Steven Schramm
(Université de Genève)

ROOT Tutorial (C++)

General introduction

This tutorial consists of three exercises. You may not finish all of them during this single three-hour session. In that case, I would encourage you to finish it later at home! These three exercises cover three very common ROOT usage scenarios:

1. Processing data from a `TTree`, filling a histogram, and writing the results to an output file
2. Reading a file that contains multiple histograms and interpreting the results, writing the final plots to a pdf file
3. Reading a file that contains a histogram and fitting the histogram in different ways, writing the results to a pdf file

The second and third step depend on the results from the first step. However, I have provided the final result of the first step in a file named "selection.root". If you get stuck on the first exercise, you can therefore use the provided result to work on the second and third exercises.

Throughout the exercises, you should be able to run the program after every step. It might not do anything visible or produce any results, but you should be able to check that it at least compiles and runs without errors at each step.

Preparation

You should each have a USB that contains both a small version of the dataset (roughly 50 MB in total) and a large version of the dataset (roughly 1 GB in total). The small version should be in a folder named "sample", while the large one should be in a folder named "full". **I strongly encourage you to use the small version of the dataset for all of these exercises**, as then it should take roughly one second to run your code. Once you are complete, you can run over the large version, which may take substantially longer than one second to run. However, this way you can see what adding more events does in particular to the quality of the fit in Exercise 3.

In order to make it easy to run your programs, **I strongly suggest that you copy the small files locally**. That is, copy them from the USB stick (or the internet) into a directory that you have created, and then write all of your macros/scripts in that same directory. This way you can avoid worrying about having long paths pointing to the files that you want to run over. If you take them from the HASCO timetable online, then the two files are simply named "Data.root" and "MC.root". The full versions totalling roughly 1 GB are not available to be downloaded - you must take those from the USB if you want to use them.

Checking your results

Sample solutions to each exercise are provided online. Details are below.

- Exercise 1:
  - Input files: `Data.root`, `MC.root`
    - Thanks to ATLAS and CERN Open Data for the data and simulation (which I then filtered)
  - Output ROOT file: `selection.root`
  - Solution code (ROOT): `selection.cpp`
  - Solution code (PyROOT): `selection.py`
- Exercise 2:
  - Input file: `selection.root`
  - Output plot file: `analysis.pdf`
  - Solution code (ROOT): `analysis.cpp`
  - Solution code (PyROOT): `analysis.py`
- Exercise 3:
  - Input file: `selection.root`
  - Output plot file: `fit.pdf`
  - Solution code (ROOT): `fit.cpp`
  - Solution code (PyROOT): `fit.py`

Exercise 1: Processing data

If you are working with ROOT, then the odds are you will need to work with reading simulated or real data from trees at some point. The `TTree` is the primary ROOT storage object which handles the interface between *transient* information (variables that you have in the computer memory) and *persistent* information (variables that you have written to your hard drive). If you ever work with data from an experiment or Monte Carlo (MC) simulation, you will likely be using files that are stored in the form of a `TTree`.

The idea of a `TTree` is that each *event*, corresponding to an interaction of interest, is independent from all other events. For example, there is no correlation between what happens in two subsequent beam crossings at the LHC. However, every event should have a given set of information that describes it. At the most basic level, this may be the electronic signals associated with a given part of a detector, or the PDG ID of a particle in MC simulation. This set of potential information should be the same for every event, and thus the variables that you need to describe an event are fixed. The `TTree` provides such functionality, by defining a `TBranch` to represent each variable, and where the value of that `TBranch` is updated each time you tell the `TTree` to go to the next event.

The `TTree` therefore provides a detailed representation of an event. On the other hand, it is possible to calculate *summary information* that describes the properties of the ensemble of all events of interest. This is typically done using histograms, all of which derive from the base class `TH1` (T for a ROOT object, H for histogram, and 1 for one-dimensional). There are also two and three dimensional histograms (`TH2` and `TH3`), as well as more specialized histogram types.

One of the most fundamental tasks is to read a full `TTree`, representing the detailed description of a given set of data, and using that data to fill a set of histograms representing the features of interest. Once this is done, it is useful to write these histograms to an output file for further study or to make publication-quality plots. This process is the focus of the first exercise: reading a `TTree` from a file, using it to fill a `TH1`, and writing that histogram to a new output file.

The final result that you are aiming to achieve is provided as the file `selection.cpp`. Please try to avoid using that file directly. Instead, the below steps will guide you towards obtaining that result, and you should only look at the relevant part of the solution if you get really stuck or if you are finished and want to compare to my version.

- Step 1: create the basic structure of a ROOT macro
  - ROOT macros should have a function with the same name as the file. If the file is named "myFunction.cpp", then the function should be named "myFunction"
  - Create the macro with something like the below to get started.

```
int myFunction()
{
  std::cout << "This is my function" << std::endl;
  return 0;
}
```

  - Run the macro using ROOT:

```
root -l "myFunction.cpp()"
```

  - For other examples of how to run a ROOT macro, see yesterday's lecture slides
- Step 2: define the macro arguments
  - We want this macro to run over a file and write the results to a new file. As such, we want two arguments, one for each file name.
  - Change the function to reflect this behaviour, similar to the below.

```
int myFunction(std::string inFileName, std::string outFileName)
{
  std::cout << "Reading from "    << inFileName  <<
               " and writing to " << outFileName << std::endl;
  return 0;
}
```

  - Run the macro using ROOT (note the escaped quotation marks to specify string values within external quotation marks):

```
root -l "myFunction.cpp(\"Data.root\",\"hist.root\")"
```

  - For other examples of how to run a ROOT macro, see yesterday's lecture slides
- Step 3: open the input file for reading
  - Once we have the file name, it is time to open the file

- ROOT handles interactions with files by using the `TFile` class
- Note that ROOT doesn't know what a `std::string` is, so you need to convert it back to an old-format string
- Open the file using:

```
TFile* inFile = TFile::Open(inFileName.c_str(),"READ");
```

- The first argument is the name of the file and the second is what we want to do with the file. In our case, we just want to read from it, not write to it.

