Kirchhoff-Insitut für Physik
Physikalisches Insitut

Winter semester 2013-14
KIP CIP Pool (1.401)

Exercises for Statistical Methods in Particle Physics

http://www.physi.uni-heidelberg.de/~nberger/teaching/ws13/statistics/statistics.php

Dr. Niklaus Berger (nberger@physi.uni-heidelberg.de)
Dr. Oleg Brandt (obrandt@kip.uni-heidelberg.de)

Exercise 0: Introduction to ROOT

14. October 2013

This tutorial should provide you with a very basic introduction to the `root` data analysis framework. It comes in two different flavours, namely in the native built-in C++ interpreter called CINT and in Python; maybe you already have a preference, if not, please give both a try and see what works best for you. Note that while CINT can interpret C++, it is not require full C++ compatibility, with all advantages and disadvantages resulting from it.

First, make sure you have your environment set up correctly, see
`http://www.physi.uni-heidelberg.de/~nberger/teaching/ws13/statistics/documentation.php`.
Also create a separate folder for your exercises in this course and go to that folder.

1 CINT: native ROOT C++ interpreter

Open a `root` session in a shell by typing

```
> root
```

(where the `>` symbol stands for your terminal prompt. This should start `root`. If not, check your environment again... You now have the `root` C++ interpreter running, so you can enter almost any valid C++ code. Or just use `root` as a calculator:

```
root[0] 1 + log(2)
```

The object in `root` we will use most is the histogram, so let's create one:

```
root[1] TH1F myhisto("myhisto","My most fabulous histogram",10,0,100)
```

This creates a histogram (of type `TH1F`), to which ROOT will refer by its name ("`myhisto`") and which will be displayed with the title given. It has ten bins of equal size, spanning the range [0, 100]. Now we can fill some values into the histogram (`root` will return the corresponding bin number, if you do not like this, terminate the statements with a semicolon):

```
root[2] myhisto.Fill(42)
root[3] myhisto.Fill(3.141592)
root[4] myhisto.Fill(66)
root[5] myhisto.Fill(99)
root[6] myhisto.Fill(69)
root[7] myhisto.Fill(17.7)
```

Now let us draw the histogram:

```
root[8] myhisto.Draw()
```

This opens a *canvas*, the surface `root` draws on. By default, the canvas will be named `c1`. In the canvas window, you can open the *Editor* from the view menu. It allows you to change the style of various objects on the canvas (which you can also move around with the mouse); be aware though that there is no undo function. All the things that can be set from Editor, can also be set from the command line, e.g.

```
root[9] myhisto.SetLineColor(2)
```

for the change to become visible, you have to draw the histo again:

```
root[9] myhisto.Draw()
```

If you want the histogram to be drawn with error bars, try

```
root[9] myhisto.Draw("E1")
```

If you do not want to rewrite all the code every time you start root, you can use macro files, i.e. files with programme code (in this case CINT-macros). They end in .C and can be executed directly from root. Make it a habit to start them with a comment stating the exercise they are for and your name and email. Create a file exercise0.C in your exercise directory and fill it similar to the following:

```
// exercise0.C: Example root programme for the root tutorial
//
// Nomen Nominandum, 14.10.2013
// nomen.nominandum@stud.uni-heidelberg.de

void fillHistogram(unsigned int nentries){
  TH1F * histo = new TH1F("histo","Another histogram",10,0,100);

  for(unsigned int i=0; i < nentries; i++){
       histo->Fill(fmod(i*777,100));
  }
}
```

Note that we create the histogram on the heap using `new`, to make sure it does not go out of scope when the function returns. In `root`, we can now load the macro file using

```
root[10] .L exercise0.C
```

(all `root` comments start with the "."). Now we can call the function like any other:

```
root[11] fillHistogram(100)
```

Let's now draw our new histogram:

```
root[12] histo->Draw()
```

We can also draw the old histogram on the same canvas

```
root[13] myhisto.Draw("same")
```

be aware that CINT, the `root` C++ interpreter, does not really differentiate between the `.` and `->` member access, which is one of its drawbacks. Compiled C++ of course does, so better do it right from the start. Now let us save the histograms to a file

```
root[14] TFile* file = new TFile("exercise0.root","RECREATE")
```

and now we can write out the histograms and close the file

```
root[15] myhisto.Write()
root[16] histo->Write()
root[17] file->Close()
root[18] delete file
```

Now try to read the histograms back in:

```
root[19] TFile* fileagain = new TFile("exercise0.root","READ")
root[20] TH1F * histoagain = (TH1F*)fileagain->Get("histo");
```

You retrieve objects from `root` files using their name (usually the first parameter in the constructor) with the `Get()` function, which will always return a pointer to a `TObject`, the base class of everything in ROOT. You have to manually cast to the type you are expecting (here a `TH1F` pointer). This is not particularly safe and cumbersome... Draw your re-loaded histogram

```
root[21] histoagain->Draw()
```

and then call it a day, you can quit root with

```
root[22] .q
```

Note that you can execute root macros directly from the command line by including them as an argument, i.e.:

```
root -l exercise0.C
```

2 And the same again using Python

Start a Python session using

```
> python
```

then import the `root` module. There are several ways of doing this, probably the cleanest is

```
>>> import ROOT
```

which imports everything from `root`, but leaves it in its own namespace, ROOT. Python can of course also serve a s a calculator:

```
>>> import math
>>> 77-math.sqrt(17)
```

Now let us create a `root` histogram

```
>>> myhisto = ROOT.TH1F("myhisto","My phytonesque histogram",10,0,100)
```

filling then works just as in the plain `root` case

```
>>> myhisto.Fill(42)
>>> myhisto.Fill(42)
>>> myhisto.Fill(17)
>>> myhisto.Fill(7)
...
```

then we can draw it

```
>>> myhisto.Draw()
```

Note that on the CIP pool machines, PyROOT uses a slightly older version of root with lots of ugly default settings. So use the editor of the canvas (in the view menu) to get rid of the grey backgrounds and the red frame. Of course you can again also do this from the command line:

```
>>> myhisto.SetLineColor(2)
>>> myhisto.Draw("E1")
```

where we have also switched on the error bars. Of course you can also write Python code into files (ending in .py), try something like the following in a file called exercise0.py:

```
# exercise0.py: Example PyROOT programme for the root tutorial
#
# Nomen Nominandum, 14.10.2013
# nomen.nominandum@stud.uni-heidelberg.de

import ROOT

histo = ROOT.TH1F("histo","My faboulous histogram",10,0,100)

def histofill(entries):
    for x in range(entries):
     histo.Fill(x)
```

You can now load that file as you would any module

```
>>> import exercise0
```

which puts everything from the file into the `exercise0` namespace. Thus

```
>>> exercise0.histofill(1000)
>>> exercise0.histo.Draw()
```

will fill and draw the histogram. Of course, drawing the old histogram on top will also work

```
>>> myhisto.Draw("SAME")
```

Now we can safe the histograms to a `root` file:

```
>>> f = ROOT.TFile("exercise0_python.root","RECREATE")
>>> myhisto.Write()
>>> exercise0.histo.Write()
>>> f.Close()
>>> del f
```

Reading back in is now syntactically much more elegant than in the plain `root` case:

```
>>> fagain = ROOT.TFile("exercise0_python.root","READ")
>>> histoagain = fagain.histo
>>> histoagain.Draw()
```

You can quit the python interpreter by pressing `CTRL + D`.
This should have given you a first glance at `root` and PyROOT, you should be able to use those skills in the first exercise. More material can be found on the course website.