# Table of Contents

# ROOT Tutorials Exercises for School


## Topics

This tutorial aims to provide a solid base to efficiently analyse data with ROOT. The main features of ROOT are presented: histogramming, data analysis using trees and advanced fitting techniques.

## Material for the course

The slides of the lectures are available in electronic form from the school Agenda page. As complementary material, one can look at

- ROOT Primer Guide, available in pdf, html or epub format. This introductory guide illustrates the main features of ROOT, relevant for the typical problems of data analysis: input and plotting of data from measurements and fitting of analytical functions.
- ROOT user guide. It can be downloaded in various format (or only individual chatters) from here.
- RooFit User Guide, available in pdf format. A coincide RooFit quick start guide is also available here.
- A tar file with all the exercise solutions (all the ROOT macro required) is available here

## Start using ROOT

We will focus first on introductory exercises for getting started working with the ROOT environment and writing our first ROOT macro. Two levels of help will be provided: a hint and a full solution. Try not to jump to the solution even if you experience some frustration. The help is organised as follow:

Hint  Hide
Here the hint is shown.

Solution  Hide
Here the solution is shown.

Some points linked to the exercises are optional and marked with a icon: they are useful to scrutinise in more detail some aspects. Try to solve them if you have the time.

### Setting up ROOT

ROOT is installed in the virtual machines prepared for the course. It should be automatically available from the Terminal. If not, one should run the fallowing lines (or put in the default login shell script):

```
export ROOTSYS=/usr/local/ROOT
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
. $ROOTSYS/bin/thisroot.sh
```

The version of ROOT available in the school virtual machines which should be 5.34.09.

## Start using the ROOT prompt

First of all open a terminal window on your computer and type

```
root
```

If this does not work, then it means ROOT is not properly installed in your system. Stop here and ask for help to fix the installation. If it works then start playing with the prompt. Use as a calculator and type:

```
root [0] 2+2
```

or whatever you like (see for example the lecture slide) or page 5 of the booklet ("A ROOT Guide for Beginners"). Note that after having issue a statement from the ROOT prompt, you can omit the `;` required by C++. This will make ROOT printing the returned value, if there is one. For example:

```
root [0] TMath::Pi();
```

will not print anything, while

```
root [0] TMath::Pi()
(Double_t)3.14159265358979312e+00
```

Afterwards start writing your first ROOT macro. A ROOT macro is a file containing some C++ code which can be run from the ROOT prompt. You need to define a function and in the function scope you write the code. For example, you create a file, which you will call `mymacro.C`. Inside the file you define a function `mymacro(int value)` and you write the code of slide 8 of the lecture (or something else, if you prefer). The you can run the macro from the ROOT prompt by typing:

```
root [0] .x mymacro.C(42)
```

Note that if the function name is different than the macro (file) name, you need two steps to run. First you load the macro:

```
root [0] .L mymacro.C
```

Then you run the function in the macro. Let's assume the function name is `test(int value)`:

```
root [1] test(42)
```

After having followed this introduction, you can try to move to the first exercise.

## Exercise 1: Plotting a Function in ROOT

Following slide 13 of the lecture or page 6 of the Introductory Guide, create a TF1 class using the `sin(x)/x` function and draw it.

Create then a function with parameters, `p0 * sin ( p1 * x) /x` and also draw it for different parameter values. You can try to change the parameter values using `TF1::SetParameters` or by using the ROOT GUI. Try also to change the style of the line and its color. You can either use the ROOT GUI and/or use the methods of the class `TAttLine` (see `TAttLine` reference documentation). Since TF1 derives from `TAttLine`, it inherits all its functions. Try for example to set the colour of the parametric function to blue.

After having drawn the function, compute for the parameter values (p0=1, p1=2):

- function value for x = 1.
- function derivative for x = 1
- integral of the function between 0 and 3.

Hint Hide
To solve the exercise you need to :

- create a TF1 object using a formula expression. In the case of a parametric functions, the two parameters are defined as [0] and [1]
- call `TF1::Draw()`
- call `TF1::Eval(x)`, `TF1::Derivative(x)` and `TF1::Integral(a,b)`

You can also find the available member functions of the class TF1, by using the Tab key on the ROOT prompt, for example

```
root [0] TF1 * f1 = ....
root [1] f1-><TAB>
```

And you can get the full signature of a method by doing for example:

```
root [1] f1->Derivative(<TAB>
```

Solution Hide

```
#include "TF1.h"

void plotFunction() {

   TF1 * f1 = new TF1("f1","sin(x)/x",0,10);
   f1->Draw();


   TF1 * fp = new TF1("fp","[0]*sin([1]*x)/x",0,10);
   fp->SetParameters(1,2);
   fp->Draw("same");
   fp->SetLineColor(kBlue);

   // to change axis y margins
   // (or invert the order of plotting the functions)
   f1->SetMaximum(2);
   f1->SetMinimum(-2);

   std::cout << "Value of f(x) at x = 1 is       " << fp->Eval(1.) << std::endl;
   std::cout << "Derivative of f(x) at x = 1 is  " << fp->Derivative(1.) << std::endl;
   std::cout << "Integral of f(x) in [0,3] is    " << fp->Integral(0,4)  << std::endl;

}
```

You can try to use a different function, for example the Gamma distribution, defined in `ROOT::Math::gamma_pdf`. Try to make a plot as the one in Wikipedia, see here, where the function is plot for different parameter values. See the here the reference documentation of the gamma distribution in ROOT.

Solution Hide

```
#include "TF1.h"
#include "Math/DistFunc.h"


void plotGamma() {

   // Note that parameter [0] is called alpha in definition of gamma_pdf or kappa in Wikipedia
   // Note that parameter [1] is  theta
```

```
   // use range [0,20] as in Wikipedia plot
   TF1 * f1 = new TF1("f","ROOT::Math::gamma_pdf(x,[0],[1])",0,20);

   f1->SetLineColor(kRed);
   f1->SetParameter(0,1);
   f1->SetParameter(1,2);

   // use DrawClone because we will plot many different copies of same object but with different
   // parameter values
   f1->DrawClone();


   // now change parameters and draw at different parameter values
   f1->SetLineColor(kGreen);
   f1->SetParameter(0,2);
   f1->DrawClone("SAME");

   f1->SetLineColor(kBlue);
   f1->SetParameter(0,3);
   f1->DrawClone("SAME");

   f1->SetLineColor(kCyan);
   f1->SetParameter(0,5);
   f1->SetParameter(1,1);
   f1->DrawClone("SAME");


   f1->SetLineColor(kOrange);
   f1->SetParameter(0,9);
   f1->SetParameter(1,0.5);
   f1->DrawClone("SAME");


}
```

## Exercise 2: Making an histogram in ROOT

We will learn in this exercise how to create a one-dimensional histogram in ROOT, how to fill it with data and how to plot it.

