ROOT Basics

First log in to a NAF working group server according to your group. For instance, if you work on ATLAS, type '`ssh -XY nafhh-atlas01`', or similarly for CMS, FLC, etc. (If the username on your laptop is different from your DESY username, you need to type `ssh -XY username@nafhh-atlas01` instead).

To start ROOT type `root -l` into the command line. The `-l` flag suppresses the splash screen. Try typing the command without the flag if you enjoy pointless logos and waiting for five minutes.

To quit root type `.q`. If root crashes (and it will) you can exit by typing the usual linux kill commands such as `ctrl+c` and `ctrl+z`.

All ROOT classes are documented, as well as with many tutorials, on the ROOT website:

`http://root.cern.ch`
`http://root.cern.ch/root/html534/ClassIndex.html`

You can also google your problems. Many people will have encountered the same issues before and often you can find a quick solution. Otherwise, ask your supervisor or email the summer student list.

Tutorial Files

The files for the tutorial are stored in a git repository:

https://github.com/artlbv/DESY-root-tutorial

You can inspect the contents of the package using a browser if you want. To download the package, type:

```
> git clone https://github.com/artlbv/DESY-root-tutorial.git
```

Inside the package you will find a readme file with some links to more information on git, and a directory called `ROOT_Tutorial_2015` containing a root file and a cxx file called `skeleton_macro`. The skeleton macro can be run directly without compiling. To run it type:

```
> root -l
root[] .x skeleton_macro.cxx
```

You should see a printout of "Hello World" if this worked correctly. If you want to open a file in ROOT quickly, you can pass it as an argument when opening ROOT:

```
> root -l MC.root
```

You can also quickly inspect files using the TBrowser, simply start ROOT and type:

```
root[] TBrowser tb
```

The TBrowser is fairly self-explanatory and works as you would expect most file browsers to work. Take a few minutes to explore the file structure and some of the variables in the file.

TFile Basics

The TFile is the basic input and output medium of ROOT. In it, one can store standard c++ objects (such as doubles, ints and floats) as well as any class that derives from TObject (such as TH1, TGraph, TTree).

Opening a TFile

To open a TFile you must create a pointer to it (ROOT loves pointers, most are unnecessary but get used to them). The syntax for the command looks something like this:

```
> TFile* myFile = new TFile( ‘‘file_name.root’’ , ‘‘OPTION’’ );
```

The name of the file can include or not include the full directory path. If not path is specified then the file is either created or read from the current working directory. The three main options that you will need are:

- **“READ”** Opens a file for reading only. You can retrieve objects but not write them. This is the safest options since you’ve can’t break the file this way or accidentally overwrite something.
- **“WRITE”** Opens an already existing file so that you can write to it. You probably won’t use this option very often.
- **“RECREATE”** Overwrites an already existing file with a new one. If the file does not already exist then it is created. This is the most common command when writing out your own information.

For example, to open the file MC.root so you can access what is inside, you would type:

```
> TFile* myFile = new TFile( ‘‘MC.root’’ , ‘‘READ’’ );
```

If you want to make a new file to store some histograms you’ve just made, you would type:

```
> TFile* newFile = new TFile( ‘‘hists.root’’ , ‘‘RECREATE’’ );
```

Retrieving Objects

To get an object from a TFile you can use the `Get()` method. Since c++ is rather stupid when it comes to things like this, you will have to remind it what sort of object it is that you’re trying to retrieve (i.e. you must cast it):

```
> TH1* histogram;
> histogram = (TH1*)myFile->Get("histogram_name");
```

As an example, inside the file MC.root there is a histogram named "mini_cutflow_ee". To access it, you would type: `> TFile* myFile = new TFile( ''MC.root'' , ''READ'' );`
`> TH1* histogram;`
`> histogram = (TH1*)myFile->Get("mini_cutflow_ee");`

You could then for instance draw the histogram with `> histogram->Draw()`, or do other things with it. **WARNING** If you have opened a TFile to retrieve an object, but not longer need the TFile, it is normally good practice to close the file using the method `Close()`. However, ROOT has many subtleties about when objects go in and out of scope. If you suddenly find that your histograms are empty, or encounter a strange seg fault, it might be because of this. We wont go into detail today but if you have a problem like this then send an email to the help list.

Writing to files

Eventually, you will want to save something such as a TTree or histogram. ROOT considers and open file to be similar to a directory. As long as you are inside that directory, you can write any object to that file using the `Write()` method. ROOT will write to the most recently opened file.

Exercise

1. Using the skeleton macro, open the TFile provided and retrieve any histogram object.
2. Draw the histogram.
3. Now try adding the `Close()` command to the TFile after you have retrieved the histogram but before you draw it and see what happens.
4. Try writing the histogram to a new file you create.

Histogram Basics

Histograms are the primary tool used by physicists to represent information. In ROOT, this is accomplished using the `TH1` classes. Multiple classes exist depending on the type of object stored in them; `TH1F` for floats, `TH1D` for doubles etc. In almost all cases you `TH1F` is perfectly fine.

Histograms in ROOT are just like any other c++ objects. They have constructors, member variables and various methods. They can also be operated upon in a basic way, such as addition, subtraction, and multiplication.

Constructor

The two most commonly used constructors are:

```
> TH1F* myhist = new TH1F(histogram_name, histogram_title, nbins, low, high)
> TH1F* myhist = new TH1F(histogram_name, histogram_title, nbins, bin_array)
```

The first constructor takes the `histogram_name` (which is usually the same string as myhist), the histogram title (which will the name that appears in the output root file), the number of bins, the lowest bin edge, and the highest bin edge. In this case ROOT will create nbins of equal width between xlow and xhigh. The second constructor allows you to specify each bin edge (in case you want variable bin sizes). Note that in this case, nbins is always one less than the length of the `bin_array`.

