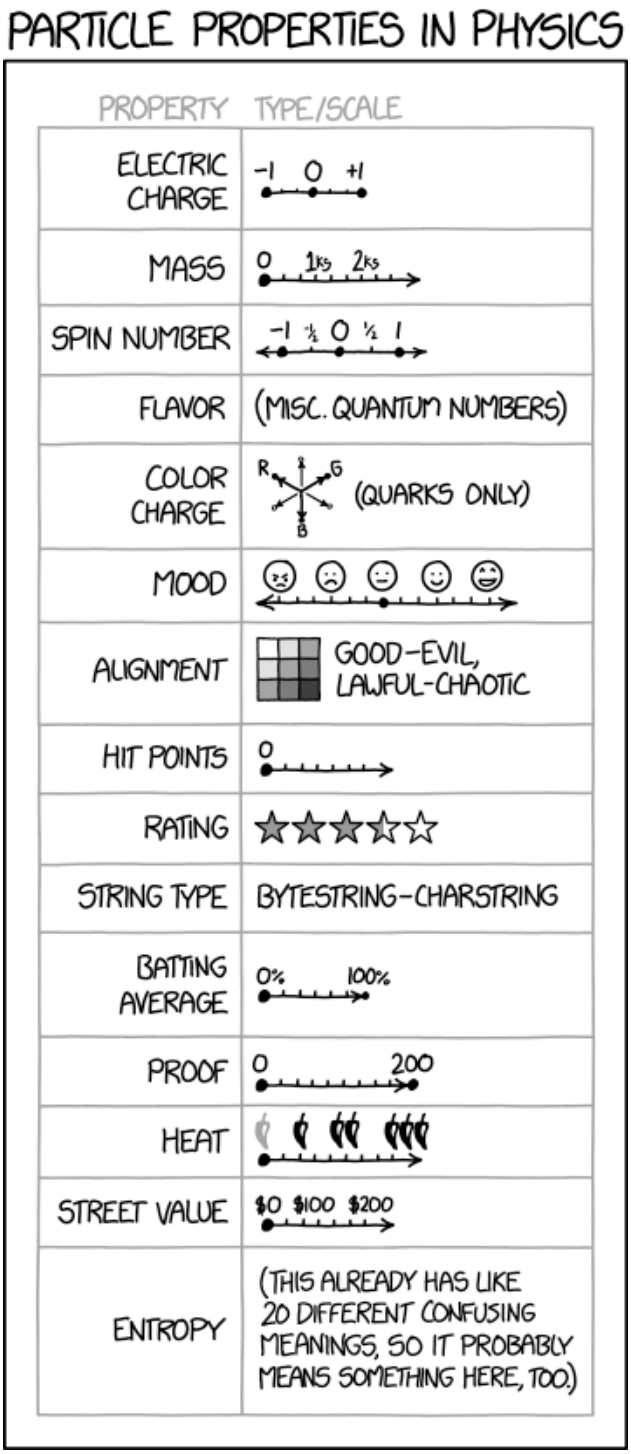

Figure 11.2:: https://xkcd.com/1862/ by Randall Munroe

CHAPTER

TWELVE

WRAP-UP

The last four exercises that make up *Advanced Exercises* and *Expert Exercises* 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.[1]

If you'd like to see how I solved those exercises, you'll find my code in PlotGraphs.C (for exercises 12-14) and MakeNtuple.C (for exercise 15).[2]

Good luck![3]

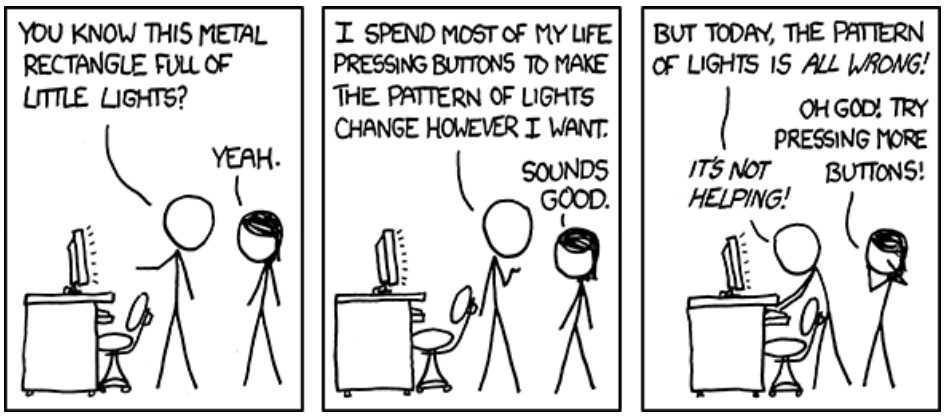

Figure 12.1:: http://xkcd.com/722 by Randall Munroe

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

[2] Maybe you're thinking, "Wow! It's lucky I went to the end before I started doing any of this stuff!" Take my word for it: reading my solutions is not a substitute for working through the problem yourself.

For example, I wrote those solutions in the mid-2000s. Since then there have been improvements in ROOT, C++, and Python. You can do better than I did!

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

CHAPTER

THIRTEEN

APPENDIX

If you ever read Lord of the Rings, you know that about a third of the third volume, The Return of the King, consists of appendices to enhance the story you've just read.

The purpose of the following is to help enhance your physics work this summer. If you've gone through *The C++ Path*, *The Python Path*, or *The RDataframe Path*, you've got some time left, and you don't feel ready to proceed to the *Advanced Exercises*, then feel free to browse through the following.

There's not much about ROOT itself, but these sections may help better prepare you for your upcoming research.[1]

[1] Unlike the appendices in *The Lord of the Rings*, you're not likely to learn much about Hobbits or the Downfall of Númenor. Sorry, but there are limits, even for me.

A too-brief and too-long introduction to statistics

This section is for those students who have had little or no exposure to statistics before taking this tutorial.

Strictly speaking, this section has nothing to do with ROOT *per se*; that's part of why it's "too long." However, I was asked to include something on these topics to hopefully give you a head start as you get involved with the research you'll do this summer.[1]

My goal is to explain the following terms of statistics "jargon"[2] that physicists use.

Gaussian

The Gaussian[1] function (sometimes called the "normal distribution" or "the bell curve," though both terms are a bit inaccurate in this case) is a standardized curve that frequently comes up in physics; for example, in random processes such as particle decay. The formula for the Gaussian function is:

$$f(x) = e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{13.1}$$

- μ is the mean of the distribution. (For a Gaussian, the mean, mode, and median are the same.)
- σ = the standard deviation; it's related to the "full width at half maximum" (FWHM) of the curve by FWHM = $2\sigma\sqrt{2\ln 2} \approx 2.35\sigma$.
- e = Euler's constant, a transcendental number that occurs often in calculations that relate to growth and increase. It's formally defined as $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$.

If you want to work with the normal distribution as a "probability density function" then you'll need to include a *normalization* so the integral $\int_{-\infty}^{\infty}\mathcal{N}(x)dx = 1$.

$$\mathcal{N}(x:\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{13.2}$$

Without the normalization factor, the maximum value ("amplitude") of $Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$ is A, which is easy to read from a graph.[2]

[1] The full subject of statistics is deep and beautiful, and you can spend a lifetime… well, you know the rest.

[2] In this discussion, when I say "physicists do this" or "physicists do that," it's not exclusive. Other fields of study use the same jargon. However, since I mostly hang out with physicists and not cultural anthropologists or librarians, I'll only speak for the folks I know.

[1] You may have noticed I'm sloppy, and use the terms "Gaussian distribution", "Gaussian function", or just plain "Gaussian" interchangeably, as if they were all exactly the same thing, or even linguistically correct. All I can do is quote Ralph Waldo Emerson: "A foolish consistency is the hobgoblin of little minds". Of course, this quote has nothing to do with physics, statistics, or the size of my little mind.

[2] When I was a graduate student, I showed a plot very much like this one at one of my first talks to the rest of my experiment's group. The spokesman thundered "Take that off the screen!" He felt that the information it conveyed was so trivial that I was insulting everyone's intelligence. Don't make the same mistake that I did!

On the other hand, if you feel that *your* intelligence has just been insulted, you may be ready to be an experiment spokesperson. Quick, write a proposal!

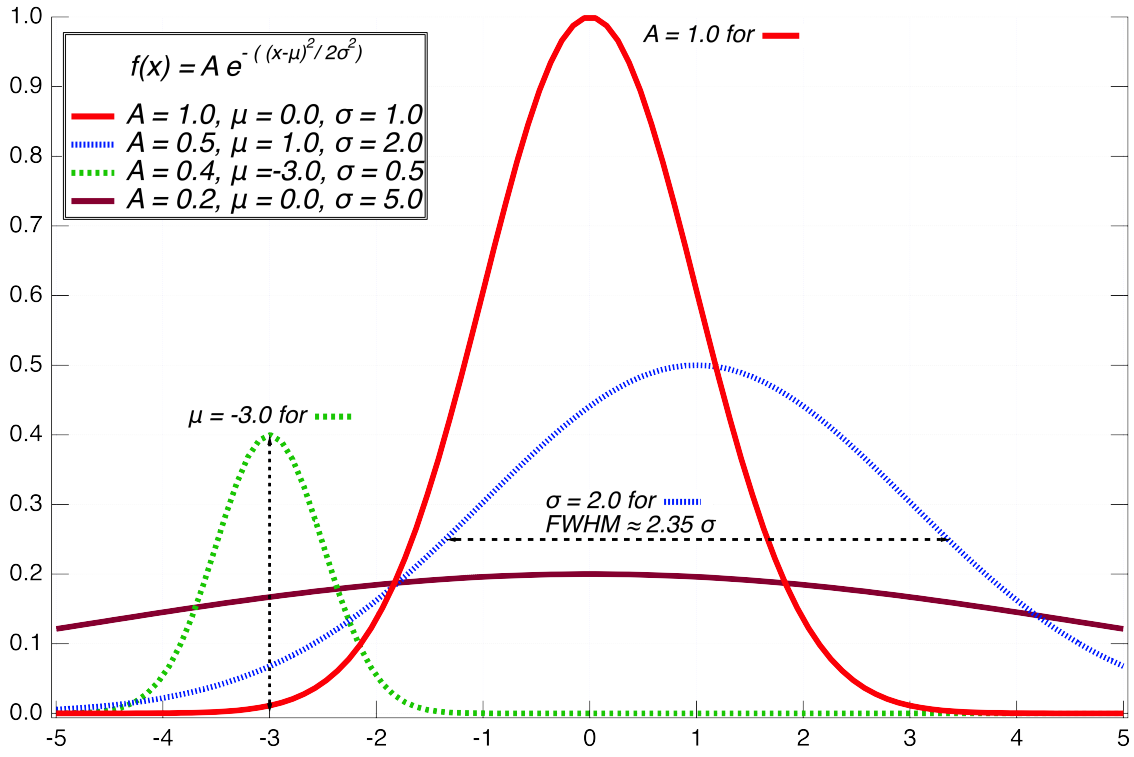

Figure 13.1:: A few Gaussian functions with annotations.

I'll repeat something I say in *The Basics*: You see the word "Gaussian" a lot in this tutorial, because it's an easy distribution to work with as I prepare plots and such. There are many other probability distributions and functional forms that are used in physics. For example, there's the Landau distribution, which describes energy loss of ionizing particles.

Gaussian distributions come up a lot because when you're making an observation that depends on the combination of many underlying probability distributions, the combination generally tends towards a Gaussian. But don't start believing that everything is normal!

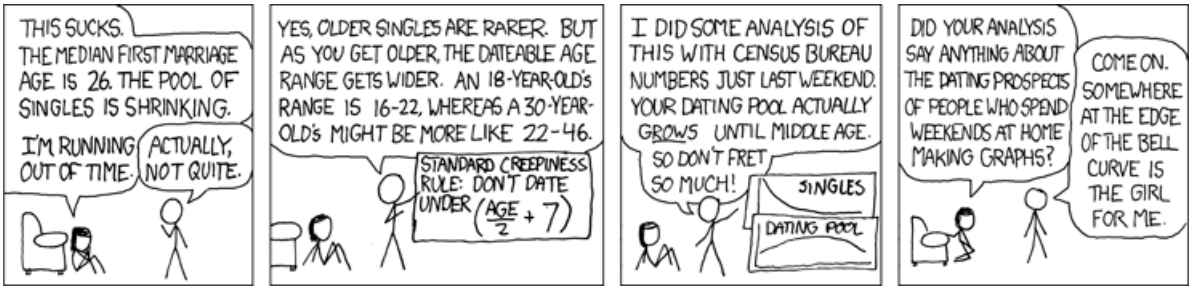

Figure 13.2:: https://xkcd.com/314/ by Randall Munroe

Poisson distributions

Let's take a brief look at one alternative distribution since it will come up later: the Poisson distribution, which models the number of discrete events seen in an interval of time or space; for example, radioactive decay. It's given by:

$$P(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{13.3}$$

where k is the number of occurrences you might measure in an interval, λ is the mean number of events you'd expect see per interval, and $P(k;\lambda)$ is the probability of seeing k events given a mean of λ.

This plot may help you visualize what's going on:

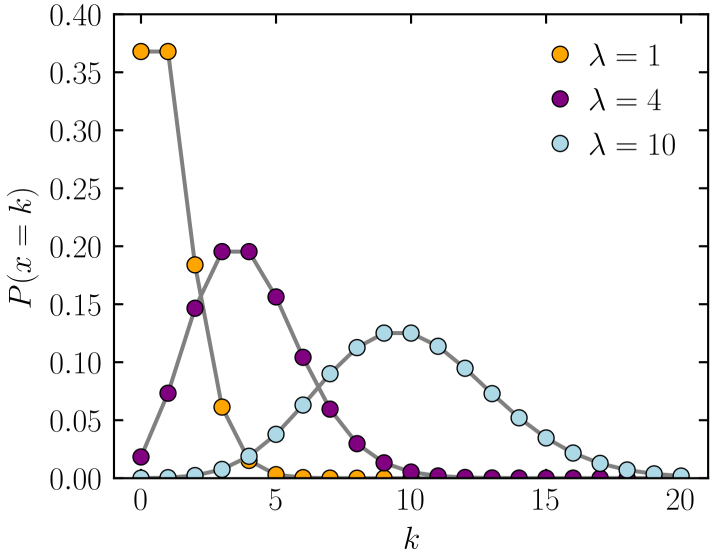

Figure 13.3:: Examples of Poisson distributions for a few values of λ. Source: Skbkekas under the CC BY 3.0 license

For example, if you're running an experiment whose observations follow a Poisson distribution, and the mean number of events you'd expect to see is λ=4, then the probability of seeing exactly 4 events in the interval is about 0.20.

You may have noticed that for λ=10, the curve looks like a Gaussian. In fact, a Gaussian can be defined as the limit for large λ of a Poisson distribution.[1]

As it happens, the particle-physics experiments that I've worked on have typically involved tens to hundreds of thousands of events, even after applying cuts,[2] so I've usually worked with Gaussian probability distributions. If you're working on an experiment with infrequent events in space or time, then you'll become more familiar with the Poisson distribution.

Let's keep that in mind as we move on to discussing fitting a function to data *in the next section*.

[1] Why is it a "Gaussian" distribution (making an adjective from the name Carl Friedrich Gauss) but a "Poisson" distribution (after Siméon-Denis Poisson) and not a "Poissonian" distribution? I'm a physicist, not a language expert. I'll leave the answer as an exercise for the student.

[2] If you don't know what "applying a cut" means yet, see either *the C++ explanation* or *the Python explanation*.

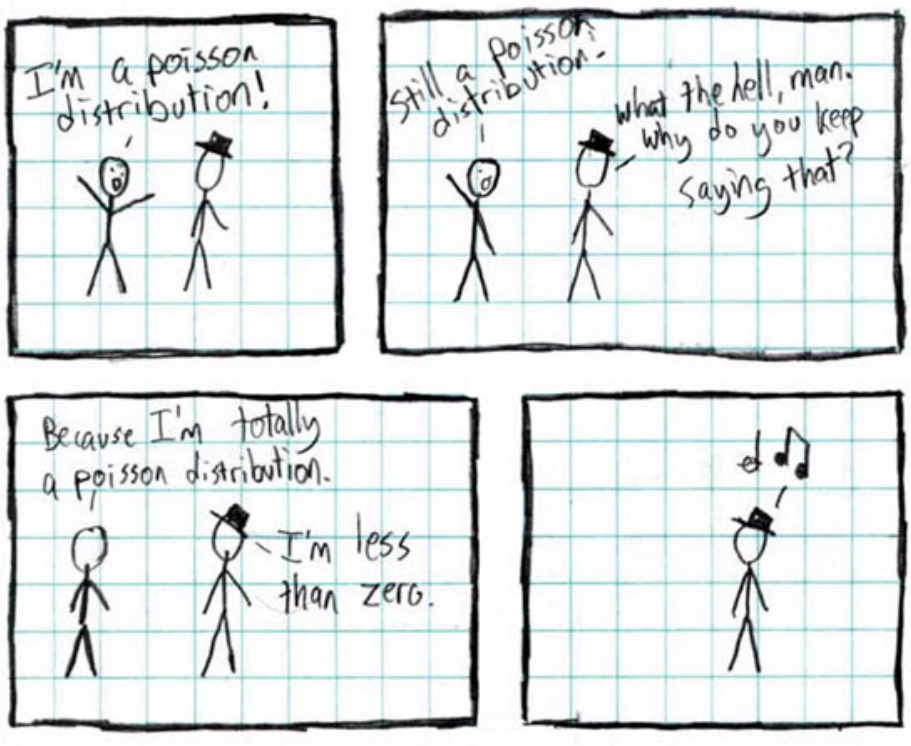

Figure 13.4:: https://xkcd.com/12/ by Randall Munroe. This early xkcd cartoon is more subtle than usual. If you're having trouble figuring it out, take another look at the above equation and plot.

Chi-squared

It's rare in physics to perform an analysis task and see nice smooth curves like those in Figure 13.1. For the most part, you make histograms of values, as in my introductory talk. Such a plot might look something like this:[1]

[1] If you jumped here from *The Basics*, you're going to make your own plots similar to the next couple of figures.

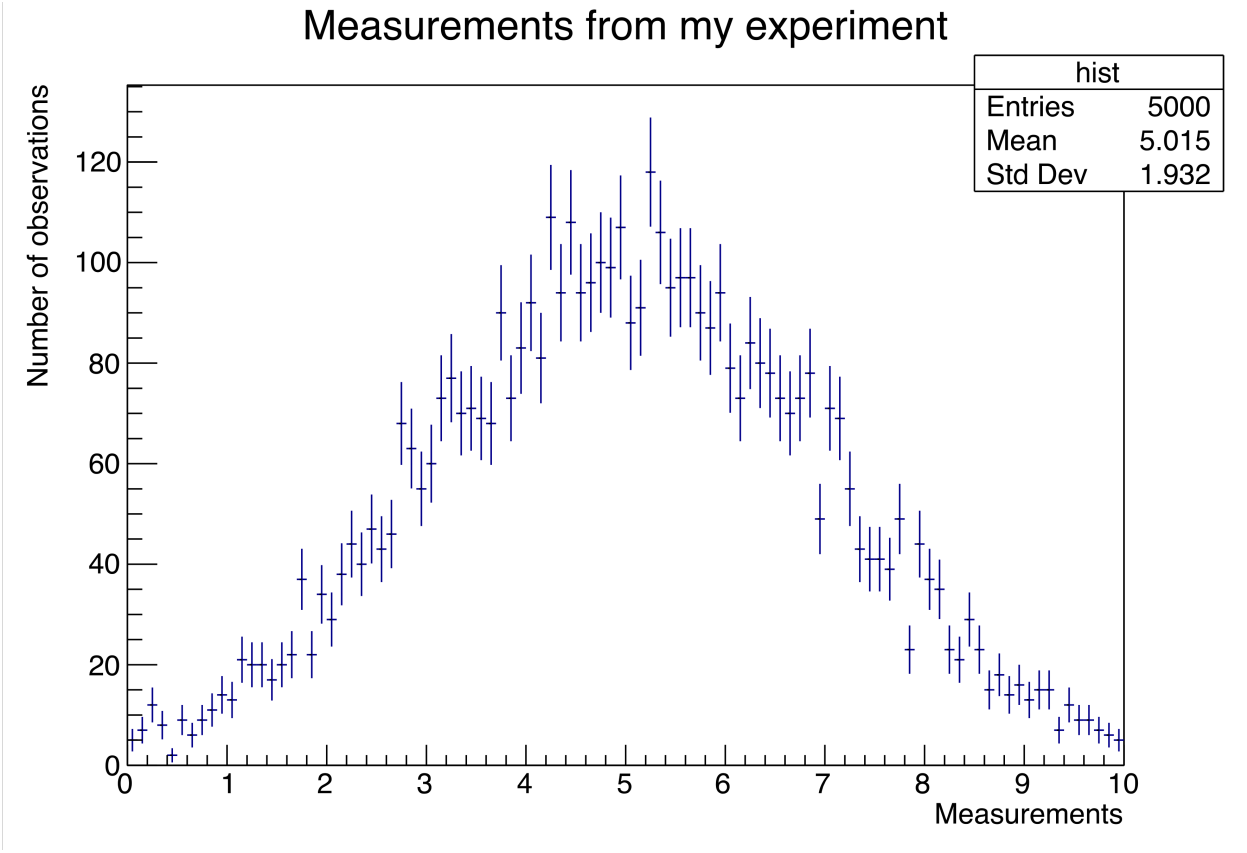

Figure 13.5:: A typical histogram that you might create during a data analysis.