Create a one-dimensional histogram with 50 bins between 0 10 and fill it with 10000 gaussian distributed random numbers with mean 5 and sigma 2. Plot the histogram and, looking at the documentation in the THistPainter, show in the statistic box the number of entries, the mean, the RMS, the integral of the histogram, the number of underflows, the number of overflows, the skewness and the kurtosis.

Hint Hide
For generating gaussian random numbers use `gRandom->Gaus(mean, sigma)`

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TStyle.h"

void plotHistogram() {

   TH1D * h1 = new TH1D("h1","h1",50,0,10);

   for (int i = 0; i < 10000; ++i) {
      double x = gRandom->Gaus(5,2);
      h1->Fill(x);
   }
```

```
   h1->Draw();

   gStyle->SetOptStat(111111110);

}
```

After calling the function `TH1::ResetStats()` , you will see that the statistics (mean, RMS,..) of the histogram is slightly different. Try to understand the reason for this, by trying for example to compute the mean of the histogram yourself.

Solution Hide
The initial statistics is computed using the original (un-binned) data, while after calling `TH1::ResetStats()`, the statistics is computed using the bin contents and centres (binned data).

The following macro, creating, filling and plotting histogram contains an error, which one ?

```
#include "TH1.h"

void testPlotHistogram() {

   TH1D h1("h1","h1",50,-5,5);
   h1.FillRandom("gaus",10000);
   h1.Draw();

}
```

Solution Hide
The histogram objected is deleted at the end of the macro, therefore it is not shown in the plot after exiting the macro. To fix it, either call `TH1::DrawClone` or create the histogram using operator new as in the previous example.

## Exercise 3: Histograms Operations

We will work in this exercise on the histogram operations like addition, subtraction and scaling.

- Make a gaussian filled histogram between 0 and 10 with 100 bis and 1000 entries with mean 5 and sigma 1.
- Make another histogram uniformly distributed between 0 and 10 with 100 bins and 10000 entries.
- Add the two histogram into a new one using `TH1::Add`
- Make another histogram, still with 100 bins but with 100000 entries. Normalize this histogram to have a total integral of 10000 using `TH1::Scale`.
- Subtract now from the histogram which contains the sum of the flat and the gaussian histograms, the flat normalised histogram, using `TH1::Add`
- Plot the result using the error option (`h1->Draw("E")`). Do the error make sense ? If not, how can you get the correct bin errors ?

Hint Hide

- Use `TH1::Add` to add the two histogram.
- Use `TH1::Scale(10000/ 100000)` to re-normalise the histogram.
- Use again `TH1::Add` to subtract the histogram, but with a second coefficient equal to -1. (`TH1::Add(h1,h2,1,-1)`).
- For getting the right errors you must call `TH1::Sumw2` before doing the operations on the histograms (i.e. before scaling and before subtracting them)

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TCanvas.h"
#include "TLine.h"
#include <iostream>

void HistoOperations() {


   TH1D * h1 = new TH1D("h1","flat histogram",100,0,10);
   for (int i = 0; i < 10000; ++i) {
      h1->Fill( gRandom->Uniform(0,10) );
   }

   TH1D * h2 = new TH1D("h2","gaus histogram",100,0,10);
   for (int i = 0; i < 1000; ++i) {
      h2->Fill( gRandom->Gaus(5,1) );
   }

   TH1D * h3 = new TH1D("h3","flat+gaus histogram",100,0,10);
   h3->Add(h1,h2);

   h3->Draw();


   TH1D * h4 = new TH1D("h4","second flat histogram",100,0,10);
   for (int i = 0; i < 100000; ++i) {
      h4->Fill( gRandom->Uniform(0,10) );
   }

   // renormalize histogram
   //!! VERY IMPORTANT - NEED TO CALL Sumw2() to get right ERRORS!!!
   h4->Sumw2();
   h3->Sumw2();
   h4->Scale(0.1);

   // plot in a new Canvas
   new TCanvas("c2","c2");


  TH1D * h5 = new TH1D("h5","(flat+gaus) histogram - flat histogram",100,0,10);
  h5->Add(h3,h4,1.,-1.);
  h5->Draw("E");

  TLine * line = new TLine(0,0,10,0);
  line->Draw();


}
```

Rebin now of the histogram (e.g. the one resulting at the end from the subtraction) in a new histogram with bins 4 times larger.

Hint Hide
For rebinning in 4 times larger bins, use `TH1::Rebin` with `ngroup=4`.

Solution Hide
Add these lines of code at the end of the macro

```
new TCanvas("c2","c2");

  TH1 * h6 = h5->Rebin(4,"h6");
  h6->SetTitle("Rebinned histogram");
  h6->Draw();
```

## Exercise 4: Multi-Dimensional Histograms and Profiles

Create a 2 dimensional histogram with x and y in the range [-5,5] and [-5,5] and 40 bins in each axis. Fill the histogram with correlated random normal numbers. To do this generate 2 random normal numbers (mean=0, sigma=1) u and w. Then use `x = u` and `y=w+0.5*u` for filling the histogram. Plot the histogram using color boxes (See documentation in THistPainter class) or choose what ever option you prefer. After having filled the histogram, compute the correlation using `TH1::GetCorrelationFactor`.

