Exercise1

July 22, 2016

1 Exercise 1: Search for H -> gamma gamma

1.1 Instructions:

- **Introduction and setup**
  - We want to perform a search for H -> gamma gamma using the invariant diphoton mass.
  - The H -> gamma gamma channel has a tiny branching ratio (10^-3) but is very pure and has a great mass resolution due to the distinct signature of two photons with good energy resolution.
  - Untar `Ex1.tar`, you will find three files `PseudoData_Histogram_100fb.root`, `Signal_1fb.root` and `Background_1fb.root`
- **Plot the data**
  - `PseudoData_Histogram_100fb.root` is the data we have measured corresponding to 100 inverse fb
  - Inside the data file is a TH1D histogram called `signal`, which shows the invariant diphoton mass, plot it.
- **Plot the background simulation**
  - `Signal_1fb.root` and `Background_1fb.root` are the signal and background simulations corresponding to 1 inverse fb.
  - Inside the simulation files a TTree is stored called `tree`, which contains two variables `invariantMass and eventWeight`, the invariant diphoton mass and the event weight, respectively.
  - Create histograms and fill them with `invariantMass` weighted by the `eventWeight`.
  - Scale the simulation to the correct integrated luminosity of the data and compare the data to the background-only hypothesis.
- **Background-only hypothesis test**
  - Perform a fit to the background in order to get a stable background model.
  - Compare the data with the background model by plotting the **difference** in a sub-plot below the main plot.
- ** Finalize the plot**
  - Add the signal simulation to the background histogram.

  - Make the plot look *nice*.

- **Optional**: Fit the signal to the difference plot and determine the signal strength.

We have measured some data, let's read it in:

```
In [1]: TFile *histoFile = new TFile("PseudoData_Histogram_100fb.root", "READ");
        TH1D  *hData     = (TH1D*) histoFile -> Get("signal");
```

Let's see how many events we have in the data:

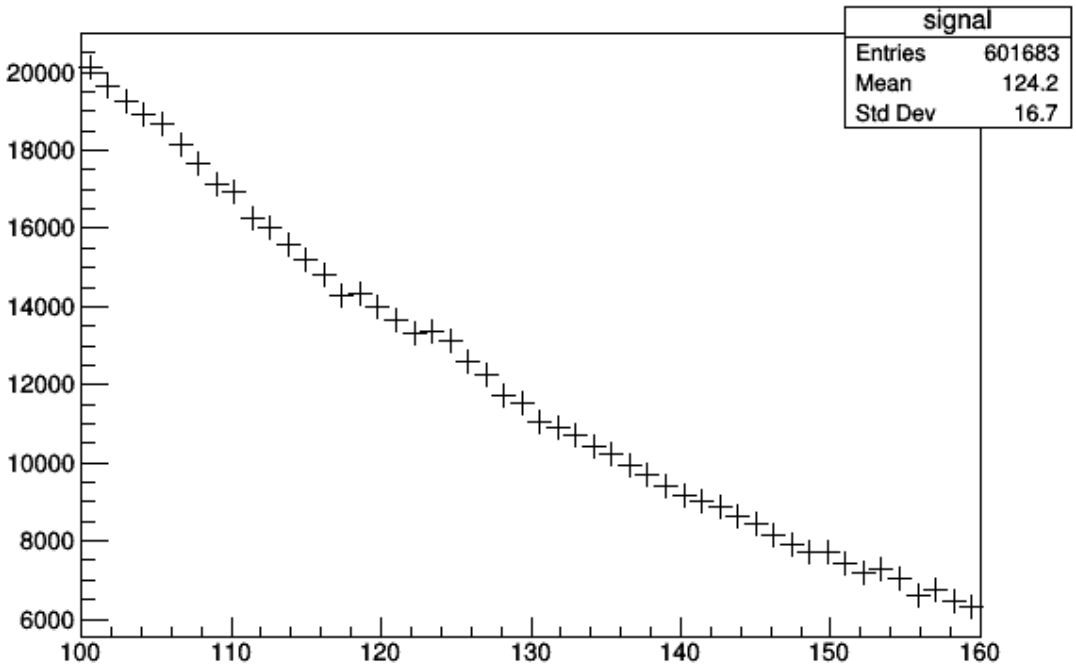
```
In [2]: std::cout << hData -> Integral() << std::endl;

601683
```

Now we want to have a look at it, so lets plot it! We will need a canvas for this, which we already divided into two pads:

```
In [3]: TCanvas *c = new TCanvas("c", "c", 600, 600);
        TPad *pad1 = new TPad("pad1","main",  0, 0.3, 1, 1.0);
        pad1->SetFillColor(0);
        pad1->Draw();

        TPad *pad2 = new TPad("pad2","ratio",  0, 0.05, 1, 0.3);
        pad2->SetFillColor(0);
        pad2->Draw();
        pad1->cd();
```

Now we can plot our data and draw the canvas!

```
In [4]: hData -> SetLineColor(kBlack);
        hData -> SetMarkerColor(kBlack);
        hData -> SetMarkerStyle(2);
        hData -> SetMarkerSize(1.5);
        hData->Draw();
        c->Draw();
```