Suppose you want to test whether the above distribution of data was generated according to a Gaussian distribution. The usual test in physics is to perform a "fit": fiddle with the Gaussian's parameters A, μ, and σ (typically called "amplitude", "mean", and "sigma" respectively) until the function best fits those data points.

The mathematical method for performing such a fit starts with computing a quantity denoted as χ^2, pronounced "chi-squared."[2] For a general 1-dimensional function with parameters $p_0, p_1, p_2 \ldots p_j$ to be fitted to a 1-dimensional histogram, this is:

$$\chi^2 = \sum_i \frac{\left(y_i - f\left(x_i; p_j\right)\right)^2}{e_i^2} \tag{13.4}$$

where:

- i means the i-th bin of the histogram (more generally, the i-th data point you've gathered).
- y_i means the data in (or the value of) the i-th bin of the histogram.
- e_i means the error in the i-th bin of the histogram (i.e., the size of the error bars).
- $f\left(x_i; p_j\right)$ means to compute the value of the function at x_i (the value on the x-axis of the center of bin i) given some assumed values of the parameters $p_0, p_1, p_2 \ldots p_j$.

The process of "fitting" means to test different values of the parameters until you find those that minimize the value of χ^2.

[2] That's "chi" with the *ch* pronounced like a *k*; the word rhymes with sky. If you pronounce it "chee" with a soft *ch*, folks at Nevis will think you're talking about a scientist named Qi.

This probably sounds quite involved. Indeed, it can be. Fortunately, physicists and mathematicians have developed many tools to relieve the computational tedium of much of this process. As you will learn in *The Basics* (if you jumped here) or learned in *The Basics* (if you're reading this tutorial from beginning to end), fitting the histogram in Figure 13.5 to a Gaussian distribution is a matter of a couple of mouse clicks:

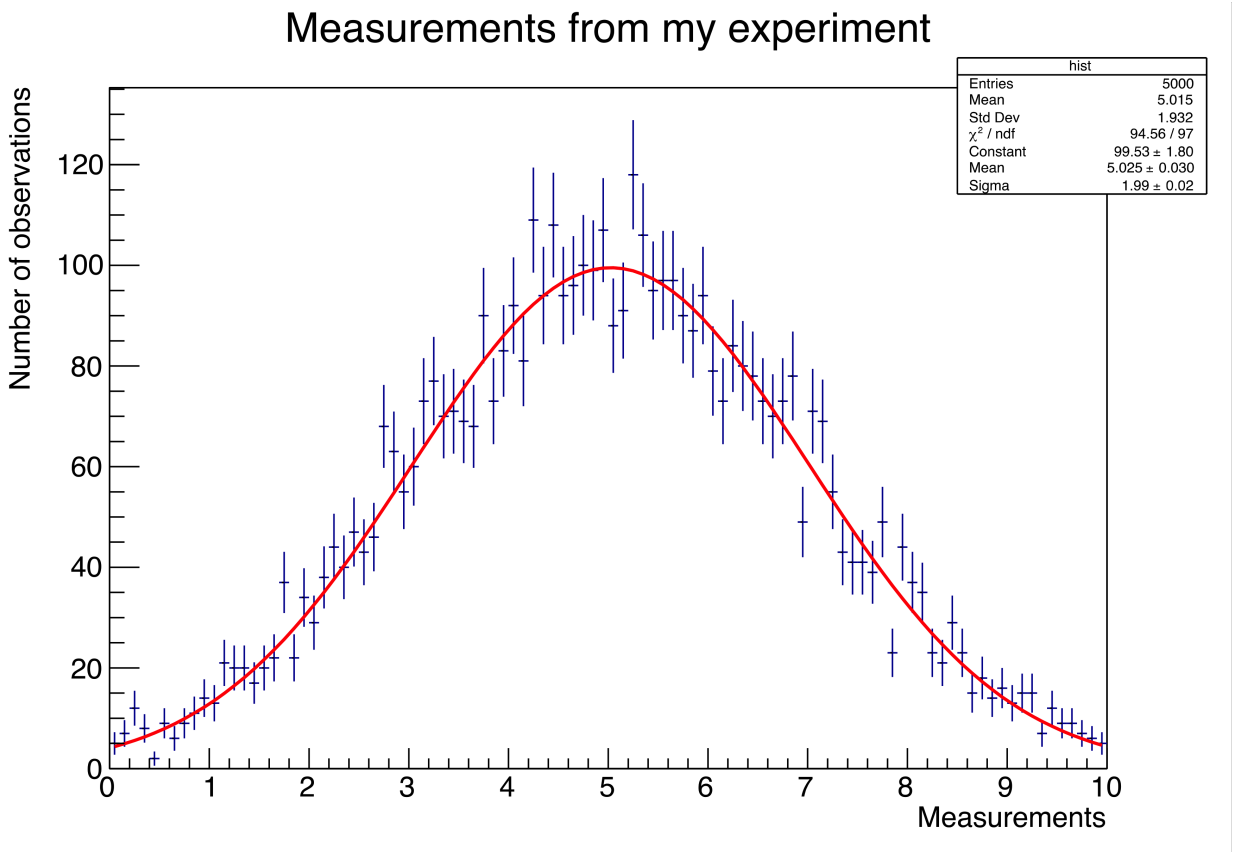

Figure 13.6:: It looks like a Gaussian function is a reasonable assumption for the underlying distribution of our data. In this case, I knew this to be true because of how I generated the "data" in the histogram. You (will do)/(have done)/(will have done) the same thing in The Basics.

Since you're a scientist, you may ask "How does ROOT fit functions?" The answer has to do with a general mathematical technique called "function minimization"; in this case, minimizing the value of χ^2. With interactive ROOT, you can fit 1-, 2-, and 3-dimensional histograms to basic distributions with a few mouse clicks. For more complex minimization tasks, you have to write your own functions to be minimized; there's a brief example of this in *The Basics*.[3]

I should mention that you don't always fit a function to your data using a χ^2calculation. That's good enough if the distribution of your data *within each individual bin* is Gaussian; it's reasonably true for the fits associated with this tutorial.

However, in physics you can have data vs. function comparisons for which the underlying data *within a bin* is not Gaussian. This frequently occurs with fits that include *Systematic errors*.

For another example, look at Figure 13.6, and consider the bins on the far left- and right-hand sides of the plot. The error bars are simply the square root of the number of events in the bin, which is a reasonable approximation for a

[3] If you've studied machine learning, some of this will sound familiar; e.g., here the "cost function" is χ^2. The procedure is still to follow deepest descent of the gradient of a function to locate a minimum. However, the nitty-gritty details aren't the same because a χ^2 fit is a different problem; I wouldn't use Tensorflow to fit a Gaussian to experimental data!

Gaussian. But the number of events in those left/right bins is less than ten, which you know from Figure 13.3 should be represented by a Poisson distribution.[4]

For these cases you may need to use a "log-likelihood" test. I won't discuss this further[5,6] but it's important that you know it exists.

For more on this topic, I prepared a Jupyter notebook on how to use a general-purpose function minimization program called Minuit from within Python and apply it to histograms I created using ROOT.[7] If you're interested in this (or writing minimization routines), you can view a static version of my Minuit notebook, or you can copy the file:

```
cp ~seligman/root-class/minuit-class.ipynb $PWD
```

Chi-squared per degree of freedom

Let's suppose your supervisor asks you to perform a fit on some data. They may ask you about the chi-squared of that fit. However, that's short-hand; what they really want to know is the chi-squared per the number of degrees of freedom.

Take a careful look at the text box in the upper right-hand corner of Figure 13.6. One of the lines in that box is "χ^2 / ndf". You've already figured that it's short for "chi-squared per the number of degrees of freedom" but what does that actually mean?

Take another look at Equation (13.4) for the Gaussian function:

$$\chi^2 = \sum_i \frac{(y_i - f(x; A, \mu, \sigma))^2}{e_i^2}$$

[4] Remember that footnote back in *Working with Histograms*? Neither do I. Wait… I think it was the one that talked about have varying bin widths in a histogram. Now you know why that can be a good idea: so that all of the bins have enough events that they can reasonably approximated by a Gaussian distribution, and a χ^2 fit would be appropriate.

[5] …because I don't fully understand it myself. Maybe, at the end of your research work at Nevis, you'll teach me!

[6] There's a package in ROOT called RooFit that easily performs log-likelihood fits. Another important feature is that RooFit can perform fits on a set of individual data points, instead of requiring that the data be binned in a histogram.

[7] If you don't know what a "Jupyter notebook" is, it means you haven't yet reached *The Notebook Server* in this tutorial.

Figure 13.7:: https://xkcd.com/2605/ by Randall Munroe. The χ^2 calculation is not a Taylor series! But given enough data points...

As I mentioned before, within each individual bin i the data forms a little Gaussian distribution of its own with a mean of y_i. The e_i acts as a scale of the difference between y_i and the function $f(x)$. So if $f(x)$ is a reasonable approximation to y_i, $\frac{(y_i - f(x))}{e_i}$ will be around ± 1. You add up those "1"s for each of the bins, and you might anticipate that χ^2 will be roughly equal to i, the number of bins.

That doesn't tell the whole story. There are three "free parameters" in the fit: A, μ, σ. They're going to be varied to make the chi-squared smaller. The net effect is that total number of "degrees of freedom" for minimizing the χ^2 is:

DOF = number of data points – number of free parameters in the function

The histogram in Figure 13.6 has 100 bins, and a Gaussian function has three parameters. If you look at the figure again, you'll see χ^2 / ndf = 94.56 / 97. It looks like ROOT knows how to count the degrees of freedom, at least for simple functions and histogram fits. If you're using a more sophisticated fitting program like Minuit, you may have to figure out the DOF yourself.

So far so good, but would it better if the χ^2 were even lower? No, it wouldn't!

Statisticians have studied the question: What is the probability that a randomly-generated set of data came from a particular underlying distribution? If you look around the web, you'll find tables that compute this probability for a given χ^2 and ndf. When you get to more than a couple dozen ndf, there's a simpler test: see if the ratio χ^2 / ndf is around 1. For the particular fit in Figure 13.6, the value of χ^2 / ndf = 94.56 / 97 is reasonably close to 1; your supervisor would probably accept it.

What might cause χ^2 / ndf to be much greater than 1?

- There's something wrong in the routine that's calculating χ^2, either in the code or the underlying model. That

probably won't happen if you're fitting histograms in ROOT with simple functions, but I can't tell you the number of days I've spent sweating over a chi-square calculation that I wrote.[1]

- The model that's being assumed for the function does not have enough freedom or has the wrong form to fit the data. This does *not* mean you can just throw additional parameters into the function for the sake of improving the χ^2![2]
- The error bars for your data are too small; in other words, there are additional sources of error (possibly *systematic error*) which you have not yet included.
- Function-minimization programs can get "stuck" in a local minimum that's not the actual true minimum; there's an example of that in *The Basics* (and in that Minuit notebook I described above).

What might cause χ^2 / ndf to be much less than 1?

- Again, something wrong in the χ^2 calculation.
- Too many free parameters in the function you're using to fit. For an extreme example, consider what would happen if we tried to fit Figure 13.6 with a 100-degree polynomial. Of course, that function would be able to go through the middle of every y_i in the plot and the resulting χ^2 would be close to zero.[3]
- The error bars for your data are too large. This can happen if you're not careful in how you add up your errors; if you simply add all your errors in quadrature $\left(\sqrt{\sigma_0^2 + \sigma_1^2 + \sigma_2^2 + \ldots}\right)$ you may have overlooked that some of your errors are correlated; i.e., there are terms $\sigma_i \sigma_j$ where $i \neq j$. If that's the case, you have to calculate χ^2 with an extension of Equation (13.4) using a covariance matrix.
- Someone has gone wrong in your data-analysis process and you're "tuning" the data to the model you want to fit.[4]

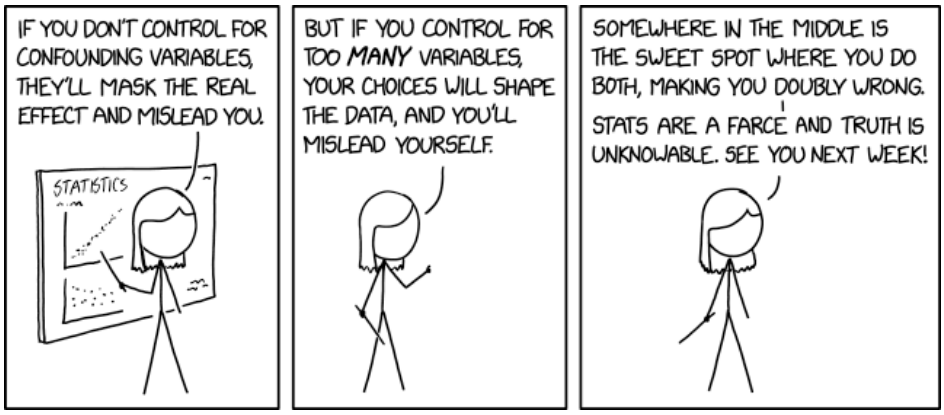

Figure 13.8:: https://xkcd.com/2560/ by Randall Munroe

[1] ...or worse, someone else wrote, then graduated quickly before anyone could ask troublesome questions.

[2] In case you think I'm joking: This actually happened during the analysis effort of my thesis experiment. No, I wasn't the one who did it. The issue remained unnoticed for a few years before I uncovered the problem; fortunately, we hadn't published any papers based on that erroneous fit.

[3] See Figure 3.9 for an example.

[4] This can happen accidentally, but there are times it's deliberate. Five decades after the death of the monk Gregor Mendel, a statistical analysis of his results showed that he was faking his data to agree with his model of genetic inheritance. As it turned out, his model was correct. That makes him the luckiest fraud in the history of science.

"We're looking for a five-sigma effect"

Suppose after a long and detailed physics analysis, you finally have a result. Assume it's in the form of a measurement and its associated error.[1] Let's further assume that there's a null hypothesis associated with this measurement: If a particular property did not exist, the measurement would have been different.

Figure 13.9:: https://xkcd.com/892/ by Randall Munroe

When physicists consider such a measurement, they ask the question: What is the probability is that their measurement is consistent with the null hypothesis; that is, what are the chances that the null hypothesis is correct, and that their measurement is just a random statistical fluctuation?

Let's take another look at our friend, the Gaussian function. As a probability distribution, it indicates the likelihood of a particular measurement being different from the actual underlying value.

[1] As you'll learn in the section on *Systematic errors*, the errors are the tough part.

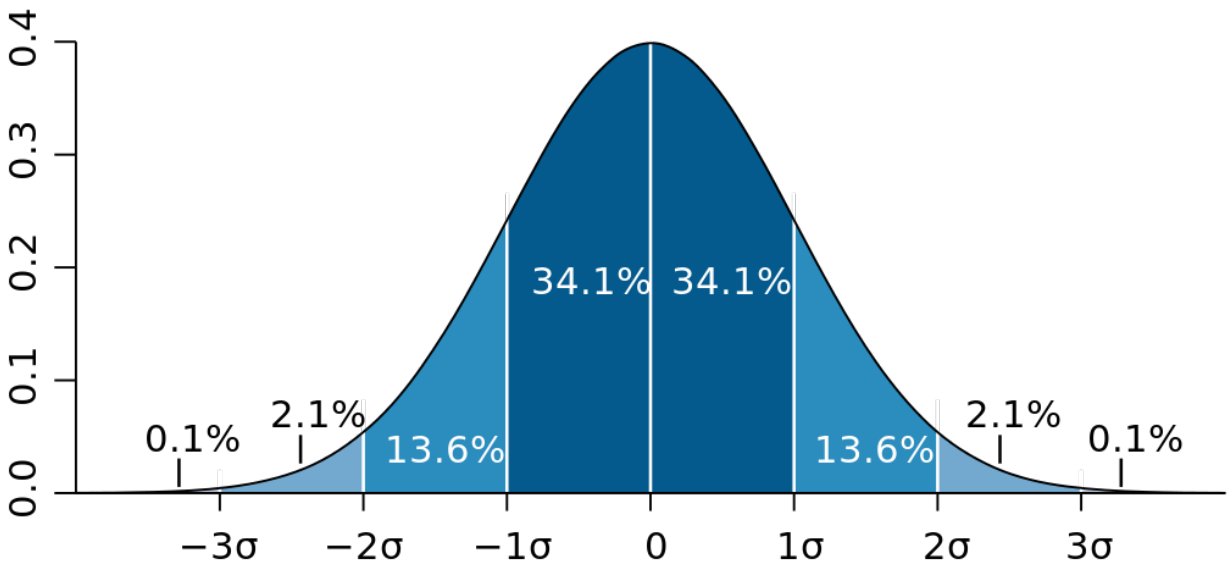

Figure 13.10:: A plot of the normal distribution where each band has a width of 1 standard deviation. Source: M. W. Toews, under the CC-by-4.0 license

Think about this in the context of the following cartoon. The scientists perform 20 tests each with a significance at the 5% level. You'd expect that one of the tests would randomly be far enough from the "mean of the normal distribution" that you'd get an anomalous result. Physicists often can't perform multiple measurements of a given quantity,[2] so they look at their results in a different way.

[2] The Large Hadron Collider at CERN costs about $9 billion. It's obvious that they should build 100 similar nine-mile-wide particle colliders so we can make multiple independent measurements of the Higgs boson mass. Why they haven't is completely beyond me.