- Step 4: retrieve the `TTree` from the file
  - After opening the file, we want to get the `TTree` from it. This is done by using "Get" on the file and telling it that the resulting object is a `TTree`. This last step is needed because the "Get" method can actually return other objects, as we will see in the next exercise.
  - The name of the `TTree` in the file is "HASCO"
  - Get the `TTree` using:

```
TTree* tree = (TTree*)inFile->Get("HASCO");
```

- Step 5: declare the variables that we want to read from the file
  - Now we have the `TTree` that we want to use, but we need to specify the variables in the tree that we want to access
  - There are several variables that we will want to use, and we need to define them all in advance
  - In particular, we are retrieving information related to the leptons present in the event. The file was pre-selected for muons, so by leptons in this case we actually mean muons.
  - Any given event may have any number of muons from 0 to much larger values. As such, the information is stored in arrays. Arrays have the unfortunate feature of being fixed-size, meaning that we have to say how large the array is before we know how many objects there are. We also need it to be large enough for any event. Therefore, we will decide that there are up to 100 leptons at most in an event, and only read out the first 100 in case there are more. This is then used to create the arrays, as below:

```
const unsigned int lep_n_MAX = 100;
unsigned int lep_n;
float pT[lep_n_MAX];
float eta[lep_n_MAX];
float phi[lep_n_MAX];
float nrg[lep_n_MAX];
```

  - After defining all of the variables that we want to use, we have to link them to the `TTree`, thus specifying that these variables are where we want each event to be stored. This is done as below:

```
tree->SetBranchAddress("lep_n",&lep_n);
tree->SetBranchAddress("lep_pt",&pT);
tree->SetBranchAddress("lep_eta",&eta);
tree->SetBranchAddress("lep_phi",&phi);
tree->SetBranchAddress("lep_E",&nrg);
```

  - In the above, the first part in quotation marks is the name of the branch in the `TTree` that we want to access, and the second value is the name of the variable where we should store the information. In this case I gave them the same name, but this is not necessary. The first part must be exactly the name used in the `TTree`. You can just use the strings provided above as they are correct, but if you want to see how to check the names that a `TTree` contains, then you can see the lecture slides on how to print out the contents of a `TTree`.

- Step 6: create the histogram that we want to fill
  - We want to look at the di-lepton invariant mass. This is a floating point quantity, so we want either `TH1F` (F for float) or `TH1D` (D for double). Let's use a double-precision histogram in this case. We want to plot this quantity by itself, not with respect to another quantity, so we only need a one-dimensional histogram.
  - Let's create a 1D histogram. Note that the variables in the tree are in units of MeV, but the LHC is high energy and thus everything will be in the thousands of MeV (aka in GeV). As such, we will need to create bins in the GeV range.
  - Let's create 150 bins, starting from 50 GeV (50×10^3 MeV) and ending at 200 GeV (200×10^3 MeV):

```
TH1D mll("data","m_{ll}, data",150,50.e3,200.e3);
mll.Sumw2();
```

- The last part is configuring the way that we want to handle statistical uncertainties in the histogram. Most of the time, you want uncertainties to be calculated using the sum of weights squared, or Sumw2. That is also true in this case.

- Step 7: loop over the `TTree` entries
  - We now have a tree, a histogram, and the variables we want to use in the tree. The next step is to actually loop over all of the events in a tree.
  - The number of entries in a `TTree` in ROOT is stored as a special very long integer type that comes with ROOT
  - Loop over the events in the `TTree` as follows:

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  tree->GetEntry(entry);
}
```

- Step 8: calculate the pairs of lepton four-vectors for each event and the di-lepton invariant mass
  - Each time GetEntry is called, all of the variables we defined are overwritten with new values for the new event. We can now check that there are two leptons in the event and make their four-vectors, then use that to calculate the di-lepton invariant mass.
  - We have to check that there are actually two leptons, as we can't calculate a di-lepton mass if there is only one lepton in the event. If there are not two leptons, then we should skip this event (*continue* to the next event).

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  if (lep_n != 2)
    continue;
}
```

  - After we have confirmed that there are two leptons, we can build their four-vectors and store them as `TLorentzVector` objects, which contain lots of useful functions.

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  TLorentzVector lepton0, lepton1;
  lepton0.SetPtEtaPhiE(pT[0],eta[0],phi[0],nrg[0]);
  lepton1.SetPtEtaPhiE(pT[1],eta[1],phi[1],nrg[1]);
}
```

  - We now have four-vector representations of the two leptons. However, we want the di-lepton system, which is the four-vector sum of the two leptons. Then, we want to store the mass of that system.

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  TLorentzVector dilepton = lepton0 + lepton1;
  double dileptonMass = dilepton.M();
}
```

- Step 9: fill the histogram
  - We now have the di-lepton mass, so we want to fill the histogram. This is the last thing that we need to do in the loop.

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  mll.Fill(dileptonMass);
}
```

- Step 10: change the histogram scope/lifetime
  - Once the histogram has been filled, we want to make sure that it doesn't disappear. By default, histograms are linked to the last file that was opened, so when you close the file the histogram may disappear.

- We want to change this behaviour, to say that the histogram we just created has no parent file, and thus should not be removed when any files are closed.

```
mll.SetDirectory(0);
```

- Step 11: close the input file
  - Now that we have set the histogram to have a global scope, we can safely close the input file.
  - It is a good practice to close files when we are done using them to help with memory management and to avoid multiple file access problems.

```
inFile->Close();
```

- Step 12: write the histogram to the output file
  - Now that we have filled the histogram and closed the input file, we want to open a new file to store the results. We will open it using the "RECREATE" mode, which means that we are opening it for writing AND if a file with the specified name already exists, we are ok to overwrite it with our new file.

```
TFile* outHistFile = TFile::Open(outFileName.c_str(),"RECREATE");
```

  - After opening the file, we need to switch our scope to being within that file

```
outHistFile->cd();
```

  - Now we can write our histogram (and as many other histograms as we want) to the file

```
mll.Write();
```

  - Finally, we have to close the output file, which is what triggers it to be written to disk rather than living in memory

```
outHistFile->Close();
```

- Step 13: run the program
  - At this point, the program is complete! Hopefully you have been running it throughout the steps to ensure that nothing was going wrong (syntax errors or similar).
  - Make sure to run the program now. If everything went well, you should now have an output file with the name that you specified as the second argument to the program.
- Step 14: check the output file
  - Now that we have made a file, we should check that it actually has done what we wanted. Does it contain a histogram? Does that histogram look reasonable?
  - Let's open the output file and open it with a `TBrowser`

```
root -l hist.root
```

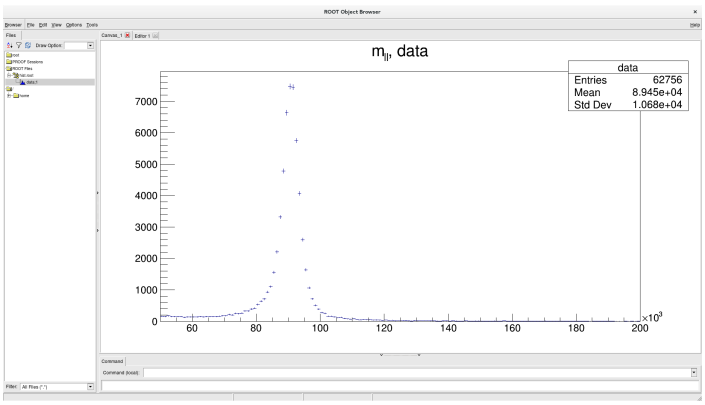
  - You should see messages like the following:

```
root [0]
Attaching file hist.root as _file0...
(TFile *) 0x3234d10
root [1]
```

  - Now, open the `TBrowser`:

```
TBrowser t;
```

  - This will give you a window with the file listed. Double click on the file, then double click on the name of the histogram. You should then see the histogram, and it should look something like this:



- You've found the Z-boson! Good job!

- **NOTE - this is the end of the main part of this exercise, while the below is adding more complex functionality. If you are short on time, you can download the file named "selection.root" and use that for the next two exercises.**

- Step 15: extend the program to handle both data and MC
  - Normally in physics, we want to compare our data with our simulation, representing the Standard Model expectation
  - For example, is the peak we saw in the last part representative of the Z, or is it too small or too large?
  - To do this, we need our program to be able to run over both data and MC
  - Steps 3 through 11 can be placed in a function (read a file, fill a histogram, close the file, and return the histogram)
  - This function can then be used for both data and MC
  - Add one more argument to the macro (so that the user can specify the names of both the data and MC input files)
  - Once this is done, you can run the program again and get two histograms in the output file (make sure to name them differently).

- Step 16: add weights for the MC
  - The MC might be different than the data, either because of how it was generated or because of non-perfect modelling of physics effects. The former effect is generally addressed with a MC weight, while the later is addressed by the use of scale factors. We have this information in the `TTree` as well, and need to declare them as extra information that we want to use.