Ok great, but how do we know if we see a signal in the data? We have a background and signal simulation, let's load the files and read in the trees:

```
In [5]: TFile *fSig = new TFile("Signal_1fb.root",     "READ");
        TFile *fBkg = new TFile("Background_1fb.root", "READ");

        TTree *tSig = (TTree*) fSig -> Get("tree");
        TTree *tBkg = (TTree*) fBkg -> Get("tree");
```

Inside the trees we see two variables:
`invariantMass and eventWeight`
Link the branch variables to some local variables!

```
In [6]: double mass_sig; double eventWeight_sig;
        double mass_bkg; double eventWeight_bkg;
```

```
        tSig -> SetBranchAddress("invariantMass", &mass_sig);
        tSig -> SetBranchAddress("eventWeight",   &eventWeight_sig);

        tBkg -> SetBranchAddress("invariantMass", &mass_bkg);
        tBkg -> SetBranchAddress("eventWeight",   &eventWeight_bkg);
```

Now define two histograms and fill them with the invariant mass weighted by the event weight. You will need to loop over the trees!

```
In [7]: // define two histograms
        TH1D *hSig = new TH1D("signal", "", 50, 100, 160);
        TH1D *hBkg = new TH1D("bkg",    "", 50, 100, 160);

        int nEntries_Sig = tSig -> GetEntries();
        int nEntries_Bkg = tBkg -> GetEntries();
```

Now loop over the trees:

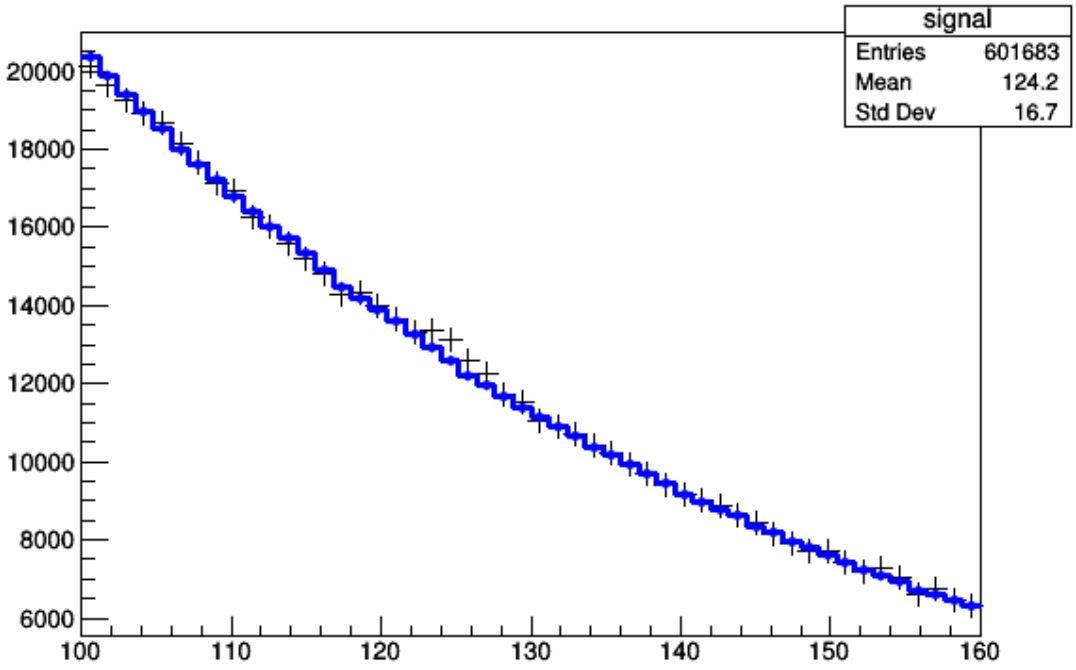
```
In [8]: for(int i = 0; i < nEntries_Sig; ++i){
            tSig -> GetEntry(i);
            hSig -> Fill(mass_sig, eventWeight_sig);
        }

        for(int i = 0; i < nEntries_Bkg; ++i){
            tBkg -> GetEntry(i);
            hBkg -> Fill(mass_bkg, eventWeight_bkg);
        }
```

The simulated events correspond to 1 inverse fb, however we have recorded 100 inverse fb, so we need to scale it

```
In [9]: std::cout << "Before reweighting: " << std::endl;
        std::cout << "Signal:\t\t" << hSig -> Integral() << "\n" << "Background:\t'

        hSig -> Scale(100.0);
        hBkg -> Scale(100.0);

        std::cout << "After reweighting: " << std::endl;
        std::cout << "Signal:\t\t" << hSig -> Integral() << "\n" << "Background:\t'

Before reweighting:
Signal:                 16.8333
Background:         6000
After reweighting:
Signal:                 1683.33
Background:         600000
```

Ok, great, that makes sense, since we had around 600k data events, lets plot the background model now:

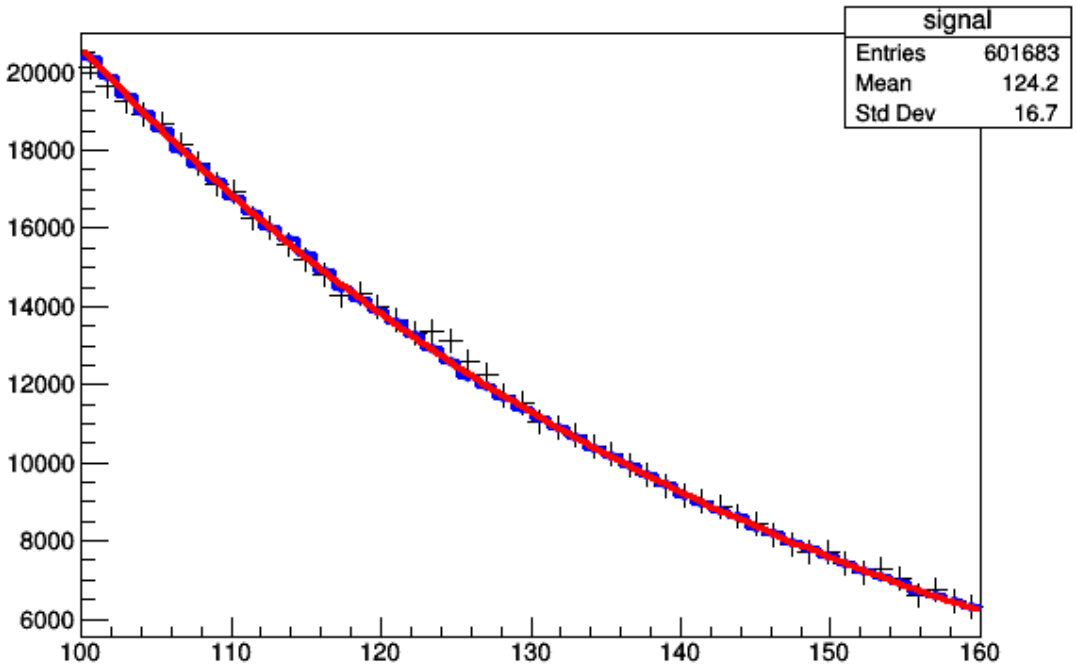
```
In [10]: hBkg -> SetLineColor(kBlue);
         hBkg -> Draw("HSAME");
         c ->Draw()
```



To neglect fluctuations, we want to fit the background now to get a good background model, it looks very exponential:

```
In [11]: TF1 *fit = new TF1("f1", "[0] + exp([2]*x+[1])", -1, 12);
         hBkg -> Fit(fit);
         fit -> SetLineColor(kRed);
         c->Draw();
```

```
 FCN=55.13 FROM MIGRAD    STATUS=CONVERGED      195 CALLS         196 TOTAL
                     EDM=9.21112e-08    STRATEGY= 1  ERROR MATRIX UNCERTAINTY   2.6
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           6.08110e+01   8.76850e+01  -2.05472e-01   3.02846e-05
   2  p1           1.19443e+01   1.22978e-02  -3.58586e-05  -7.02661e-02
   3  p2          -2.01293e-02   1.57061e-04   4.16136e-07   1.26766e+01
```



Great, if we now look at the difference between data und background we should be able to see a possible signal, let's create a new histogram in which we store the difference:

```
In [12]: TH1D *hDiff = (TH1D*)hData -> Clone(0);
         int nBins = hData->GetNbinsX();
```

We need to loop over the bins and calculate the difference between data and fit for each bin. This is how you can get the information:

```
   hData->GetXaxis()->GetBinCenter(iBin); hData->GetBinContent(iBin);
fit -> Eval(binCenter);
```

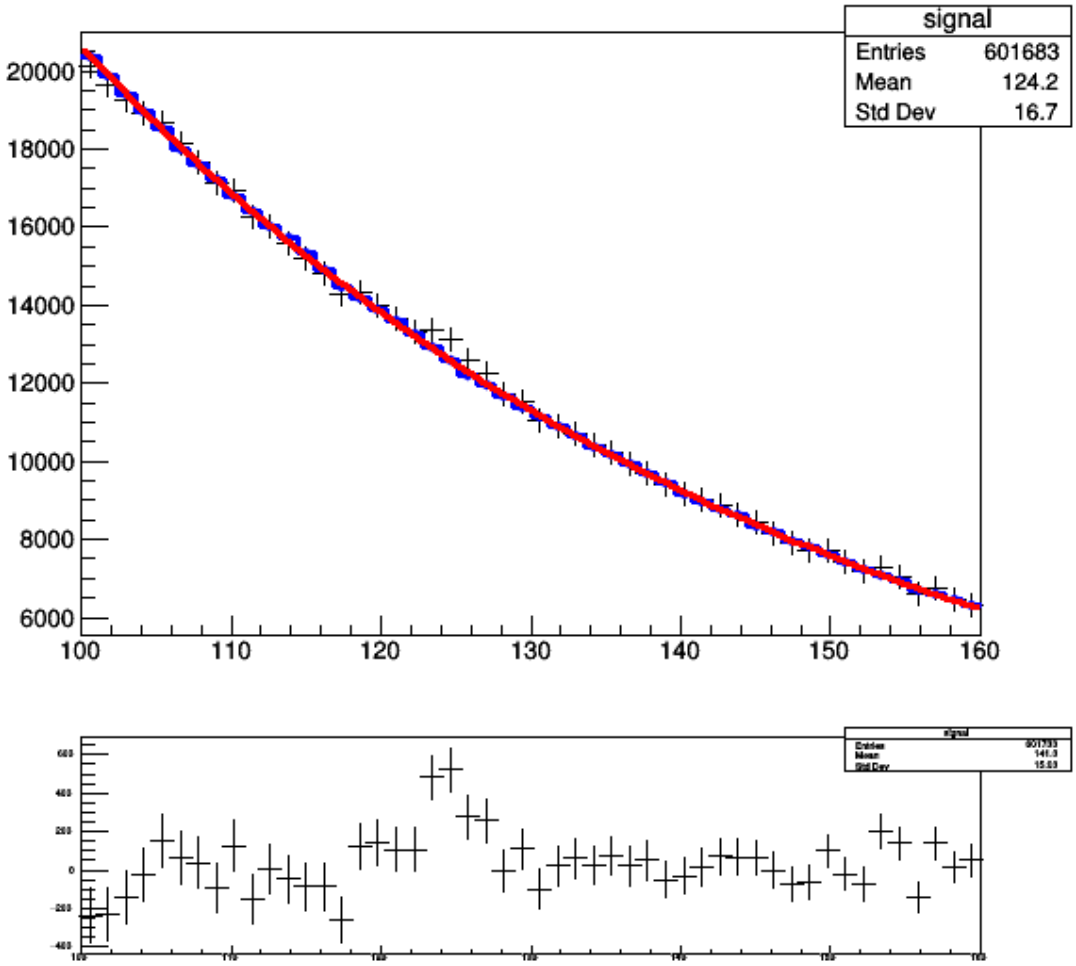
```
In [13]: for(int iBin = 1; iBin <= nBins; ++iBin){
             double binCenter   = hData->GetXaxis()->GetBinCenter(iBin);
             double dataValue   = hData->GetBinContent(iBin);
             double functionVal = fit -> Eval(binCenter);
             double difference  = dataValue - functionVal;
             hDiff -> SetBinContent(iBin, difference);
         }
```

