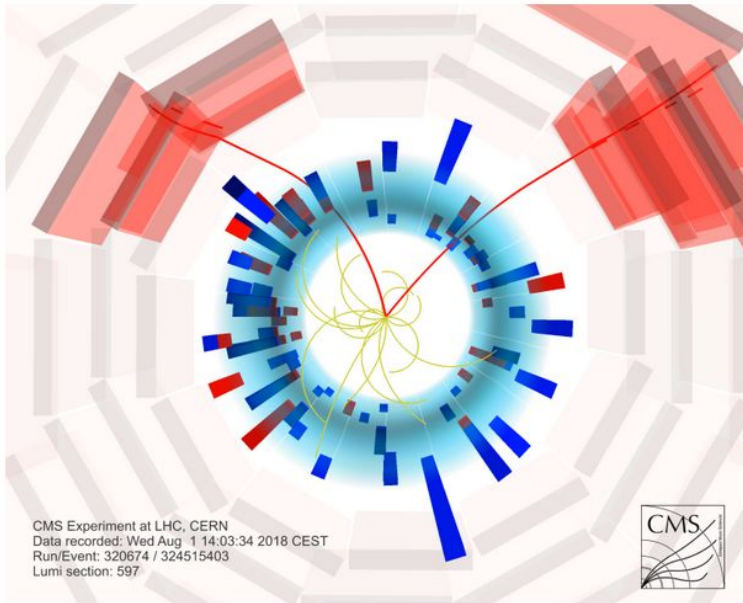


ROOT

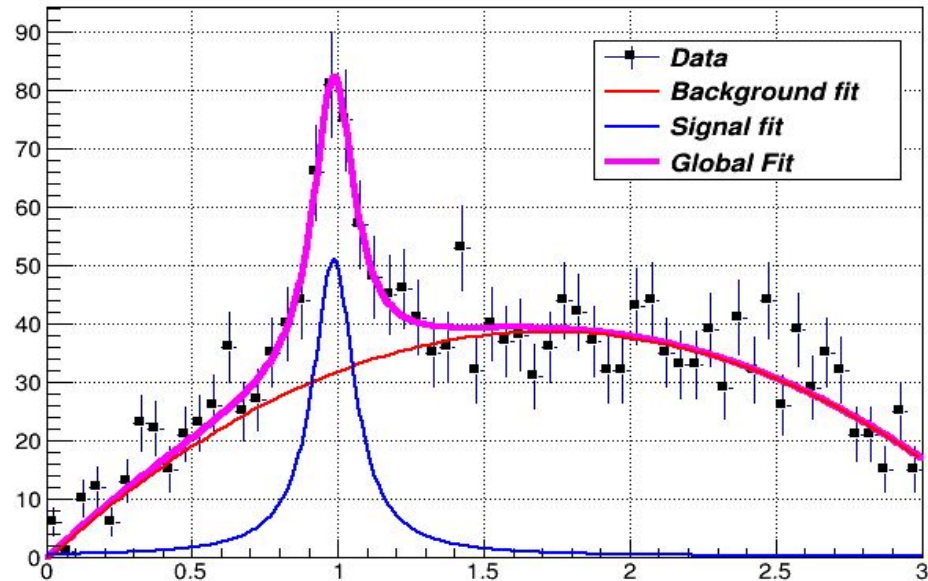
An Object-Oriented
Data Analysis Framework



CMS Experiment at LHC, CERN
Data recorded: Wed Aug 1 14:03:34 2018 CEST
Run/Event: 320674 / 324515403
Lumi section: 597



Lorentzian Peak on Quadratic Background



Big data en el CERN y otros contextos

Jhovanny Andres Mejia Guisao
UNIVERSIDAD DE ANTIOQUIA, COLOMBIA

What we hope to discuss about scientific data analysis?

- **Advanced graphical user interface**
- **Interpreter for the C++ programming language**
- **Persistency mechanism for C++ objects**
- **Used to write every year petabytes of data recorded by the Large Hadron Collider experiments**

Input and plotting of data from measurements and fitting of analytical functions.

My first Canvas

```
root[0] new TCanvas ← "quick" creation of graphical window with generic properties
(class TCanvas*) 0x2c1e5e0 ← Notice: automatic name assignment ("c1")
root[1] c1->Set
root[2] c1->SetTitle ("HelloCanvas")
root[3] c1->GetTitle ()
(const char* 0x1557339) "HelloCanvas"
root[4] c1->Is ()
root[5] c1->Close ()
root[6] TCanvas c2 ← We create a new window. Before we used pointer. Now – object.
root[7] c2.GetName()
(const char* 0x16632b1) "c1_n2" ← Title is different than variable name!
root[8] TCanvas c3 (
TCanvas TCanvas(Bool_t build = kTRUE)
TCanvas TCanvas(const char* name, const char* title = "", Int_t form = 1)
TCanvas TCanvas(const char* name, const char* title, Int_t ww, Int_t wh)
TCanvas TCanvas(const char* name, const char* title, Int_t wtopx, Int_t wtopy,
Int_t ww, Int_t wh)
TCanvas TCanvas(const char* name, Int_t ww, Int_t wh, Int_t winid)
root[7] TCanvas c3 ("c3canvas", "My canvas", 600, 400);
```

↑
Proper name
of object
(within C++).

↑
"Name"
(identifier)
(within ROOT).

↑
Displayed title
(just a c-string)

Multitude of constructors

```
root[8] c3Canvas ← Name as identifier or replacement of C++ name
(class TCanvas*) 0x16964d0
root[9] TCanvas* c4 = new TCanvas ("c4canv", "2nd Canvas", 600, 400);
```

↑
Dynamic allocation (we then use a pointer to an object)

TCanvas

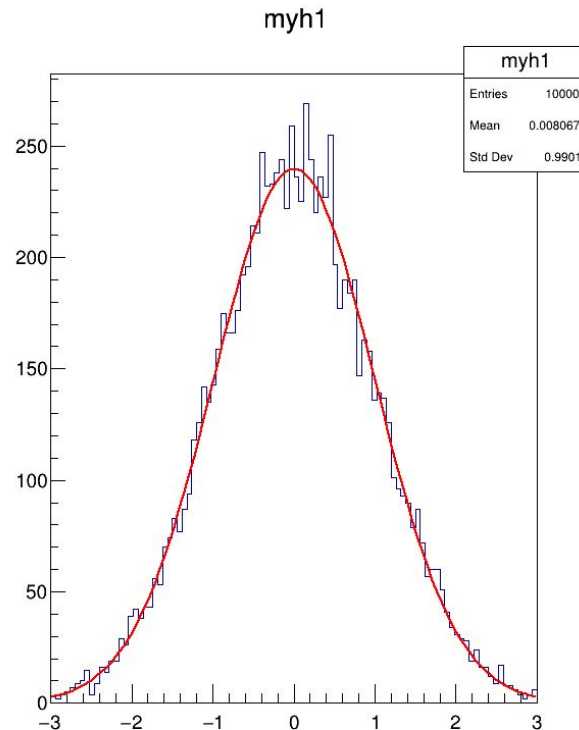
My first Canvas

<https://root.cern.ch/doc/master/classTCanvas.html>

```
auto myc1 = new  
TCanvas("myc1","myc1",600,800);  
myc1->cd();  
myc1->SetLeftMargin(0.15);  
myc1->SetRightMargin(0.06);  
myc1->SetTopMargin(0.09);  
myc1->SetBottomMargin(0.14);
```

Please, take a look to this example:

https://root.cern/doc/master/canvas_8C.html



Example: myfirtsCanvas.C

Graphs

A graph is a graphics object made of two arrays X and Y, holding the x, y coordinates of n points. There are several graph classes, they are:

[TGraph](#), [TGraphErrors](#), [TGraphAsymmErrors](#), [TMultiGraph](#), and [TGraphMultiErrors](#).

```
Int_t n = 20;
Double_t x[n], y[n];
for (Int_t i=0; i<n; i++) {
  x[i] = i*0.1;
  y[i] = 10*sin(x[i]+0.2);
}
TGraph *gr1 = new TGraph (n, x, y);
TCanvas *c1 = new TCanvas("c1","Grap
// draw the graph with axis, contineous lir
gr->Draw("AC*");
```

```
TGraph *gr2 = new TGraph (n, x, y);
gr2->SetFillColor(40);
gr2->Draw("AB");
```