```
float mcWeight;
float sf_PILEUP;
float sf_MUON;
float sf_TRIGGER;
```

  - After defining these additional variable, we have to link them to the `TTree` just like before.

```
tree->SetBranchAddress("mcWeight",&mcWeight);
tree->SetBranchAddress("scaleFactor_PILEUP",&sf_PILEUP);
tree->SetBranchAddress("scaleFactor_MUON",&sf_MUON);
tree->SetBranchAddress("scaleFactor_TRIGGER",&sf_TRIGGER);
```

  - We then have to calculate the total weight to apply, which is just the product of these four values.

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  double weight = 1;
  if (isMC)
  {
    weight = mcWeight * sf_PILEUP * sf_MUON * sf_TRIGGER;
  }
  // Later code in the loop is here
}
```

  - Then, when we are filling the histogram, apply this weight:

```
for (Long64_t entry = 0; entry < tree->GetEntries(); ++entry)
{
  // Previous code in the loop is here
  mll.Fill(dileptonMass,weight);
}
```

  - Now your MC has been corrected to more correctly model the expected data distribution.

- That's it! You're done exercise 1. You should now have a ROOT file that contains two histograms, one for the MC and one for the data.

Exercise 2: Data analysis – plotting

Your thesis will certainly involve some plots, and many (or all) of them will likely be made using ROOT. Plots are the way in which you convey your scientific results to the larger community, and it is very important that they are clear and easy to understand. Your plots may also be shown independent of their accompanying captions, such as in conference talks. As such, it is important that any key information required to understand the plot is listed directly on the plot in the form of labels. In this exercise, we will go through the process of creating a nice publication-style plot.

- Step 1: create the basic structure of a ROOT macro, with arguments
  - ROOT macros should have a function with the same name as the file. If the file is named "myPlotter.cpp", then the function should be named "myPlotter"
  - Create the macro with something like the below to get started.

```
int myPlotter(std::string histFileName, std::string plotFileName)
{
  std::cout << "Reading from "    << histFileName  <<
               " and writing to " << plotFileName << std::endl;
  return 0;
}
```

  - Run the macro using ROOT:

```
root -l "myPlotter.cpp(\"selection.root\",\"plots.pdf\")"
```

  - The input ROOT file should be the one produced at the end of Exercise 1, or the one that I have provided for you.
  - For other examples of how to run a ROOT macro, see yesterday's lecture slides
- Step 2: open the input histogram file for reading
  - Once we have the file name, it is time to open the file
  - ROOT handles interactions with files by using the `TFile` class
  - Note that ROOT doesn't know what a `std::string` is, so you need to convert it back to an old-format string
  - Open the file using:

```
TFile* histFile = TFile::Open(histFileName.c_str(),"READ");
```

  - The first argument is the name of the file and the second is what we want to do with the file. In our case, we just want to read from it, not write to it.
- Step 3: retrieve the two histograms from the file
  - Just like in exercise 1, where we were retrieving the `TTree`, we will use the Get function. This function returns the object of the requested name, assuming that it exists. If you are using my input file, then the names of the two histograms are "data" and "MC".

```
TH1D* dataHisto = (TH1D*)histFile->Get("data");
TH1D* mcHisto   = (TH1D*)histFile->Get("MC");
```

  - It is also important to check that we retrieved the histograms, as we don't want the program to crash if we get the name of the histogram wrong.

```
if (!dataHisto)
{
  std::cout << "Failed to get data histogram" << std::endl;
  return 1;
}
if (!mcHisto)
{
  std::cout << "Failed to get MC histogram" << std::endl;
  return 1;
}
```

- Step 4: set histogram scope and close the input file
  - Once the histograms have been retrieved from the file, we want to make sure that they don't disappear. By default, histograms are linked to the last file that was opened, so when you close the file the histogram may disappear.

- We want to change this behaviour, to say that the histograms we just retrieved should live beyond their parent file, and thus should not be removed when any files are closed.
- Then, close the input file.

```
dataHisto->SetDirectory(0);
mcHisto->SetDirectory(0);
histFile->Close();
```

- Step 5: prepare to save plots to an output file
  - Any time you draw a plot in ROOT, you are using a `TCanvas`. There is a default canvas, but you can't manipulate the default canvas so easily. Instead, it is useful to define your own canvas and use that instead.
  - We have to give the `TCanvas` a unique name, but it doesn't have to be anything original. Below, we have named it simply "canvas".
  - We also have to specify that we want to use the canvas, which is the second command below.

```
TCanvas canvas("canvas");
canvas.cd();
```

  - Once we have a canvas, we can also set it to be written out to a plot file. In particular, pdf and ps files are very useful because they can have multiple pages (unlike eps or png). We can therefore write several plots to the same file.
  - If we want to write several plots to the same file, then we can use the below command. The name of a file followed by "`[`" tells ROOT that we want to print several plots to the same file. The same command without the "`[`" will write the current canvas to the file and that's it, so that's what you would use for eps, png, or similar.

```
canvas.Print(Form("%s[",plotFileName.c_str()));
```

  - The above is a fancy way to use the std::string and add a "`[`" to the end. If `plotFileName` is "plots.pdf", then the above is equivalent to:

```
canvas.Print("plots.pdf[");
```

  - After you have drawn all of your plots (see below), the very last command you should use is the one to close the plot file. Without this, your plots may not be readable, as the program will exit without actually writing the plots from memory to disk. This is done using a closing bracket, "`]`".

```
canvas.Print(Form("%s]",plotFileName.c_str()));
```

  - Again, if `plotFileName` is "plots.pdf", then the above is equivalent to:

```
canvas.Print("plots.pdf]");
```