Let's plot this in the second pad below the first one:

```
In [14]: pad2 ->cd();
```

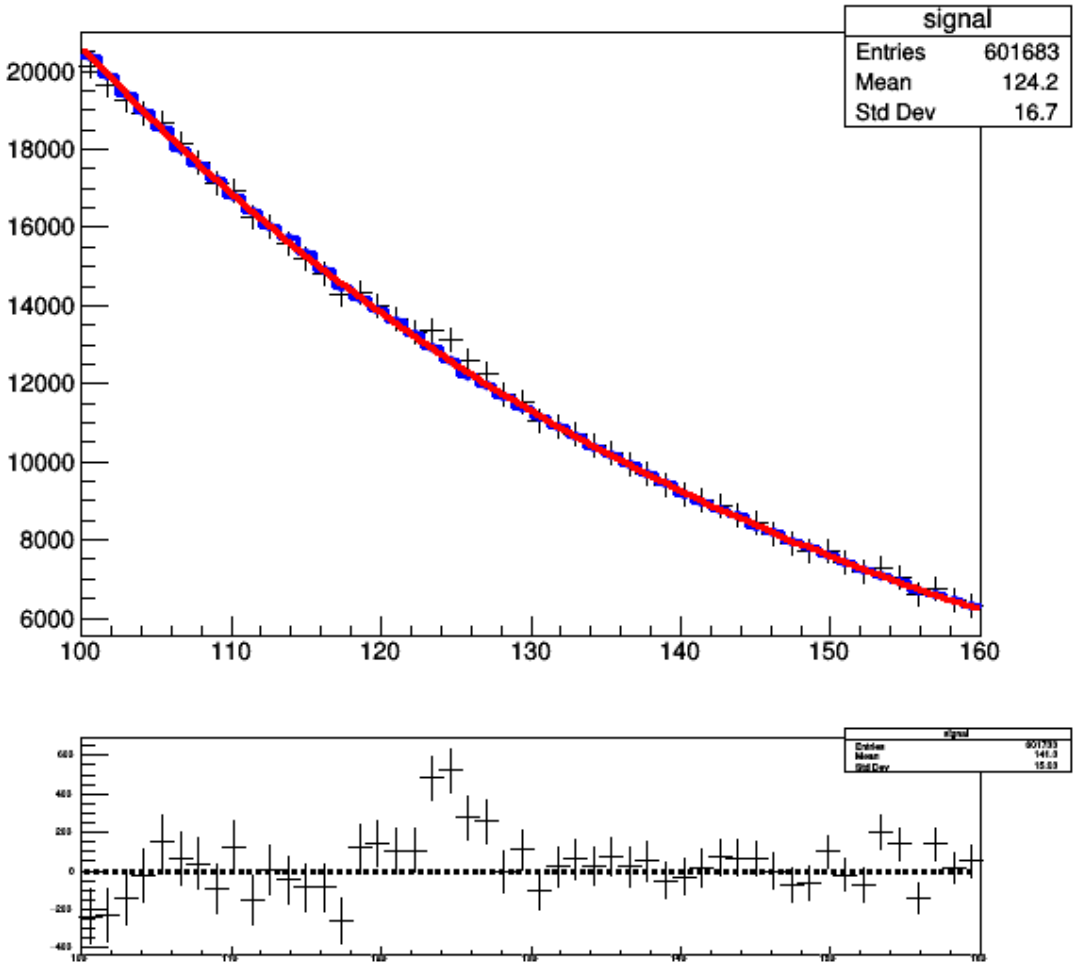
Make the histogram a bit nicer and draw:

```
In [15]: hDiff -> Draw();
         c -> Draw();
```