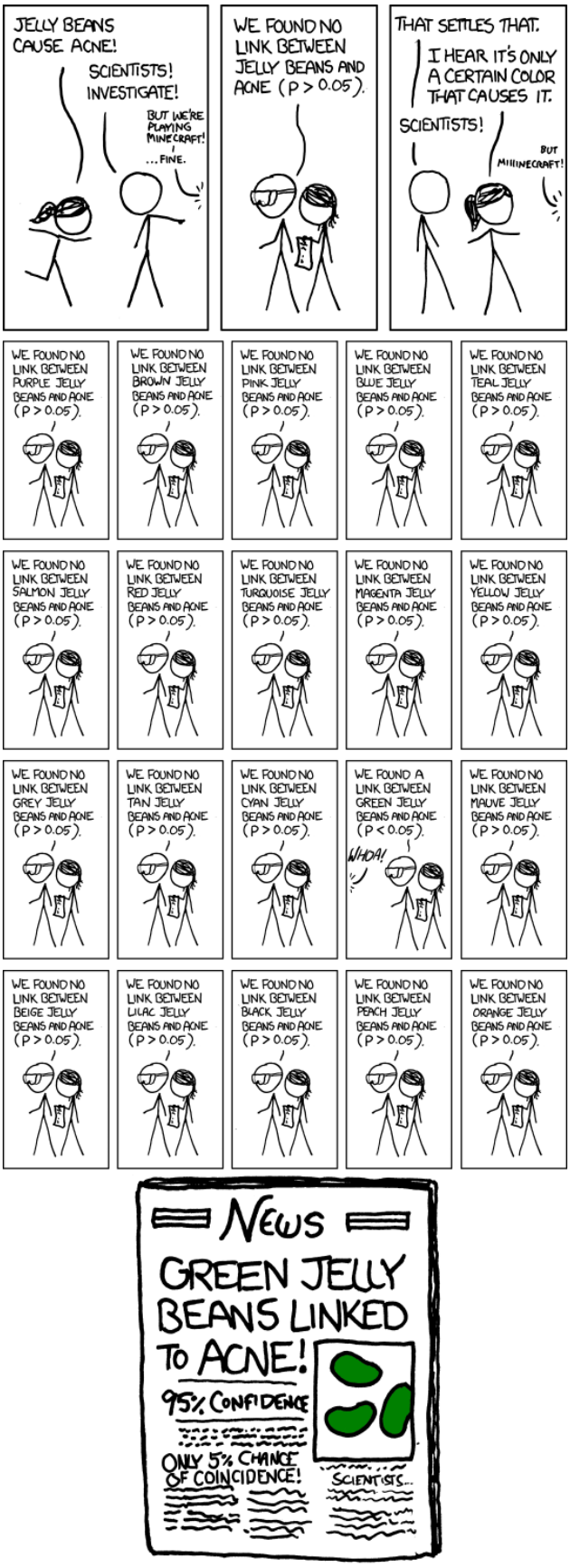

Figure 13.11:: https://xkcd.com/882/ by Randall Munroe

When physicists express a measurement versus a null hypothesis, they usually state it in terms of "sigma" or the number of standard deviations that it is from that hypothesis. This is how I think of it, even though they don't usually show it in this way:

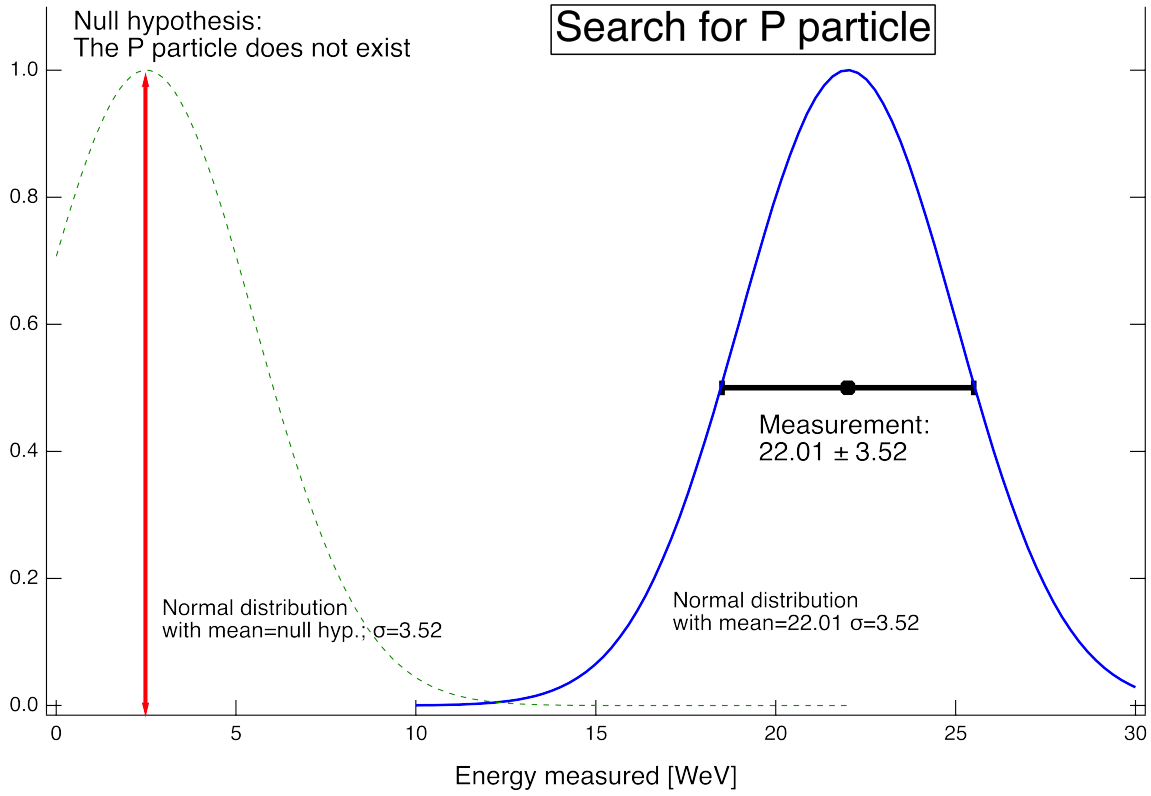

Figure 13.12:: What I picture when a physicist says they've observed an effect at more than a 5σ level of significance.

In this imaginary experiment, the result is reported at 22.01 $\pm$ 3.52 *WeV*. I picture a normal distribution with a mean of 22.01 and σ=3.52 superimposed on the measurement. Then I count off the number of sigmas between the mean and the null hypothesis. In this particular imaginary experiment, the difference is more than 5σ and therefore refutes the null hypothesis. Hence the P particle exists!

In other words, I imagine an assumption that the same normal distribution applies to the null hypothesis (the dashed curve) and ask if what we actually observed could be within 5σ of the mean of the null hypothesis.

At this point, you may have compared Figure 13.10 and Figure 13.12, including the captions, and thought, "Wait a second. A 3σ effect would mean that the odds that the null hypothesis was correct would be something like 0.1%, right? In fact, I just looked it up, and the exact value is closer to 0.3%. Isn't that good enough? It's much better than that xkcd cartoon. Why do physicists insist on a 5σ effect, which is around 0.00003% or one in 3.5 million?"

A 3σ effect is indeed only considered "evidence," while a 5σ effect is necessary for a "discovery." The reason why is summarized in a quote from a colleague on my thesis experiment: "We see 3σ effects go away all the time."

The reason why 3σ effects "go away" is a deeper study of the data and its analysis procedure. One potential cause of such a shift is a change in systematic bias as you work to understand your *Systematic errors*.

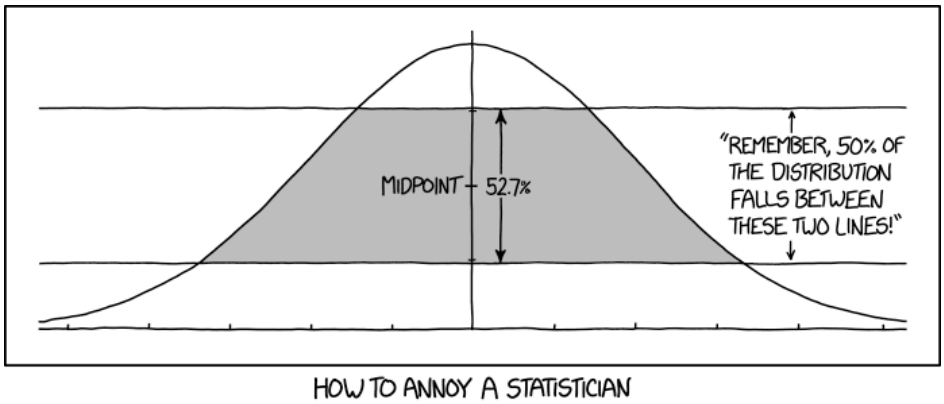

Figure 13.13:: https://xkcd.com/2118/ by Randall Munroe

Systematic errors

A *statistical* error is one that's due to some inherit randomness in your process of making a measurement. A *systematic* error comes from a consistent bias in that measurement, but you don't know how much that bias is. The systematic error is the limit you assign to the potential range of that bias.

To explain this concept, I like to start with that old statistics example: measuring the size of a table with a ruler. You repeat this measurement every day. There is some variance in the day-to-day measurement: you tilt your head differently, the light in the room depends on time of day, you're feeling tired that day, etc.

There's a reasonable chance that if you were to plot these measurements, the result would look like a Gaussian distribution. The standard deviation of that distribution would be related to the *statistical* error in your measurement.

To understand the *systematic* error in the measurement, you have to ask: How do we know that 20*cm* as measured on your ruler is the same as 20*cm* as measured on mine? Or 20*cm* as measured by the Physics Department of Polytechnic Prep in Birnin Zana? Or the International Committee for Weights and Measures in Saint-Cloud, Hauts-de-Seine, France?

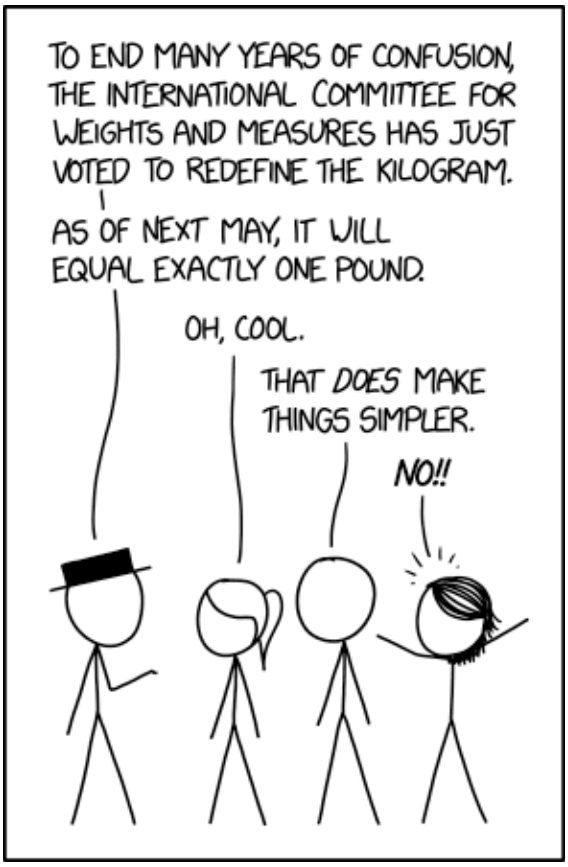

Figure 13.14:: https://xkcd.com/2073/ by Randall Munroe

I'll give you a hypothetical chain of reasoning along the lines that a physicist (or a metrologist) might use to think about systematic errors. Assume that your ruler is similar to the one sitting next to my desk right now, a cheap one I purchased at a drug store 30 years ago.

- The ruler is made of plastic. I assume liquid plastic was poured into a mold then allowed to harden. What are the thermal characteristics of this particular type of plastic? Does it shrink when it's cooled? Does its shape distort when it gets hot in my apartment? Has it become warped over the past 30 years due to the age of the plastic or the conditions in which I've stored it?
- If a metal mold was used to shape the plastic, does it have thermal characteristics of its own? It might have been shaped at room temperature, yet plastic is poured into it at some higher temperature. Is this temperature variation enough to distort the mold to some degree?
- How was that mold made? Did it start out as a block and then was shaped at a tool-and-die factory? What was the precision of the drill, mill, or press used to create that mold?
- Who manufactured that drill, mill, or press? How accurate was the tool that made it?

And so on.

Your probable reaction to the above list is that all these effects are too small to worry about for an actual 30*cm* plastic ruler being used to measure a typical living-room table. Let's consider a more realistic scenario: the imaginary experiment mentioned in Figure 13.12, the discovery of the P particle.

For the purposes of this example, the P particle is hypothesized to be emitted by a rare decay of Vb299. The energies of the decay products of Vb299 are measured with a calorimeter. The detector setup is located under the Jabari mountains, but even so enough cosmic rays get through to be a substantial background for the rare signal they're trying to detect.

The calorimeter measures the energy of the particles and returns some value in millivolts. You have to calibrate the calorimeter, to translate those millivolts into *WeV*. The typical way to do this is to shoot a beam of particles of known energy at the calorimeter, and see how many *WeV* corresponds to the calorimeter output in millivolts.

- A calorimeter has some energy resolution. Even if you shoot a beam of known energy into one, you're going to see a spread in the resulting detector response. Perhaps that distribution will look like a Gaussian, but you'll still have to fit it. Take another look at Figure 13.6. There's a fitting error associated with the mean and sigma of the distribution. The width of that distribution is your energy resolution; the error in the mean is a systematic error of your energy calibration.[1,2]
- What is the exact energy of that beam of electrons used to calibrate the calorimeter? The electrons might be extracted from an ARC reactor and sent through a chain of focusing and steering magnets. The final step is to point the calibration beam at the calorimeter with a bending magnet to select those electrons with a given energy. The mean energy of the beam will depend on the magnetic field of the final bending magnet. How well do you know that magnetic field? That will be another source of systematic error.
- You have to separate the energy signatures of the P particle from those of the cosmic rays that pass through the calorimeter. How well can you identify the event type? You'll apply various analysis cuts (there are examples of this in *The C++ Path* and *The Python Path*), but there's always a limit to their efficiency, for another source of systematic error.
- The above were sources of *experimental* systematic error. Now let's consider a *theoretical* systematic error: Both Dr. Shuri Wright and Dr. William Ginter Riva have published models of the predicted energy spectrum from Vb299 decays involving the P particle. The separation of your signal from the cosmic-ray background depends on the model. You must perform your analysis with both models and treat the difference as a theoretical systematic error.

You may feel that these examples are as unimportant as the systematic errors I hypothesized for the ruler,[3] but they were adapted from cases within experiments I've worked on; the relative sizes of such errors are much larger than the errors in a plastic ruler due to a milling machine. If anything, I've underestimated the number of systematic errors considered in a typical physics experiment.

In case my fictional example left you dubious about the concept of systematic errors, here's a systematic error table from a real physics analysis. Note how the total error at the bottom is dominated by the systematic errors over the statistical errors. In particular, the largest systematic error is "ISR and FSR" (Initial State Radiation and Final State Radiation) which is a theoretical systematic error.[4]

[1] For a χ^2 fit, the uncertainty in a parameter comes from shifting that parameter and looking at its change about the minimum when χ^2 varies by ± 1. I don't expect you absorb that bit of arcane trivia right now; it's enough to know that any fits to points with error bars will necessarily have error estimates in the fit results.

By the way, this is the answer to the *statistics question* I posed back in *Fitting a Histogram*.

[2] The error in the mean from fitting to the detector response is usually reported as the "energy calibration." The standard deviation of that fit is the "energy resolution." You'll usually see these two reported separately (as in Figure 13.15).

On my thesis experiment, it took us years to understand both the energy calibration and the energy resolution, and their correlation with each other. In part this is because they're also a function of energy; e.g., the energy resolution is often reported using a formula like $\sigma(E)/E = K/\sqrt{E}$, where K has to be determined by the analysis.

[3] You might be justified in this impression given the obscure pop-culture references. If you didn't get the references, do a web search on Birnin Zana, then ask yourself which element is Vb299 and what *WeV* stands for.

[4] Let's add up those individual errors. Wait… the total is 7.27, not 2.27! What's happening?

The answer is that the errors are being added in quadrature. If a given error is δ_i, then adding them in quadrature means to compute $\left(\sum_i \delta_i^2\right)^{1/2}$.

But that computation assumes that none of the systematic errors are correlated; i.e., there are no $\delta_i \delta_j$ terms with $i \neq j$. Is that necessarily true? For example, in Figure 13.15, what if the "Jet energy scale" was correlated with the "Jet energy resolution"?

At this point, you may be coming to understand the complexity of handling errors in a physics analysis.

Uncertainty	Δm_{top} [GeV]
Statistics	0.94
Method calibration	0.40
Signal MC generator	0.62
Single-top Wt generator	0.28
Hadronisation and parton shower	0.55
ISR and FSR	1.39
Underlying Event	0.67
Colour Reconnection	0.23
Parton distribution function	0.42
Single-top contribution	0.10
Leptons	0.50
$E_{\text{T}}^{\text{miss}}$	0.12
b-tagging	0.08
Jet energy scale	0.60
Jet energy resolution	0.32
Jet vertex fraction	0.05
Total	2.27

Figure 13.15:: From ATLAS PUB Note ATL-PHYS-PUB-2018-001 31st January 2018 Investigation of systematic uncertainties on the measurement of the top-quark mass using lepton transverse momenta

I did a lot of hand-waving to condense what little I know about statistics into these pages without (I hope) getting too bogged down in the math. If you'd like more rigorous explanations of these concepts, see my list of statistics books.

Anyone can make a measurement. Understanding the error on that measurement is the true skill of a physicist.

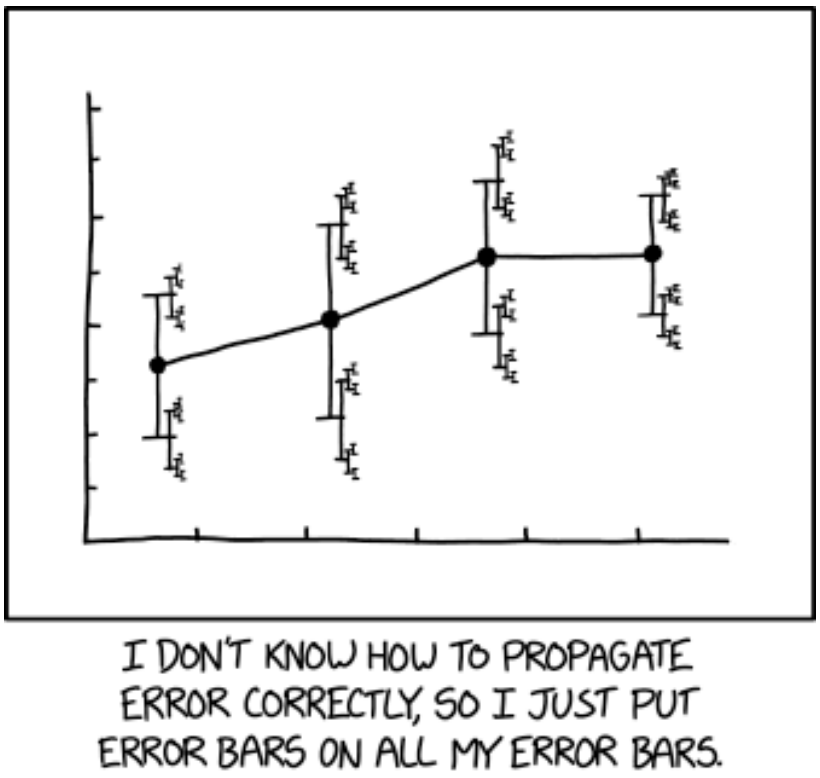

Figure 13.16:: https://xkcd.com/2110/ by Randall Munroe

Note: Don't laugh too quickly. Adding statistical and systematic errors can be a tricky business. Often an experiment will report them separately, and sometimes will plot them in a similar way as this cartoon.

Programming Tips

Over the past few years, I've asked the faculty at Nevis, "What non-ROOT topics would you like me to cover in my computing tutorial?" In 2021, I got the response, "Teach them how to make their code faster."

That's a huge topic. It's not quite big enough for me to make the "lifetime" joke again, but it's hard to decide what exactly to teach.

The topics below are a scattering of tidbits that I've picked up over the years. I hope you find at least some of them useful.

Bad Code

If my task is to help you "speed up code", then I feel that the first part is to help you maintain it.

This is one of the 10/90 principles of programming: 10% of the time will be spent writing a program; 90% of the time will be spent maintaining it.[1]

Let's start by looking at a code fragment. For the sake of argument, assume that it "works," in the sense that it's part of a larger program that produces the results that you expect.

Pick the Python version or the C++ version. Can you figure out what, in my opinion, is wrong with this code?

Listing 13.1: What's wrong with this Python code?

```
# Compute the vector product of the two velocities
i = 10
while j < i+1:
    k = i + 3
    s[j] = s[j] + k*j
    j = j + 1
```

Listing 13.2: What's wrong with this C++ code?

```
// Compute the vector product of the two velocities
int i = 10;
while (j < i+1) {
    int k = i + 3
    s[j] = s[j] + k*j
    j = j + 1
}
```

Hint: In my opinion:

- There are at two execution-speed problems with this code (though they affect the Python version more than the C++ version). There is also a separate speed problem with only the Python code.
- There are also at least two programming-style problems with the above code.

My answers are in the next section.

Is this worth it?

[1] The other 10/90 principle is that 10% of the code is responsible for 90% of a program's execution time. We'll deal with creating faster code in the next section.

It's natural to ask this question. We're all busy. If the code works, why make a fuss about it?

I offer the following in response:

- If code is unclear, it will be harder to maintain. You may feel that's the next person's problem. Except that the next person is likely to be you!

  Suppose you've have a program that performs a cluster-finding algorithm. It works, but it's quickly and sloppily written. You save it and move on to your next project.

  Six months later, your team comes back to you: "You're the cluster-finding expert. Update your code so that it accommodates this revision." Of course, you've forgotten about your old code.[2] You look at it now... and realize that five seconds worth of effort back then would have saved you hours or days re-analyzing your old code to figure out what it did.

- Typically, physics groups don't do code review. But professional programming environments do.

  A "code review" is when a group of programmers looks over your code to see if it's acceptable to add to a project, in much the same way a physics group will review a paper to see if it's acceptable for publication. If you find yourself in a programming group, they're not going to accept your code if has the kinds of flaws I discuss here.

- When looking at code, physicists tend to treat it as a "black box":[3] They assume it just works, and use it as-is without thinking or even understanding what it does.

  Maybe this sounds fine, but put it in the context of quickly-written, sloppy, or slow code. In 2021, I wrote a quick-and-dirty bit of analysis code to answer a single question. In 2022, students copied that code, treated it as "perfect"[4] and used it in a loop to analyze 150 times more events than I did in my brief study. Result: Their analysis code ran slowly, because I hadn't put in the time to write proper code.