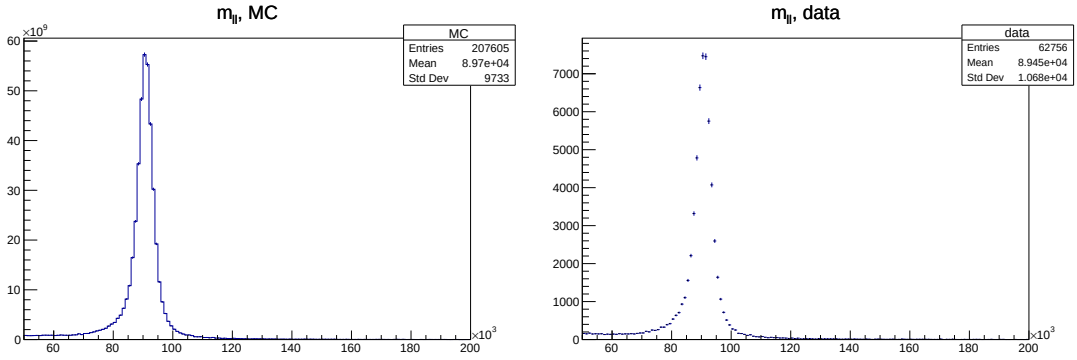
- **NOTE - for all of the below steps, the new code should be added between the opening and closing commands listed in the previous step. That is, all plotting commands should be after the opening of the plot file, and before the closing of the plot file.**
- Step 6: draw the two histograms and write them as two separate plots to the same pdf file
  - Draw the MC histogram as a **h**istogram (connected lines with vertical transitions at each bin boundary) and then write it to the output file:

```
mcHisto->Draw("h");
canvas.Print(plotFileName.c_str());
```

  - Now draw the data histogram as **p**oints with **e**rror bars and write it to the output file:

```
dataHisto->Draw("pe");
canvas.Print(plotFileName.c_str());
```

  - If you now run the macro, you should get a file with two plots:



- **NOTE - whenever you are changing the style of a plot, you have to add the new code to modify the plots before the relevant Draw and Print commands.**
- Step 7: fix the basic style
  - The statistics box in the top right corner of each plot is not useful for these plots. Let's remove it.

```
mcHisto->SetStats(0);
dataHisto->SetStats(0);
```

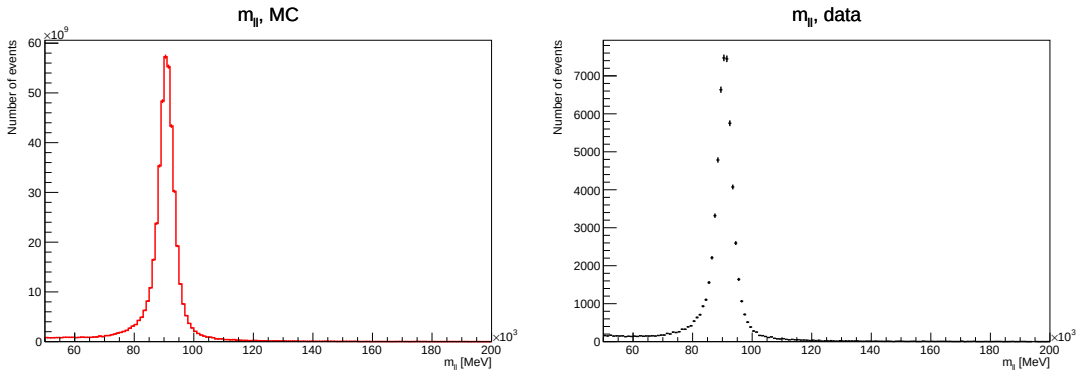
  - Let's also change the colours of the two histograms so that they can be easily told apart. Let's also increase their width from "1" to "2" such that they are thicker and easier to see.

```
mcHisto->SetLineColor(kRed);
dataHisto->SetLineColor(kBlack);
mcHisto->SetLineWidth(2);
dataHisto->SetLineWidth(2);
```

  - Finally, let's add axis labels to the plots. You may know what your plot shows, but if you are showing it to someone else, they will not understand the plot without axis labels. This should really be done for every single plot you make. Even if you make a plot and know what it is now, when you go to use it a few months or years later in your thesis, you will probably forget the specific details.

```
mcHisto->GetYaxis()->SetTitle("Number of events");
dataHisto->GetYaxis()->SetTitle("Number of events");
mcHisto->GetXaxis()->SetTitle("m_{ll} [MeV]");
dataHisto->GetXaxis()->SetTitle("m_{ll} [MeV]");
```

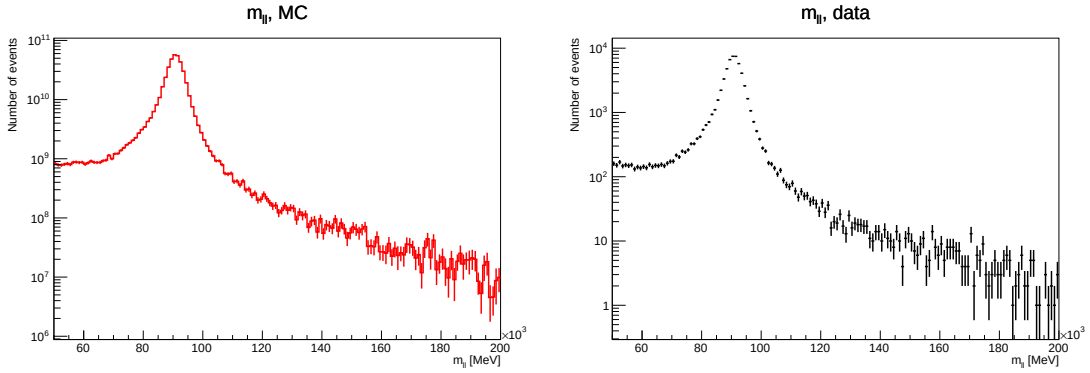
  - If you now run the macro, you should get a file with two plots:



- Step 8: change to a logarithmic view
  - The plots are very sharply peaked, which means that it is hard to see anything other than the peak. We can change this by using a logarithmic y-axis, which is done by changing the canvas (not the histogram).

```
canvas.SetLogy(true);
```

  - Now if you run the macro, you should get a file with two plots:



- Step 9: normalize the MC histogram
  - If you look at the plots, you will see that the y-axis has a very different range. The y-axis of the MC plot seems to be roughly 7 orders of magnitude larger than the data y-axis!
  - There are several reasons for this, but for now, the main point is that this is an artificial difference. As such, we want to normalize the MC histogram to data to remove this offset.
  - We don't want to normalize it in a way that makes the distributions agree bin-by-bin, as they we could never search for new physics or measure the standard model properties. Instead, we want to just get the overall scale correct. One way to do this is to scale by the total number of events in data vs the number of events in MC. This is done using the integral of the histograms (which is the total number of events, as the y-axis is just the number of events in a given bin).

