MIT HEP Starter Kit

Yutaro Iiyama

March 18, 2015

Before you begin

The goal of the starter kit (`https://github.com/yiiyama/starterkit.git`) is to introduce the basic tools and ideas in high-energy physics analyses to intereted learners. We hope you enjoy the exercises and become comfortable with the basics by the time you are done!

A successful high-energy physics experiment requires an extremely broad range of expertise (hence the extraordinary collaboration size). It is motivated by the theory of particle physics, which is arguably one of the most esoteric field of study with no obvious practical application in human society. But to execute the experiment, one must also solve highly nontrivial civil engineering problems (e.g. drilling a vertical shaft while freezing the shaft wall to stop groundwater from flooding in), build superconductive magnets, cryogenics (magnets must be cooled down), and ultra-high vacuum, develop cutting-edge particle detectors and electronics, and finally know the statistical methods to interpret the data and write a software to implement those methods. This package is intended to help you get started with the last bit, i.e., writing the software for HEP analyses.

Please note that this document will contain mostly background information and not much of examples and solutions. Source code templates will be provided, but you are encouraged to modify and rewrite them as much as possible – if things cease to work, you can always recover the original file by
`$ git checkout HEAD` *`path_to_file`*
However, be careful when issuing above command, since the command removes all of the changes you made.

The discussion points in this document are there to bring up key points and ways of thinking. Some questions are answerable without HEP-specific knowledge, but some are not. The most important thing is to talk about these points, so come and ask frequently!

Requirements

You need a command-line terminal, an SSH program, and a text editor.

Installation

You have to first log in to one of the HEP cluster machines (MIT CMS tier-3 computing cluster):

```
$ ssh user_name@t3btch***.mit.edu
```

If your local user name is the same as your account name at T3, you can ommit the part `user_name@`. You need to work on a node that has Scientific Linux 6 installed as the OS. For now use `*** = 092`.

Once you are in one of the cluster machines, download the starter kit package:

```
$ git clone https://github.com/yiiyama/starterkit.git
```

Install script will execute the necessary initial setups:

```
$ cd starterkit
$ ./install.sh
```

This is a one-time procedure. No more installation is necessary in future sessions. However, you always have to set up the environment after each log in (including the first):

```
$ cd starterkit
$ source init.sh
```

Exercise 1. Plotting out distributions

We will familiarize ourselves with ROOT in this exercise. ROOT is a software library developed at CERN specifically for HEP analysis. It is written in `C++` and is intended to be mixed with user applications to create softwares for specific analysis purposes, as opposed to being used as a standalone program. Thus we will be writing `C++` codes using ROOT objects, and compiling them together with ROOT libraries.

Compilation can be done with `g++` or any other `C++` compiler by letting the compiler know where to find the ROOT source code and the pre-compiled ROOT binary libraries. Alternatively, one can make ROOT itself to compile your code using its internal compiler (which is actually `g++`):

```
$ root -q your_program.cc+
```

The plus sign at the end of the file name tells ROOT to compile the program (sometimes also called macro).

If you call ROOT without the plus sign, instead of compiling and creating a binary, ROOT will try to "interpret" your program line-by-line as a script written in `C++`. ROOT has a built-in interpreter named CINT to do this. CINT is fairly reliable but has some limitations (mainly because `C++` was never intended to be used as a scripting language), so this mode of operation is not recommended unless the macro is very simple.

If ROOT is called without any argument, it will enter the interactive mode, where command-line inputs are interpreted by CINT line-by-line. ROOT will enter this mode also after the macro execution if you call ROOT without the `-q` option. This mode is useful when e.g. making small changes to plots.

Another popular way to use ROOT functionalities is to import the ROOT module to python. We will see a lot of this later.

As a first exercise, start ROOT with an input file and open the graphical user interface (GUI) to inspect the content of the file:

```
$ root /scratch/yiiyama/starterkit/singlemuA.root
```

```
root [0] new TBrowser
```

On the left-hand side of the TBrowser, you will a folder icon with title `ROOT Files`. Under the folder is the file that has been just opened by ROOT. Double-click on the file icon, and you will see a folder-like icon saying `events` (If there are more than one icons, use the one with the highest number). This is a `TTree` object, often called "ntuples", which is something like a big spreadsheet. The rows of the spreadsheet, or entries in the ROOT terminology, are *events*, i.e., snapshots of the CMS detector output at each proton-proton collision (bunch crossing). In fact, this particular file contains events collected with a single-muon trigger during the first two months of 2012 CMS data-taking.

Discussions

- What is a trigger? What does it do, and why is it necessary?
- What is a muon? What are some of its key properties? How do we detect and reconstruct muons?

The columns of the spreadsheet are called branches in ROOT, and represent various measurable quantities in the detector. Since it is very difficult to deal with raw outputs of the detector electronics, ntuple files like the current one often contain highly processed quantities which are ready for physics analysis. Double-click on the `events` icon and you see that this ntuples contain branches for momentum (p_x, p_y, p_z, E) of four kinds of objects (photon, electron, muon, and jet) plus some other variables that will be

explained shortly. Note that these are reconstructed *objects* and are not necessarily true photons, electrons, muons, and jets. We never know with 100% certainty what particle caused specific detector response, but by combining information from various parts of the detector, we can guess what came out of the interaction point.