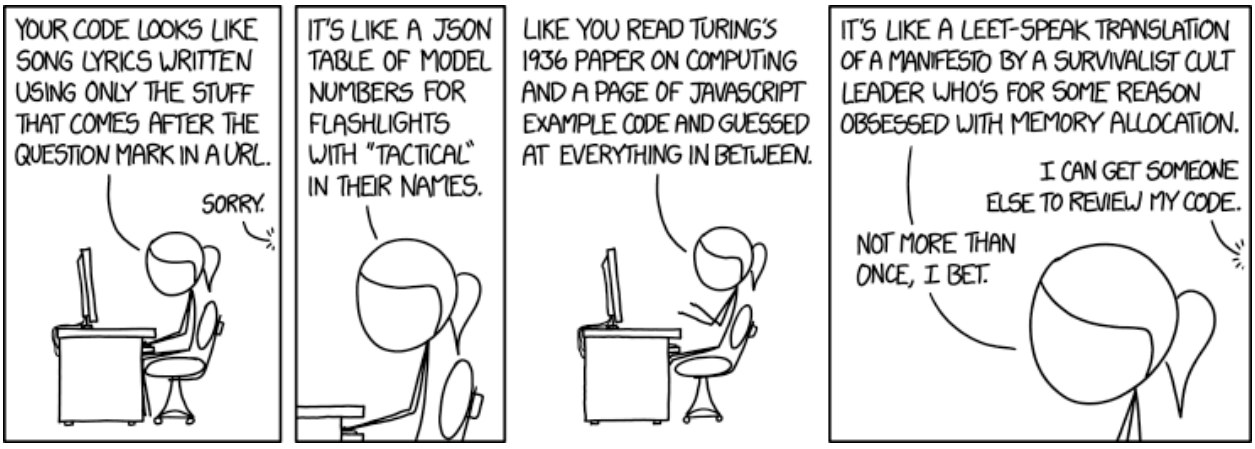

Figure 13.17:: https://xkcd.com/1833/ by Randall Munroe

[2] I know you think you'll remember. You won't. Although arguments from authority are always suspect, please take it from someone who's been programming for over five decades.

[3] In this discussion, when I say "physicists like to put things in a black box," it's not exclusive. Other fields of study also tend to put things in a black box. However, since I mostly hang out with physicists and not pharmacists or airline pilots, I'll only speak for the folks I know.

[4] For some reason they thought that, because I had written it, my code was superior to anything they could write. I can't begin to tell you how false that reasoning is.

Faster Code

Note: I won't claim that my suggestions will make the code perfect, merely that they will make the code better (at least in my biased opinion).

Take another look at the code from the previous section. To keep things simple, let's just consider the Python version:

```
# Compute the vector product of the two velocities
i = 10
while j < i+1:
    k = i + 3
    s[j] = s[j] + k*j
    j = j + 1
```

I'll start with this line:

```
while j < i+1:
```

What is the value of `i+1`? It's 11, since we have `i = 10` on the line above. Is the value of `i+1` going to change as we go through the loop? No, there's nothing that changes the value of `i` in the loop, so `i+1` will be unchanged.

But Python doesn't "know" that. Python is an interpreted computer language: The Python interpreter looks at each line, one-by-one, converts that line into the native computer machine language, and executes it.[1]

That means that every time the Python interpreter goes through the loop, and interprets the statement `while j < i+1:`, it's re-computing the value of `i+1`. Since `i` never changes (in the loop), `i+1` never changes. Yet Python must go through the additional step of adding 1 to `i` each time.

Therefore, I claim:

Speed problem 1: The loop performs an unnecessary computation of `i+1` for every iteration of the loop.

Now you've probably already guessed:

Speed problem 2: The loop performs an unnecessary computation for `k = i+3` for every iteration of the loop.

Again, since the value of `i+3` never changes, the value of `k` never changes. Yet Python must spend time re-calculating `k`.[2]

Note: Why this fuss over a simple addition? Computers are fast, aren't they? And it's just an addition, not calculating the space-time tensor in the vicinity of a black-hole merger.

But:

- Programs and goals change over time. Right now a loop may be going through 11 entries in a table. Tomorrow it may be going over 10,000 entries.
- In my experience, paying attention to what's actually modified in a loop helps one write code that is easier to read and debug.

Let's revise the Python version with those two points in mind:

[1] For anyone familiar with compilers and interpreters, I'm skipping over a lot of details here. However, the details (as important as they are to the overall science of computing) still support my point… I think.

[2] If you're a C++ programmer, your reaction may be that these two points are meaningless for an *optimizing compiler*. That's a legitimate reaction, but I maintain it's no excuse for sloppy code.

```
# Compute the vector product of the two velocities
i = 10
k = i + 3
m = i + 1
while j < m:
    s[j] = s[j] + k*j
    j = j + 1
```

+=

We can do just a bit better. After you've been programming for a while, you'll notice the following kind of statement is very common:

```
a = a + b
```

To make lines of code a bit shorter, and to allow language interpreters or compilers a chance at improving efficiency, both Python and C++ implement a set of operators:

```
+=, -=, *=, /=, %=
```

The first one is by far the one I've seen most often. The following two statements do the same thing:

```
a = a + b

a += b
```

As do these two statements:

```
value = value + offset

value += offset
```

C++ goes a step further. Since adding (or subtracting) 1 is very common, the language has ++ (and `--`) operators. The following statements do the same thing:

```
a = a + 1;

a += 1;

a++;

++a;
```

There's a difference between `a++` and `++a`, but I'll leave that topic to a more formal C++ course.

Now you know why the language is called C++!

With that in mind, we may get a bit more speed by using += in our code:

```
# Compute the vector product of the two velocities
i = 10
k = i + 3
m = i + 1
while j < m:
```

(continues on next page)

(continued from previous page)

```
    s[j] += k*j
    j += 1
```

Well, our code is faster... but there are still problems with it. We'll look at them in the next section.

Clearer Code

Here's a repeat of the "faster code" from the previous section:

```
# Compute the vector product of the two velocities
i = 10
k = i + 3
m = i + 1
while j < m:
    s[j] += k*j
    j += 1
```

I'm going to start with an "iffy" style issue: From this code, we can't tell what the initial value of `j` is supposed to be.

Since `j` is presumably assigned outside this fragment from a larger program, it may be that the calculation of `j` is complicated.

However, it may also be that the programmer stuck `j=0` somewhere near the top of the program. This leaves us, the code's reviewers, in doubt about the limits of the loop.

So I'll only go as far as to claim:

Style problem 0.5: The initial loop limit may be defined too far from the start of the loop. In general, one would like the loop limits to be defined near the loop statement.

My next style complaint has to do with the single-letter variable names. A single letter may be appropriate for the loop index,[1] but for the other variables a single letter just hides what's going on.

Style problem 1: The variable names tell us nothing about what the variables do.

Style problem 2: The comment has nothing to do with the code.

Warning: I'm about to go off the deep end. What follows is very subjective. But if you're willing to wade through it, you might come out writing better programs.

Program comments are my pet peeve. In my view, the comments are part of the code. When you write the code, write the comments. When you revise the code, you revise the comments.[2]

The comment says "Compute the vector product of the two velocities" but the code has nothing to do with vector products. When you see something like this, it means that someone revised the code, but left the comment unchanged.

This is just as bad as having no comments at all, if not worse. Anyone else looking at the code is going to be confused, since the comment says one thing and the code says another.

[1] Back in the good ol' days, scientists programmed in FORTRAN. As the name implies (FORTRAN = "FORmula TRANslation"), the language was designed to implement mathematical expressions as directly as possible, given the punch-card technology of the time. Back then, FORTRAN variable names could not be more than six letters long (yes, really!), and mathematical notation mainly uses single letters for indexing matrices and such.

That's why, to this day, there is a tendency to use `i`, `j`, `k`, etc. as the loop variables in programs, along with simple repeats like `ii`, `jjj`, `k3`.

Writing comments can be tedious. But revising uncommented or badly-commented code is a nightmare.

One more time:

- Put comments in your code.
- Describe the purpose of the section of code. You don't have to describe the syntax of each line (unless it uses an obscure feature of the language); there are plenty of web sites on programming-language syntax.
- The comments are part of the code. When you revise the code, revise the comments.[3,4]

With all that said, let's take a look at what the revised code might look like:

```
# For each user, add a displacement to the distance array
interval = 10
scale = interval + 3
limit = interval + 1
j = 0 # or previously calculated elsewhere
while j < limit:
   distance[j] += scale*j
   j += 1
```

We're faster and clearer than when we started. In the next section, we'll see if we can use the programming language make things even faster for us.

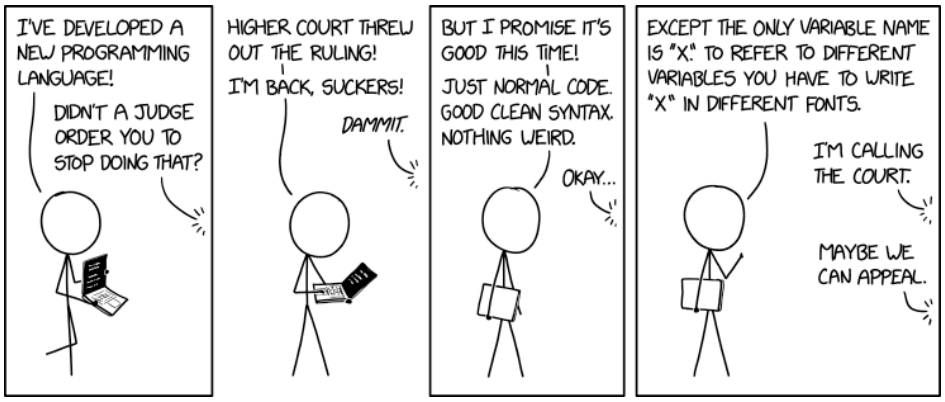

Figure 13.18:: https://xkcd.com/2309/ by Randall Munroe

[2] I know there's a strong impulse to get to it "later." As I note in *one of the expert exercises*, there's no experimental evidence of the existence of "later." I'm an experimentalist, not a theorist, which is why I put comments in my code.

[3] Remember when they made you take an English class in college, even though you're a science major? I imagine you thought that this was to teach you to think critically, to better enjoy literature, to understand the world with nuance and conviction, to appreciate diversity, to develop sensitivity, and to contribute to the future of culture.

Well, yeah, I suppose so. But the real reason was to teach you how to write and revise program comments.

[4] Occasionally you will meet people who claim that there's no need to comment their code, because their choice of variable names is so good that the code is "self-documenting." There are also people who claim to have seen Bigfoot or the Loch Ness Monster. I lump all those claims into the same category.

To put it another way: The goal of writing comments is to document the code's purpose and to save time for the next person who has to work with it. If you're forcing the next person to analyze your variable names and deduce the code's function, you haven't saved anyone any time.

Loops

Note: There's a saying: "In C, loops are your friend. In Python, loops are your enemy."

C++ is a compiled language. In the process of turning C++ code into machine language, the compiler can do many things to automatically optimize your code; for example, it can identify small loops (like the one in our example code fragment) and arrange for it to be executed in a CPU's memory cache.

Python can't do this. It has to interpret each line one-by-one. In a loop, it goes through each line, then "backtracks" to the beginning of the loop and freshly interprets each line again.

There are some tricks to get around this. Mostly they involve using Python to call routines written in C.[1]

As a first step, let's get rid of one "cheat" that I put into our code fragements. The `while` statement has its uses, but they're associated with potentially varying logical conditions like reading a file. If you're incrementing a number by a constant interval, then both Python and C++ have a better way to write a loop.[2]

Listing 13.3: A `for` loop in Python

```
# For each user, add a displacement to the distance array
interval = 10
scale = interval + 3
limit = interval + 1
# Assuming we want j=0:
for j in range(0,limit):
   distance[j] += scale*j
```

Listing 13.4: A `for` loop in C++

```
// For each user, add a displacement to the distance array
int interval = 10;
int scale = interval + 3;
int limit = interval + 1;
// Assuming that we want j=0:
for ( int j=0; j < limit; ++j ) {
   distance[j] += scale*j
}
```

There's not much more we can do for the C++ code,[3] so we'll focus on the Python version of the rest of this section.

I didn't supply a defintion for `distance`. Given the use of `distance[j]`, it could be simple Python array. But if you've used Python before, you're probably screaming at me: "Use numpy!"

The length of the `distance` array isn't specified in the code fragment. In theory, it could be larger than `limit`. Let's be general for the moment, and assume the length of `distance` is greater than `limit`.

[1] At this point in the tutorial, you already know that I'm biased in favor of C++ over Python. So I can't help but be snide and point out that if you're using Python to call C, write not write your code in C in the first place?

The usual reason to prefer Python is its development cycle: You can quickly test small fragments and integrate them into your larger program. But now you know ROOT, which has a C++ interpreter that lets you do the same thing.

With that foolishness off my chest, I will continue to support and inform your use of Python. I've got at least one more lifetime I can spend learning more about the language!

[2] I'm now using a lower limit of `j=0` in the loop, which I did not explictly do when I first introduced the fragments. In the "real" world, if the value of `j` was more complicated, it would not change the loop code by much… except that I would not have continued to use the simple letter `j` for the name of the variable.

[3] Though as we'll see in the next section, there's a way to improve the C++ *compilation*, as opposed to the code.

```
import numpy as np
distances = 20 # or some other value
# Create a numpy array of length 'distances'
distance = np.zeros((distances))
```

Given the artificial nature of the original code fragment, the fastest way to perform this task is to use numpy's array features:

Listing 13.5: The Python code using numpy

```
# For each user, add a displacement to the distance array
interval = 10
scale = interval + 3
limit = interval + 1
j = 0 # or previously calculated elsewhere
distance[j:limit] += scale * np.arange(j,limit)
```

This will give us C-level speed. It's up to you to decide whether this gives us Python-level clarity.[4]

Here's a more detailed tutorial on using numpy to perform faster computations.

[4] Although we've achieved faster code, we haven't necessarily achieved more robust code. What if the value of `limit` is greater than the length of `distance`?

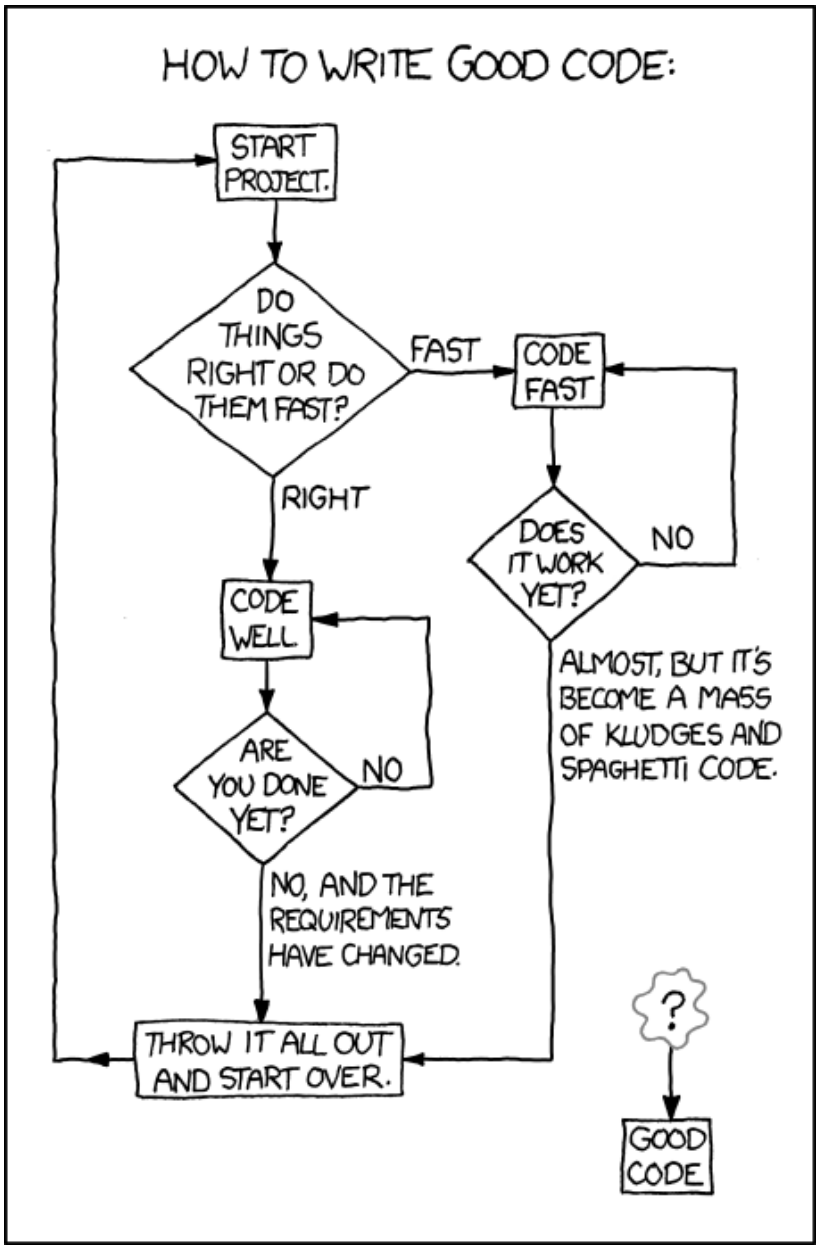

Figure 13.19:: https://xkcd.com/844/ by Randall Munroe

Compiling Your Program

Note: We've come to the end of the programming tips to make your code run faster. The rest of this page offers methods of compiling your code in ways that can increase its speed. Note that I have not personally explored the Python-based options.

Python

If you want the friendliness of the Python language syntax, but the speed of a compiled language, there are compilers for Python.

These are installed on the Nevis particle-physics Linux cluster. If you visit the links below, you can ignore the installation instructions if you're using one of the computers on the cluster, or you're using *The Notebook Server*.

Numba

Numba is a "just-in-time" compiler for Python. Basically, you create some code within a Python decorator that Numba will compile the first time the decorated function is called. After the compilation, the function will execute at C-level speed.

If you went through *The RDataframe Path* to the end, you've already seen a practical example of using Numba.

Cython

Cython is both a compiler and a language extension for Python.

In general, you'll want to use Cython from the command line, as you would C++. But you can use it on our notebook server. The Nevis wiki Jupyter page has an example.

C++

On the Nevis particle-physics Linux cluster, the GNU C++ compiler is standard. Its major competitor is clang.

By default, both compilers produce code that is optimized for debugging. But once you're convinced that your code works, you may want to get some extra speed by setting the `-O3` option, which optimizes the code for execution speed.[1]

For example:[2]

```
g++ `root-config --cflags --libs` analysis.cc -o analysis -O3
```

[1] Why should optimizing the code for speed make it harder to debug? Take another look at Listing 13.2, the example of "bad" C++ code. I *described* how the code computed values more than once, even though their values were unchanged in the loop. An optimizing compiler would pull out any code that was independent of the loop, and execute it before the loop started.

However, it would do this on the machine-language level (or on the LLVM level, a topic way beyond the scope of both this tutorial and this footnote). There would no longer be a correpondence between a bit of code that the computer executed and a given line in your program. This means that any error messages generated during the execution of your program would be harder to track down; an error message could not include text like "segmentation fault in line 17".

[2] Did I just give away an answer to a question I posed back in *Exercise 11*? Oops!