Hint Hide
The option for plotting colour boxes is "COLZ", which draws also a palette for the scale on the Z axis (the bin content)

Solution Hide

```
#include "TH2.h"
#include "TProfile.h"
#include "TRandom.h"
#include "TCanvas.h"
#include <iostream>

void Histogram2D() {


   TH2D * h2d = new TH2D("h2d","2d histogram",40,-5,5, 40, -5, 5);
   for (int i = 0; i < 100000; ++i) {
      double u =  gRandom->Gaus(0,1);
      double w =  gRandom->Gaus(0,1);
      double x = u;
      double y = w + 0.5 * u;
      h2d->Fill( x,y );
   }

   h2d->Draw("COLZ");

   std::cout << "correlation factor " << h2d->GetCorrelationFactor() << std::endl;


}
```

Make then a projection of the 2-dimensional histogram on the x. Make also a projection of the y axis into a profile. Plot the resulting projected histograms in a new canvas separated in 2 pads.

Hint Hide
For making the projection call `TH1::ProjectionX` and for making the profile call `TH1::ProfileX`

For dividing the canvas call `TCanvas::Divide(1,2)` and navigate in the pad contained in the canvas by calling `TCanvas::cd(pad_number)`.

Solution Hide
Add these lines of code at the end of the macro

```
TH1 * hx = h2d->ProjectionX();
   TH1 * px = h2d->ProfileX();

   TCanvas * c2 = new TCanvas("c2","c2");
   // divide in 2 pad in x and one in y
   c2->Divide(2,1);
   c2->cd(1);
   hx->Draw();

   c2->cd(2);
   px->Draw();
```

Look at the bin error of the profile. Do you know how it is computed ? You can find this answer in the TProfile reference documentation.

If you have still time, after having finished the exercises you can look at some of the tutorials in the ROOT distribution directory, $ROOTSYS/tutorials/hist. They are available on the Web at this location. For example look at:

- hlabels1.C
- hstack.C
- rebin.C
- sparsehist.C

# ROOT I/O and Trees

This is the session dedicated on working with the Trees in ROOT. First will start with an exercise on the I/O of ROOT by storing and reading an histogram from a file

## Exercise 5: Writing and Reading Histogram from a file

Open a file then create a simple histogram, for example an histogram generated with exponential distribution. Fit it and write it in a file. Why the ROOT Canvas does not show the histogram ?

Hint Hide
Use `TFile::Open` to open the file or just create a `TFile object`. Call `TH1::Write` to write the histogram in the file after having filled it.

Solution Hide
```
#include "TFile.h"
#include "TH1.h"
#include "TRandom.h"

void histogramWrite() {

   TFile f("histogram.root","RECREATE");

   TH1D * h1 = new TH1D("h1","h1",100,0,10);
   for (int i = 0; i < 10000; ++i)
      h1->Fill(gRandom->Exp(5) );

   h1->Fit("expo");
   h1->Draw();

   f.Write("h1");
   f.Close();
}
```

The histogram is not shown, becausem when the file is close, it is automatically deleted.

Now read the histogram from the file and plot it.

Hint Hide
Create a file object (or call `TFile::Open`) and then `TFile::Get`

Solution Hide
```
void histogramRead() {

   TFile * file = new TFile("histogram.root");
```

```
   TH1 * h1 = 0;
   file->GetObject("h1",h1);
   // you can also use nut you need to cast if you compile the code
   //TH1 * h1 = (TH1*) file->Get("h1");

   h1->Draw();
}
```

You can also use the `TBrowser` to open the file and display the histogram.

What is going to happen if you delete the file after having retrieved the histogram from the file ?

## Exercise 5: Creating a ROOT Tree

Create a ROOT tree which contains an EventData object. The Event data object is defined in the file below

Show Hide

```
#include <vector>

class Particle {
public:
   Particle() {memset(fTags, 0, sizeof(fTags)); }
   double fPosX,fPosY,fPosZ; // particle position nearest to interaction point
   double fMomentum; // particle momentum
   double fMomentumPhi; // particle direction (phi)
   double fMomentumEta; // particle direction (eta)
   Long64_t fTags[128]; // particle tags
};

class EventData {
public:
   std::vector<Particle> fParticles; // particles of the event
   int fEventSize; // size (in bytes) of the event

   void SetSize() {
      fEventSize = sizeof(EventData) + fParticles.size() * sizeof(Particle);
   }
   void Clear() {
      fParticles.clear();
   }
   void AddParticle(const Particle& p) { fParticles.push_back(p); }

   //ClassDef(EventData,1); // Data for an event
};

#ifdef __MAKECINT__
#pragma link C++ class Particle+;
#pragma link C++ class EventData+;
#endif
```

The macro creating the tree and filling with them with the events is below

Show Hide

```
// Prefer compiled:
#include "TTree.h"
#include "TFile.h"
#include "TRandom.h"
#include "TMath.h"
#include <vector>

#include "EventData.h"

void createTree(ULong64_t numEvents = 200) {
```

```
   TFile* f = new TFile("eventdata.root", "RECREATE");
   TTree* tree = new TTree("EventTree", "Tutorial tree");

   EventData* event = new EventData();
   tree->Branch("event", &event);

   Particle p;

   for (ULong64_t i = 0; i < numEvents; ++i) {       event->Clear();

      // generate event information
      int nParticles = 10 * gRandom->Exp(10.);

      // for every particle in the event generate the particle data
      for (int ip = 0; ip < nParticles; ++ip) {          do {             p.fPosX = gRandom->Gaus
               + gRandom->BreitWigner(0.1, 0.1);;
         } while (fabs(p.fPosX) > 10.);
         p.fPosY = gRandom->Gaus(gRandom->PoissonD(0.01), .7);
         p.fPosZ = gRandom->Gaus(gRandom->PoissonD(10), 19.);

         p.fMomentum = gRandom->Exp(12);
         p.fMomentumPhi = gRandom->Uniform(2*TMath::Pi());
         do {
            p.fMomentumEta = gRandom->BreitWigner(0.01, 10.);
         } while (fabs(p.fMomentumEta) > 12.);


         event->AddParticle(p);
      }
      event->SetSize();

      tree->Fill();

      if (i % (numEvents/50) == 0) {
         printf("*");
         fflush(stdout);
      }
   }
   printf("\n");
   tree->Write();
   delete f;
}
```

Look at the macro and try to understand. Run the macro to create and write the tree in the file.

## Exercise 6: Reading a ROOT Tree

Read the tree from the file and by using the TBrowser or TTree::Draw plot for example the momentum of all the particle or the event size.

Now read the Tree with a macro and calculate the sum of all event sizes.