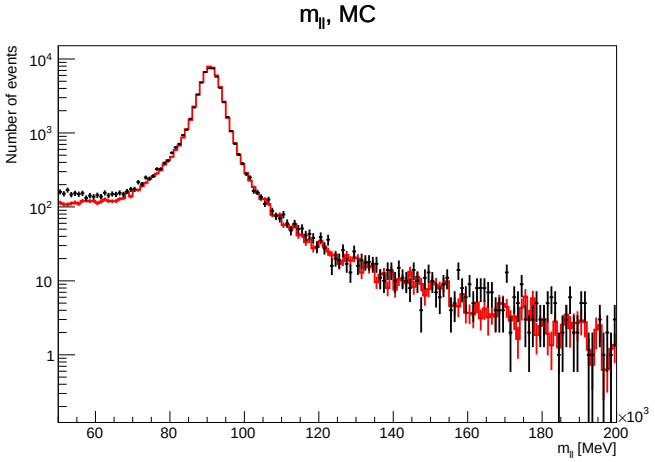
```
mcHisto->Scale(dataHisto->Integral()/mcHisto->Integral());
```

  - Now if you run the macro, you should get a file with the same two plots as before, but now the y-axis range for the MC and data are essentially identical (up to roughly 10^4 events).

- Step 10: draw the data and MC on the same plot
  - Now that we have set the histograms to the same scale, we can compare them in a sensible manner. Let's start by drawing them on the same plot (a third plot, created after the previous two).
  - We are drawing the MC histogram with solid lines and the data as points, so we should draw the MC histogram first. This way, we can see points on top of solid lines where the two histograms overlap, instead of the solid lines hiding any points underneath.
  - The "same" option is what tells ROOT to draw a given object on top of the previous one(s).

```
mcHisto->Draw("h");
dataHisto->Draw("pe,same");
canvas.Print(plotFileName.c_str());
```

  - Now if you run the macro, you should get a file with three plots, where the third plot is as below.



- Step 11: Make a split-panel plot
  - There is reasonably good agreement between the data and MC. To quantify how good the agreement is, we may want to take a ratio. However, we want to also see the non-ratio plot to understand the situation better, and so we can see whether we are in the peak or not. As such, we want a split-panel plot, with the current plot on top and a ratio on the bottom.

- First, let's prepare by creating a histogram which is the ratio of the data divided by the MC. Do this by creating a copy of the data histogram (to avoid modifying the original one) and dividing that histogram by the MC.

```
TH1D* ratio = (TH1D*)dataHisto->Clone();
ratio->Divide(mcHisto);
ratio->SetLineColor(kRed);
```

- Next, we should clear the current canvas, as we are going to do something more complicated and don't want the old plot to be hiding in the background

```
canvas.Clear();
```

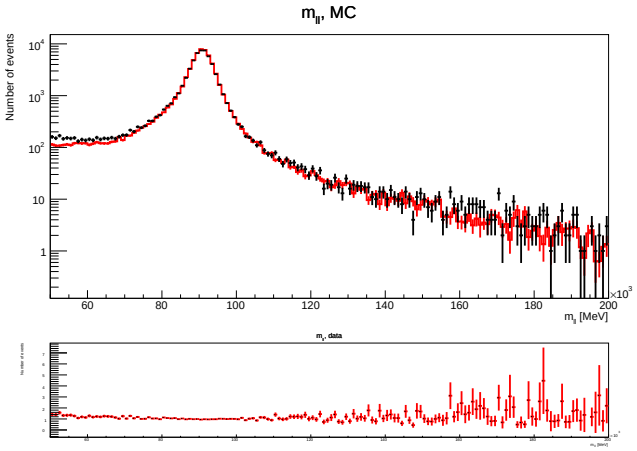
- Now, create a `TPad` (a sub-canvas) with a horizontal extent of the full range (0, y, 1, y) and a vertical extend of the upper 70% of the range (x, 0.3, x, 1). That is, this sub-canvas should be the upper 70% of the full canvas.
- Set the sub-canvas to have a logarithmic y-axis, draw the sub-canvas on the canvas, set the sub-canvas to be the owner of whatever is drawn next, and then draw the MC and data histograms (not their ratios) on this upper sub-canvas.

```
TPad pad1("pad1","pad1",0,0.3,1,1);
pad1.SetLogy(true);
pad1.Draw();
pad1.cd();
mcHisto->Draw("h");
dataHisto->Draw("pe,same");
```

- Now, go back to the full canvas and make a second sub-canvas covering the bottom 30% of the plot, and then draw the ratio on that sub-canvas. Note that this sub-canvas is a linear scale, not logarithmic, as this is a ratio.

```
canvas.cd();
TPad pad2("pad2","pad2",0,0.05,1,0.3);
pad2.Draw();
pad2.cd();
ratio->Draw("pe");
canvas.Print(plotFileName.c_str());
```

- With all of this done, if you run the macro, you should get the following fourth plot in the same file after the previous three. It's a nice plot, but it can definitely be improved.



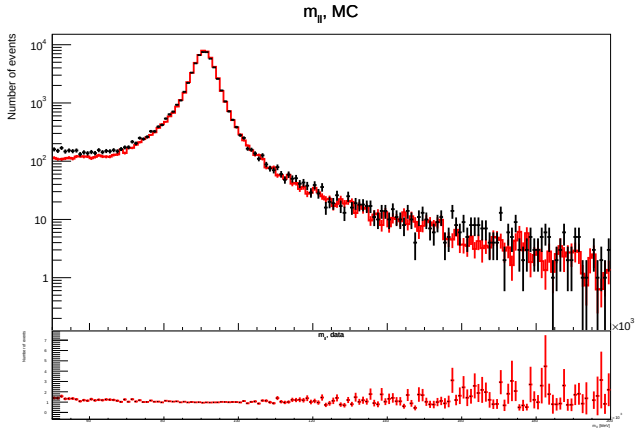
- **NOTE - whenever you are changing the style of a plot or pad, you must change the style before calling the Draw() command. This applies to both histograms and pads.**
- Step 12: Change the pad spacing to better use canvas space
  - First, remove the border spacing on the bottom of the top pad

```
pad1.SetBottomMargin(0);
```

  - Now, remove the border spacing on the top of the bottom pad. Also add a bit of extra space on the bottom of the top pad so that we have some space for later.

```
pad2.SetTopMargin(0);
pad2.SetBottomMargin(0.25);
```

– Once this is done, the upper and lower plots touch, which makes sense as you only need to have one x-axis label for the two plots (they are the same axis). You should now have a plot like the below.