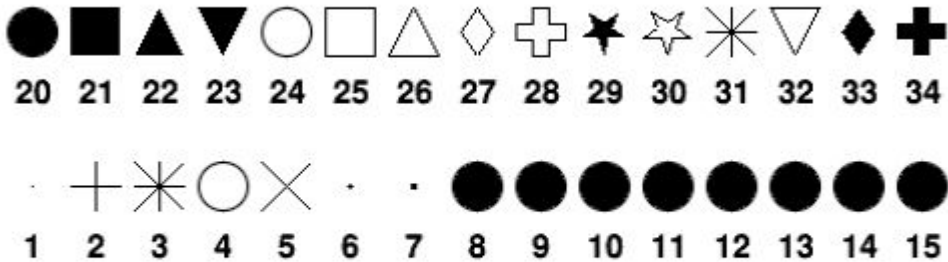
Graph Draw Options

The various draw options for a graph are explained in **TGraph::PaintGraph**. They are:

- "L" A simple poly-line between every points is drawn
- "F" A fill area is drawn
- "F1" Idem as "F" but fill area is no more repartee around X=0 or Y=0
- "F2" draw a fill area poly line connecting the center of bins
- "A" Axis are drawn around the graph
- "C" A smooth curve is drawn
- "*" A star is plotted at each point
- "P" The current marker of the graph is plotted at each point
- "B" A bar chart is drawn at each point
- "[" Only the end vertical/horizontal lines of the error bars are drawn. This option applies to the **TGraphAsymmErrors**.
- "1" ylow = rwymin

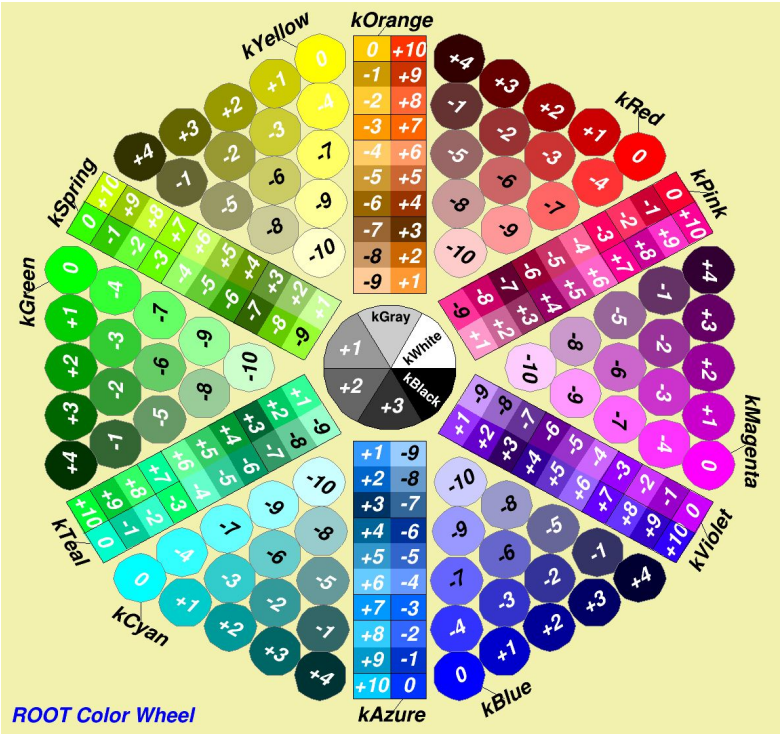
```
TGraph *gr3 = new TGraph(n,x,y);
gr3->SetFillColor(45);
gr3->Draw("AF")
```


Graphs



Marker styles

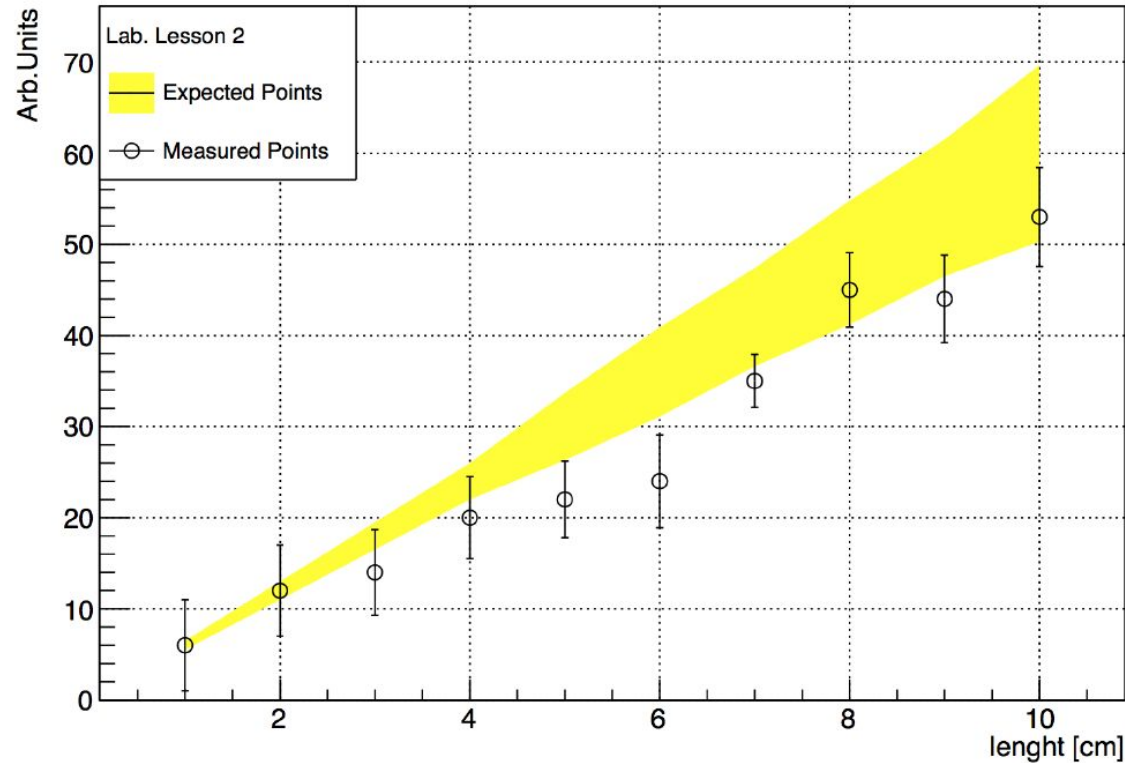
TAttMarker Class



Example: TwoGraph.C

"Good Plot" example

Measurement XYZ and Expectation



```
//Example: MyFirtsTgraph.C  
void MyFirtsTgraph() {  
    TGraphErrors myTgr2("ExampleData.txt");  
    myTgr2.Draw("AP");; // shows histogram  
}
```

ROOT doesn't show my histogram!


```
//Example: MyFirtsTgraph.C
```

```
void MyFirtsTgraph() {  
    TGraphErrors *myTgr = new TGraphErrors("ExampleData.txt");  
    //auto myTgr = new TGraphErrors("ExampleData.txt");  
    myTgr->Draw("AP");  
}
```

`new` puts object on “heap”, escapes scope

```
//Example: MyTgraph2.C
```

```
{
```

```
  TGraphErrors myTgr2("ExampleData.txt");
```

```
  myTgr2.Draw("AP");; // shows histogram
```

```
}
```

if "anonymous" macro, this problem disappear?

```
//On the window
{
root [0] TGraphErrors myTgr2("ExampleData.txt");
root [1] myTgr2.Draw("AP");
root [2] myTgr2.SetMarkerSize(0.8);
root [3] myTgr2.SetMarkerStyle(20);
root [4] myTgr2.Draw("AP");
}
```

```
root[1] TGraph g ("tgraph2.dat", "%*s %lg %*lg %lg");
```

```
root[2] g.Draw ("AP");
```

```
root[3] TGraph gr2 (
(...)
```



%lg : read column as double precision
%*lg: column type is double; omit it.
%*s : column type is string; omit it.

let's move step by step

The data set

```
void Grap_StepbyStep() {  
    // The number of points in the data set  
    const int n_points = 10;  
    // The values along X and Y axis  
    double x_vals[n_points] = {1,2,3,4,5,6,7,8,9,10};  
    double y_vals[n_points] = {6,12,14,20,22,24,35,45,44,53};  
    // The errors on the Y axis  
    double y_errs[n_points] = {5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};  
  
    // Instance of the graph  
    auto graph = new TGraphErrors(n_points,x_vals,y_vals,nullptr,y_errs);  
}
```

This code creates the **data set**.

⇒ Create a macro called "Grap_StepbyStep.C" and execute it.

The data drawing

As this graph has error bars along the Y axis an obvious choice will be to draw it as an **error bars plot**. The command to do it is:

```
graph->Draw("APE");
```

The Draw() method is invoked with three options:

- "A" the **axis** coordinates are automatically computed to fit the graph data
- "P" **points** are drawn as marker
- "E" the **error bars** are drawn

⇒ Add this command to the macro and execute it again.

⇒ Try to change the options (e.g. "APEL", "APEC", "APE4").