Hint  Hide

- Open the file using its file name in `TFile::Open()` and get the Tree. Remember to check if the file pointer is not null. If it is null means the file is not existing.
- Get then a pointer to the tree.
- Connect a Tree Branch with the Data Member.We have to somehow connect the branch we want to read with the variables used to actually store the data by calling `TTree::SetBranchAddress()`.
- Load the TTree data. For the analysis example we need to access the events' size, which is stored in the variable `eventSize`. But the TTree first needs to load the data for each event it contains. For that call TBranch::GetEntry(entry) in a loop, passing the TTree entry number from the loop index to

`GetEntry()`. Again `TBranch` is the class name, but you obviously need to call it on an object. To know how many entries the tree contains, simply call TTree::GetEntries(). The branch is stored in eventSizeBranch.

- In the same loop, compute the total size of all events (simply add the current event size to the total size)
- Without the call to `GetEntry()`, the variables will not contain data. `GetEntry()` loads the data into the variables connected with the tree by the call to `SetBranchAddress()`.
- Access the Analysis Result. At the end of the loop, print the sum of all event sizes. This sum shows you the real power of a TTree: even though you can analyze large amounts of data (our example tree with 22MB is tiny!) ROOT needs just a few MB of your RAM, no matter how many events you analyze. Imagine what it would be like if you had to load all data into memory, e.g. using a simple vector<EventData>

Solution Hide

```
#include "TFile.h"
#include "TTree.h"
#include "TBranch.h"

void AnalyzeTree()
{
   // Variables used to store the data
   Int_t     totalSize = 0;        // Sum of data size (in bytes) of all events
   Int_t     eventSize = 0;        // Size of the current event
   TBranch  *eventSizeBranch = 0;  // Pointer to the event.fEventsize branch


   // open the file
   TFile *f = TFile::Open("eventdata.root");
   if (f == 0) {
      // if we cannot open the file, print an error message and return immediatly
      printf("Error: cannot open eventdata.root!\n");
      return;
   }
   // get a pointer to the tree
   TTree *tree = (TTree *)f->Get("EventTree");



   // To use SetBranchAddress() with simple types (e.g. double, int)
   // instead of objects (e.g. std::vector<Particle>).
   tree->SetMakeClass(1);

   // Connect the branch "fEventSize" with the variable
   // eventSize that we want to contain the data.
   // While we are at it, ask the tree to save the branch
   // in eventSizeBranch
   tree->SetBranchAddress("fEventSize", &eventSize, &eventSizeBranch);



   // First, get the total number of entries
   Long64_t nentries = tree->GetEntries();
   // then loop over all of them
   for (Long64_t i=0;i<nentries;i++) {
      // Load the data for TTree entry number "i" from branch
      // fEventSize into the connected variable (eventSize):
      eventSizeBranch->GetEntry(i);
      // compute the total size of all events
      totalSize += eventSize;
   }


   Int_t sizeInMB = totalSize/1024/1024;
   printf("Total size of all events: %d MB\n", sizeInMB);
```

```
}
```

You can also download the macro AnalyzeTree.C.

## Exercise 7: Chaining ROOT Trees.

Create a TTree now containing for each event simple double variables (like a tuple), for example, `x`, `y`, `z` and `t`. Run the macro few times, but changing always the name of the file where the tree is stored. Use then the =TChain class to merge the trees

Hint Hide
You can look at the tutorial `tree/tree1.C` as example, in particular how `TTree:Branch` is used to define the tree branches containing the variables. Then see the example in the lecture slide on how to use `TTree::Chain`.

Solution Hide
Here is for example the macro to create several trees

```
#include "TRandom.h"
#include "TFile.h"
#include "TTree.h"

void SimpleTree(const char * filename= "tree.root") {

   TTree data("tree","Example TTree");
   double x, y, z, t;
   data.Branch("x",&x,"x/D");
   data.Branch("y",&y,"y/D");
   data.Branch("z",&z,"z/D");
   data.Branch("t",&t,"t/D");

// fill it with random data
   for (int i = 0; i<10000; ++i) {
      x = gRandom->Uniform(-10,10);
      y = gRandom->Gaus(0,5);
      z = gRandom->Exp(10);
      t = gRandom->Landau(0,2);

      data.Fill();
   }
// write in a file
   TFile f(filename,"RECREATE");
   data.Write();
   f.Close();

}
```

%

This are the few lines to create the TChain, that you can run directly from the prompt. You can also use wildcard's to chain many files

```
TChain chain("tree");
chain.Add("tree*.root")
chain.Draw("t")
```

## Exercise 8: Using Tree Friends

Make a Tree with three variables (x,u) and you fill with some random variables that you prefer. Read from the file the Tree used before and add as a friend to this tree. Plot the x variable of the first tree versus the x

variable of the second one.

Solution Hide
Here is the macro to create a second tree, containing x, u:

```
#include "TRandom.h"
#include "TFile.h"
#include "TTree.h"

void SimpleTree2() {

   TTree data("tree_2","Example TTree");
   double x, u;
   data.Branch("x",&x,"x/D");
   data.Branch("u",&u,"u/D");


// fill it with random data
   for (int i = 0; i<10000; ++i) {
      x = gRandom->Exp(100);
      u = gRandom->Uniform(0,10);

      data.Fill();
   }
// write in a file
   TFile f("tree_2.root","RECREATE");
   data.Write();
   f.Close();

}
```

Here are the lines of codes to Draw the `x` of the first tree versus the `x` of the second tree with a selection depending on `u` and `t`. You can run these lines from the ROOT prompt.

```
TFile f("tree.root");    // to get  the first tree
 tree->AddFriend("tree_2","tree_2.root");
tree->Draw("x:tree_2.x","t<100 && tree_2.u<6","COLZ");
```

## Exercise 9: Using the TSelector class for analysing a TTree

Create your own Selector for the simple (x,y,z,t) TTree made before. Use TTree::MakeSelector to create your own Selector class. Inside the code of your Selector do the following:

- book an histogram in the initialisation routine, for one of the variable of the tree (e.g. the variable `t` )
- fill the histogram in the Process function
- draw the histogram in the Terminate function

Hint Hide
Here is what you need to do, after having opened the file with the tree

```
tree->MakeSelector("MySelector.C");
```

The file `MySelector.h` and `MySelector.C` will be created. Add in `MySelector.h`, inside the class `MySelector`, a new data member, the histogram you want to create,

```
TH1D * h_t;
```

Edit then the file MySelector.C and add in `MySelector::SlaveBegin` the booking of the histogram.

```
h_t = new TH1D("h_t","t",100,0,100);
```

In `MySelector::Process` the filling of the histogram after calling `TSelector::GetEntry()`

```
GetEntry(entry);
h_t->Fill(t);
```

In `MySelector::Terminate` the drawing of the histogram.