- Step 13: adjust the size and contents of axis labels and markers
  - First, get rid of the x-axis labels on the top plot, this removing the random $\times 10^3$ on the far right of the plot

```
mcHisto->SetTitle("");
mcHisto->GetXaxis()->SetLabelSize(0);
mcHisto->GetXaxis()->SetTitleSize(0);
```

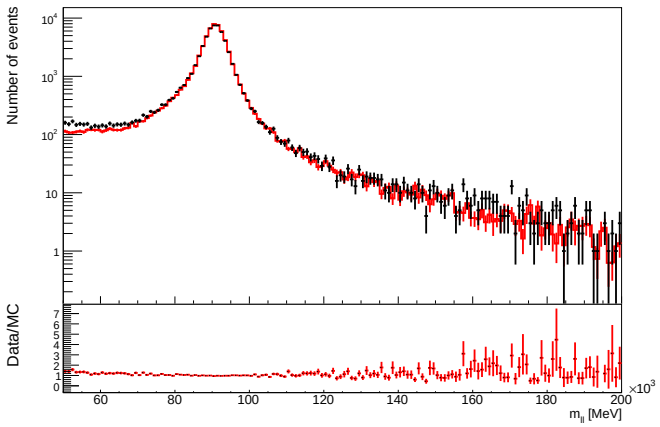
  - Then increase the size of the y-axis label, which is necessary because the pad is only 70% of the full vertical range, so we need to increase the text size to get back to a reasonable value.

```
mcHisto->GetYaxis()->SetTitleSize(0.05);
```

  - Now do the similar things to the bottom panel, also increasing the text sizes everywhere as this is now only 30% of the full vertical range. Note that the title offset is changed at one point to bring the label closer to the axis (so it doesn't go off the canvas).

```
ratio->SetTitle("");
ratio->GetXaxis()->SetLabelSize(0.12);
ratio->GetXaxis()->SetTitleSize(0.12);
ratio->GetYaxis()->SetLabelSize(0.1);
ratio->GetYaxis()->SetTitleSize(0.15);
ratio->GetYaxis()->SetTitle("Data/MC");
ratio->GetYaxis()->SetTitleOffset(0.3);
```

  - Once this is done, you should have a plot like the below.



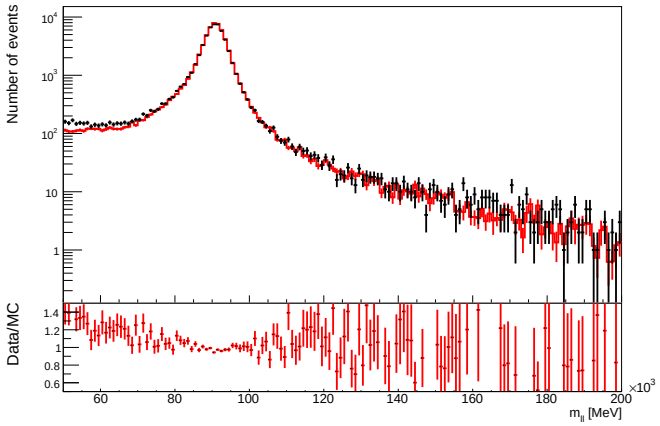
- Step 14: improve the ratio plot range and axis
  - The ratio plot right now isn't very useful, as there are very large uncertainties for large values of the di-lepton mass which are making the y-axis range quite large. Let's specify the range we want to see, focusing in on the range close to the value of 1, such that we can see where the two distributions agree or not. Let's use a range between 0.5 and 1.5.

```
ratio->GetYaxis()->SetRangeUser(0.5,1.5);
```

– Next, the y-axis labels are quite dense. Let's reduce this by manually specifying the number of divisions to use. See the `TAxis` documentation to understand how to use the `SetNdivisions` function.

```
ratio->GetYaxis()->SetNdivisions(207);
```

– Once this is done, you should have a plot like the below.



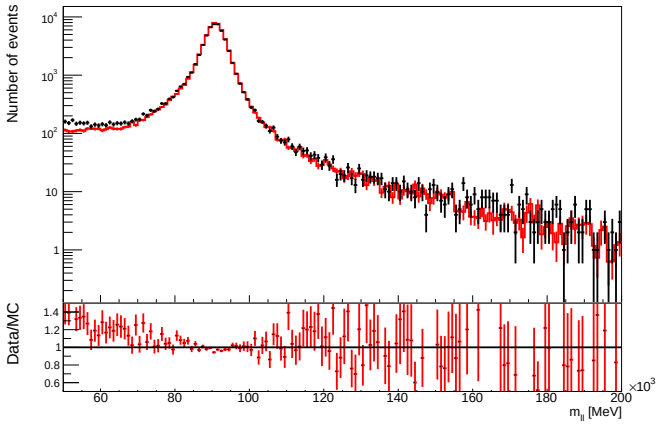
- Step 15: add a line at 1 on the ratio plot

– To guide the eye, it is useful to add a flat line at 1 on ratio plots. This is done using the `TLine` object, where you specify the (x_{start}, y_{start}, x_{end}, and y_{end}) coordinates of the line.

– Note that this should be created **after** you draw the ratio histogram. That is, we are adding a line **on top of** the ratio plot.

```
TLine line(50.e3,1,200.e3,1);
line.SetLineColor(kBlack);
line.SetLineWidth(2);
line.Draw("same");
```

– Once this is done, you should have a plot like the below.



– Now you can see that there is generally agreement for di-lepton masses above roughly 70 GeV (70×10^3 MeV) within the statistical precision. However, there is a significant difference at lower mass values. This is because the MC sample we have used is actually only for one process, $Z \to \ell\ell$, and is thus ignoring the Standard Model $\gamma \to \ell\ell$ process. That is why data is above MC, namely we are neglecting a real process present in data when calculating our MC expectation.

- Step 16: add a legend to the plot

– Right now, we know that the data is black and the MC is red. However, if we show this plot to someone else, they won't know this. It is thus very important to add legends to all of your plots.

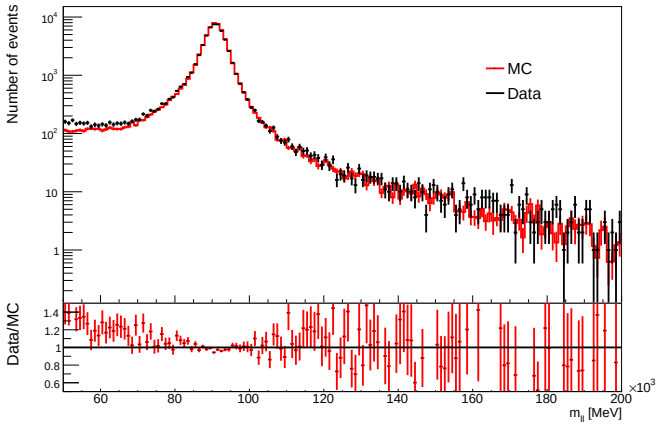
– Legends should be added **after** drawing all of the relevant histograms, adding a legend **on top of** the plot.

– We also set the width of the line around the box to 0 to remove it.

– Note that the coordinates that you use to specify the legend location are pad-global coordinates. That means that 0 is the bottom/left of the pad, 1 is the top/right of the pad, and any point in between is a fractional value. These numbers are relevant to the pad it is being drawn on, not the full canvas.

```
TLegend legend(0.7,0.6,0.85,0.75);
legend.AddEntry(mcHisto,"MC");
legend.AddEntry(dataHisto,"Data");
legend.SetLineWidth(0);
legend.Draw("same");
```

– Once this is done, you should have a plot like the below.



- Step 17: add other labels to the plot

– The plot is looking great now, but as mentioned earlier, people will look at plots without captions. As such, it is often useful to add additional labels to plots. This can be done using `TLatex`, which allows you to write text at arbitrary locations on the plot.