Customizing the data drawing

Graphical attributes can be set to customize the visual aspect of the plot. Here we change the **marker style** and **color**.

```
// Make the plot esthetically better  
graph->SetMarkerStyle(kOpenCircle);  
graph->SetMarkerColor(kBlue);  
graph->SetLineColor(kBlue);
```

⇒ Add this code to the macro. Notice the visual changes.

⇒ Play a bit with the possible values of the attributes.

Add a Function

The data set we built up looks very linear. It could be interesting to **compare it with a line** to see what the linear law behind this data set could be.

```
// Define a linear function
auto f = new TF1("Linear law", "[0]+x*[1]", .5, 10.5);
// Let's make the function line nicer
f->SetLineColor(kRed);
f->SetLineStyle(2);
// Set parameters
f->SetParameters(-1, 5);
f->Draw("Same")
```

The function "f" graphical aspect is customized the same way the graph was before.

⇒ Add this code to the macro. Execute it again.

⇒ Play with the graphical attributes for the "f" function.

Plot Titles

The graph was created without any titles. It is time to define them. The following command **define the three titles** (separated by a ";") in one go. The format is:

"Main title ; x axis title ; y axis title"

```
graph->SetTitle("Measurement XYZ;length [cm];Arb.Units");
```

The axis titles can be also set individually:

```
graph->GetXaxis()->SetTitle("length [cm]");  
graph->GetYaxis()->SetTitle("Arb.Units");
```

- ⇒ Add this code to the macro. You can choose one or the other way.
- ⇒ Play with the text. You can even try to add TLatex special characters.

Axis labelling (1)

The axis labels must be very **clear**, **unambiguous**, and **adapted to the data**.

- ROOT by default provides a powerful mechanism of **labelling optimisation**.
- Axis have three levels of divisions: Primary, Secondary, Tertiary.
- The axis labels are on the Primary divisions.
- The method `SetNdivisions` change the number of axis divisions

```
graph->GetXaxis()->SetNdivisions(10, 5, 0);
```

⇒ Add this command. nothing changes because they are the default values :-)

⇒ Change the number of primary divisions to 20. Do you get 20 divisions ? Why ?

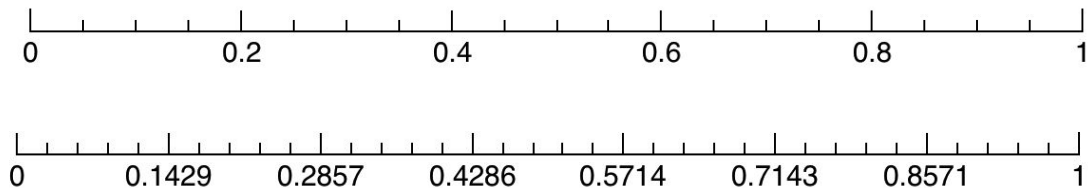
Axis labelling (2)

The number of divisions passed to the `SetNdivisions` method are **by default optimized** to get a comprehensive labelling. Which means that the value passed to `SetNdivisions` is a maximum number not the exact value we will get.

To turn off the optimisation the fourth parameter of `SetNdivisions` to `kFALSE`

⇒ try it

As an example, the two following axis both have been made with **seven** primary divisions. The first one is optimized and the second one is not. The second one is not



Nevertheless, in some cases, it is useful to have non optimized labelling.

Other kinds of axis

In addition to the normal linear numeric axis axis ROOT provides also:

- **Alphanumeric labelled axis.** They are produced when a histogram with alphanumeric labels is plotted.
- **Logarithmic axis.** Set with `gPad->SetLogx()` \Rightarrow **try it** (`gPad->SetLogy()` for the Y axis)
- **Time axis.**

Fine tuning of axis labels

When a specific **fine tuning** of an individual label is required, for instance changing its color, its font, its angle or even changing the label itself the method `ChangeLabel` can be used on any axis.

⇒ For instance, in our example, imagine we would like to highlight more the X label with the maximum deviation by putting it in red. It is enough to do:

```
graph->GetXaxis() ->ChangeLabel (3, -1, -1, -1, kRed);
```

See here: https://root.cern/doc/master/gaxis3_8C_source.html

Legend (1)

ROOT provides a powerful class to build a legend: **TLegend**. In our example the legend is build as follow:

```
auto legend = new TLegend(.1,.7,.3,.9,"Lab. Lesson 1");  
legend->AddEntry(graph,"Exp. Points","PE");  
legend->AddEntry(f,"Th. Law","L");  
legend->Draw();
```

⇒ Add this code, try it and play with the options

- The **TLegend** constructor defines the **legend position and its title**.
- Each legend item is added with method **AddEntry** which specify
 - **an object to be part of the legend** (by name or by pointer),
 - **the text of the legend**
 - **the graphics attributes to be legended**. "L" for TAttLine, "P" for TAttMarker, "E" for error bars etc ...

Legend (2)

ROOT also provides an **automatic way to produce a legend** from the graphics objects present in the pad. The method `TPad::BuildLegend()`.

⇒ In our example try `gPad->BuildLegend()`

This is a quick way to proceed but for a plot ready for publication it is recommended to use the method previously described.

⇒ Keep only the legend defined in the previous slide

Annotations (1)

Often the various parts of a plot we already saw are not enough to convey the complete message this plot should. Additional **annotations** elements are needed **to complete and reinforce the message** the plot should give.

ROOT provides a collection of basic graphics primitives allowing to draw such annotations. Here a non exhaustive list:

- `TText` and `TLatex` to draw text.
- `TArrow` to draw all kinds of arrows
- `TBox`, `TEllipse` to draw boxes and ellipses.
- Etc ...

Annotations (2)

In our example we added an annotation pointing the "maximum deviation". It consists of a simple arrow and a text:

```
// Draw an arrow on the canvas
auto arrow = new TArrow(8,8,6.2,23,0.02,"|>");
arrow->SetLineWidth(2);
arrow->Draw();

// Add some text to the plot
auto text = new TLatex(8.2,7.5,"#splitline{Maximum}{Deviation}");
text->Draw();
```

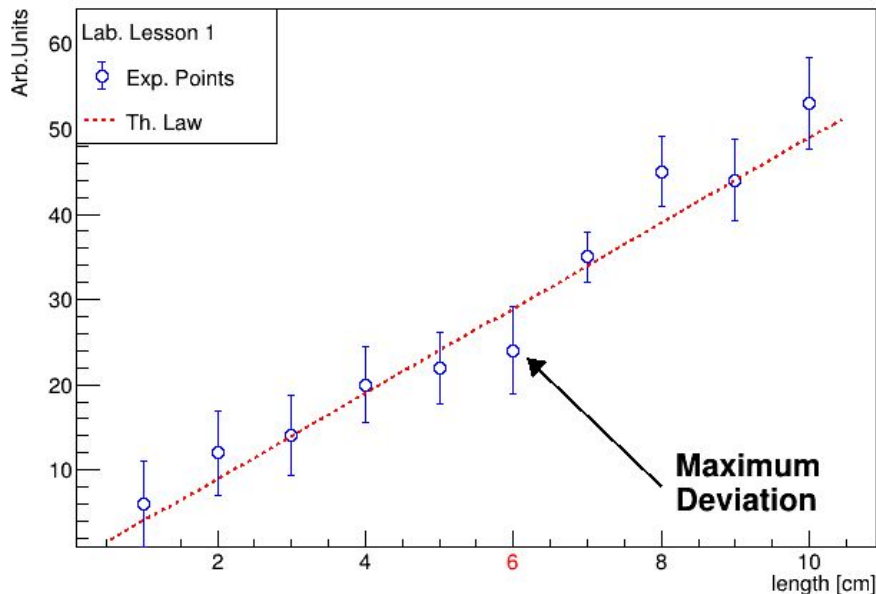
⇒ Try this code adding it in the macro.

⇒ Notice changing the canvas size does not affect the pointed position.

Seguimiento1

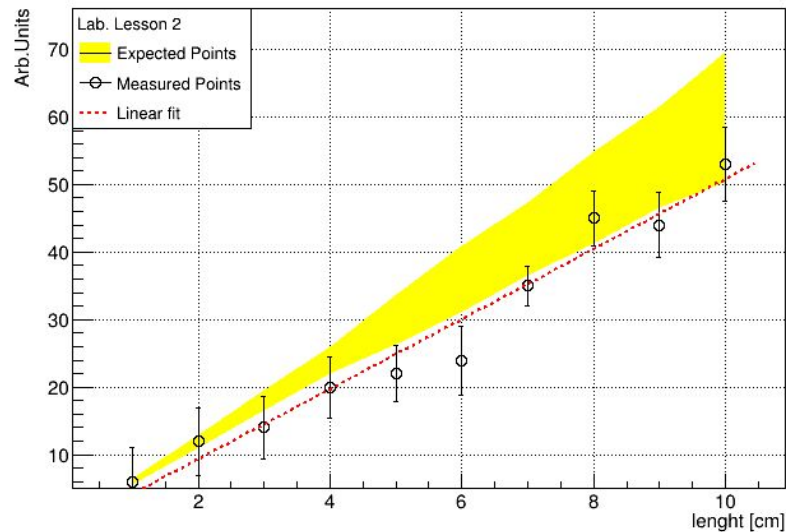
1)

Measurement XYZ



2)

Measurement XYZ and Expectation



Teniendo en cuenta los archivos: "macro2_input_expected.txt" y "macro2_input_mio.txt", y el ejemplo realizado para el plot "1)", haga el plot "2)". Además, el conjunto de datos que construimos parece muy lineal. Podría ser interesante ajustarlo con una línea para ver cuál podría ser la ley lineal detrás de este conjunto de datos.