After having saved the file run the selection by doing (for example from the ROOT prompt):

```
TFile f("tree.root");
tree->Process("MySelector.C+");
```

Solution Hide
See the attached file MySelector.h and MySelector.C.

# Fitting in ROOT

Welcome to the hands-on session dedicated on fitting in ROOT

## Exercise 10: Gaussian fit of an histogram

We will start with an exercise where we fin an histogram with a simple function, to get familiar with the fitting options in ROOT.

- Start creating an histogram with 50 bins in [-5,5] and fill with 1000 Gaussian distributed number
- Fit the histogram using a Gaussian function
- Get the value and error of the width of the gaussian
- Retrieve the fit result and print the correlation matrix.

Hint Hide
To solve the exercise you need first to create first the TF1 object, either using the pre-defined gaus function or by using a formula expression with `"[0]*ROOT::Math::normal_pdf(x,[2],[1])"`. Remember that in the second case you need to set the initial function parameters (e.g. `f1->SetParameters(1,0,1)` ).

To get access to the `TFitResult` object after fitting use option "S", as shown in slide 10 of the lecture.

Use `gStyle->SetOptFit(1)` to display the fit result in the statistics box.

Solution Hide

```
#include "TF1.h"
#include "TH1.h"
#include "TFitResult.h"
#include "TMatrixDSym.h"
#include "TStyle.h"

void gausFit() {

   TH1D * h1 = new TH1D("h1","h1",50,-5,5);

   h1->FillRandom("gaus",1000);

   TF1 * f1 = new TF1("f1","gaus");

   // add also option "Q" (quite) to avoid prrinting two time the result
   TFitResultPtr r = h1->Fit(f1,"S Q");

   // print the result
   r->Print();
```

```
   // get the correlation matrix and print it
   TMatrixDSym corrMatrix = r->GetCorrelationMatrix();
   corrMatrix.Print();

   gStyle->SetOptFit(1);

  // to get the sigma of the gaussian
  std::cout  << "Gaussian sigma = " << f1->GetParameter("sigma") << "  +/- " << f1->GetParError(f

}
```

If you repeat the fit few times (without exiting ROOT) you will see that the sigma you obtain is almost always less than 1. The result is then slightly bias. Try to perform a likelihood fit (option "L"). Is the result better ?

Solution Hide
The reason is that the lest square fit is not correct in case of low statistics. The bins with zero events are not included in the fit and this bias the result. The likelihood fit is the correct method for fitting count histograms.

You can notice that the sigma and the amplitude of the gaussian are quite correlated. Can you have a better parametrisation ?

Solution Hide
You can fit using a normalised Gaussian. In this case you get a much smaller correlation. To do this create the TF1 as following:

```
TF1 * f1 = new TF1("f1","[0]*ROOT::Math::normal_pdf(x,[2],[1])");
// set the parameters (needed if not using a pre-defined function as "gauss")
 f1->SetParameters(1,0,1);
```

## Exercise 11: Fit a peak histogram

We are going to fit the histogram with a more complicated function. First we create an histogram with a gaussian peak and an exponential background

- Create an histogram with 50 bins between 0 and 10.
- Fill with 100 gaussian number with mean 2 and width 0.3.
- Fill with 1000 exponential number with a decay length of 2.
- Create a function composed of the gaussian plus the exponential and fit to the histogram. Do the fit works ? What do you need to do to make the fit working ?
- Compute the number of peak events, by using the integral of the Gaussian function. Use `TF1::IntegralError` to compute also its error.

Hint Hide
Before fitting you need to set sensible parameter values. You can do this by fitting first a single gaussian in the range [1,3] and then the exponential separately. If you don't set good initial parameter values, the fit will probably not converge.

After the fit works, you can compute the number of peak events, by using `TF1::Integral` on the Gaussian only function. For the error you can use `TF1::IntegralError`, but you need to extract the correlation matrix from the fit and use the sub-matrix referring to the gaussian part.

Solution Hide

```
#include "TF1.h"
#include "TH1.h"
#include "TFitResult.h"
#include "TMatrixDSym.h"
```

```
#include "TRandom.h"
#include "TStyle.h"

void PeakFit() {

   TH1D * h1 = new TH1D("h1","h1",50,0,10);

   for (int i = 0; i < 100; ++i) {         h1->Fill(gRandom->Gaus(2,0.3));
   }
   for (int i = 0; i < 1000; ++i) {          h1->Fill(gRandom->Exp(2));
   }



   // fit first a single gaussian in range [0,5]
   TFitResultPtr r1 = h1->Fit("gaus","S","",1,3);  // first fit of gaussian
   TFitResultPtr r2 = h1->Fit("expo","S");  // first exponential

   TF1 * f1 = new TF1("fitFunc","gaus(0)+expo(3)");
   // get parameters and set in global TF1

   // parameters of first gaussian
   f1->SetParameter(0,r1->Parameter(0));
   f1->SetParameter(1,r1->Parameter(1));
   f1->SetParameter(2,r1->Parameter(2));
   // parameters of second gaussian
   f1->SetParameter(3,r2->Parameter(0));
   f1->SetParameter(4,r2->Parameter(1));



   TFitResultPtr res = h1->Fit(f1,"SL");

   TMatrixDSym cov = res->GetCovarianceMatrix();

   // compute number of signal events
   // fitted gaussian function
   TF1 * peakFunc = new TF1("peakFunc","gaus");
   peakFunc->SetParameter(0, f1->GetParameter(0));
   peakFunc->SetParameter(1, f1->GetParameter(1));
   peakFunc->SetParameter(2, f1->GetParameter(2));

   TMatrixDSym covPeak = cov.GetSub(0,2,0,2);

   double nsignal = peakFunc->Integral(0,10) / h1->GetBinWidth(1);
   double err  = peakFunc->IntegralError(0,10,peakFunc->GetParameters(), covPeak.GetMatrixArray()

   std::cout << "number of signal events = " << nsignal << " +/- " << err << std::endl;









}
```

You can try to use a different minimiser, like Minuit2. To run an alternative minimiser do:

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

## Exercise 12: Using the Fit Panel GUI

Repeat the previous exercise, but by using the Fit Panel GUI

Hint  Hide

- Select the fit panel in the Canvas/Tools menu or by right-clicking on the histogram
- Make the fit function (gaussian plus exponential) using the Operation/Add button
- Set the initial parameters by playing with the sliders
- Press the Fit button

Explore the other functionalities of the fit panel like changing the fit method, use a different minimiser, plotting contours in the parameters.