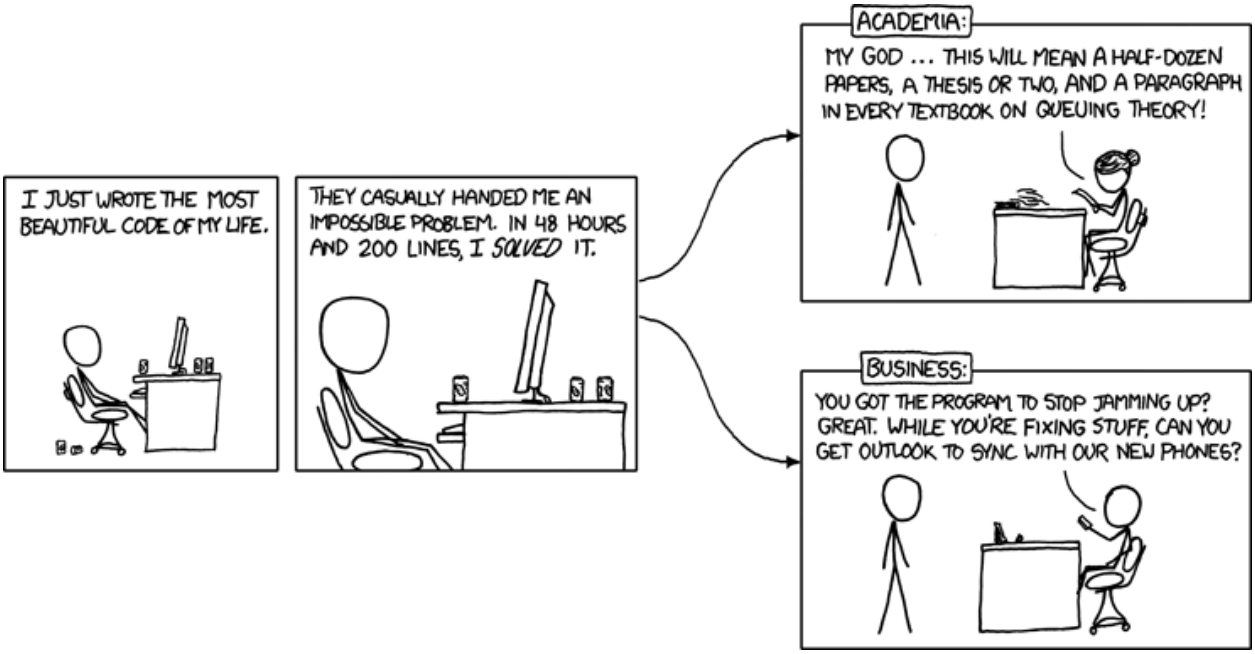

Figure 13.20:: https://xkcd.com/664/ by Randall Munroe

Batch Systems

All of the Nevis particle-physics research groups have to work with "batch systems" as part of their analysis chain. You probably won't be asked to use a batch system as part of a summer project.

However, if you continue your work beyond that, and/or join a larger project, almost certainly there will be a batch system in your future. This section will help you understand what the term "batch system" means, and give you some insight into the planning you'll need to do in order to use one.

Why batch?

For the sake of the discussion on this page and the next several pages, assume you have a program. The program has at least one input file and at least one output file.[1]

You can run this program on your own computer, like so:

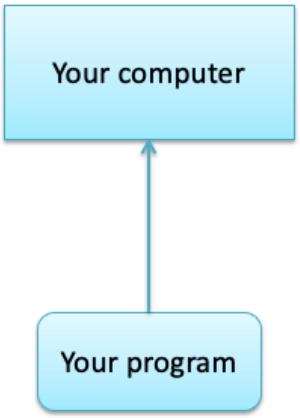

Figure 13.21:: Let's start from here: You've got a program. You run it on your computer.

Eventually, there's one thing that every program needs to have: more. In physics, the typical need is to process more data, which may in turn require more memory, larger disk files, more calculations, etc.

For now, let's consider the need to process more data in the same amount of time. For example, if your program has to process ten times more events to generate a histogram, it's going to take longer to run.

Why not just let the program run longer? Part of the answer is that a longer program becomes a more vulnerable program. I've seen programs that take weeks to run.[2] What if something goes wrong during that time; e.g., a power outage on the computer; you close your laptop's cover; someone else runs a job and crashes the computer. Another reason is impatience: Why wait days to run a program when there are methods to split up the calculations and run the process in minutes?

One way to approach this issue is to note that computers now come with more processing cores. It's a rare laptop that doesn't have at least four cores; the login systems at Nevis have at least 16 cores each (if not more); many systems on modern server farms have hundreds of cores.

What you can do is to re-write your program so it can take advantage of multiple cores, with each core representing one execution thread. Your program might then be viewed like this:

[1] If you've gone through the ROOT tutorial, *of course* you have a program!
[2] If you think I'm joking, ask an astrophysicist or someone working with the GENIE neutrino simulation about "spline files".

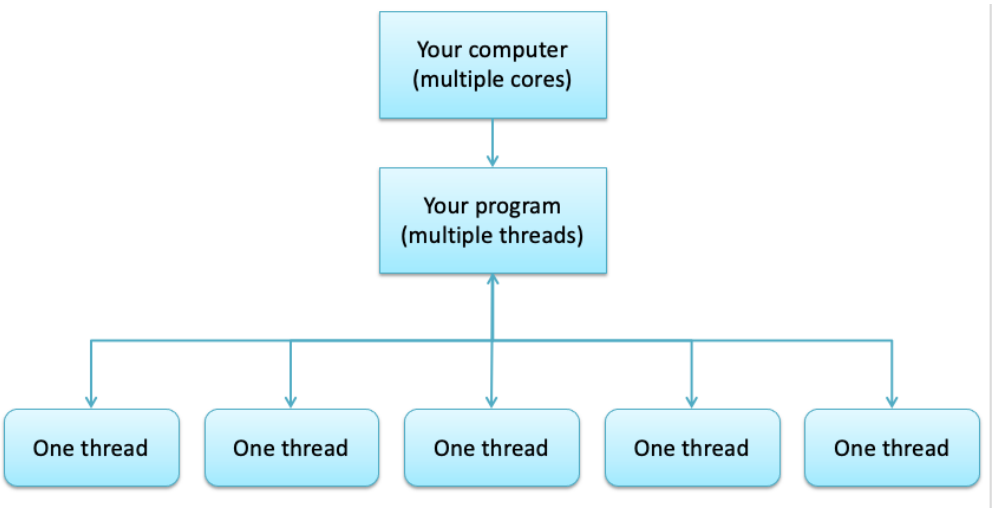

Figure 13.22:: Multiple programming threads on a single computer.

However, writing a multi-threaded program is not easy. You have to consider whether your code is thread-safe, as opposed to making uncoordinated accesses to the same region of computer memory.[3]

Apart from that, the multi-thread approach works well if you're satisfied with the number of cores on your computer. What you if you want more?[4,5]

Let's consider a different method: running your program more than once simultaneously on the same computer.

[3] "Hey, don't just say thread-safety is hard! Show it!" Okay, whatever you say.

Assume that your program is going to count something, as you did when you learned about applying a cut. That means that each of your threads might execute a line such as:

```
a = a + 1
```

Remember, each one of your execution threads is running simulteously and asynchronously (meaning the threads aren't "talking" to each other).

Now consider what happens when you execute `a = a + 1`. The computer takes the value stored inside the variable **a**. The computer then adds 1 to that value. Then the computer goes to store that new value into the memory location occupied by the variable **a**.

Except that while one computer thread is doing this, another thread may be doing the same thing. What's the value of **a**? It's the value that was stored by the last thread to execute that statement. In other words, the value of **a** has become time-dependent; you'll get different results depending on how fast each thread executes.

"Isn't there some way around this?" Of course there is! For example, you can define **a** in such a way that each thread has their own copy of **a**, then sum the copies of **a** after all the threads have finished. Or you can use a lock so that only one thread can execute `a = a + 1` at any given instant.

If you're writing multi-threaded code, you have to think about these issues for every line of code that you write. That's why it's hard.

QED!

[4] Asking "what if" here is redundant. Physicists always want more. Always.

[5] In discussing threads in this way, I'm skipping over GPU cards. These contain thousands of processors, each of which can represent a separate thread. You may be familiar with this approach because it's the basis of machine learning.

However, GPU cards are limited to those cases where each thread has a relatively simple and independent execution task (e.g., acting as a node in a neural network). If each of your threads has to calculate the relativistic stochastic and non-stochastic energy loss of a muon and the changes in its trajectory as it passes through liquid argon in a non-uniform magnetic field, then a GPU card may not be able to help.

Also, don't forget the previous footnote: Even if your task can be accomplished by a GPU card, eventually you will want more. You'll still want to consider batch job submission to a "farm" of dedicated computers, each with its own set of GPUs.

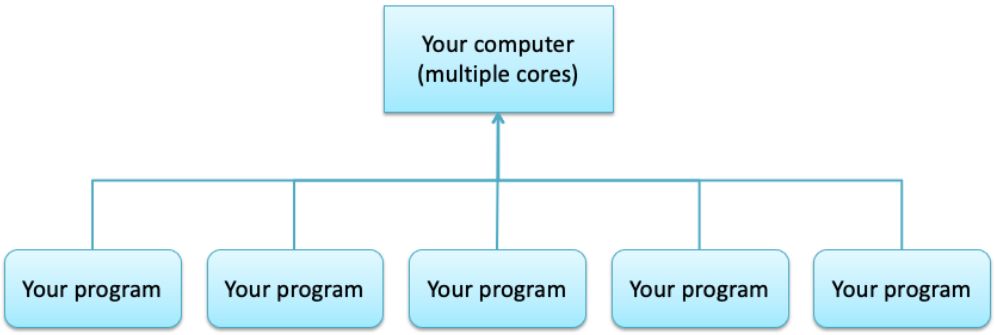

Figure 13.23:: Multiple instances of the same program on the same computer. You could manage this with the Linux `at` command, but as I note below, I don't recommend this approach.

This avoids the threading issues. Each program is separate… and therefore requires a separate region of computer memory. That's part of the "more"; this time it's "more RAM."

Let's think about this for a moment: If you run multiple simultaneous copies of the same program, each one will produce an identical result. The usual way to handle this issue is to have the program accept a parameter of some kind, and make sure that each instance of the program receives a different value of that parameter. We'll get into how that can be done in the *Condor Tutorial*.

Problem solved? Not quite. There's nothing in this approach that keeps you from submitting any number of programs. What happens at some point is that you'll overload your computer, because it's swapping between one program and another, and probably running out of memory.[6]

While you could plan carefully and not run more programs than your computer has cores or memory, you've already learned that physicists want "more."[7] In particular, you probably want to just submit your program more than a dozen times and have the computer manage it somehow.

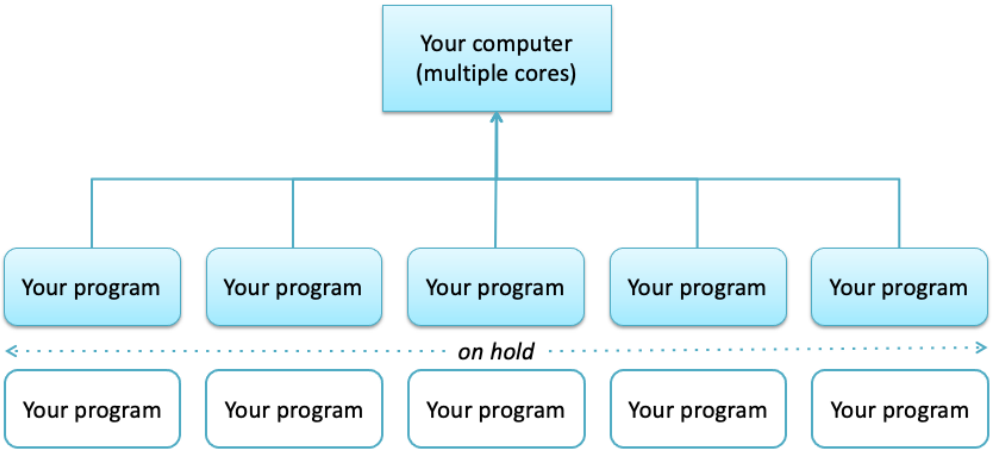

Figure 13.24:: Managing multiple programs on the same computer, with the computer controlling which ones are active. On a single computer, this can be done with the Linux `batch` command. Again, I don't recommend that you do this.[Page 238, 8]

[6] Now you know one reason you get the eternal "beach ball" or "hourglass" on your laptop… or why it can freeze entirely.

[7] In this discussion, when I say "physicists want more," it's not exclusive. Other fields of study also want more resources. However, since I mostly hang out with physicists and not geologists or architects, I'll only speak for the folks I know.

This is a batch system: A process by which a computer monitors its resources and only executes a given job when there's enough resources to run it.

This answers the question "Why batch?": So you can schedule lots of programs at once without overloading your computer. However, so far we've been limited to using a single computer. In the next section, we'll move to the next level: running your program on multiple computers at the same time.

Batch Farm

Let's take a look at a model of how you can submit a job from one computer and have it run on many other computers.

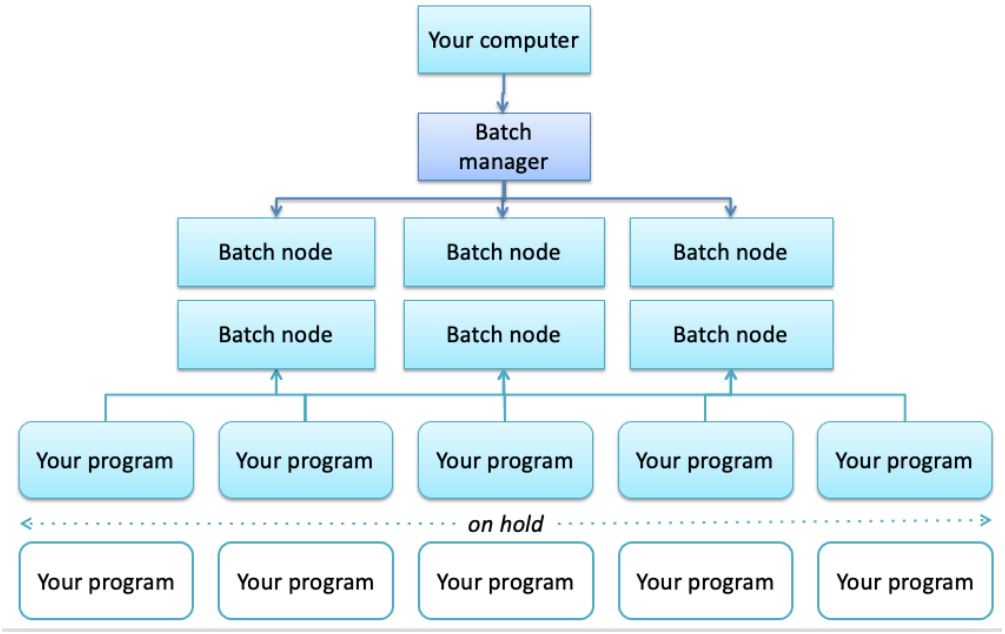

Figure 13.25:: An example of a batch system managing multiple instances of a program on multiple computers.

You can probably guess how things flow from top to bottom in the above figure: You submit a program from your computer; we'll get into how that happens later. The computer that receives the submission is the "batch manager". The batch manager looks over its collection of "batch nodes", that is, a group of computers that do nothing else but run users' program remotely.

The batch manager keeps track of which copy of your program has been submitted to which node. It also controls which copy is actually running on a given node. Other copies of your program are "on hold", waiting until a queue on that batch node is free for your program to start running.

Of course, your programs are being interleaved with the programs submitted by all the other users of this "batch farm."

[8] Why don't I recommend that you do this? Most computers used in the sciences are shared. You can submit several dozen jobs using `batch`, but what happens when another user on that computer does the same thing? Generally, one of two things: The computer slows down, which means it annoys everyone else in the working group; you become aware of all those other users (whose work, of course, is less important than yours by definition) whose programs are blocking yours from executing.

Even if you manage to get a working development and execution environment on your own computer, you'll find that using `batch` may eventually slow your system down.

A solution, as we'll see, is to distribute all these programs from multiple users onto other computers that no one uses interactively.

"Hey, if I have to run my program just five times, what's the point of all this fuss?" Don't forget the "more" principle: I've seen researchers submit 20,000 instances of their program to be executed on a batch farm. You may not need a batch system now, but if you continue your scientific research, you almost certainly will in the future.

Figure 13.26:: https://xkcd.com/2021/ by Randall Munroe

Condor

Now that you have some idea of what a *Batch Farm* is, let's consider a particular batch farm.

HTCondor is a suite of batch-related software tools. It's maintained by the Computer Science Department of the University of Wisconsin-Madison. It's probably the most popular batch software in use by scientific institutions.[1,2]

Here's how the abstract "batch farm" in Figure 13.25 looks in condor.

[1] I base this statement on the fact that I know of no other large-scale batch software manager used by any of the particle-physics institutions associated with Nevis.

The probable reason for this is that HTCondor is free. You can't beat that price!

[2] You may have noticed that the title of this section is **Condor**, but the name of the software suite is **HTCondor**. The maintainers of HTCondor had to change the name from Condor a few years ago due to legal conflicts.

This is the sort of thing that can happen in the legal world when you name your software after a species of vulture.

Since the names of all the HTCondor-related commands begin with the string `condor_`, I'm probably going to lapse into calling the suite condor. Please forgive the inaccuracy.

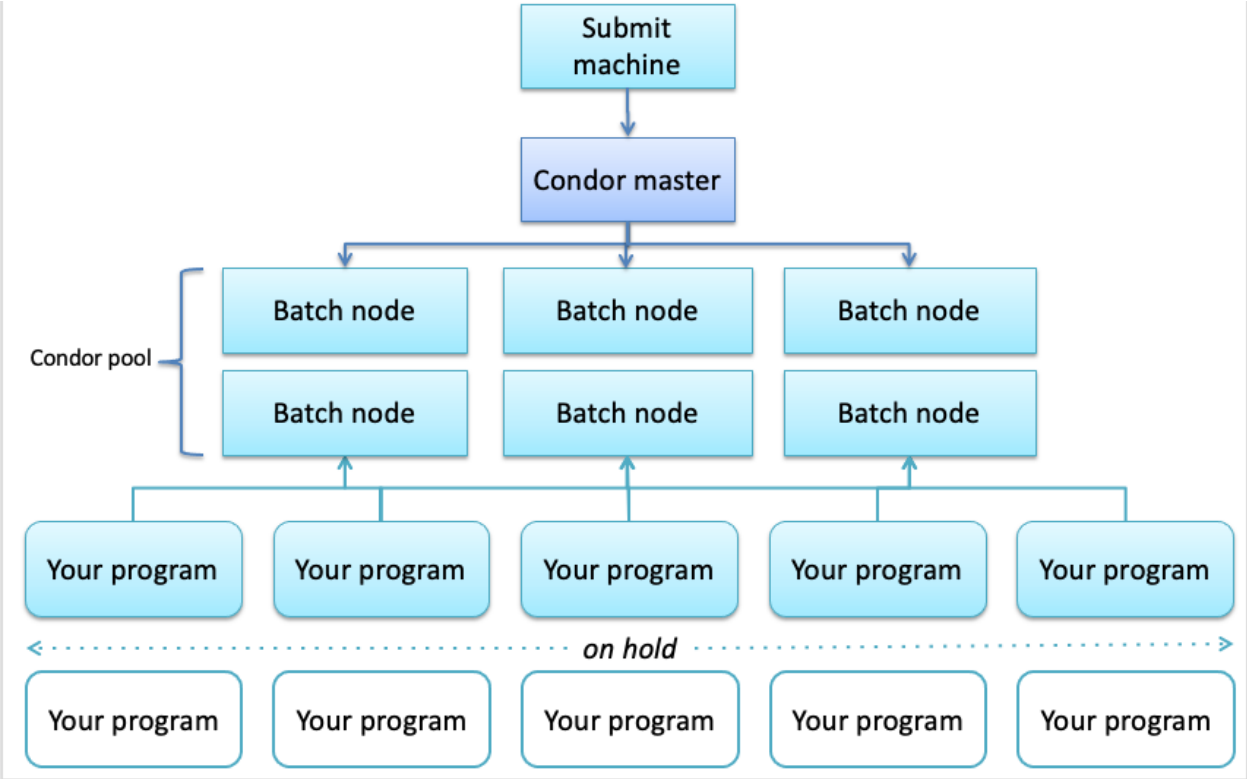

Figure 13.27:: This figure defines a couple of terms: the "condor master" is the central computer that manages all the others; the "condor pool" are the groups of computers dedicated to running the jobs scheduled by the condor master.