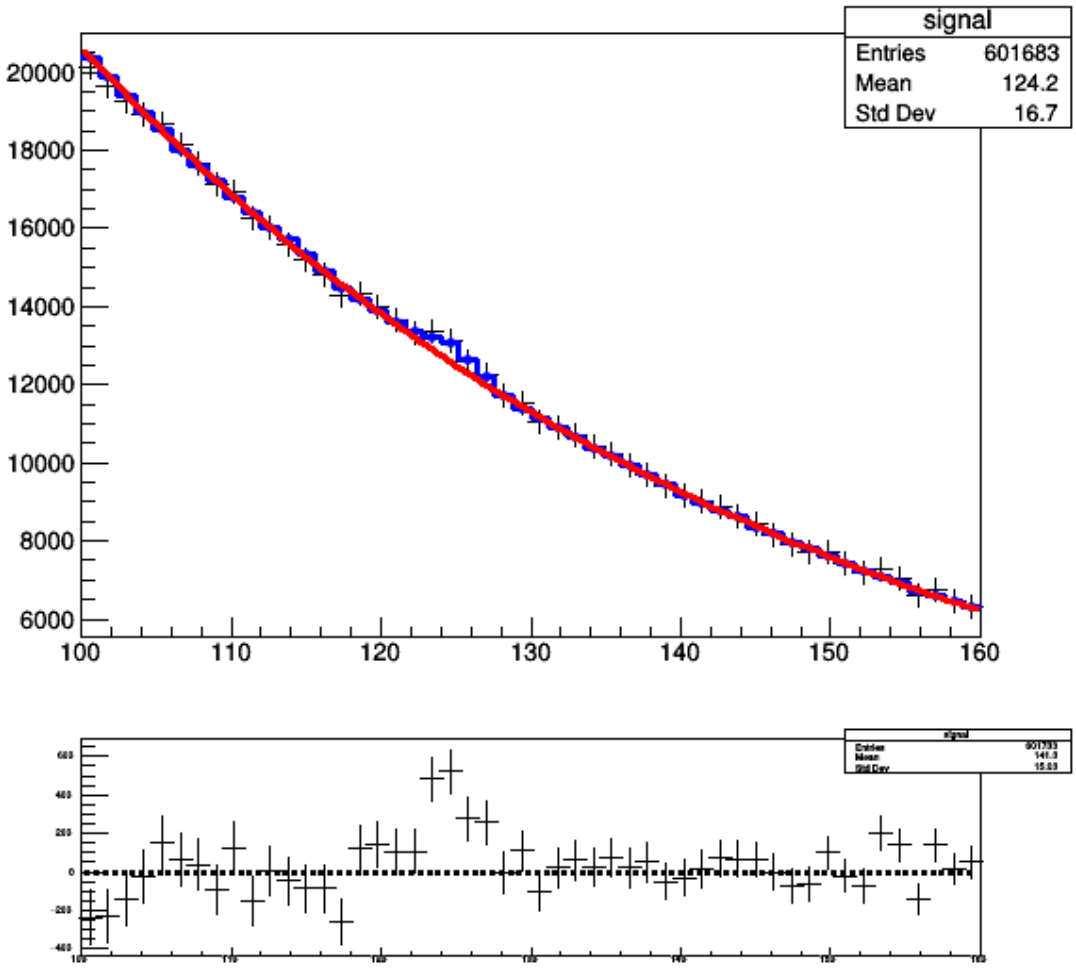
In order to better visualize an excess or deficiency, we can draw a line at zero.

```
In [16]: TF1 *zero = new TF1("zero", "0", 100, 160);
         zero -> SetLineColor(kBlack);
         zero -> SetLineWidth(3);
         zero -> SetLineStyle(2);
         zero -> Draw("SAME");
         c->Draw();
```