If you want to know more about fitting, you can look at some of the tutorials in the ROOT distribution directory, `$ROOTSYS/tutorials/fit`. They are available on the Web at this location. For example look at:

- FittingDemo.C
- NumericalMinimization.C
- ErrorIntegral.C
- fitMultiGraph.C

# Fitting using RooFit

Welcome to the hands-on session dedicated on fitting using RooFit. The aim is to start familiarizing with RooFit and trying understand the basic syntax of creating models using the workspace factory. We will also see how to save a workspace in a ROOT file which, allowing to perform the fitting analysis at a later stage or to share the models with other people.

RooFit provides also a separate User Guide (unfortunately it does not cover yet the workspace factory syntax), but it exists also a reduced booklet, showing the main functionality, which you can find here.

## Exercise 13: Gaussian model and fit it to random generated data

We will start with a similar exercise we did for Root fitting. We will create a Gaussian model from which we will generate a pseudo-data set and then we will fit this data set.

Start directly creating the Gaussian model using the workspace factory, thus the syntax introduced in the lecture slide 15. One you have created the model, use the `generate()` method of the `RooAbsPdf` class to generate 1000 events. Try to plot the data set using `RooPlot` as shown in slide 12.

After, fit the model to the data and show the resulting fitted function as in slide 14.

At the end save the `RooWorkspace` object in a file, but before remember to import, by calling `RooWorkspace::import(data)`, the data set you have generated in the workspace. The workspace does not contains only the model, but also the data, allowing then to re-perform the analysis later on.

Hint  Hide

- Use the syntax of the `RooWorkspace` factory to create first the variables (observables and parameters) of the Gaussian probability density function (p.d.f.), as shown in slide 15.

Every variable in RooFIt, when created needs to be created with a value a min and a max allowed range. Use

very large value if you don't know the range. If you provided only a value, the variable is considered constant. If you provide only the minimum and the maximum, the initial value will be taken as half the range. To avoid undesired side effect the value given should be defined between the min and max of the given range.

- You need to define the [value, min, max] of a variable only the first time you create in the factory. Afterwards you can reference the variable by its name.

See also slide 21 to create the p.d.f.

- After you have created the variable and p.d.f in the workspace, you can access their pointers, by using `RooWorkspace::var` for variables or `=RooWorkspace::pdf` for p.d.f.
- You can then do as in slide 12 to generate the data set and plot it.
- To fit the data set with the pdf, you need to call the `RooAbsPdf::fitTo`. See slide 14 on how to fit and on how to plot the function after the fit.
- To save the model call, `RooWorkspace::write(file_name)`. See slide 19.

Below is the code solution:

Solution Hide

```
// make a simple Gaussian model

#include "RooWorkspace.h"
#include "RooRealVar.h"
#include "RooAbsPdf.h"
#include "RooDataSet.h"
#include "RooPlot.h"

#include "TCanvas.h"

using namespace RooFit;

void GaussianModel(int n = 1000) {


   RooWorkspace w("w");
   // define a Gaussian pdf
   w.factory("Gaussian:pdf(x[-10,10],mu[1,-1000,1000],s[2,0,1000])");

   RooAbsPdf * pdf = w.pdf("pdf");   // access object from workspace
   RooRealVar * x = w.var("x");   // access object from workspace


   // generate n gaussian measurement for a Gaussian(x, 1, 1);
   RooDataSet * data = pdf->generate( *x, n);

   data->SetName("data");

   // RooFit plotting capabilities
   RooPlot * pl = x->frame();
   data->plotOn(pl);
   pl->Draw();


   // remove this line if you want to fit the data
   return;

   // now fit the data set
   pdf->fitTo(*data);


   // plot the pdf on the same RooPlot object we have plotted the data
   pdf->plotOn(pl);
```

```
   pl->Draw();

   // import data in workspace (IMPORTANT for saving it )
   w.import(*data);

   w.Print();

   // write workspace in the file (recreate file if already existing)
   w.writeToFile("GaussianModel.root", true);

   cout << "model written to file " << endl;

}
```

## Exercise 14: Reading a workspace from a file

Open the file you have just created in the previous exercise and get the `RooWorkspace` object ffrom the file. Get a pointer to the p.d.f describing your model, and a pointer to the data. Re-fit the data, but this time in the range [0,10] and plot the result.

Hint Hide
To read and analyse the workspace you need to do:

- Open the `TFile` object and use `TFile::Get` to get a pointer to the workspace using its name
- Once you have the workspace in memory, retrieve from it the p.d.f. and the data set with their names using `RooWorkspace::pdf` and `RooWorkspace::data`.
- Re-issue again the call to `RooAbsPdf::fitTo`. You can set the fit range using the `RooFit::Range(xmim,xmas)` command arg option in `fitTo`. See the reference documentation for all the possible options that you can pass (some are shown in the solution code).

Here is the solution. This macro (apart form the Range fit) can work and fit whatever workspace you have in the file. You just need to set the right names for file, workspace, global p.d.f. and data set.