The uniform color and shapes of the "Batch nodes" in the above figure may give you the impression that all the nodes have to be identical: the same amount of memory, disk space, CPU, etc. However, one of the nice features of condor is that it can manage a heterogenous collection of nodes. Consider the following:

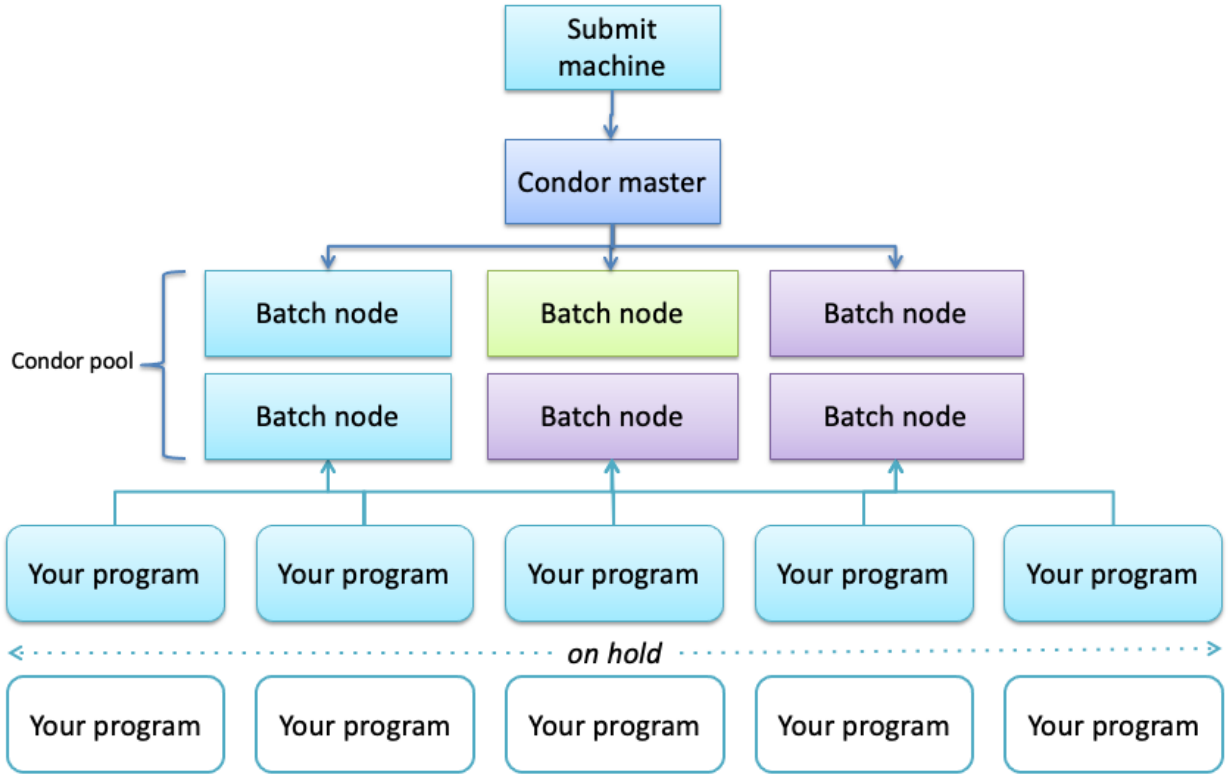

Figure 13.28:: In this figure, the different colors of the batch nodes represent different computer configurations. Those configurations might represent different amounts of memory or disk storage; they can also represent different CPU or GPU architectures; they can even represent different operating systems. (Yes: UNIX, Windows, and Mac OS can all exist on a single condor farm!)

This means that when you submit a job to condor (I'll describe how do this soon, I promise!) there needs to be some mechanism by which condor can match what you need to run your program with what a batch node offers. This mechanism is called ClassAds.

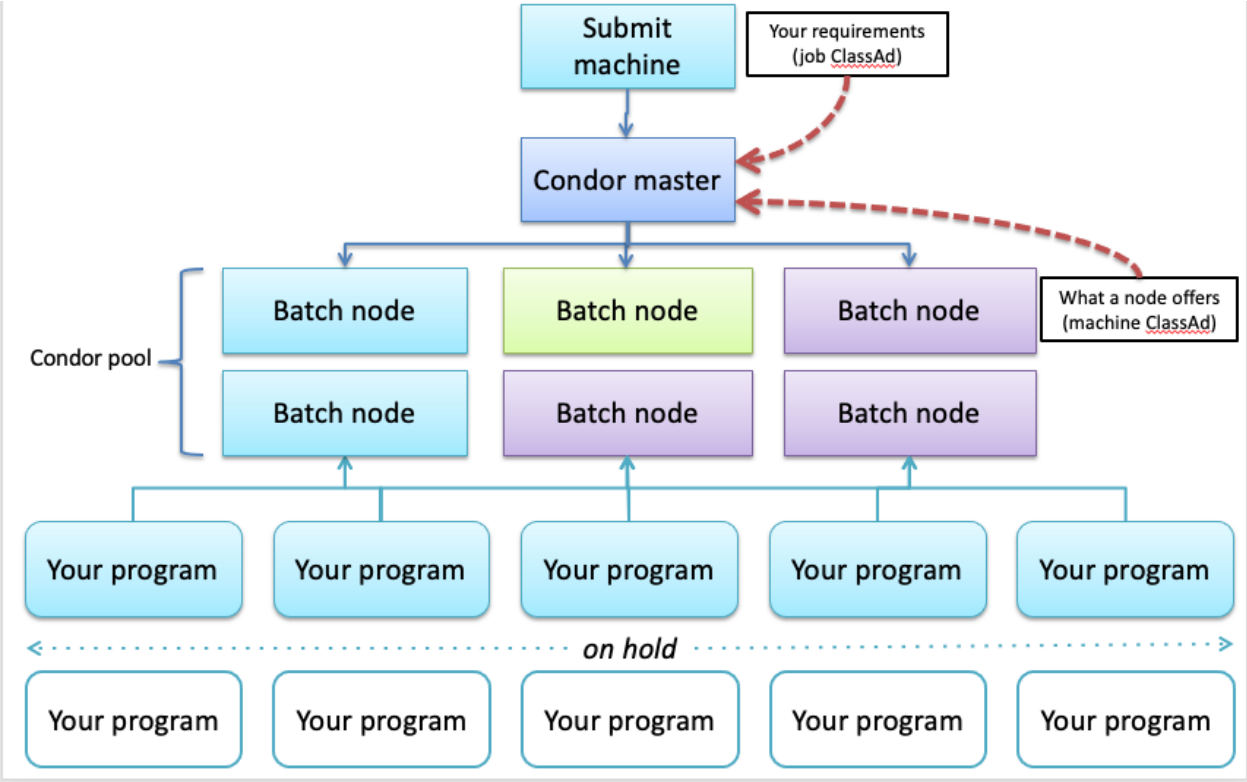

Figure 13.29:: Here's the ClassAd mechanism in action. Your program ("job") has some set of ClassAds, even if you don't specify any explicitly. Each batch node has its own set of ClassAds, such as the amount of memory it offers per job. Condor matches the job to run on an appropriate computer.

In the next section we'll go over how to actually use HTCondor.

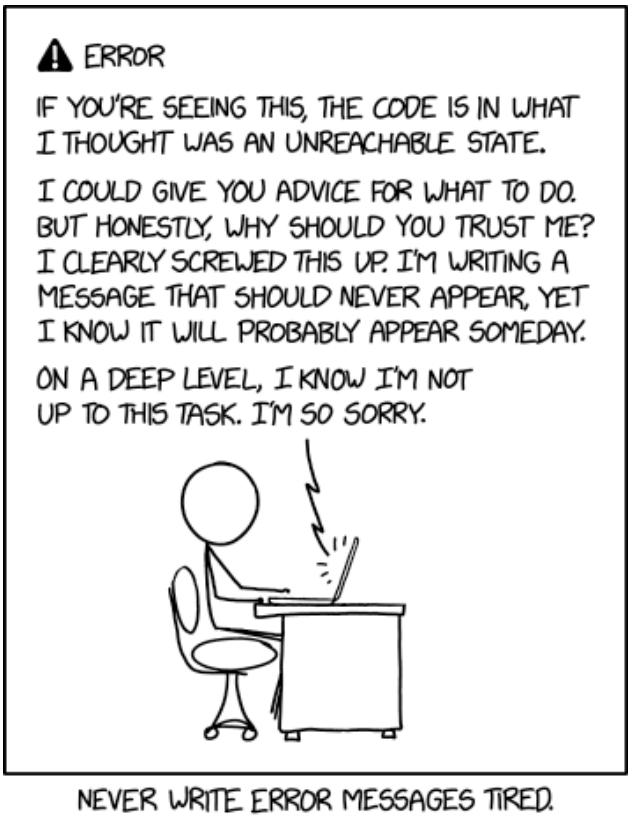

Figure 13.30:: https://xkcd.com/2200/ by Randall Munroe

Using Condor

Let's start with the main documentation:

- HTCondor Quick-Start Guide
- Full HTCondor Manual

From here, I'm going to become gradually more focused on the Nevis particle-physics procedure. It's not because the our condor system is different in any special way. It's the opposite: because Nevis is a relatively small site (its condor farms[1] have roughly 300 queues), we use a "vanilla" version of condor.[2] Sites with large farms of batch nodes, such as Fermilab and CERN, have special "wrapper" scripts around their condor job submission.

We have to start somewhere, and it might as well be with the Nevis particle-physics environment. This is a typical structure of a set of files you'd prepare for a condor job:

[1] I say "farms" because the Nevis particle-physics groups have three condor batch farms:

- ATLAS Tier3 cluster
- Neutrino cluster
- a "general" cluster

The reasons for the separate farms are mostly historical, and have to do with dull things like *moving the job to the data* and funding issues ("Why is this other group's job running on a system that our group paid for?").

[2] As you get to understand condor better, you'll realize that this is a pun. There are several different execution environments in condor, and the default universe is called "vanilla".

If you clicked on the link in the above paragraph to learn about condor universes, I'll make things easy for you: Unless your group explicitly instructs you differently for your batch system, you'll always want to use the vanilla universe.

- At the lowest level, the program(s) you want to execute.

  For example, this might be a python program or a compiled C++ binary. It would be the same kind of program that you used to complete Exercise 11.

- Above that, you'll probably need a "shell script," that is, a set of UNIX commands to set up the environment in which the programs will execute.

  It may help to picture this: When you run your program interactively by using `ssh` to get to a remote server, you generally have to type in some commands before you can actually run a program. The shell script contains those commands.[3]

  As a simple example, in this tutorial we run programs that use ROOT. On the Nevis particle-physics systems, the command to *set up ROOT* is:

  ```
  module load root
  ```

  So our script will contain that shell command (or whatever command sets up ROOT for your site).

- At the highest level is the condor command file. This is the file that contains the commands to control what condor does with your program as it transfers it to a batch node.

  For example, it's this file that contains the *ClassAds*, in the form of a `requirements` statement, to select the properties of the batch node.

Yes, it's a file within a file within a file. Once you get the hang of it, it's less complex than it sounds, especially if you start with an example. We'll look at one in the next section.

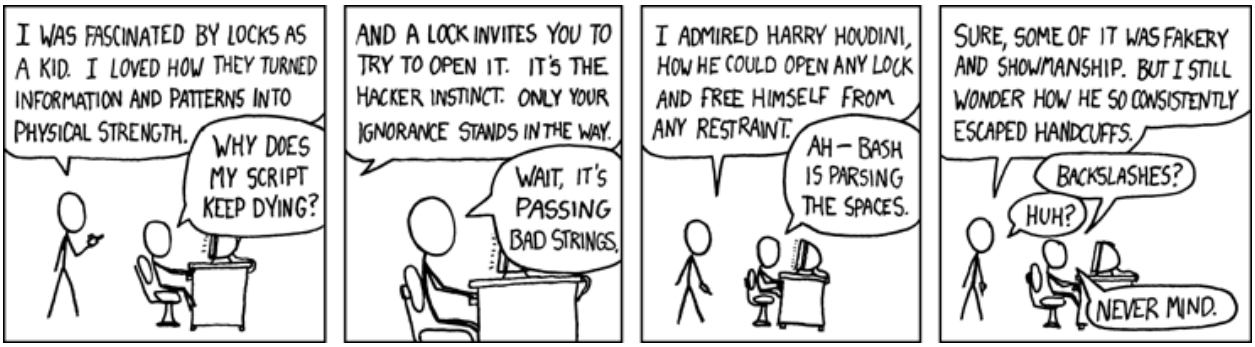

Figure 13.31:: https://xkcd.com/234/ by Randall Munroe

Condor Tutorial

Clarification: If you are not in one of the Nevis particle-physics groups, the following instructions won't work as-is. However, if you're trying to come up with an initial set of a condor command file (a `.cmd` file), a shell script (a `.sh` file)[1]*, and a program (here a `.py` file), then going over these instructions will still be useful on other condor-based batch farms.*

[3] There is an important exception to this. When you log in to a remote server, you may be typing commands like `cd` to go to the directory that contains your program. As we'll see in *Resource Planning*, in condor you cannot do this, because a batch node won't have access to any of your directories. If your program needs any files to run, you must tell condor to transfer them to the batch node.

[1] For the rest of this discussion, I'm going to assume any wrapper script is written in a bash-style shell. There's nothing wrong with having the condor executable written in a csh-style shell, or in any other scripting or programming language.

Login to your file server

For the purposes of illustration, I'm going to assume that your file server is olga (a machine name that does not exist for the Nevis particle-physics systems), and that your Nevis account name is $USER (a variable that's automatically assigned in most UNIX-related operating systems).

```
> ssh $USER@olga.nevis.columbia.edu
```

Create a directory on your file server

Disks are divided into partitions, and directories are created within partitions. All the Nevis systems have a `/data` partition. Let's go there, create our test directory, and go into the new directory:

```
> cd /nevis/olga/data
> mkdir $USER
> cd $USER
```

Copy the files used in this tutorial

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

For a list of files you've copied:

```
> ls -lh
```

Look at the files

As noted *before*, typically condor jobs require at least three files: the condor command file, a shell script executed by the command file, and a program executed by the shell script. Take a look at these files and read the comments:

```
> less condor-example.cmd
> less condor-example.sh
> less condor-example.py
```

The command file (the `.cmd` file) submits the shell script to be executed on some machine in the condor pool. The shell script (the `.sh` file) sets up the environment for the program to execute. The program (here a `.py` file), when it executes, writes an output file. That output file is copied by condor to the directory from which you originally executed `condor_submit`.

If you're not at Nevis: if you click on the following links, you should be able to see these files in your web browser. It's worth reviewing them for both the commands and the comments:

- condor-example.cmd
- condor-example.sh
- condor-example.py

However you're looking at the files, note that each "instance" of this job has its own `Process` parameter. This is one way to differentiate between multiple instances of the same program running on different job queues; i.e., process 0 gives a different result than process 1, and process 1 is different from process 2, etc.

The Process number

Realistically, at most you're just going to skim the contents of these example files. But there's one aspect I ask you to pay special attention to: process.

This is what makes each run of your program unique. If you didn't make use of `$(Process)` in some way, as shown in the examples, then every run of your program would produce identical results. This will wreak havoc on your analysis.

How will `$(Process)` affect your program? That's up to you. Among the most common is to have a program read a selected number of events from a file; e.g., if the process is 0 read the first 10,000 events, if the process is 1 read events 10,001 to 20,000, etc.[2]

If you're running a simulation, your program will be generating lots of random numbers. Typically you'd have the process be part of the random-number seed.

Another common use of `$(Process)` is to make it part of a file name. In this way, your program will generate a unique output file. After all the instances of your program have run, you'll have to combine the outputs in some way. There's a ROOT feature designed for this: *TChain*.

Make sure the files are executable

For a file to be executed as a program, it must be executable. I've already made sure that condor-example.sh and condor-example.py are executable programs via the following commands, but I suggest you type them in again to both be certain and to know how to do this when you start writing your own scripts.

```
> chmod +x condor-example.sh
> chmod +x condor-example.py
```

Let's try it

Submit your condor command file to the condor cluster:[3]

```
> condor_submit condor-example.cmd
```

Quickly (before the program has a chance to finish), type

```
> condor_q
> condor_q -run
```

The first command shows all the jobs you submitted on this computer. The second command shows the jobs you submitted which are currently executing, and on which computer.

Within a minute or so, the job will complete and there'll be no result with your account ID from `condor_q`. Take a look at the contents of your directory:

```
> ls -lrth
```

The files are listed in ascending order by date. Note the new files at the bottom of the list. Compare these files names to the ones given in condor-example.sh. Can you see how `condor-example-test-0.root` got its name?

[2] If you're working with *RDataFrame*, the `Range` method described on the RDataFrame web page makes this approach easy.

[3] "After all those *figures* you showed us, don't we have to know the name of the condor master or any of the nodes?" If everything works, no! Condor handles it all for you.

If you want to see how condor handles it, you can look at the following files on your server:

```
> less /etc/condor/condor_config
> less /etc/condor/condor_config.local
```

Multiple jobs

Edit the file condor-example.cmd and change the last line to read

```
queue 10
```

This means to submit 10 jobs.[4] Save the file and execute the `condor_submit` command again. Note how the submitted jobs are "counted off" by periods. Type `condor_q` and `condor_q --run` to see which computers execute the jobs. When they're all done, look at the contents of your directory to see all the new files.

Run ROOT and look at the contents of `condor-example-test-9.root`. Does it contain the histogram you expect? Look at the mean and the histogram limits.

Aborting a job

It happens all the time: You submit 10,000 jobs, and then realize that something is wrong. Fortunately, you can quickly abort a cluster of condor jobs.

Do `condor_submit` again. The message that comes out looks something like this:

```
> condor_submit condor-example.cmd
Submitting job(s)..........
Logging submit event(s)..........
10 job(s) submitted to cluster 14.
```

The identifier for this particular cluster is "14" (you'll almost certainly see a different number). If you want to cancel all the jobs in that cluster at once, the command is:

```
> condor_rm 14
```

If you forget the cluster ID, you can always remind yourself with `condor_q`.

Clean up

Finished? Get rid of the files you no longer need:

```
> rm condor-example-test*
```

Or if you really want to wipe a directory that you're never going to use again:

```
> cd /nevis/olga/data
> rm -rf $USER
```

[4] If you're going to frequently change the number of jobs you're going to submit, there's another way to approach this. Delete the `queue` line from the `.cmd` file. Then include a `-queue N` option in `condor_submit`, where N is the number of jobs you want to submit. For example:

```
condor_submit condor-example.cmd -queue 10
```

A couple of tricks

Here are a couple of extra tricks you can do with python to improve this process a little bit.

- Eliminating the "middle-man", that is, the `.sh` file. You can do this by placing all the environment set-up commands in your `.py` file.
- A different method of getting command-line arguments into your python program.

If you've deleted your temporary directory in `/data`, create it again and `cd` to it. Copy over these example files:

```
> cp ~seligman/root-class/root-python-setup.* $PWD
```

Take a look at root-python-setup.cmd. It looks pretty much the same as that other condor command file, with one big difference: instead of executing a shell script that will execute another program, this command file will execute the python program directly.

Now look at root-python-setup.py and look at the comments. Two new things are happening in this program:

- The python program is setting up its own environment. This requires the "stupid python trick" I mention in the comments (causing the program to run itself again). This altering of an external environment is something python can do but C++ cannot, at least not without even more trickery than you see here.
- The python program is parsing its arguments; that is, it's looking for options and arguments instead of just assuming that the first argument has a particular meaning. This can be done in C++ as well. When I'm writing code that requires only a couple of parameters, I like to use "getopt" or "argparse" methods because they help tell a user what a program is doing. Which is clearer to you?

```
> condor-example.py 5 myfile-5.root

> root-python-setup.py --mean=5 --outputfile=myfile.root
```

Resource Planning

If you've made it this far, then:

1. Congratulations!
2. You now know how to submit a condor job with very simple input/output requirements.
3. Your head must be spinning with all these files: `.cmd` files, `.sh` files, `.py` files, etc.

There's a reason for the nested scripts. It has to do with resource planning: Telling condor what inputs are needed to run your program, and how to handle any outputs.

Physicists hate resource planning.[1] But if you're going to use condor for any real task, you'll have to do it.

[1] In this discussion, when I say "physicists hate resource planning," it's not exclusive. Other fields of study also hate resource planning. However, since I mostly hang out with physicists and not physicians or morticians, I'll only speak for the folks I know.

Understanding the execution environment

Let's take a look at some lines from condor-example.cmd, with a couple of modifications for the sake of discussion:

```
executable     = condor-example.sh
transfer_input_files = condor-example.py
input          = experiment.root
transfer_output_files = condor-example-$(Process).root
output         = condor-example-test-$(Process).out
error          = condor-example-test-$(Process).err
log            = condor-example-test-$(Process).log
```

The purpose of these lines is to instruct condor which files have to be transferred to the batch node for your program to work, and which files should be copied back to you after your program is finished.[2]

This diagram illustrates what's going on:

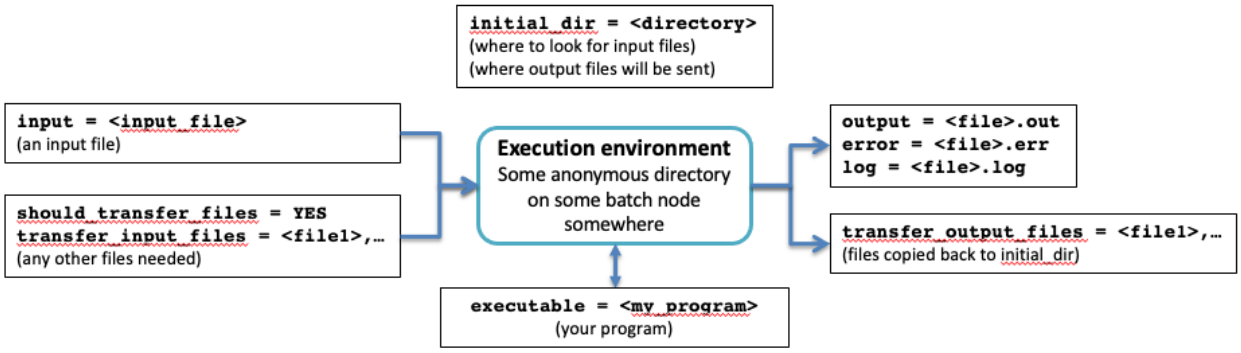

Figure 13.32:: The relationship between the lines in a `.cmd` file and the condor execution environment.