We can now add the signal to our background model

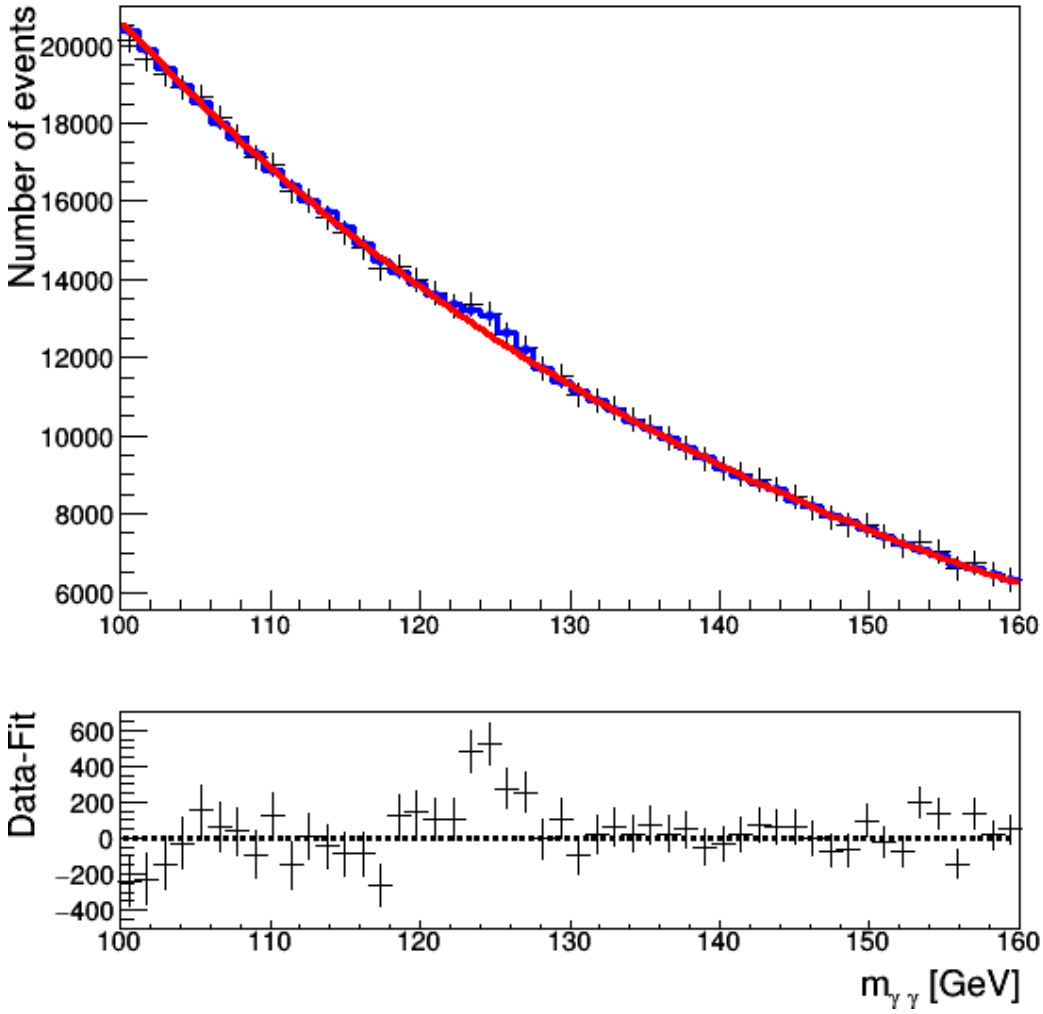
```
In [17]: hBkg -> Add(hSig);
         c-> Draw();
```



Almost there, lets make everything more beautiful, let's start with the axis labels.

```
In [18]: gStyle->SetOptStat(0);
         hData -> GetYaxis() -> SetTitle("Number of events");
         hData -> GetYaxis() -> SetTitleOffset(1.55);
         hData -> GetYaxis() -> SetTitleSize(20);
         hData -> GetYaxis() -> SetTitleFont(43);
         hData -> GetYaxis() -> SetLabelSize(15);
         hData -> GetYaxis() -> SetLabelFont(43);

         hDiff -> GetYaxis() -> SetTitle("Data-Fit");
         hDiff -> GetYaxis() -> SetTitleSize(20);
         hDiff -> GetYaxis() -> SetTitleFont(43);
         hDiff -> GetYaxis() -> SetTitleOffset(1.55);
         hDiff -> GetYaxis() -> SetLabelSize(15);
```

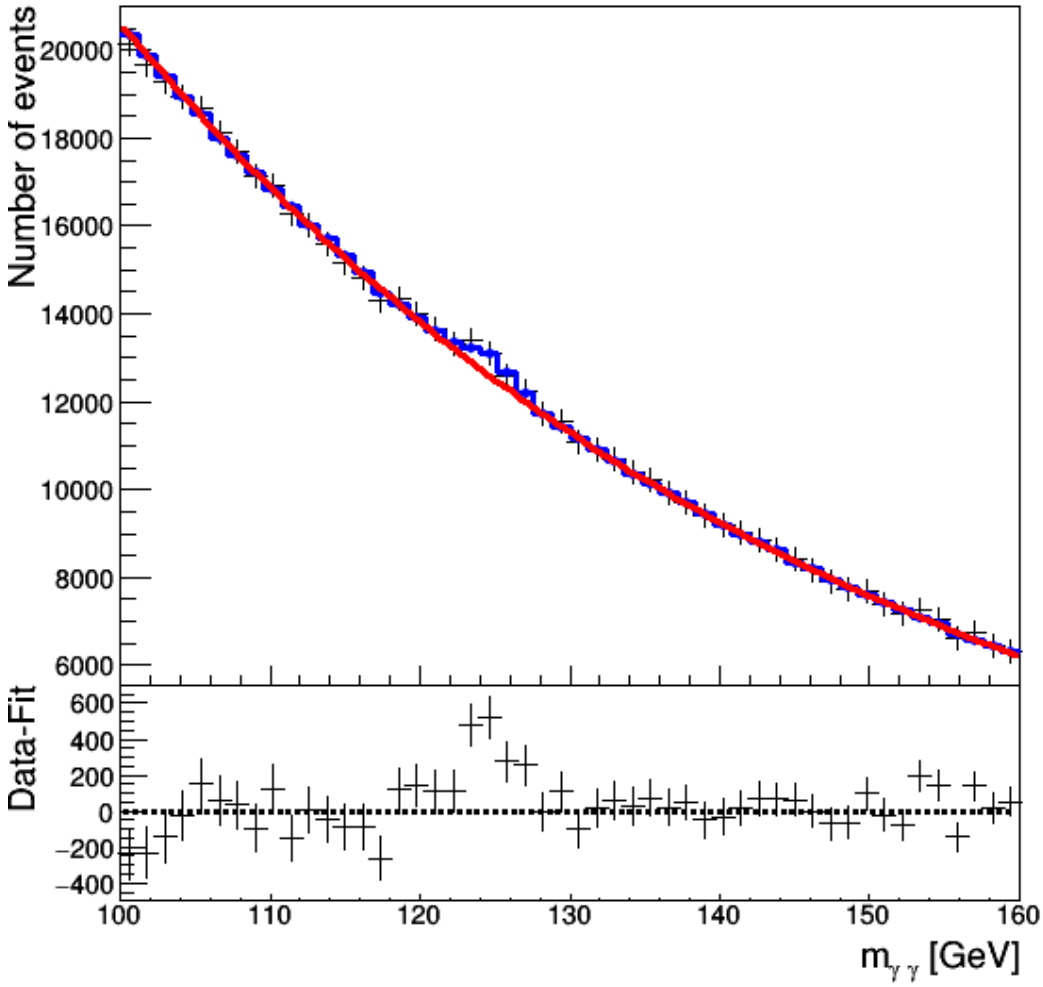
```
        hDiff -> GetYaxis() -> SetLabelFont(43);
        hDiff -> GetYaxis() ->SetRangeUser(-500, 700);

        hDiff -> GetXaxis() -> SetTitle("m_{#gamma #gamma} [GeV]");
        hDiff -> GetXaxis() -> SetTitleSize(20);
        hDiff -> GetXaxis() -> SetTitleFont(43);
        hDiff -> GetXaxis() -> SetTitleOffset(4.);
        hDiff -> GetXaxis() -> SetLabelSize(15);
        hDiff -> GetXaxis() -> SetLabelFont(43);

        c->Draw();
```