Solution Hide

```
#include "RooWorkspace.h"
#include "RooAbsPdf.h"
#include "RooRealVar.h"
#include "RooPlot.h"
#include "RooDataSet.h"
#include "RooFitResult.h"

#include "TFile.h"


// roofit tutorial showing how to fit whatever model we get from a file

// we assume the name of the workspace is w
// the name of the pdf is pdf
// the name of the data is data
const char * workspaceName = "w";
const char * pdfName = "pdf";
const char * dataName = "data";


using namespace RooFit;



void fitModel(const char * filename = "GaussianModel.root" ) {
```

```
// read file:
// following lines are for reading workspace
// and to check that is fine

// Check if example input file exists
TFile *file = TFile::Open(filename);

// if input file was specified but not found, quit
if(!file ){
   cout <<"file " << filename << " not found" << endl;
   return;
}

// get the workspace out of the file
RooWorkspace* w = (RooWorkspace*) file->Get(workspaceName);
if(!w){
   cout <<"workspace with name " << workspaceName << " not found" << endl;
   return;
}

// fit a pdf from workspace with name pdfName

RooAbsPdf * pdf = w->pdf(pdfName);
if (!pdf) {
   w->Print();
   cout << "pdf with name " << pdfName << " does not exist in workspace " << endl;
   return;
}

// get the data out of the file
RooAbsData* data = w->data(dataName);

if(!data ){
   w->Print();
   cout << "data " << dataName << " was not found" <<endl;
   return;
}

//-------------------------------------------------------------------------
//// real code starts here

// get variable x (is the first of the data)
RooRealVar * x = w->var("x");
RooPlot * plot = x->frame();

data->plotOn(plot);
plot->Draw();

// fit pdf - (example using option: save result and using diffferent minimizer

// global fit

pdf->fitTo( *data );

// for doing a reduce fit in a Range (plus other options)

RooFitResult * r = pdf->fitTo( *data, Save(true), Minimizer("Minuit2","Migrad"),Range(0.,10.)

pdf->plotOn(plot);
pdf->paramOn(plot);

plot->Draw();

// if we have a result we can do

r->Print();
```

```
   r->correlationMatrix().Print();


}
```

## Exercise 15: Fit of a Signal over an Exponential Background

The aim of tis exercise is to learn how to build a composite model in RooFit made of two p.d.f, one representing a signal and one a background distributions. We want to determine the number of signal events. For this we need to perform an extended maximum likelihood fit, where the signal events is one of the fit parameter. We will create the composite model formed by a Gaussian signal over a falling exponential background. We will have ~ 100 Gaussian-distributed signal event over ~ 1000 exponential-distributed background. We assume that the location and width of the signal are known. The "mass" observable is distributed in range [0 - 10] and the signal has mean ~ 2 and sigma 0.3. The exponential decay of the background is 2.

We will follow the same step as in the previous exercise:

- Start creating the model using the workspace factory. Create the separate signal and background pdf and then combine together using the `RooAddPdf` class.
- Use the `generate(...)` method of the `RooAbsPdf` class to generate a data set with 1000 events.
- Plot the data set using `RooPlot` in a ROOT Canvas.
- Fit the generate data using the `fitTo(...)` method of the `RooAbsPdf` class
- Plot the resulting fit function on top of the data
- Save the workspace in a file

Hint  Hide
Creating the model

- Use the syntax of the `RooWorkspace` factory to create first the Gaussian p.d.f. function with the variables (observables and parameters) and the Exponential
- Create then a `RooAddPdf` using the Gaussian and the Exponential and the relative number of events. Note that we could instead of the number of events a relative fraction. In this last case we would fit only for the shape and not the normalisation.
- The `RooAddPdf` is created using the `SUM` operator of the factory syntax.

```
// create model
RooWorkspace w("w");
 w.factory("Exponential:bkg_pdf(x[0,10], a[-0.5,-2,-0.2])");
 w.factory("Gaussian:sig_pdf(x, mass[2], sigma[0.3])");

// create RooAddPdf
 w.factory("SUM:model(nsig[100,0,10000]*sig_pdf, nbkg[1000,0,10000]*bkg_pdf)");  // for extended
```

Retrieving objects from the workspace

- After you have created the variable and p.d.f in the workspace, you can access their pointers, by using `RooWorkspace::var` for variables or `RooWorkspace::pdf` for p.d.f. Example:

```
// retrieving model components
RooAbsPdf * pdf = w.pdf("pdf");   // access object from workspace
RooRealVar * x = w.var("x");   // access object from workspace
```

Generating a dataset

- We generate the dataset using the `generate` function passing the observables we want to generate and the number of events. In case of an extended pdf (as the `RooAddPdf` we have created), if we don't pass

the number of events, the number of events will be generated according to a Poisson distribution given the expected events (`nsig+nbkg`).

- Note that you can generate also binned data sets. In that case you have two options:
  - ♦ ◊ use a `RooDataHist` and `generateBinned`: `RooDataHist * binData = pdf->generateBinned(*x, 1000);`.
  - ♦ ◊ use the option `AllBinned()` which will generate a weighted data set: `RooDataSet * data = pdf->generate(*x, NumEvents(1000), AllBinned());`.

```
// generate an unbinned dataset of 1000 events
RooDataSet * data = pdf->generate( *x, 1000);
```

Plot the data:

```
// create a RooPlot object and plot the data
RooPlot * plot = x->frame();
data->plotOn(plot);
```

Fit the data: you need to call the `RooAbsPdf::fitTo`.

```
// fit the data and save its result. Use the optional =Minuit2= minimiser
RooFitResult * res = pdf->fitTo(*data, Minimizer("Minuit2","Migrad"), Save(true) );
```

- Plot the resulting fit function in the same plot.
- Plot also the signal and background fit components with different colour and style

```
pdf->plotOn(plot);
//draw the two separate pdf's
pdf->plotOn(plot, RooFit::Components("bkg_pdf"), RooFit::LineStyle(kDashed) );
pdf->plotOn(plot, RooFit::Components("sig_pdf"), RooFit::LineColor(kRed), RooFit::LineStyle(kDash
plot->Draw();   // to show the RooPlot in the current ROOT Canvas
```

Save the workspace in a file for re-using it without the need to generate the data

```
w.writeToFile("GausExpModel.root",true);  // true means the file is re-created
```