¿Que valores de los parametros del fit obtiene?

TLatex (advanced text: mathematical expressions, etc.)

```
root [0] TLatex myl;  
root [1] myl.SetTextSize (0.06);  
root [2] myl.SetTextSize (0.06);  
root [3] myl.SetTextAngle (45.);  
root [4] myl.SetTextColor (4);  
root [5] myl.DrawLatex (0.5, 0.6, "E^{2} = m^{2} + p^{2}")  
root [6] myl.DrawLatex (0.5,0.6,"#gamma_{cm} = #frac{1}{#sqrt{1-#beta^{2}_{cm}}}")
```

A few basic rules:

$\wedge\{\dots\}$: top index

$_ \{\dots\}$: bottom index

$\#bf\{\dots\}$: bold font

$\#it\{\dots\}$: italic font

$\#vec\{\dots\}$: vector

$\#(\)\{\dots\}$: large brackets

$\#frac\{\text{Numerator}\}\{\text{Denom.}\}$: Horizontal fraction

$\#sqrt\{x\}$, $\#sqrt\{N\}\{x\}$: root of degree 2 and higher

$\#splitline\{\text{Above}\}\{\text{Below}\}$: two lines, one above the other

$\#color[4]\{\text{Blue}\}$: local change of color

$\#font[12]\{\text{Font}\}$: local change of font size

$\#scale[1.2]\{\text{Larger}\}$: local rescaling of font size

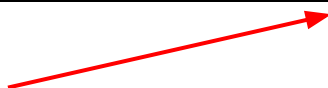
Example: mylatexexample.C

TStyle Object managing the graphical style

In ROOT session, the gStyle object of TStyle class is available. It keeps settings of graphical style. The settings concern e.g. the attributes of canvas, lines, markers, stats. Access is through the getters and setters.

```
root [0] gStyle->SetLabelSize(0.07,"XY")
root [1] gStyle->SetLabelOffset(0.01,"Y")
root [2] gStyle->SetNdivisions(2,"X")
root [3] TH1F h1 ("h1", "h1", 10,-5,5)
root [4] h1.Draw()
```

```
root [5] gStyle->SetNdivisions ( 8 , "X" );
root [6] h1.Draw()
root [7] h1.UseCurrentStyle ();
root [8] h1.Draw()
```



However, if we first defined a histogram, and changed the style later on, we need to inform that histogram in order it to update the style:

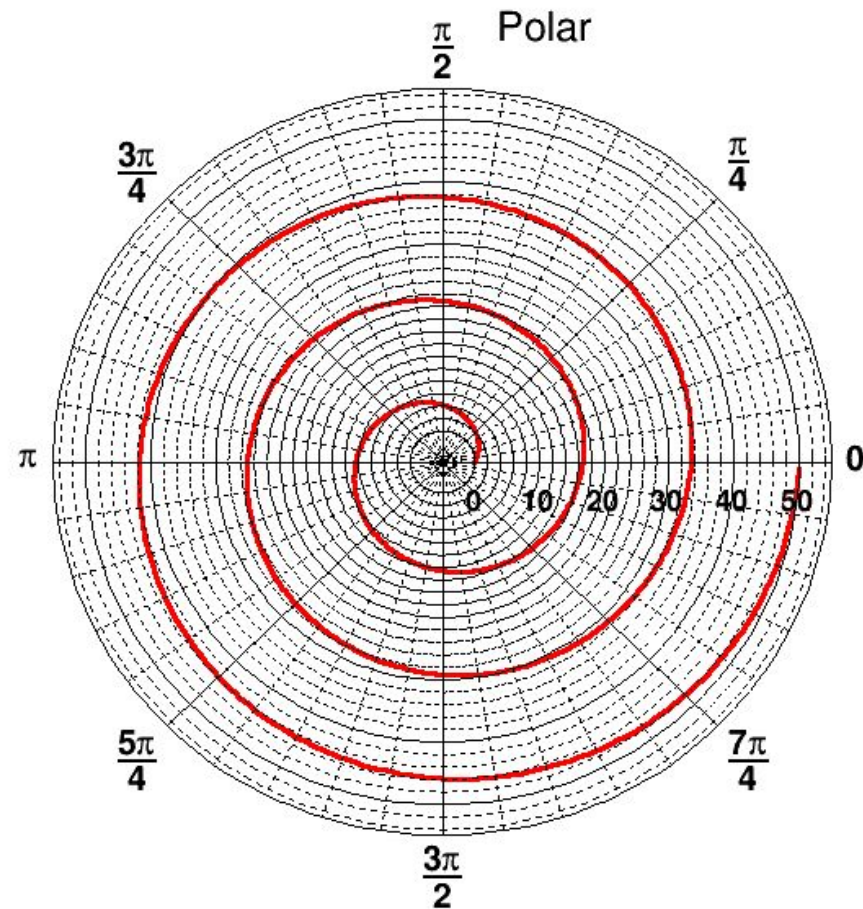
Important: TStyle allows to modify the contents of displayed statistics:

gStyle->SetOptStat ({rmen});

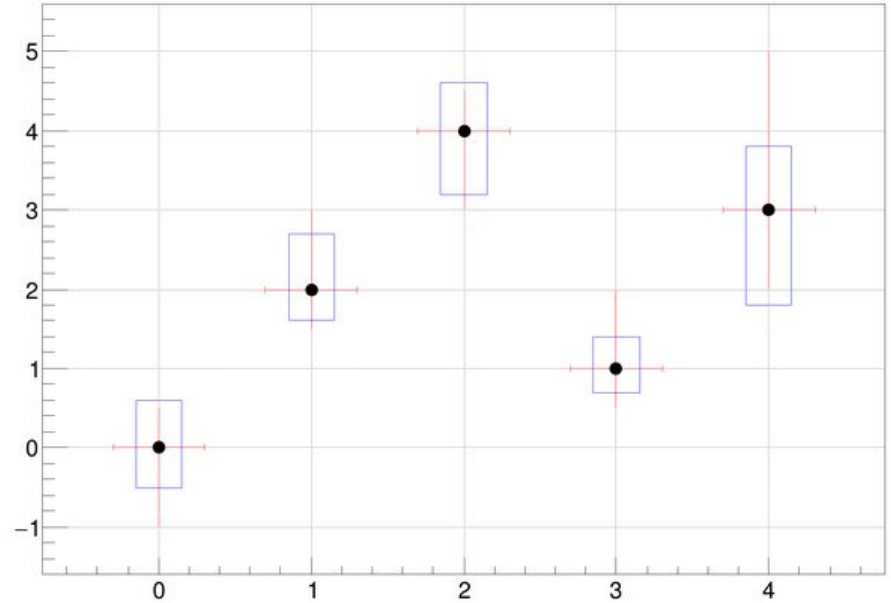
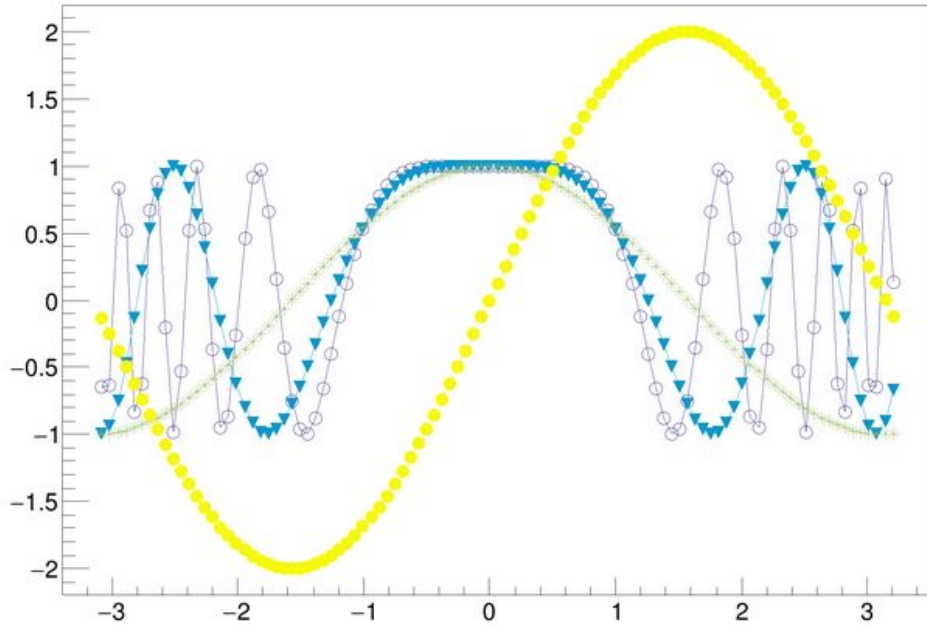
Symbols {r m e n} are the basic 4 of 9 attributes to display. They can take values {0, 1}, sometimes 2.