– Note that we use the `SetNDC()` function to specify that we want to use global coordinates, not specific points of a given plot. This is canvas-level, not pad-level. As such, the values for the legend and the values for the canvas are not the same even if you wanted to draw them at the same location.

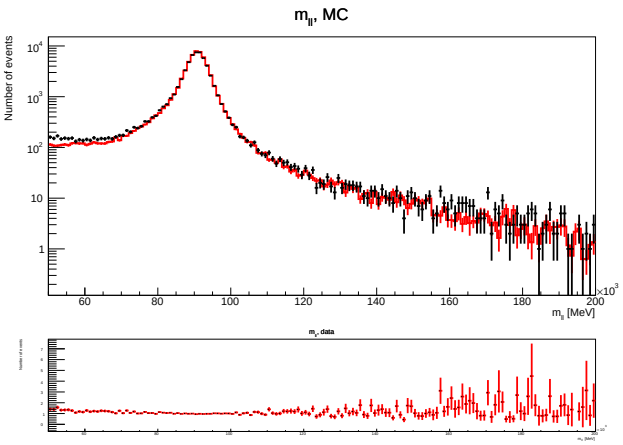
– In this case, you only need to specify the starting x and y coordinates, not also the ending coordinates.

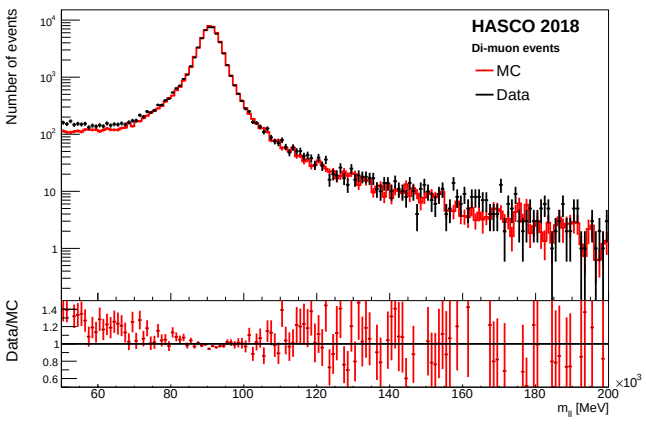
– Labels should be added after everything else, before writing the final plot to the output pdf file.

```
TLatex latex;
latex.SetNDC();
latex.SetTextSize(0.06);
latex.DrawText(0.7,0.83,"HASCO 2018");
latex.SetTextSize(0.04);
latex.DrawText(0.7,0.77,"Di-muon events");
```

- Step 18: admire your final plot

– As a reminder, here's a comparison with the first ratio plot with the final ratio plot





Exercise 3: Data analysis – fitting

We saw in exercise 2 that there is a sizable peak in the di-lepton mass spectrum, corresponding to the Z boson. If we wanted to quantify the properties of this peak, how would we do so? One way is by trying to fit a well-defined function to the peak in order to extract important parameters, such as its central value or width. This is important for far more applications than just fitting resonance peaks, but resonances are one clear example.

This exercise will contain a lot of plotting too. You should take what you have learned in the previous exercise and put it to use. I won't detail all of the reasons for why plotting commands are used in this exercise.

- Step 1: create the basic structure of a ROOT macro, open the file, read the data histogram, and close the file
  - Create the macro with something like the below. See the previous exercises for more details and motivation on the commands used.

```
int myFitter(std::string histFileName, std::string plotFileName)
{
  std::cout << "Reading from "    << histFileName  <<
               " and writing to " << plotFileName << std::endl;

  TFile* histFile = TFile::Open(histFileName.c_str(),"READ");
  TH1D* dataHisto = (TH1D*)histFile->Get("data");
  if (!dataHisto)
  {
    std::cout << "Failed to get data histogram" << std::endl;
    return 1;
  }
  dataHisto->SetDirectory(0);
  histFile->Close();

  return 0;
}
```

  - Run the macro using ROOT:

```
root -l "myFitter.cpp(\"selection.root\",\"plots.pdf\")"
```

  - The input ROOT file should be the one produced in Exercise 1, or the one that I have provided for you.
  - For other examples of how to run a ROOT macro, see yesterday's lecture slides

- Step 2: create the `TCanvas` and prepare the output file
  - Create the canvas

```
TCanvas canvas("canvas");
canvas.cd();
canvas.SetLogy(true);
```

  - Before drawing any plots, open the output file for multiple writing with the following command.

```
canvas.Print(Form("%s[",plotFileName.c_str()));
```

  - After you have drawn all of your plots, the very last command you should use is the one to close the plot file.

```
canvas.Print(Form("%s]",plotFileName.c_str()));
```

- Step 3: plot the basic histogram as our starting point
  - Write the basic plot to the file, so now we have a one-page pdf.

```
dataHisto->SetStats(0);
dataHisto->SetLineColor(kBlack);
dataHisto->SetLineWidth(2);
dataHisto->GetYaxis()->SetTitle("Number of events");
dataHisto->GetXaxis()->SetTitle("m_{ll} [MeV]");
dataHisto->Draw("pe");
canvas.Print(plotFileName.c_str());
```

- Step 4: fit a Gaussian to the peak

- Fitting Gaussians in ROOT is remarkably easy, and this is a very common task.
- The Gaussian function is pre-defined in ROOT. Create a `TF1` (T for ROOT, F for function, 1 for 1-dimensional) as below. The first argument is an arbitrary name that we need to give it, the second is the type of function where we are using the pre-defined "gaus" for Gaussian, the third is the lower boundary to use for the fit, and the fourth is the upper boundary to use for the fit.

```
TF1 gaussFit("gaussfit","gaus",81.e3,101.e3);
```

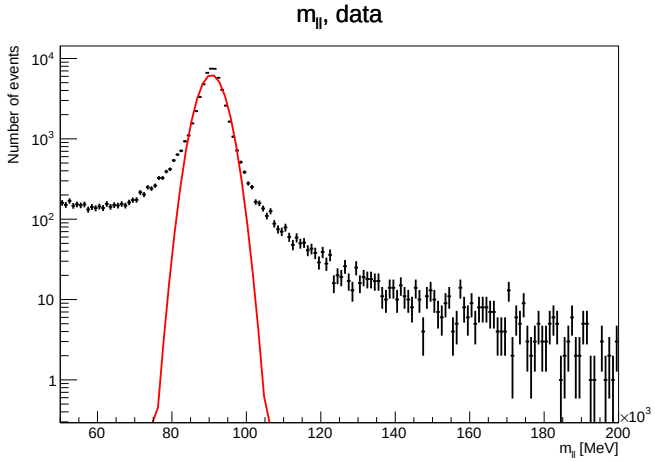
- Now, let's actually fit the Gaussian to the histogram. We are using the "E" option, which uses improved error handling. In general, you should always use this option unless you have a specific reason not to.

```
dataHisto->Fit(&gaussFit,"E");
```

- Finally, let's write out the plot (which includes the most recent fit):

```
dataHisto->Draw("pe");
canvas.Print(plotFileName.c_str());
```