Connect the pads

```
In [19]: pad1 -> SetBottomMargin(0);
         pad2 -> SetTopMargin(0);
```

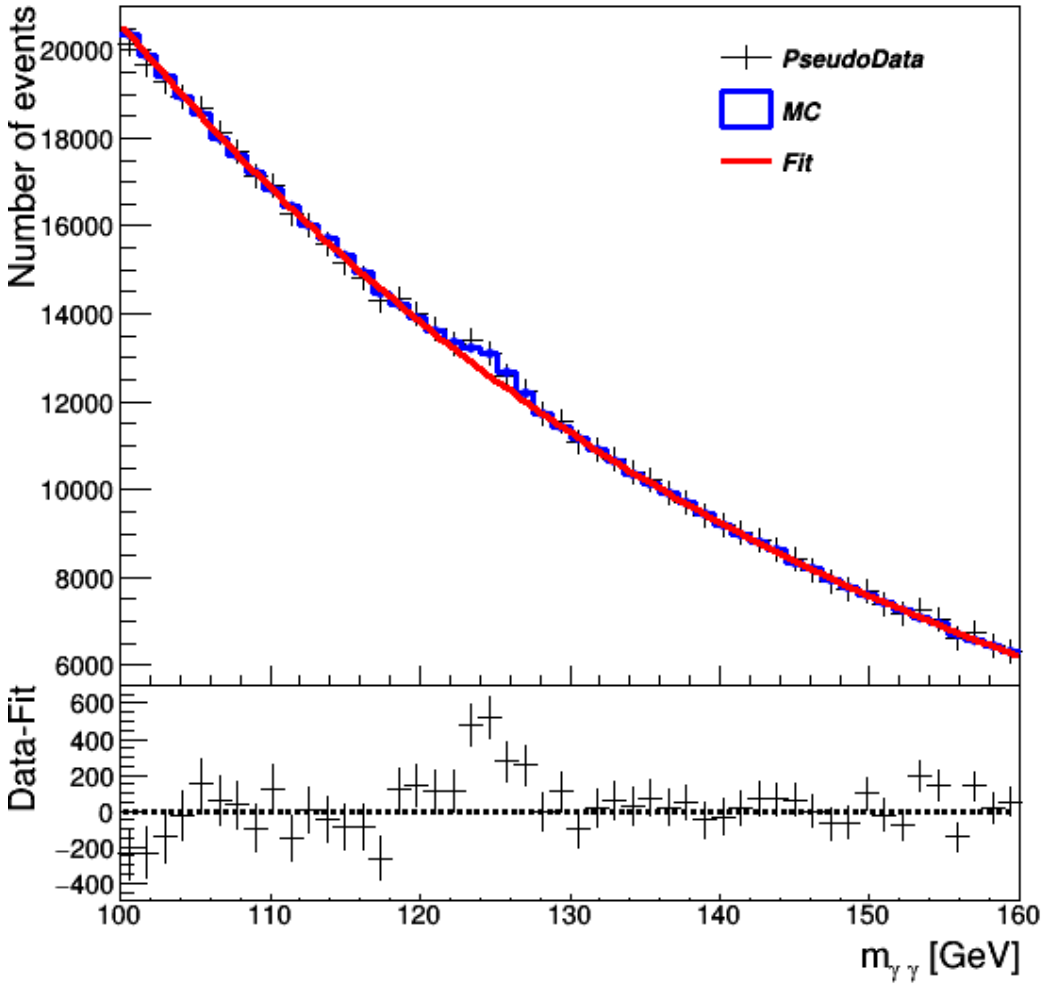
```
        pad2 -> SetBottomMargin(0.2);
        c -> Draw();
```