[Useful link: list properties](#)

r	=	(0)	1:	(do not) display RMS
m	=	(0)	1:	(do not) display the mean
e	=	(0)	1:	(do not) display the count numbers
n	=	(0)	1:	(do not) display the histogram name



Multiple graphs: TMultiGraph, TGraphMultiErrors.



Tarea: estudiar estos objetos y sus metodos

TGraphPainter:

<https://root.cern/doc/master/classTGraphPainter.html>

Interlude: TRandomN , $N = \{ 1, 2, 3 \}$ Pseudorandom numbers

<https://root.cern.ch/doc/master/classTRandom.html>

Usage of [TRandom3](#) is advised in the documentation.

```
root [0] TRandom3 r1;  
root [1] r1.SetSeed();
```

← setting the pseudorandom seed

TRandomN have predefined distributions, e.g. :

- `Exp(Double_t tau)`
- `Integer(UInt_t imax)`
- `Gaus(Double_t mean, Double_t sigma)`
- `Rndm()`
- `Uniform(Double_t)`
- `Landau(Double_t mean, Double_t sigma)`
- `Poisson(Double_t mean)`
- `Binomial(Int_t ntot, Double_t prob)`

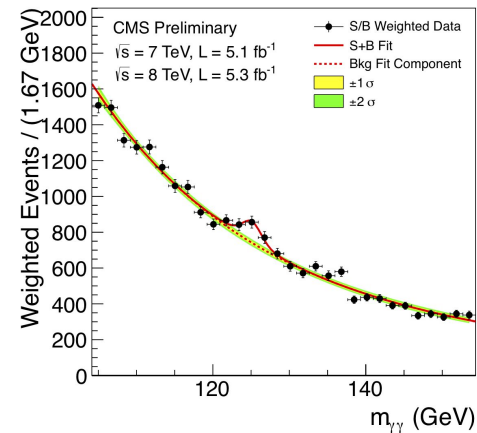
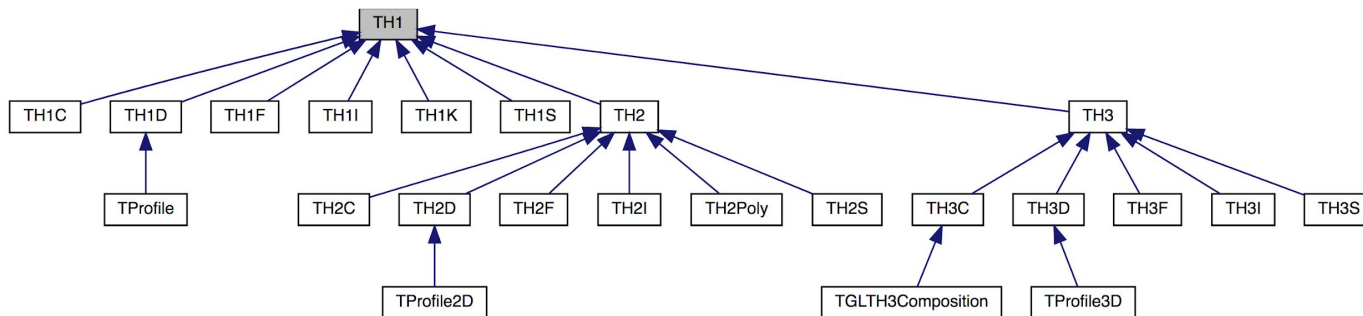
```
root [2] for (int i=0; i<5; i++){cout<< r1.Gaus(1.0,1.0) << endl;}
```

Moreover, you can pull numbers from distribution defined by the user in form of TF1 object, e.g.

```
root[3] TF1 fun1 ( " fun1 " , " x*x*exp(-x/0.5) " , 0., 5.)  
root[4] fun1.GetRandom ()
```

Histograms

- Simplest form of data reduction
 - Can have billions of collisions, the Physics displayed in a few histograms
 - Possible to calculate momenta: mean, rms, skewness, kurtosis ...
- Collect quantities in discrete categories, the bins
- ROOT Provides a rich set of histogram types
 - We'll focus on histogram holding a *float* per bin



Histograms

In general **THdt** where :

d=dimension(1,2,3) and

t:type{C,S,I,F,D} of variables that stores the bin content

#bins from to



```
root [0] auto myth1f = new TH1F("myth1f " , " My histogram " , 100, -10., 10.);
```

```
root [1] myth1f->Fill(5.0);
```

Fill the bin containing $x = 5.0$, with weight of 1.

```
root [2] myth1f->Draw();
```

```
root [3] myth1f->Fill(3.0,0.5);
```

Fill the bin containing $x = 5.0$, with weight of 0.5

```
root [3] myth1f->Draw();
```

```
root [4] myth1f->Fill(-3.0,0.8); myth1f->Draw();
```

```
root [0] auto myth1f = new TH1F("myth1f " , " My histogram " , 100, -10., 10.);
```

```
root [1] TRandom3 myr; myr.SetSeed();
```

```
root [2] for(int i=0; i<100; i++){ myth1f->Fill(myr.Gaus()); }
```

```
root [3] myth1f->Draw();
```


Histograms content

```
root [0] auto myth = new TH1F("myth " , " My histogram " , 100, -10., 10.);
root [1] TRandom3 myr; myr.SetSeed();
root [2] for(int i=0; i<1000; i++){ myth->Fill(myr.Gaus()); }
root [3] myth->Draw();
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [4] myth->SetMinimum(-10.0); myth->SetMaximum(250);
root [5] myth->Draw();
root [6] myth->SetLineColor(20); myth->SetLineStyle(250); myth->SetLineWidth(3);
root [7] myth->Draw();
root [8] myth->SetLineColor(20); myth->SetLineStyle(3); myth->SetLineWidth(3);
root [9] myth->Draw();
root [10] gStyle->SetOptStat(1101); ← (for details, see description of TStyle)
root [11] gStyle->SetOptStat(1111);
root [12] gStyle->SetOptStat(1101);
root [13] gStyle->SetOptStat(1001);
root [14] gStyle->SetOptStat(1000);
root [23] myth.SetNdivisions(505,"y"); ←
```

Code = $N1 + 100 \cdot N2 + 10000 \cdot N3$, where:
N1: (expected) number of leading divisions
N2: (expected) number of 2. rank divisions
N3: (expected) number of 3. rank divisions

Histograms content

```
root [0] auto myth = new TH1F("myth ", " My histogram ", 100, -10., 10.);
root [1] TRandom3 myr; myr.SetSeed();
root [2] for(int i=0; i<1000; i++){ myth->Fill(myr.Gaus()); }
root [3] myth->Draw();
root [4] cout << myth.GetMean() << "\t" << myth.GetRMS() << "\t" << myth.GetNbinsX() << endl;
root [5] cout << myth.FindBin(10) << endl;
root [6] cout << myth.FindBin(-0.1) << endl;
root [7] cout << myth.Integral() << endl;
root [8] cout << myth.Integral(myth.FindBin(-2.0), myth.FindBin(2.0)) << endl;
root [9] cout << myth.Integral(myth.FindBin(-2.0), myth.FindBin(2.0), "width") << endl;
```

Caution: Integral without specifying option – does not integrate ($\sum h_i \cdot \Delta$) but sums up the contents of bins ($\sum h_i$). In order to calculate integral, you have to type "width" option.

```
cout << myth.GetBinContent(50) << "\t" << myth.GetBinError(50) << endl;
```

As you can see, uncertainties are Poissonian

Histograms content

```
root [0] auto myth = new TH1F("myth " , " My histogram " , 100, -10., 10.);  
root [1] TRandom3 myr; myr.SetSeed();  
root [2] for(int i=0; i<5; i++){ myth->Fill(myr.Gaus()); }  
root [3] Float_t* hy = myth->GetArray();  
root [4] for(int i=45; i<65; i++){ cout << hy[i] << endl; }
```



Will return the array with bin contents.