Note that `output` refers to the direct text output of your job (such as `print` statements in python or `cout` statements in C++). The `error` file will contain any error messages. The `log` file will include some condor status messages, including the name of the batch node on which your job executed.[3]

You can learn more about these statements in the HTCondor manual.

[2] You might initially think that if your program produces any outputs, condor should simply transfer everything back to you. In reality, an execution environment often contains a lot of "junk"; one classic example are conda work files. It's generally better to explictly tell condor which output files are relevant to you.

Why I didn't do this in condor-example.cmd? When I first created that file, selective transfer of output files was not available in condor. And now… well, perhaps I'll get around to it in time for the next edition of this tutorial.

[3] The `.log` output from condor is a bit different than the `.out` and `.err` output. If you run your job more than once with identical values for `output=` and `error=` (in this example it would mean than you're running the same with the same process number), the `.out` and `.err` files will be overwritten.

However, the `.log` is always appended to, not overwritten. If you see your `.log` file gradually become larger and larger as you re-submit your jobs, it's because the file is basically maintaining a "history" of every time you've run the job.

Stay within the execution environment

You're not in Kansas anymore

Or to be less flippant, you're not in your home directory anymore. When your job is executing on a batch node, your program (and any shell script that runs it) doesn't have access to the files and environments that you might have set up in your home directory.

Take another look at condor-example.sh. The first few non-comment lines set up the execution environment for the Nevis particle-physics systems.[4]

If you're not at Nevis, of course, you'll have learn what form of preamble or set-up is required for your institution's batch farm to replicate your environment.

In general, it's not a good idea if your shell script (the `.sh` file) references any directory outside the execution environment for that particular process. The reason is that you may not know which batch nodes will run your job, or if they have access to any external directories.

You can fiddle with directories in condor, but you'll want to keep it within the condor environment; e.g.:

```
# Create a new sub-directory within the condor execution environment.
mkdir myDirectory

# Visit this directory.
cd myDirectory

# Do something in that directory.
echo "This is a temporary file" > file.txt

# Leave the directory
cd ..

# Run something that uses that directory.
./myprogram.py --inputfile=myDirectory/file.txt
```

Test to see if everything works

This may sound trivial, but you'd be surprised at how often I see people skip this step: Test your condor job submission with 2-3 jobs before submitting a huge quantity like 20,000.

The reason why I'm giving this advice is that I've seen what happens when someone doesn't do it, and there's a problem with their job:

- When a condor job fails, it sends an error message.
- 20,000 failed jobs means 20,000 error messages.
- Sending out 20,000 error messages at once will clog our mail server, slowing it down. This does not make one popular.
- The 20,000 error messages are sent to your email address at your home institution. Your home institution becomes suspicious, and shuts down your email address.
- The `nevis.columbia.edu` mail server is identified as a potential source of spam, since it sent out 20,000 emails to a single address. Our mail server is added to spam block lists. Members of the Nevis faculty discover that their emails never arrive, and without any warnings. Again, this does not make one popular.

[4] Actually, on Nevis particle-physics systems, the directory `/usr/nevis/adm` *is* located outside the condor execution environment. This works on our batch farms because of the way directories are *managed* here.

All of the above have happened, though fortunately not all at once. Still, to maintain your popularity, test your scripts first!

Find reasonable execution times for your jobs

The "sweet spot" for the length of a job is about 20-60 minutes.

- Shorter than 5 minutes, and the overhead involved in condor setting up the execution environment and copying your files begins to take up a substantial percentage of the job's execution time.
- Longer than that, and your job may run up against condor's time limit for a job. That limit is assigned by a systems administrator; for the Nevis particle-physics systems I've left this limit at its default of two hours.

  That's not a "hard" limit; if no other jobs are waiting to be executed, condor will let a job run over two hours. However, as soon as any other job requests to be executed, condor will "suspend" a job that's taking too long. A suspended job has to start again from the beginning.[5]

A practical way to discover how long your job takes to run is to do some test submissions with varying values of an appropriate parameter; e.g., the number of events read from an input file, or the number of events to generate in a simulation.

Too many inputs

As you work with more elaborate projects, you may discover that you need a longer and longer list of files to transfer with the `transfer_input_files` option in the `.cmd` file.

There are two ways to deal with this. One way is to let `transfer_input_files` copy an entire directory for you. For example:

```
transfer_input_files=myprogram.py,my_directory
```

If you have to transfer a lot of input files, put them in that directory. Then you reference that directory within your condor environment; e.g.:

```
./myprogram.py --inputfile=my_directory/experiment.root
```

It's easy to get sloppy and forget about the contents of any such special directories. They tend to pick up "junk" that your condor job never uses, but transfers anyway because condor can't tell the difference.

Therefore, I like to use the tar command to create an archive file that I will unpack in my condor job.[6]

Here's an example from work that I'm doing right now (May-2022). I've got a program that I want to submit to condor. I developed the program and its associated files as I worked in a directory. That directory is filled with all kinds of temporary files, test scripts, and other junk:

```
> ls work-directory
10000events.mac       example.hepmc2    grams.gdml          LArHits.h            root
atestfil.fit          example.hepmc3    gramssky            mac                  ␣
↪scripts
bin                   example.treeroot  GramsSky            Makefile             test
btestfil.fit          GDMLSchema        gramssky.hepmc3     options.xml          test.
↪mac
```

(continues on next page)

[5] At one time we had someone try to submit a week-long spline calculation on a Nevis batch farm. It never executed, because any time someone submitted a different job to the farm, the spine job was suspended and had to start from the beginning.

[6] Why don't I use zip? Mostly it's personal preference: although `tar` is a bit harder to use, it's more powerful than `zip`. Also, `zip` isn't always part of a standard Linux installation, though I make sure it's available on all the Nevis particle-physics systems.

(continued from previous page)

```
c1.gif                gdmlsearch        grams-z20.gdml       options.xml~          test.
↪root
c1.pdf                gdmlsearch.cc~    hepmc3.mac           outputs-pretrajectory ␣
↪treeViewer.C
CMakeCache.txt        gramsdetsim       hepmc3.mac~          output.txt            user.
↪properties
CMakeFiles            GramsDetSim       hepmc-ntuple.root    parsed.gdml           view.
↪mac
cmake_install.cmake   gramsdetsim.root  HepRApp.jar          radialdistance.root   ␣
↪xinclude.xml
crab-45.mac           gramsg4           heprep.mac           rd.pdf
crab.mac              GramsG4           HitRestructure.root  README.md
detector-outline.mac  gramsg4.root      LArHits.C            RestructuredEdx.root
```

I copy the work directory:

```
> cp -arv work-directory job-directory
```

Then I clean up job-directory to only contain the files I'll need to run my condor job. Here it is after I'm done.

```
> ls job-directory
bin  GDMLSchema  gramsdetsim  gramsg4  grams.gdml  gramssky  mac  options.xml
```

For me, this is a "reference directory" for the jobs I'm going to submit. I can continue to fiddle with things in work-directory, but I'll leave job-directory pristine until there's a reason to change it.

This directory still contains a lot of files:[7]

```
# Count the number of files in job-directory
# and all its sub-directories.
> find job-directory -type f | wc -l
60
```

So it's easier for me to transfer all these files as a single archive. I'll create an archive of that directory:

```
> tar -cvf jobDir.tar ./job-directory
```

It's the single `jobDir.tar` file that I'll transfer and unpack in my condor job. In my `.cmd` file, I'll have:

```
transfer_input_files=jobDir.tar
```

In my `.sh` file, I'll have:

```
tar -xf jobDir.tar
```

Then within my condor execution environment, I'll have access to all the files in job-directory. For example:

```
./job-directory/gramsky --rngseed=${process}
```

That last option to `gramssky` supplies a unique random-number seed for my sky simulation... but that's another story.

[7] Mystified by the find? By the wc? And what's the deal with that vertical bar?
Hey, I told you that you could spend a lifetime learning about UNIX!

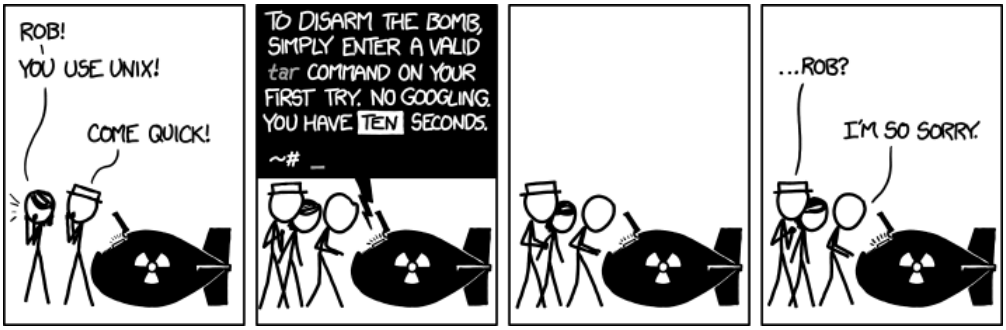

Figure 13.33:: https://xkcd.com/1168/ by Randall Munroe

Shared filesystems

We're almost finished with the subject of batch systems. Hang in there. There's a really sick xkcd cartoon at the end.

After lecturing you on keeping your execution environment clean and independent of outside directories, I have to confess: I wasn't telling you the whole truth.

The reality is that in many batch-node installations, there is an external filesystem of some sort that all the nodes share. Here are a couple of reasons why that might be needed:

- Software libraries, such as the those maintained in environment modules and in conda.

  Keeping such libraries in a shared filesystem may be the only practical way to assure that all the batch nodes have access to the same software.[1]

- Large data files.

  Condor is pretty good about transferring files, but there can be problems when those files get bigger than 1GB or so. It may be easier to read those files via a shared filesystem, even though there'll be a speed and/or bandwidth penalty for many programs reading a large file over the network at the same time.

There's no single standard for shared filesystems and condor. The two accelerator labs with which I'm the most familar, CERN and Fermilab, each have their own method. You've already guessed that I'm going to describe what I've set up on the Nevis particle-physics batch farms, because it's the one you're mostly likely to use if you've read my condor pages so far.

From this point forward, everything I describe is in the context of the Nevis particle-physics Linux cluster. If you're not in one of the Nevis particle-physics group, or you're outside Nevis, you'll have to ask how they handle their shared filesystem (if any).

[1] An example of this at Nevis, to which I aluded in *Resource Planning*, is the `/usr/nevis/adm` directory. On all the systems in the Nevis particle-physics Linux cluster, including the batch nodes, this is an external directory mounted from `/nevis/library/usr/nevis/adm`.

Nevis shared filesystem

The cluster shares its files via NFS, a standard protocol for systems to view other directories.

At this point, you may want to read the Nevis wiki page on automount. The basic summary is if a system (let's say `olga`) has a partition (perhaps `share` as an example), then to view that partition use the path `/nevis/olga/share`.[2]

Anyone can set up permissions to restrict others from viewing their directories. For the most part, you can view others' directories without having accounts on the individual machines. That's why you can view the directory `~seligman/root-class/`, which expands to `/nevis/tanya/home/seligman/root-class/`, even though you can't login to my desktop computer `tanya`.

Nevis filesystems and the batch farms

At this point, you may be thinking, "Hey, I don't have to bother with all the *Resource Planning* stuff.[3] Bill just said everything is shared, right? So I'll just copy-and-paste the same lines I use to run my program into a `.sh` file and submit that. Easy-peasy!"

Sorry, but that won't work. The reasons why:

- It may be troublesome to figure out how to use `input=`, `transfer-input-files=`, and `transfer-output-files=`. But condor's file-transfer mechanism is much more robust than NFS. I've seen systems running hundreds of shadow processes without slowing down the system from which the jobs' files came.
- The NFS file-sharing scheme has been deliberately set up in such a way that you *can't* refer to your home directory within a condor job.

It's reasonable to ask "Why not?" Consider what might happen if the batch nodes could access your home directory, and all the batch nodes on a cluster wanted to access that directory at once:

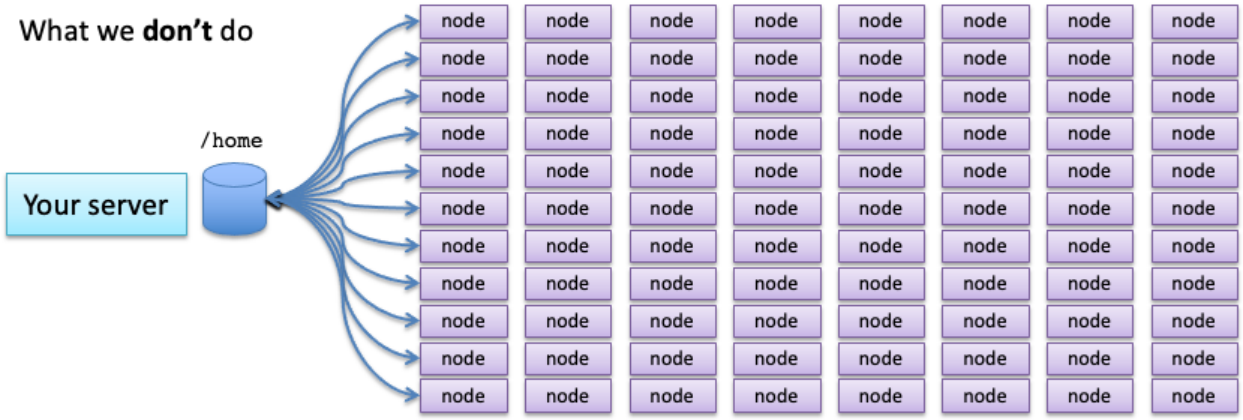

Figure 13.34:: This is what we *don't* permit the NFS-based shared filesystem to do.

NFS is a robust protocol, but handling hundreds of access requests to the same location on a single partition is a bit much for it. If it's just reading, the server may slow down so much that it becomes unusable, which irritates any users

[2] This may explain for you why your Nevis home directory is `/nevis/milne/files/<account>`. It means your home directory is named `<account>`, on partition `files`, on machine `milne`, in the Nevis cluster.

[3] I thought about using the more polite word "nonsense" here. But we're all adults, and I know you can handle the word "stuff", which (despite being slang) is more direct and to-the-point.

who are logged into that server to do their work. If those hundreds of jobs are *writing* to that directory at the same time, the server will crash.

The servers with users' home directories are called login servers, because those are the servers that users primarily login to. If a login server slows down or crashes, users can't login. Since our mail server requires a user's home directory be available to process email, if a login server slows down or crashes, our email slows down or crashes.[4]

Each Nevis particle-physics group resolves this issue by having dedicated file servers that are distinct from their login servers. Remember the diagram I showed you at the beginning of the first class?

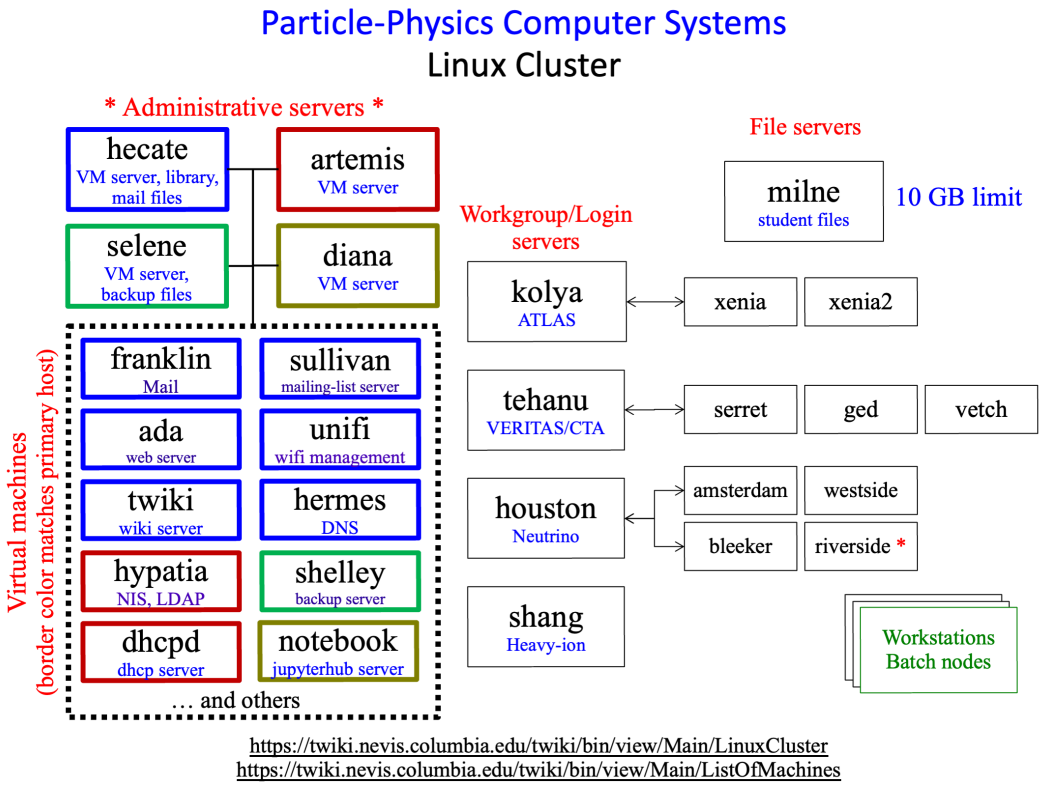

Figure 13.35:: On the first day of class, I predicted that you'd forget this diagram. Was my prediction correct?

The file servers are the smaller boxes to the right in the above figure. Each one of those file servers has at least two partitions:

- `/share`

  This partition is meant for read-only access by the batch farms. `/share` is intended for software libraries or similar resources that a job may need in order to execute.[5] The size of the `/share` partition is typically on the order of 150GB, and it's shared among the users in that research group.

- `/data`

  This partition is meant for big data files (either inputs or outputs), and any other recreatable files associated with your jobs.[6] Typically `/data` is about a dozen or more terabytes, though that varies widely between file servers.

[4] *Your* email may not handled at Nevis, but the *faculty's* email is. If you were to submit a job that would crash the faculty's email, you would not be popular.

[5] Like the `/home` partitions (and similar critical directories, such as the student accounts on `milne`), the `/share` partitions are backed up nightly. Note that `/data` directories **are not backed up**; we have neither the bandwidth nor the spare disk storage for overnight multi-terabyte backups.
Keep that in mind as you're deciding what to keep in `/share` and what to keep in `/data`.

[6] Here, "recreatable" means that if something were to happen to the `/data` partition, you could easily recreate the file by running a program

This is how it looks:

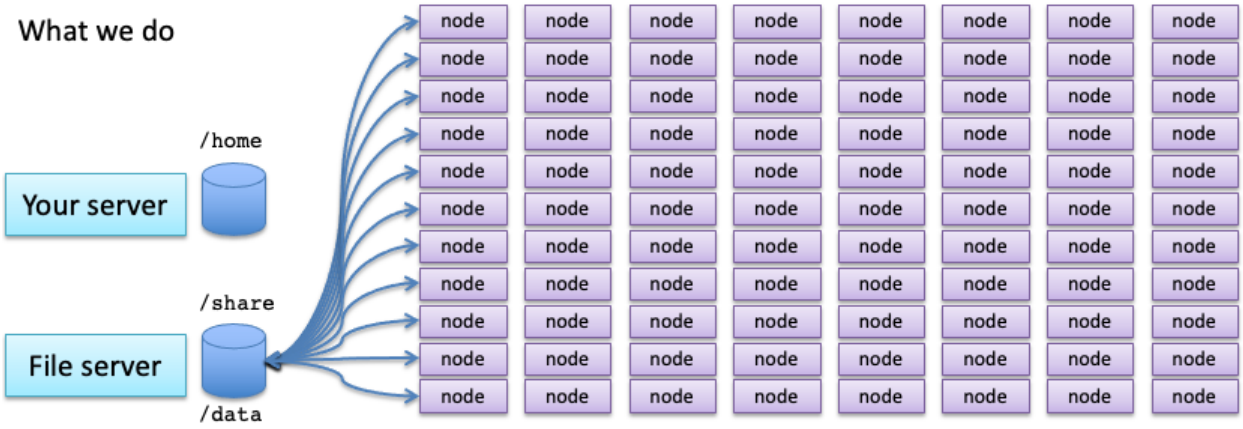

Figure 13.36:: This is what we permit the NFS-based shared filesystem to do.