- Now you should have a second page in your pdf file that looks like this:



- Step 5: adding the fit results to the plot
  - By eye it is already possible to tell that the fit isn't excellent, but it may still give us a reasonable result for the central value. Let's write out the quality of the fit (χ^2/N_{dof}) as a label on the plot, as well as the mean and width of the Gaussian fit.
  - First, create the `TLatex` as before

```
TLatex latex;
latex.SetNDC();
latex.SetTextSize(0.03);
```

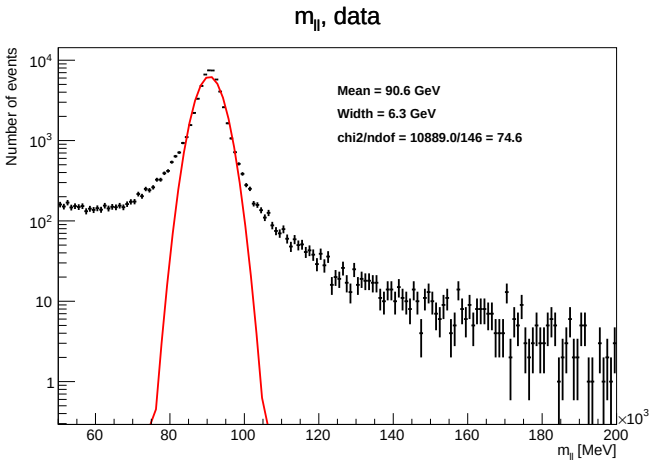
  - Then, get the χ^2, N_{dof}, mean, and width from the fit. Note that I have divided the mean and width by a factor of 1000 to convert from units of MeV to GeV.

```
double chi2 = gaussFit.GetChisquare();
int ndof = gaussFit.GetNDF();
double mean = gaussFit.GetParameter(1)/1000;
double width = gaussFit.GetParameter(0)/1000;
```

  - Finally, draw all of this on the plot (again, after drawing the histogram, but before writing the canvas to the output file)

```
latex.DrawText(0.5,0.80,Form("Mean = %.1f GeV",mean));
latex.DrawText(0.5,0.75,Form("Width = %.1f GeV",width));
latex.DrawText(0.5,0.7,Form("chi2/ndof = %.1f/%d = %.1f",
               chi2,ndof,chi2/ndof));
```

  - Note that in the above we have used `%.1f`, which is the notation to print only the first digit (1) after the decimal place (.) of a floating point number (f).
  - Now your second plot should look something like this:



- Step 6: fit a Breit-Wigner to the peak
  - We know that a Gaussian is not the correct function to model a resonance peak. We should instead be using a Breit-Wigner. This is not a pre-defined function, and so we must define it ourselves. It is also quite a complex function, so we need to be careful with our implementation.
  - When writing custom functions, "`x`" is the name of the variable, and "`[#]`" are the parameters that we want to fit. In other words, "`[0]`" is the first parameter, "`[1]`" is the second, and so on.
  - As a reminder, the Breit-Wigner equation is:

$$f(E) = \frac{k}{(E^2 - M^2)^2 + M^2\Lambda^2} \text{ with } k = \frac{2\sqrt{2}M\Gamma\gamma}{\pi\sqrt{M^2+\gamma}} \text{ and } \gamma = \sqrt{M^2(M^2+\Gamma^2)}$$

  - In other words, if we split this up into three pieces $f(E) = (A/B)/C$, then:

$$A = 2\sqrt{2}M\Gamma\sqrt{M^2(M^2+\Gamma^2)}, \quad B = \pi\sqrt{M^2+\sqrt{M^2(M^2+\Gamma^2)}}, \quad C = (E^2-M^2)^2 + M^2\Lambda^2$$

  - Converting this into the ROOT notation gives:

```
std::string bw_A = "2*sqrt(2)*[0]*[1]*sqrt([0]*[0]*([0]*[0]+[1]*[1]))";
std::string bw_B = "3.14159*sqrt([0]*[0] + sqrt([0]*[0]*([0]*[0]+[1]*[1])))";
std::string bw_C = "(x*x-[0]*[0])*(x*x-[0]*[0]) + [0]*[0]*[1]*[1]";
std::string bw   = Form("[2]*((%s)/(%s))/(%s)",
                        bw_A.c_str(), bw_B.c_str(), bw_C.c_str());
```

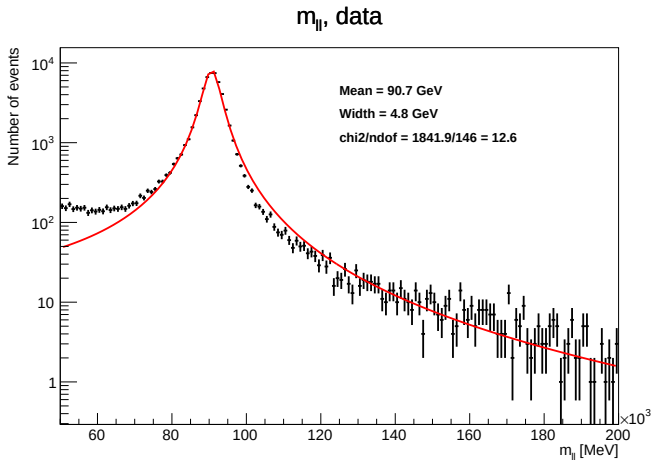
  - In the above, the final "`[2]`" is a normalization factor applied to the entire function.
  - Now, we create the actual fit function, but in this case we also specify starting points for two of the parameters to give it a hand (as the Breit-Wigner is much more complex than the Gaussian function)

```
TF1 bwFit("bwfit",bw.c_str(),50.e3,200.e3);
bwFit.SetParameter(0,100.e3);
bwFit.SetParameter(1,1.e3);
```

  - Now fit the Breit-Wigner to the histogram, write the results as plot labels, and write the plot to the output file.

```
dataHisto->Fit(&bwFit,"E");
dataHisto->Draw("pe");
chi2 = bwFit.GetChisquare();
ndof = bwFit.GetNDF();
mean = bwFit.GetParameter(0)/1000;
width = bwFit.GetParameter(1)/1000;
latex.DrawText(0.5,0.80,Form("Mean = %.1f GeV",mean));
latex.DrawText(0.5,0.75,Form("Width = %.1f GeV",width));
latex.DrawText(0.5,0.7,Form("chi2/ndof = %.1f/%d = %.1f",
                chi2,ndof,chi2/ndof));
canvas.Print(plotFileName.c_str());
```

  - Now you should have a third plot in the pdf that looks like this



– The Breit-Wigner is clearly a better fit than the Gaussian, but it's still not perfect. There are many reasons from this, from physics to detector effects, but now you have an idea of how you can fit both pre-defined (Gaussian) and user-specified (Breit-Wigner) functions to histograms.

- Bonus exercise:

  – If you have managed to complete everything so far, use what you learned from exercise 2 to make a split-panel plot of the data and its fit in the top panel, and the ratio of the data to its fit in the bottom panel. Try to get something that looks like the following.