Let's add a legend:

```
In [20]: pad1->cd();
         TLegend *fLegend = new TLegend(0.625, 0.66, 0.625+0.255, 0.865);
         fLegend -> AddEntry(hData,    "PseudoData", "lp");
         fLegend -> AddEntry(hBkg,     "MC",         "f");
         fLegend -> AddEntry(fit,      "Fit",        "l");
         fLegend -> SetFillColor(0);
         fLegend -> SetLineColor(0);
         fLegend -> SetBorderSize(0);
         fLegend -> SetTextFont(72);
         fLegend -> SetTextSize(0.035);
```

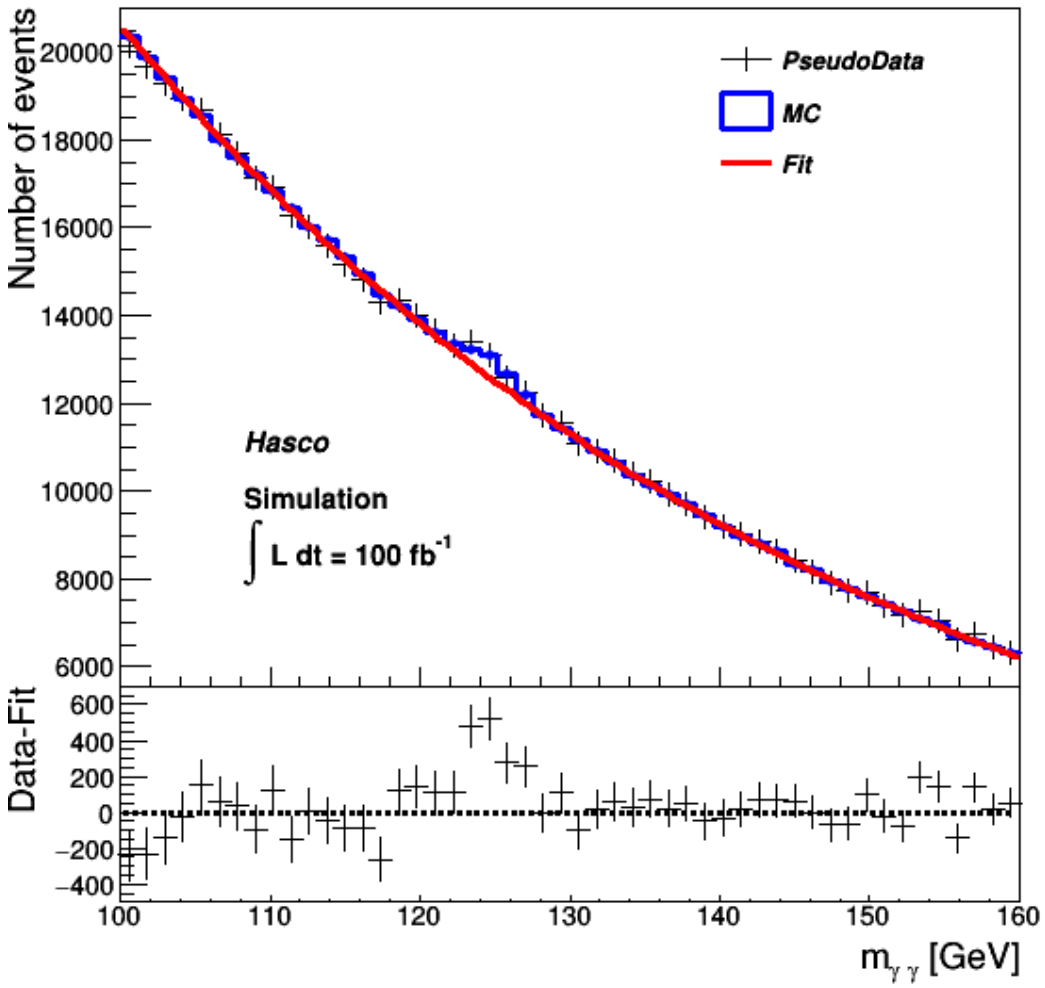
```
        fLegend -> Draw("SAME");
        c->Draw();
```



Let's add a text label for the Lumi, Hasco, Simulation

```
In [21]: TLatex l1;
         l1.SetTextAlign(9);
         l1.SetTextSize(0.04);
         l1.SetNDC();
         l1.DrawLatex(0.21, 0.160, "#int L dt = 100 fb^{-1}");

         TLatex l2;
         l2.SetTextAlign(9);
         l2.SetTextFont(72);
         l2.SetTextSize(0.04);
```

```
l2.SetNDC();
l2.DrawLatex(0.21, 0.310, "Hasco");

TLatex l3;
l3.SetTextAlign(9);
l3.SetTextSize(0.04);
l3.SetNDC();
l3.DrawLatex(0.21, 0.235, "Simulation");
c->Draw();
```



Ok great, that looks awesome, but how do we know how strong our signal excess is?