Discussions

- What is a jet?
- How are different objects reconstructed? How are momenta of the objects measured?

Double-click on the `photon.n` branch. This branch holds the number of photon objects reconstructed in the event. The distribution of this variable appears on the right-hand side of the window. The first bin of the histogram corresponds to 0. As you can see, majority of the events do not have a photon object.

Now double-click on the `photon.pt` branch. The plot shows the distribution of the transverse momentum $p_{\mathrm{T}} := \sqrt{p_x^2 + p_y^2}$ in GeV of all photons in the event. Transverse momentum is one of the most important variables in hadron collider experiments. Bring your mouse close to the x axis until the pointer changes to a hand icon, then click at 0 and drag up to 100 GeV. You will see that the first bin of the histogram has no entry. This is because experimentally we cannot reconstruct photons to arbitrary low energy; we only count photons with $p_{\mathrm{T}} > 10\,\mathrm{GeV}$ in CMS.

Discussions

- Why is transverse momentum important, when p_z and E are also measureable?

A finer control on the distributions can be achieved from the command line. On your terminal, enter

```
root [1] events->Draw("photon.pt>>histo_photon_pt(100, 0.0, 100.0)")
```

Here, `histo_photon_pt` is the name of the histogram object you just created, with a binning of 100 bins from 0 to 100 GeV. We will look later at how to handle histogram objects.

Exercise 2. Looking at muons

Double-click on the `muon.n` branch. Here you see that there are no entries with 0 *or* 1 muons. This is simply because I selected only the events that

contain at least two reconstructed muon objects from the original dataset; such reduction of data with specific criteria is often called a skim. So what you have now is "a dimuon skim of single-muon triggered dataset". Now plot the muon pt distribution from 0 to 100 GeV from command line.

Discussions

- In what way(s) does the muon p_{T} distribution look different from the photon p_{T} distribution?
- Why is/are there this/these difference(s)?

As already mentioned, the branch variables are quantities from all reconstructed objects. In the reconstruction step, identification of particles are done rather conservatively, i.e., tracks and calorimeter clusters that are due to hadrons are identified as photons, electrons, and muons. The idea is that the latter three particles are typically more interesting for physics, so we try to pick up every detector hits that can potentially be due to those particles. We then apply identification selections to the reconstructed objects to reduce the fraction of hadron "fakes". You can tune the selections depending on whether you want to keep as many objects as possible paying the price of larger hadronic fakes (high efficiency, low purity) or have only the objects that you are very confident about their genuineness at the price of throwing away some "true" particles because of low-quality reconstruction (low efficiency, high purity). Such choices are specific to individual physics analyses.

The selection "cuts" typically use values of multiple variables. In the current ntuples, the selections are already run and the results are stored in the tree as boolean (true or false) "flag" branches. For muons, `muon.isTight` and `muon.isSoft` branches represent the results of applying two different selection criteria. "Soft" criteria is looser, which results in more muon objects passing the cuts. Check this by plotting out pass/fail (shown in the plot as 1/0) of the flags.

Next, plot the p_{T} of the muons that pass the selection criteria. On your terminal, enter

```
root [2] events->Draw("muon.pt>>histo_muon_pt(100, 0.0, 100.0)",
"muon.isTight")
```

The second argument to the Draw function (which is empty by default) specifies which muons should be used to make the histogram. Try with the `isSoft` flag too.

You can overlay the results of two `Draw` commands for comparison.

```
root [3] events->Draw("muon.pt>>histo_muon_pt_soft(100, 0.0, 100.0)",
"muon.isSoft", "same")
```

The keyword same in the third argument tells ROOT to draw the new histogram (which must have a name different from the previous one) on the same "canvas". To further make the comparison easier, we can change the line color, line style, fill color, etc. of the histograms.

```
root [4] TH1* soft = gDirectory->Get("histo_muon_pt_soft")
root [5] soft->SetLineColor(kRed)
```

The first line retrieves the histogram object that was created in the memory and creates a handle (variable `soft`) for it. The second line calls the function that sets the line color on this object. The line color of the soft muon distribution should now change to red once you bring the focus back to the TBrowser window.

Discussions

- What is the difference in the p_{T} distributions for two different selections?
- From this observation, what can you say about the two criteria?

Quit root by

```
root [6] .q
```

Exercise 3. Going beyond `TTree::Draw`

If anything more complicated than just plotting the distributions of variables possibly with some simple selections is desired, one must write a `C++` code. The `C++` code version of the last example in the last exercise is written in `muonpt.cc`. Run it with

```
$ root muonpt.cc+
```

As you can see in the file, this version requires quite a bit of overhead before reaching the body of the program, which is to loop over the entries in the ntuples. On the other hand, you have full control of what to do with the branches as promised, since all branch values can be assigned to standard `C++` variables.

Discussions

- Add p_x and p_y variables to the code and plot the p_{T} calculated for each muon from these two variables. Confirm that the result is identical to what you get from simply plotting the p_{T} variable.

Exercise 4. Dimuon invariant mass

Starting from the `muonpt.cc` example, write a code that plots the distribution of the invariant mass of all muon pairs found in each event.

Discussions

- Why is invariant mass a more interesting quantity than, say, the energy of the dimuon system?
- How do we loop over pairs of muons?
- How do we implement the calculation of invariant mass?