It's still possible to crash a file server in this way. But if you do, it only affects your research group, not all the Nevis particle-physics groups, faculty, or email. Your *group* may irritated with you, but that's a different story.[7]

Planning a batch job

Here's the general work flow:

- Develop a program or procedure in your home directory. Get the program to work on a small scale.
- Once you're confident you have a working program, copy the relevant files to a `/share` partition. Typically you can do this (using `olga` as an example file server):

```
> mkdir -p /nevis/olga/share/$USER
> cp -arv <my-program-directory> /nevis/olga/share/$USER
> cd /nevis/olga/share/$USER
```

  Then clean things up in `<my-program-directory>` in preparation for the batch farm.

- In your `.cmd` file, refer to any distinct input files by their absolute path to the `/share` directory in which you keep them. For example:

```
executable=/nevis/olga/share/$USER/myjob.sh
transfer_input_files=/nevis/olga/share/$USER/myprog.py,/nevis/olga/share/$USER/
↪jobDir.tar
```

- Create a directory on `/data` for your output files; e.g.:

```
> mkdir -p /nevis/olga/data/$USER
```

- Set `initialdir` in the `.cmd` file to that directory, so that output files will go there; e.g.:

```
initialdir=/nevis/olga/data/$USER
```

again. An example of a file that is *not* easy to recreate is a research paper that you write; keep that and its associated plots in your home directory!

[7] If your group hassles you about crashing a file server, ask them if any of them have ever crashed a server? They'll look embarassed, mumble an apology, and politely offer to work with you so that it doesn't happen again.

- Do a small test run of your job to see that it all works; e.g. (in `.cmd`):

```
queue 4
```

  Then:

```
> condor_submit myjob.cmd
```

- Examine the files in /nevis/olga/data/$USER to make sure that everything works the way you want it to. Then go to town!

```
queue 1000
```

Bringing the job to the data

There's one last wrinkle on the whole job-design process.

Suppose your analysis requires reading large disk files. They're too big to let condor do the transfer, and you also don't want to hog the network bandwidth by reading them via NFS.

One answer is to make sure that a job that requires access to a big file runs on the machine with the partition that holds that file. This requires:

- A list of which large file is located on which batch node. That list, in turn, must come from some earlier procedure that created/downloaded the file onto selected nodes.
- A wrapper script around the `.cmd` file to insert or modify a `requirements=` line, to force the job to run on a particular node.

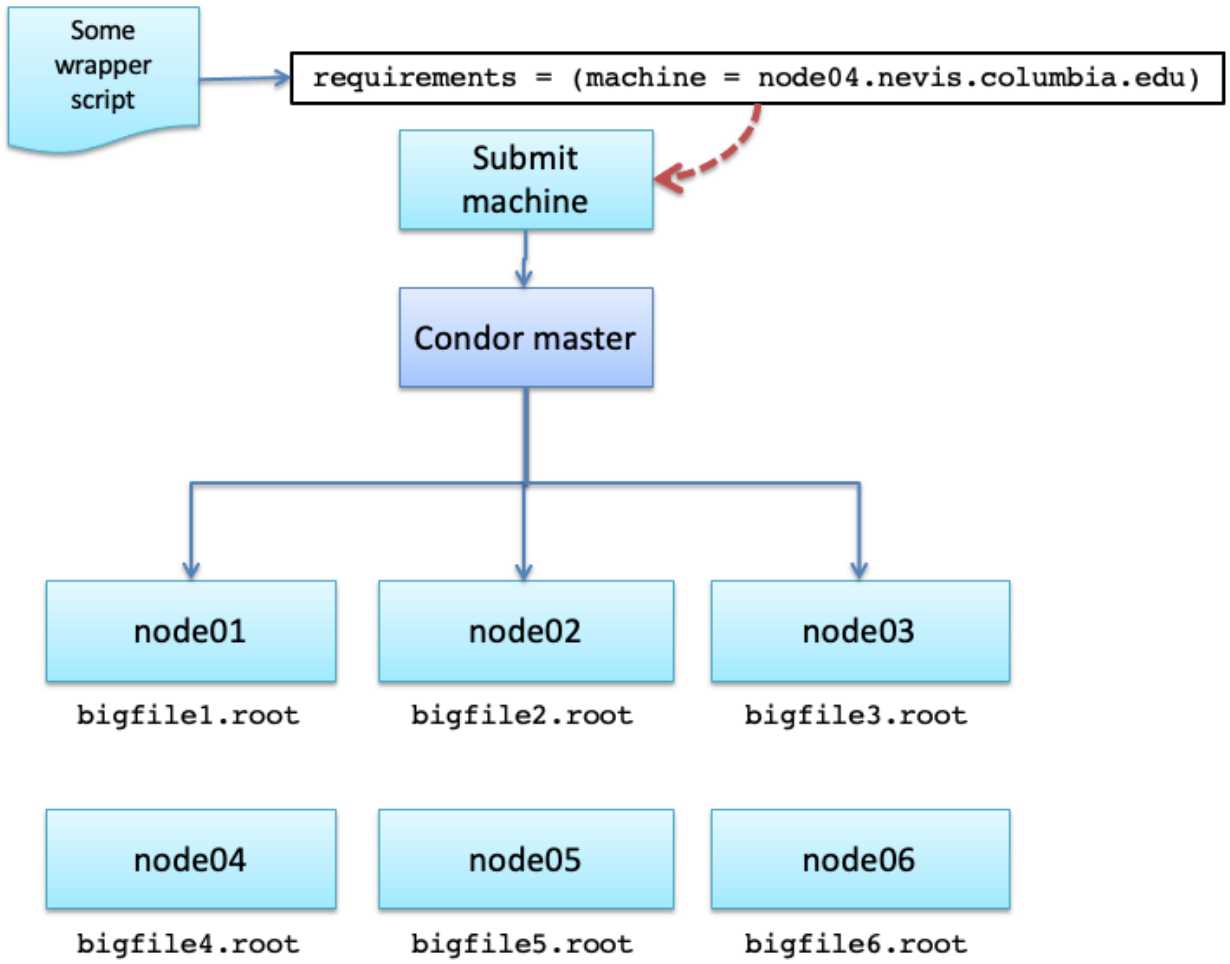

Figure 13.37:: A sketch of how one might "bring the job to the data." In this example, our program needs to access `bigfile4.root`.

To some degree this brings us all the way back to Figure 13.24: a bunch of programs with the same input file all being forced to run on a single computer. The main difference is that the process of downloading the files and submitting the jobs can be automated. The details of how that's automated depend on each research group and the tools they use.

Disk space

Physicists like to live in an idealized world where there's infinite disk space.[8] The reality is that there's only so much disk space available to you, or that is shared among your research group.

Batch jobs tend to consume large amounts of disk space. In particular, if you submit N jobs, you're going to get at least 3*N files written to `initialdir`; each job will write a `.out`, a `.err`, and a `.log` file. These files, and other miscellaneous outputs associated with different projects, can accumulate and be forgotten. They passively take up space and are never looked at again.

At some point, please consider deleting these "scrap" files once you no longer need them. Only the most intelligent and clever physicists remember to do this… and we have high hopes for you!

[8] In this discussion, when I say "physicists like to live in an idealized world"… Oh, never mind. By this point I'm sure you've got the joke.

Figure 13.38:: https://xkcd.com/669/ by Randall Munroe. This is another example of an idealized environment that, for practical reasons, physicists can't use.

If you're using the Nevis particle-physics systems, this same material is covered on the Nevis particle-physics wiki (though without the cartoons and jokes):

- Condor
- Batch details
- Disk sharing

As you work your way through these sections, your initial reaction may be, "Why does this have to be so complicated?"

I have a horrifying answer for you: At Nevis, batch systems and job submissions are relatively simple. At the large national labs, or if you're using the Open Science Grid, the available resources are more powerful but it's more complex to manage them.

A description of the Nevis particle-physics batch set-up, even if you don't have access to it, may help bridge your understanding towards the more advanced environments.

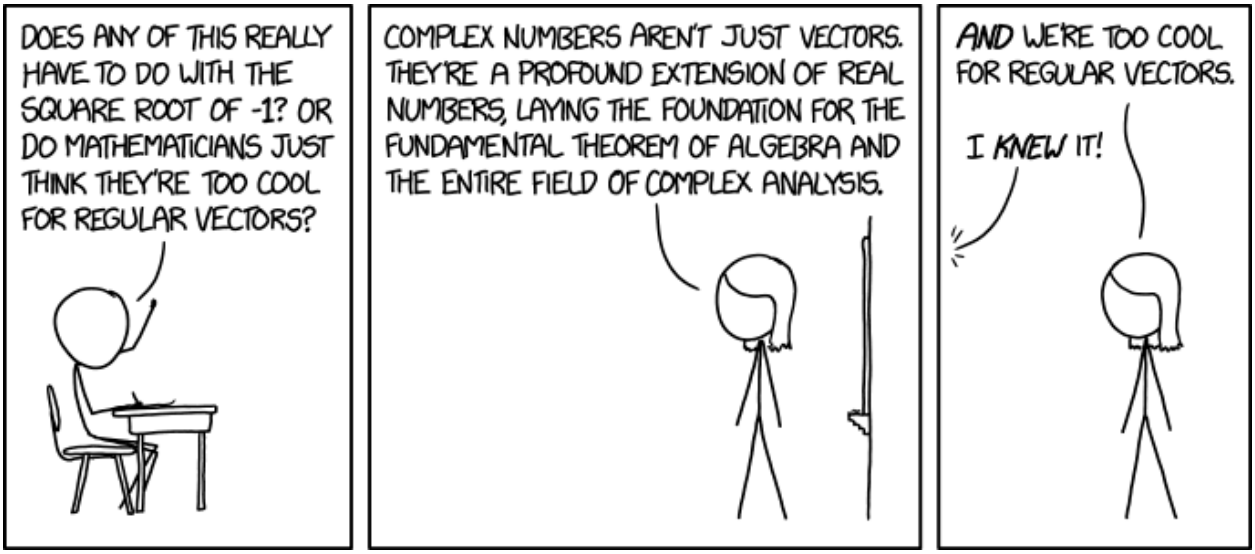

Figure 13.39:: https://xkcd.com/2028/ by Randall Munroe. This is another reason for the complexity of batch systems.

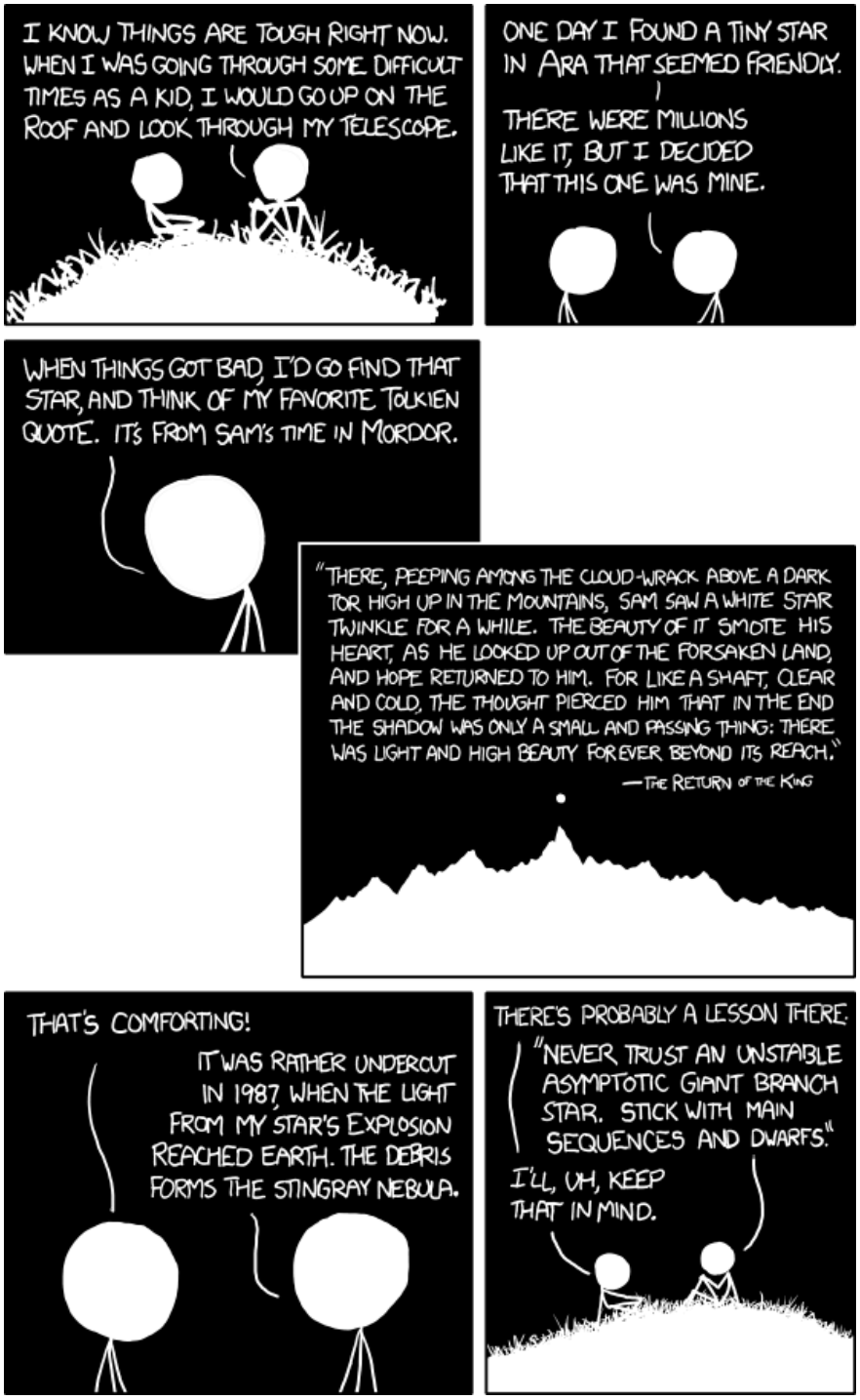

Figure 13.40:: https://xkcd.com/847/ by Randall Munroe

CHAPTER

FOURTEEN

VERSION HISTORY

2023

The biggest change for 2023 is a big new section on using *RDataFrame* to analyze an n-tuple. This was in response to the ROOT developers stating that `RDataFrame` (and a still-experimental class, RNtuple) represented the future of ROOT-based analysis.

I also went over the instructions on *Installing ROOT on your own computer*. In particular, I re-ordered the material to emphasize the approach (using conda) that the students were most likely to use on their laptops. I also address some issues with trying to install ROOT on Windows systems.

2022

People have reported to me that this tutorial is being more widely used outside of Nevis. Considering all the resources that CERN and other institutions have put into their own ROOT tutorials, this is quite a compliment! In recognition of this, I've tried to make the ROOT setup information and installation information less Nevis-centric.

I've added some material about directories in the "Introduction to Linux" section. This is in response to my (perhaps false or biased) impression that an increasing number of students have no background in using the command line on their laptops; they're accustomed to their GUIs and saving all their files on the desktop.

On the advice of Prof. Schellman of Oregon State University, I added a few paragraphs on Poisson distributions to the section on statistics.

The biggest change to the tutorial for 2022 was also due to an idea from Prof. Schellman: In prior years, the tutorial was a PDF file generated from a Microsoft Word document. It is now maintained using the Sphinx documentation management system and written in a combination of MyST and reStructuredText, both of which are variants of Markdown. This enables the main web-based presentation of the tutorial, with a simultaneous release of a supplementary PDF version.

I added expansions of the three additional talks that I give during the ROOT tutorial (on statistics, speeding up code, and batch systems). These are now available in an appendix.

2021

This year I had some extra time to teach the tutorial. I asked for suggestions from the working groups at Nevis for additional topics. One of them was to teach the students some common concepts that physicists use in statistics. The result was longer than I'd like, since statistics is a big topic that's worthy of its own courses and texts. This doesn't have much to do with ROOT, so I created a new section on the topic.

2020

I sincerely hope that by the time you read this, the impact of the 2020 pandemic will have faded into irrelevance. There could be no organized summer student program at Nevis that year, and certainly no in-person ROOT tutorial. However, I decided to update the tutorial anyway.

The primary changes were to the intermediate topics:

- I moved more optional material into this section, so it would be less distracting for those going through the tutorial page-by-page, but still be available as a reference.
- I offered more detailed options for installing ROOT and Jupyter on your laptop.

2019

Included a new "intermediate topics" section, to act as a reference for useful material that the students may not immediately need for summer research.

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 notebooks 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 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.

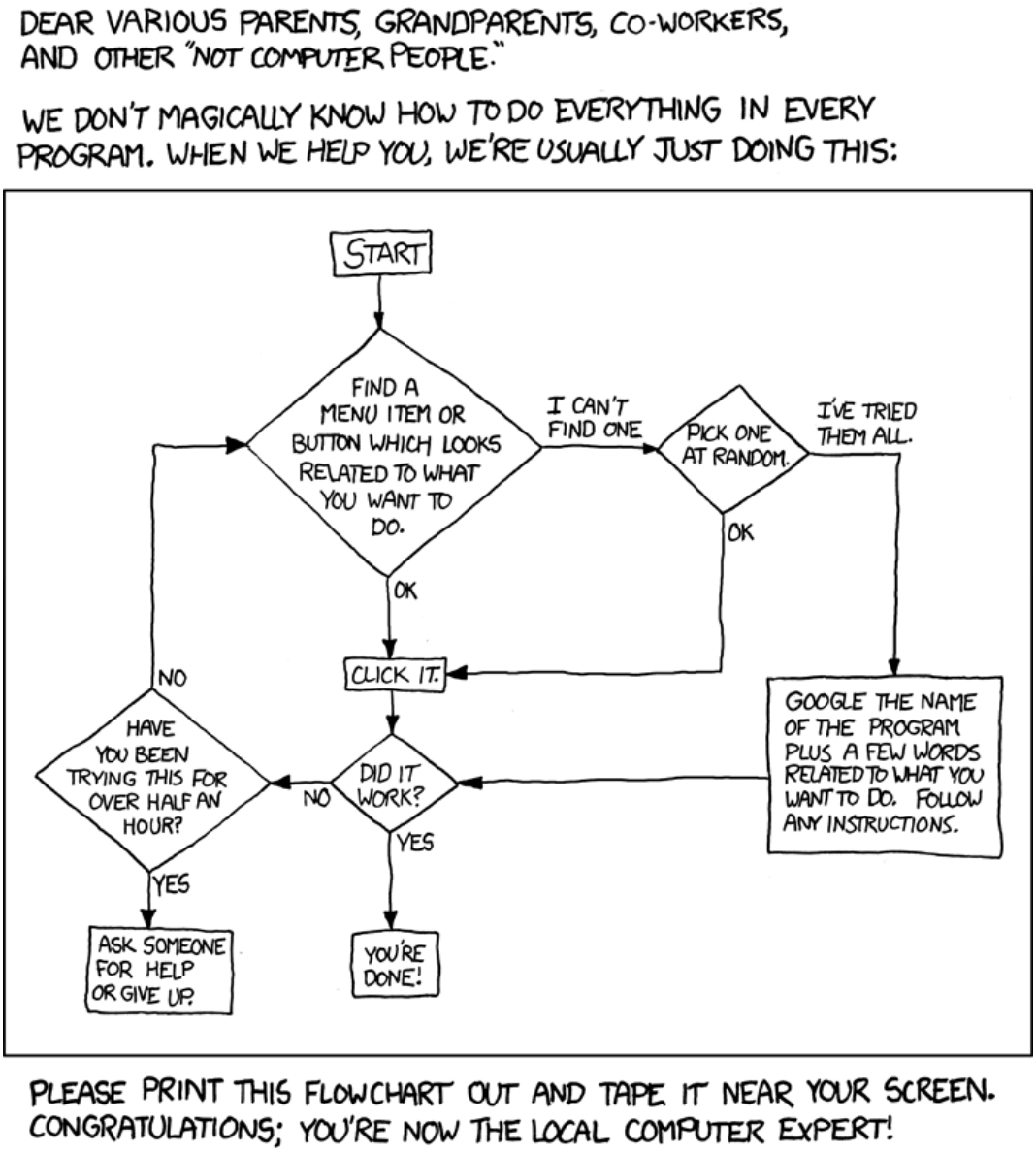

Figure 14.1:: http://xkcd.com/627 by Randall Munroe. Your reward for reaching the end of the very last page is to learn what I do most of the day.