Filling

Filling a histogram is done using the `hist->Fill(value)` method. You can also include a weight (which is often very useful when dealing with MC) by specifying the weight as a second argument to the method `hist->Fill(value,weight)`. If you plan on scaling your histograms (and you probably will at some point) and in particular if you plan to divide one histogram by another, it is important to make sure that you have the following line in your macro `TH1::SetDefaultSumw2(True)`. This ensures that the errors in your histogram are calculated correctly.

Drawing

You will spend a lot of time trying to make your histograms look pretty, but drawing them is very simple. A histogram is drawn to a canvas using the `hist->Draw()` method. It should just pop up on your screen. There are numerous things that can be done with the draw method and we leave you to explore these on your own. The ROOT reference guide has many details on the topic and you should read it as soon as possible.

Histograms in ROOT are drawn on a TCanvas. If you do not draw a TCanvas on your own, when you type `hist->Draw()` ROOT will automatically create a TCanvas named "c1" for you. If you want to save your histogram to a graphics file, you need to "print" the TCanvas that it's drawn on:

```
> c1->Print("myHist.pdf");
```

Root can print to most common formats (pdf, eps, png). ROOT will determine the appropriate format based on the string you provide (i.e. if you put .pdf it will print in pdf).

Exercise

1. In the skeleton macro, create a histogram with 10 bins between -1 and 1.
2. Fill the histogram with a random Gaussian (there is an example already in the macro on how to do this).
3. Print the histogram to a pdf file (you can view it using the unix command `display`).
4. Using the Draw options (look on the ROOT website) print the histogram again but this time with error bars.
5. Find out how to change the color of the histogram (look at the class documentation on the ROOT website).

TTrees

TTrees are the main way in which LHC physics data is stored and accessed for user-end physics analysis. Each "entry" in a TTree corresponds to a physics event (either in data or MC). For each event many different types of information can be stored in what are called "branches". If you want to see what information is inside a TTree you can either inspect it with TBrowser or, when using interactive ROOT, you can simply type:

```
root[] myTree->Print();
```

and a list of all the available branches and their types will appear in the window. (hint: TTrees are retrieved from TFiles in exactly the same way that you would retrieve a histogram).

Setting Up Branches

Once you have your TTree you are probably going to want to loop over the events. In order to do this, you must first declare some variables that are going to store the information that you want. Let's say you want to store the number of jets per events. First, you need to declare a suitable object (an int in this case) that is going to hold your information:

```
> UInt_t my_jet_n;
```

Next you need to tell your TTree that you want the information from it's jet multiplicity branch to go into this object each time you look at an event:

```
> myTree->SetBranchAddress("jet_n",&my_jet_n);
```

Now the TTree knows that each time you look at an event it should put the information stored in it's `jet_n` branch for that event, into the `my_jet_n` object. This also means that each time you look at an event, the information in the `my_jet_n` object will be overwritten. Better save it in a histogram! All that is left to do now is to loop over the events:

```
> for(int i = 0; i < myTree->GetEntries(); ++i)
>     myTree->GetEntry(i);
>     myHist->Fill(my_jet_n);
```

The second line tells the TTree to fill the objects with what is stored in the branches for event number `i`. Sometimes you will be working with large TTrees that have many branches. This can take a long time to loop over (not so much for `c++` but certainly in pyROOT). If you only want to take the information from a few branches then you can turn the other branches off and speed up your execution time:

```
> myTree->SetBranchStatus("*", 0);
> myTree->SetBranchStatus("branch_you_want", 1);
```

All types of standard `c++` objects can be stored in TTrees, as well as ROOT objects. Often these will be in the form of arrays or vectors (see examples).

Projection and Drawing

Sometimes we just want to quickly look at the data inside a TTree. This can be accomplished with the TBrowser (not covered in this tutorial) or through the `Draw()` command. Learning to use `TTree::Draw()` can save you a lot of time. Let's say that we want to see what the jet p_{T} spectra looks like, but we don't want to spend time writing a macro and setting up the branches. Well, you can do this with the `Draw()` command very easily in interactive ROOT:

```
root[] myTree->Draw("jet_pt");
root[] myTree->Draw("jet_pt[0]");
root[] myTree->Draw("jet_pt","jet_n > 2");
root[] myTree->Draw("jet_pt","","PE2");
```

Each entry in the tree is an event, and each event may have many (or no) jets, so the jet p_{T} is stored as a vector of floats. The first line above will draw the p_{T} of any jet in any event, while the second line will only draw the jet p_{T} for the first jet in each event. The third line plots the jet p_{T} of all jets, but only for events with at least two jets present. There's a lot you can do with the `Draw()` command and it's worth taking a look at the ROOT documentation and asking you supervisor.

Exercise

1. Get the `"mini"` TTree from the ROOT file in the tarball. How many entries ("events") does it have?
2. Use `Print()` to see what branches are inside the TTree.
3. Using the skeleton macro, loop over the tree and fill a histogram with the lepton p_{T} information. Print the histogram with a unique name. Note that like jets, there may be more than one lepton in an event.
4. Try the same again, only this time use interactive ROOT (not the macro). Does the histogram look the same? If not, do you know why?
5. In the macro, turn off all branches except the lepton p_{T} branch. Does the macro run any faster?
6. Make the lepton p_{T} plot again (either in the macro or interactively), but this time only for leptons with $\eta > 1.0$.