Attention for the numbering: hy[1] ... hy[N]

In hy[0] , hy[N+1] the underflow / overflow are stored



Unfortunately, there is no method directly extracting the array of uncertainties.
Instead you can:

```
Float_t* hyerr = new Float_t [myth->GetNbinsX() + 2];  
for (int i=0; i<= myth->GetNbinsX()+1; i++) {hyerr[i] = myth->GetBinError(i);}
```



To create a separate data structure for uncertainties,
issue this before filling the histogram

```
root[0] myth.Sumw2 ()
```

Operations on histograms (Addition, multiplication, division)

```
Bool_t Add (TF1 *h1, Double_t c1=1, Option_t *option="")
Bool_t Add (const TH1 *h, const TH1 *h2, Double_t c1=1, Double_t c2=1)
Bool_t Add (const TH1 *h1, Double_t c1=1)
```

this = $c1 \cdot h1 + c2 \cdot h2$

this = this + $c1 \cdot h1$

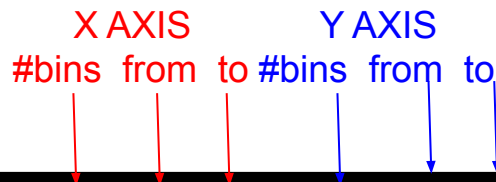
```
root [0] TH1F myth("myth ", " My histogram ", 100, -10., 10.);
root [1] TRandom3 myr; myr.SetSeed();
root [2] for(int i=0; i<10000; i++){ myth.Fill(myr.Gaus()); }
root [3] TH1F myth2(myth); myth2.Reset() ← New histogram: such as h1, but empty.
root [4] for(int i=0; i<10000; i++){ myth2.Fill(myr.Gaus(5.0,1)); }
root [5] myth2.Draw();
root [6] myth2.Add(&myth,0.1); myth2.Draw("e1"); ← we are adding

root [6] myth2.Add(&myth,0.1); myth2.Draw("e1");
root [7] TH1F myth3(myth); myth3.Reset();
root [8] for(int i=0; i<1000; i++){ myth3.Fill(-9.99+0.02*i); }
```

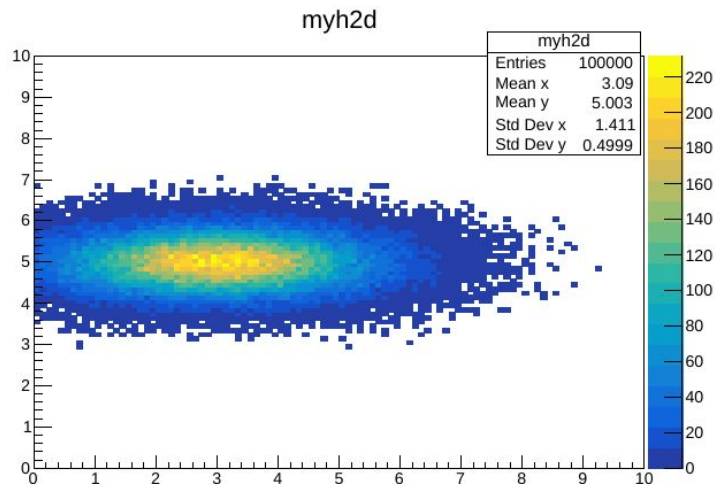
Example: **HistoOperations.C**

2-D histograms

X AXIS Y AXIS
#bins from to #bins from to



```
root [0] TH2F myh2d ("myh2d", "myh2d", 100, 0., 10., 100, 0., 10.);  
root [2] TRandom3 myr; myr.SetSeed();  
root [3] for (int i=0; i<1e5; i++){ myh2d.Fill(myr.Gaus(3,1.5), myr.Gaus(5,0.5));}  
root [4] h2d.Draw ("colz")
```

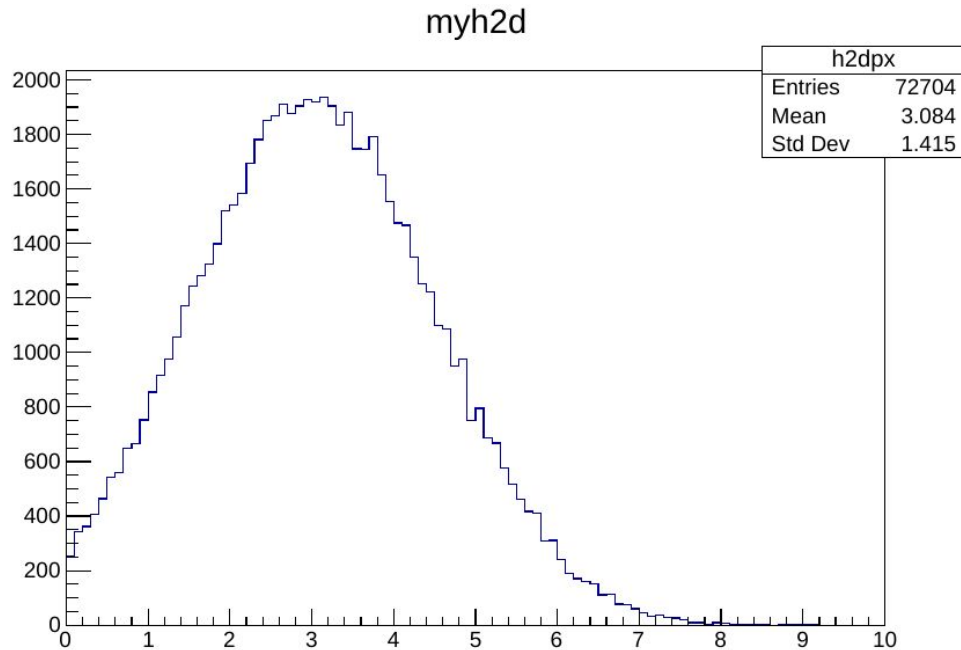


```
root [5] cout << myh2d.GetNbinsX() << '\t' <<  
myh2d.GetNbinsY() << endl;
```

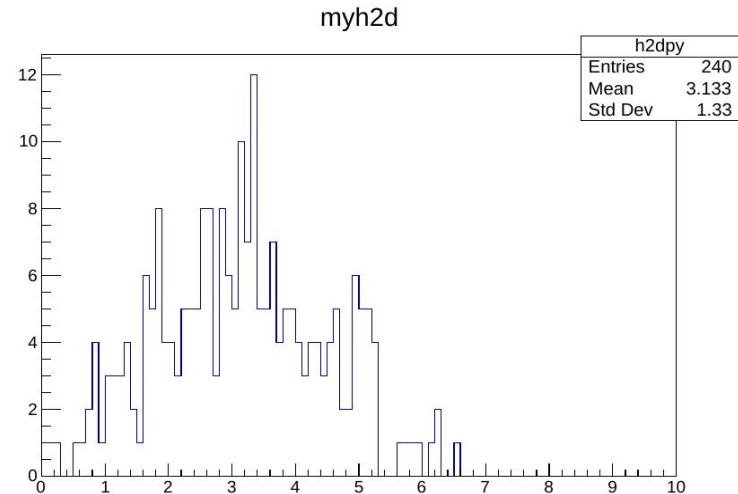
Example:myh2d.C

2dim \rightarrow 1dim projections (into X axis or Y axis)

```
root [3] myh2d.ProjectionX( "h2dpx", myh2d.GetYaxis()->FindBin(4.5) ,  
myh2d.GetYaxis()->FindBin(5.5) )  
root [4] h2dpx->Draw();
```



```
root [5] myh2d.ProjectionX( "h2dpy",  
myh2d.GetXaxis()->FindBin(2.5) ,  
myh2d.GetXaxis()->FindBin(3.5) )
```



Parameter Estimation and Fitting

- ▶ Understand the problem of parameter estimation
- ▶ Know the difference between a likelihood or a χ^2 based approach
- ▶ Become familiar with the tools ROOT offers to perform a fit: fit panel and programmatic steering of the procedure

https://cds.cern.ch/record/1019859/files/9789812567956_TOC.pdf

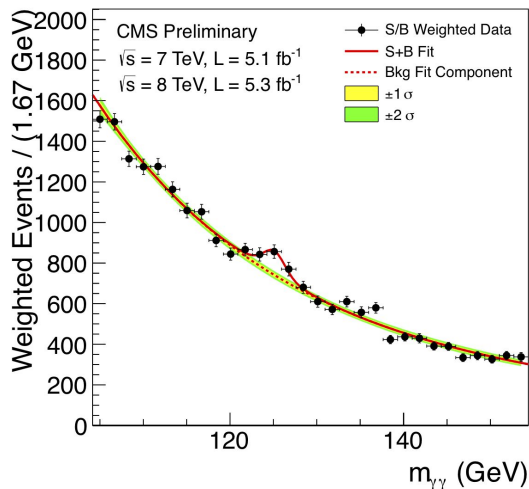
(F. James, Statistical Methods in Experimental Physics, *2nd Edition*)

Parameter Estimation and Fitting

- ▶ Introduction to Fitting:
 - fitting methods in ROOT
 - how to fit a histogram in ROOT,
 - how to retrieve the fit result.
- ▶ Building fit functions in ROOT.
- ▶ Interface to Minimization.
- ▶ Common Fitting problems.
- ▶ Using the ROOT Fit GUI (Fit Panel).

What is Fitting ?

- ▶ Estimate parameters of an hypothetical distribution from the observed data distribution
 - $y = f(x | \theta)$ is the fit model function
- ▶ Find the best estimate of the parameters θ assuming $f(x | \theta)$



Example

Higgs $\rightarrow \gamma\gamma$ spectrum

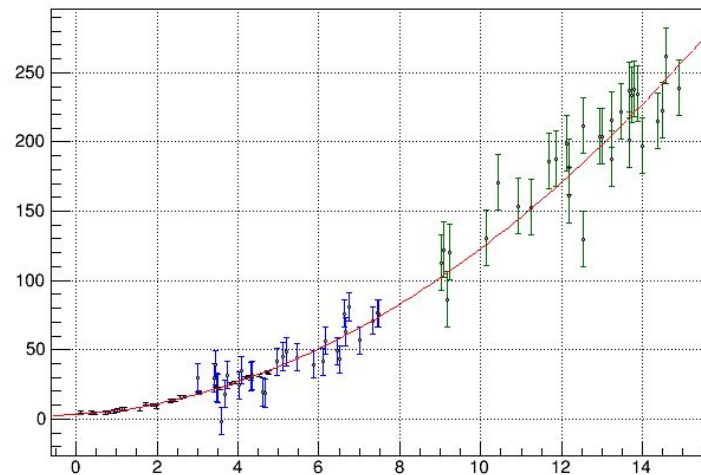
We can fit for:

- the expected number of Higgs events
- the Higgs mass

Least Square (χ^2) Fit

- ▶ Minimizes the deviations between the observed y and the predicted function values:
- ▶ **Least square fit (χ^2)**: minimize square deviation weighted by the uncertainties

$$\chi^2 = \sum_i \frac{(Y_i - f(X_i, \theta))^2}{\sigma_i^2}$$



Maximum Likelihood Fit

- The parameters are estimated by finding the maximum of the likelihood function (or minimum of the negative log-likelihood function).

- Likelihood:
- Find best value θ ,
the maximum of:

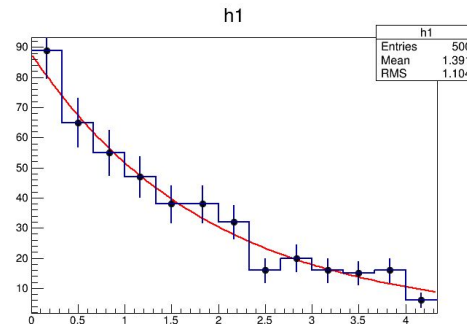
$$L(x|\theta) = \prod P(x_i|\theta)$$

$$\log L = \sum_i \log(f(x_i, \theta))$$

- The least-square fit and the maximum likelihood fit are equivalent when the error model is Gaussian
 - E.g. the observed events in each bin is normal $f(\mathbf{x} | \boldsymbol{\theta})$ is gaussian

ML Fit of an Histogram

- ▶ The Likelihood for a histogram is obtained by assuming a Poisson distribution in every bin:
 - **Poisson($n_i | v_i$)**
 - n_i is the observed bin content.
 - v_i is the expected bin content, $v_i = f(x_i | \theta)$, where x_i is the bin center, assuming a linear function within the bin. Otherwise it is obtained from the integral of the function in the bin.
- ▶ For large histogram statistics (large bin contents) bin distribution can be considered normal
 - equivalent to least square fit
- ▶ For low histogram statistics the ML method is the correct one !



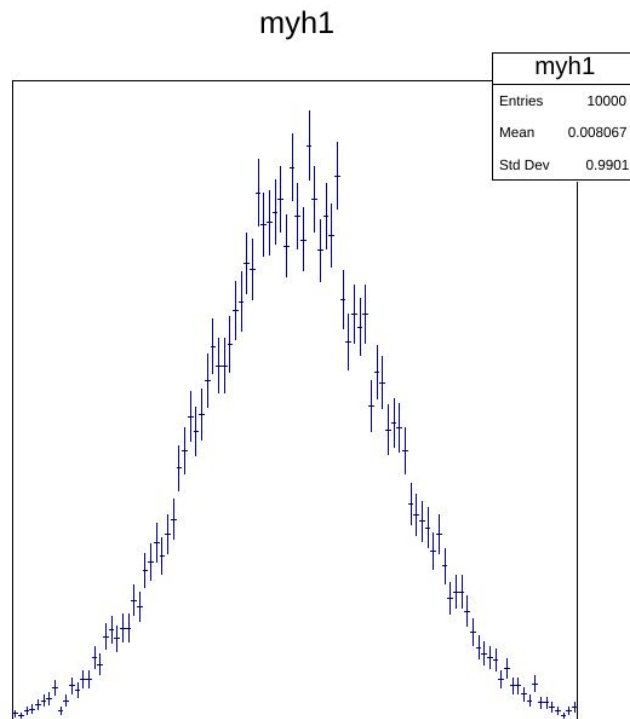
Fitting in ROOT

How do we do fit in ROOT:

- ▶ Create first a parametric function object, **TF1**, which represents our model, *i.e.* the fit function.
 - need to set the initial values of the function parameters.
- ▶ Fit the data object (Histogram or Graph):
 - call the **Fit** method passing the function object
 - various options are possibles (see the [TH1::Fit](#) documentation)
 - e.g select type of fit : least-square (default) or likelihood (option "L")
- ▶ Examine result:
 - get parameter values;
 - get parameter errors (e.g. their confidence level);
 - get parameter correlation;
 - get fit quality.
- ▶ The resulting fit function is also draw automatically on top of the Histogram or the Graph when calling **TH1::Fit** or **TGraph::Fit**

Simple Gaussian Fitting

- ▶ Suppose we have this histogram
 - we want to estimate the mean and sigma of the underlying gaussian distribution.



Fitting Histograms

```
TF1 f1("f1","gaus");
```

```
h1.Fit(f1);
```

```
Processing myGaussFit.C...
```

```
FCN=78.849 FROM MIGRAD STATUS=CONVERGED 61 CALLS 62 TOTAL
```

```
EDM=2.87322e-07 STRATEGY= 1 ERROR MATRIX ACCURATE
```

```
EXT PARAMETER
```

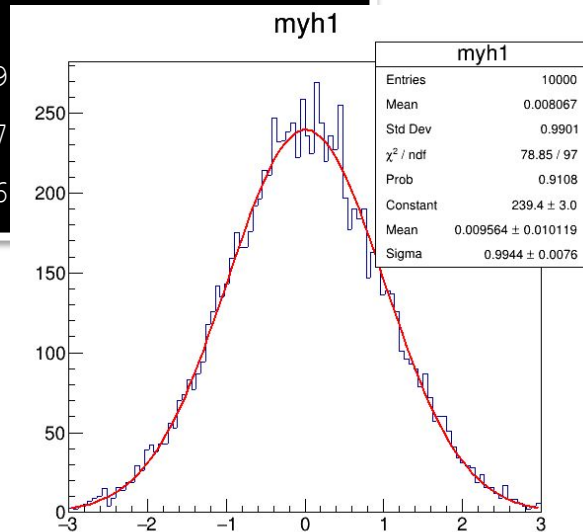
```
STEP
```

```
FIRST
```

NO.	NAME	VALUE	ERROR	SIZE	DERIVATIVE
1	Constant	2.39420e+02	2.98901e+00	1.04878e-02	3.079
2	Mean	9.56366e-03	1.01191e-02	4.41268e-05	-1.047
3	Sigma	9.94429e-01	7.57122e-03	8.92460e-06	2.656

For displaying the fit parameters:

```
gStyle->SetOptFit(1111);
```



Creating the Fit Function

- How to create the parametric function object (**TF1**) :
 - we can write formula expressions using functions available from the C++ standard library and those available in ROOT (e.g. TMath)

```
TF1 f1("f1", "[0]*TMath::Gaus(x, [1], [2])");
```

- we can use the available functions in ROOT library
 - [0],[1],[2] indicate the parameters.
 - We could also use meaningful names, like [a],[mean],[sigma]
- we can also use pre-defined functions

```
TF1("f1", "gaus");
```

- with pre-defined functions the parameter names and value are automatically set to meaningful values (whenever possible)
- pre-defined functions available: *gaus*, *expo*, *landau*, *breitwigner*, *crystal_ball*, *pol0*, *1..,10*, *cheb0*, *1*, *xygaus*, *xylanday*, *bigaus*

Building More Complex Functions

- Sometimes better to write directly the functions in C/C++
 - but in this case the function object cannot be fully stored to disk and reused later
- Example: using a general free function with parameters:

```
double function(double *x, double *p){  
    return p[0]*TMath::Gaus(x[0],p[0],p[1]);  
}  
TF1 f1("f1",function,xmin,xmax,npar);
```

Retrieving The Fit Result

- The main results from the fit are stored in the fit function, which is attached to the histogram; it can be saved in a file (except for C/C++ functions where only points are saved).
- The fit function can be retrieved using its name:

```
auto fitFunc = h1->GetFunction("f1");
```

- The parameter values/error using indices (or their names):

```
fitFunc->GetParameter(par_index);  
fitFunc->GetParError(par_index);
```

Retrieving The Fit Result

- It is also possible to access the **TFitResult** class which has all information about the fit, if we use the fit option "S":

```
TFitResultPtr r = h1->Fit(f1,"S");  
TMatrixDSym C = r->GetCorrelationMatrix();  
r->Print();
```

C++ Note: the TFitResult class is accessed by using operator-> of TFitResultPtr

Fitting Options

- Likelihood fit for histograms
 - option “L” for count histograms;

```
h1->Fit("gaus","L");
```
 - option “WL” in case of weighted counts.

```
h1->Fit("gaus","LW");
```
- Default is chi-square with observed errors (and skipping empty bins)
 - option “P” for Pearson chi-square
expected errors, and including empty bins

```
h1->Fit("gaus","P");
```
- Use integral function of the function in bin

```
h1->Fit("gaus","L I");
```
- Compute MINOS errors : option “E”

```
h1->Fit("gaus","L E");
```

Some More Fitting Options

- Fitting in a Range

- `h1->Fit("gaus","",",-1.5,1.5);`

- For doing several fits

- `h1->Fit("expo","+","",2.,4);`

- Quiet / Verbose: option “Q”/“V”

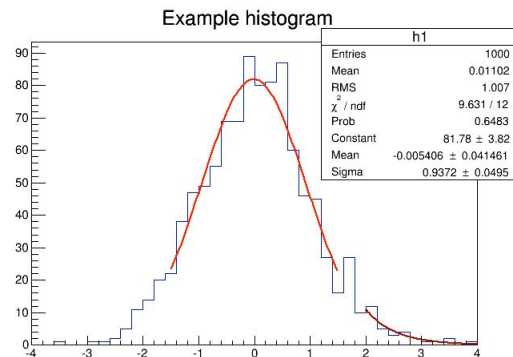
- `h1->Fit("gaus","V");`

- Avoid storing and drawing fit function (useful when fitting many times)

- `h1->Fit("gaus","L N 0");`

- Save result of the fit, option “S”

- `auto result = h1->Fit("gaus","L S");
result->Print("V");`



Parameter Errors

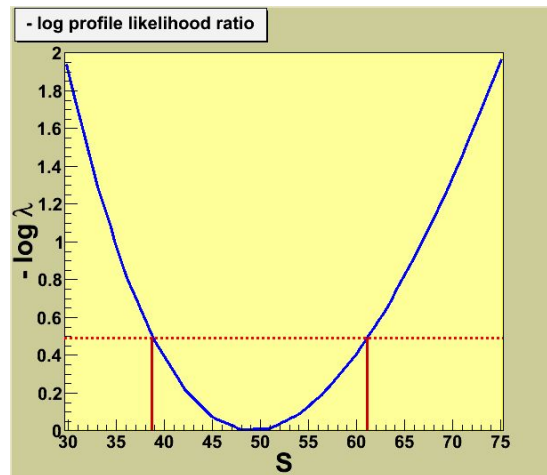
- Errors returned by the fit are computed from the second derivatives of the log-likelihood function
 - Assume the negative log-likelihood function is a parabola around the minimum
 - This is true asymptotically and in this case the parameter estimates are also normally distributed.
 - The estimated correlation matrix is then:

$$\hat{\mathbf{V}}(\hat{\boldsymbol{\theta}}) = \left[\left(-\frac{\partial^2 \ln L(\mathbf{x}; \boldsymbol{\theta})}{\partial^2 \boldsymbol{\theta}} \right)_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} \right]^{-1} = \mathbf{H}^{-1}$$

Parameter Errors

- A better approximation to estimate the confidence level of the parameter is to use directly the log-likelihood function and look at the difference from the minimum.
 - Method of Minuit/Minos (Fit option “E”)
 - obtain a confidence interval which is in general not symmetric around the best parameter estimate

```
auto r = h1->Fit(f1,"E S");  
r->          (par_number);  
r->          (par_number);
```



Minimization

- The fit is done by minimizing the least-square or likelihood function.
- A direct solution exists only in case of linear fitting
 - it is done automatically in such cases (e.g fitting polynomials).
- Otherwise an iterative algorithm is used:
 - Minuit is the minimization algorithm used by default
 - ROOT provides two implementations: Minuit and Minuit2
 - other algorithms exists: Fumili, or minimizers based on GSL, genetic and simulated annealing algorithms
 - To change the minimizer:

```
ROOT::Math::MinimizerOptions::SetDefaultMinimizer("Minuit2");
```

- Other commands are also available to control the minimization:

```
ROOT::Math::MinimizerOptions::SetDefaultTolerance(1.E-6);
```


Function Minimization

- Common interface class (**ROOT::Math::Minimizer**)
- Existing implementations available as plug-ins:
 - **Minuit** (based on class **TMinuit**, direct translation from Fortran code)
 - with Migrad, Simplex, Minimize algorithms
 - **Minuit2** (C++ implementation with OO design)
 - with Migrad, Simplex, Minimize and Fumili2
 - **Fumili** (only for least-square or log-likelihood minimizations)
 - **GSLMultiMin**: conjugate gradient minimization algorithm from GSL (Fletcher-Reeves, BFGS)
 - **GSLMultiFit**: Levenberg-Marquardt (for minimizing least square functions) from GSL
 - **Linear** for least square functions (direct solution, non-iterative method)
 - **GSLSimAn**: Simulated Annealing from GSL
 - **Genetic**: based on a genetic algorithm implemented in TMVA
- All these are available for ROOT fitting and in RooFit/RooStats
- Possible to combine them (e.g. use Minuit and Genetic)

Comments on Minimization

- Sometimes fit does not converge

Warning in <Fit>: Abnormal termination of minimization.

- can happen because the Hessian matrix is not positive defined
 - e.g. there are no minimum in that region → wrong initial parameters;
- numerical precision problems in the function evaluation
 - need to check and re-think on how to implement better the fit model function;
- highly correlated parameters in the fit. In case of 100% correlation the point solution becomes a line (or an hyper-surface) in parameter space. The minimization problem is no longer well defined.

PARAMETER		CORRELATION COEFFICIENTS		
NO.	GLOBAL	1	2	
1	0.99835	1.000	0.998	
2	0.99835	0.998	1.000	

Signs of trouble...

Mitigating fit stability problems

- When using a polynomial parametrization:
 - $a_0 + a_1x + a_2x^2 + a_3x^3$ nearly always results in strong correlations between the coefficients.
 - problems in fit stability and inability to find the right solution at high order
- This can be solved using a better polynomial parametrization:
 - e.g. Chebychev polynomials

$$T_0(x) = 1$$

$$T_1(x) = x$$

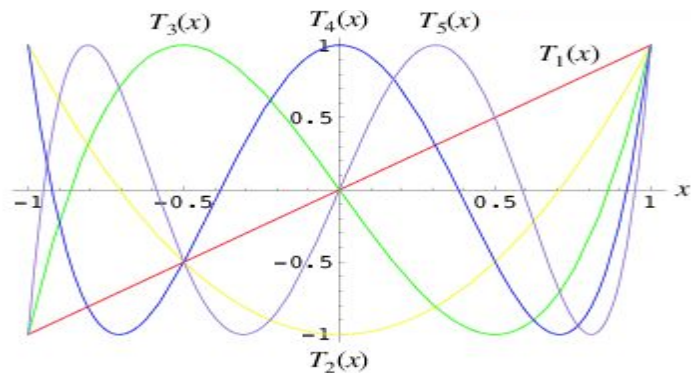
$$T_2(x) = 2x^2 - 1$$

$$T_3(x) = 4x^3 - 3x$$

$$T_4(x) = 8x^4 - 8x^2 + 1$$

$$T_5(x) = 16x^5 - 20x^3 + 5x$$

$$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1.$$



The Fit Panel

- The fitting in ROOT using the FitPanel GUI
 - GUI for fitting all ROOT data objects
- Using the GUI we can:
 - select data object to fit
 - choose (or create) fit model function
 - set initial parameters
 - choose:
 - fit method (likelihood, chi2)
 - fit options (e.g Minos errors)
 - drawing options
 - change the fit range