Solution Hide

Note that in the solution we also create a `ModelConfig` object that we save in the workspace. This will be useful for the RooStats exercises (see later the RooStats exercise 1)

```
#include "RooWorkspace.h"
#include "RooAbsPdf.h"
#include "RooDataSet.h"
#include "RooFitResult.h"
#include "RooPlot.h"
#include "RooRealVar.h"
#include "RooRandom.h"

#include "RooStats/ModelConfig.h"

using namespace RooFit;



void GausExpModel(int nsig = 100,    // number of signal events
                  int nbkg = 1000 )  // number of background events
{

   RooWorkspace w("w");
   w.factory("Exponential:bkg_pdf(x[0,10], a[-0.5,-2,-0.2])");
   w.factory("Gaussian:sig_pdf(x, mass[2], sigma[0.3])");
```

```
   w.factory("SUM:model(nsig[0,10000]*sig_pdf, nbkg[0,10000]*bkg_pdf)");  // for extended model

   RooAbsPdf * pdf = w.pdf("model");
   RooRealVar * x = w.var("x");  // the observable

   // set the desired value of signal and background events
   w.var("nsig")->setVal(nsig);
   w.var("nbkg")->setVal(nbkg);

   // generate the data

   // use fixed random numbers for reproducibility (use 0 for changing every time)
   RooRandom::randomGenerator()->SetSeed(111);

   // fix number of bins to 50 to plot or to generate data (default is 100 bins)
   x->setBins(50);

   RooDataSet * data = pdf->generate( *x);  // will generate accordint to total S+B events
   //RooDataSet * data = pdf->generate( *x, AllBinned());  // will generate accordint to total S+
   data->SetName("data");
   w.import(*data);

   data->Print();


   RooPlot * plot = x->frame(Title("Gaussian Signal over Exponential Background"));
   data->plotOn(plot);
   plot->Draw();

   RooFitResult * r = pdf->fitTo(*data, RooFit::Save(true), RooFit::Minimizer("Minuit2","Migrad")
   r->Print();

   pdf->plotOn(plot);
   //draw the two separate pdf's
   pdf->plotOn(plot, RooFit::Components("bkg_pdf"), RooFit::LineStyle(kDashed) );
   pdf->plotOn(plot, RooFit::Components("sig_pdf"), RooFit::LineColor(kRed), RooFit::LineStyle(kD

   pdf->paramOn(plot,Layout(0.5,0.9,0.85));

   plot->Draw();


   RooStats::ModelConfig mc("ModelConfig",&w);
   mc.SetPdf(*pdf);
   mc.SetParametersOfInterest(*w.var("nsig"));
   mc.SetObservables(*w.var("x"));
   // define set of nuisance parameters
   w.defineSet("nuisParams","a,nbkg");

   mc.SetNuisanceParameters(*w.set("nuisParams"));

   // import model in the workspace
   w.import(mc);

   // write the workspace in the file
   TString fileName = "GausExpModel.root";
   w.writeToFile(fileName,true);
   cout << "model written to file " << fileName << endl;
}
```

- Gaussian plus Exponential fit result:

Error: (3) can't find GausExpModelFit.png in Main

# Exercise 16: Compute the significance of the signal peak using RooStats

The aim of this exercise is to compute the significance of observing a signal in the Gaussian plus Exponential model. It is better to re-generate the model using a lower value for the number of signal events (e.g. 50), in order not to have a too high significance, which will then be difficult to estimate if we use pseudo-experiments. To estimate the significance, we need to perform an hypothesis test. We want to disprove the null model, i.e the background only model against the alternate model, the background plus the signal. In RooStats, we do this by defining two two `ModelConfig` objects, one for the null model (the background only model in this case) and one for the alternate model (the signal plus background).

We have defined in the workspace saved in the ROOT file only the signal plus background model. We can create the background model from the signal plus background model by setting the value of `nsig=0`. We do this by cloning the S+B model and by defining a snapshot in the `ModelConfig` for `nsig` with a different value:

```
ModelConfig*  sbModel = (RooStats::ModelConfig*) w->obj(modelConfigName);
RooRealVar* poi = (RooRealVar*) sbModel->GetParametersOfInterest()->first();
// define the S+B snapshot (this is used for computing the expected significance)
poi->setVal(50);
sbModel->SetSnapshot(*poi);
// create B only model
ModelConfig * bModel = (ModelConfig*) sbModel->Clone();
bModel->SetName("B_only_model");
poi->setVal(0);
bModel->SetSnapshot( *poi  );
```

Using the given models plus the observed data set, we can create a RooStats hypothesis test calculator to compute the significance. We have the choice between

- Frequentist calculator
- Hybrid Calculator
- Asymptotic calculator

The first two require generate pseudo-experiment, while the Asymptotic calculator, requires only a fit to the two models. For the Frequentist and the Hybrid calculators we need to choose also the test statistics. We have various choices in RooStats. Since it is faster, we will use for the exercise the Asymptotic calculator. The Asymptotic calculator it is based on the Profile Likelihood test statistics, the one used at LHC (see slides for its definition). For testing the significance, we must use the one-side test statistic (we need to configure this explicitly in the class). Summarising, we will do:

- create the `AsymptoticCalculator` class using the two models and the data set;
- run the test of hypothesis using the `GetHypoTest` function.
- Look at the result obtained as a `HypoTestResult` object

Hint  Hide

```
AsymptoticCalculator   ac(*data, *sbModel, *bModel);
ac.SetOneSidedDiscovery(true);  // for one-side discovery test

// this run the hypothesis test and print the result
HypoTestResult * result = ac.GetHypoTest();
result->Print();
```

Asymptotic calculator Solution  Hide

```
using namespace RooStats;
using namespace RooFit;

void SimpleHypoTest( const char* infile =  "GausExpModel.root",
                     const char* workspaceName = "w",
```

```
                    const char* modelConfigName = "ModelConfig",
                    const char* dataName = "data" )
{

  ///////////////////////////////////////////////////////////
  // First part is just to access the workspace file
  ///////////////////////////////////////////////////////////

  // open input file
  TFile *file = TFile::Open(infile);
  if (!file) return;

  // get the workspace out of the file
  RooWorkspace* w = (RooWorkspace*) file->Get(workspaceName);


  // get the data  out of the file
  RooAbsData* data = w->data(dataName);

  // get the modelConfig (S+B) out of the file
  // and create the B model from the S+B model
  ModelConfig*  sbModel = (RooStats::ModelConfig*) w->obj(modelConfigName);
  sbModel->SetName("S+B Model");
  RooRealVar* poi = (RooRealVar*) sbModel->GetParametersOfInterest()->first();
  poi->setVal(50);  // set POI snapshot in S+B model for expected significance
  sbModel->SetSnapshot(*poi);
  ModelConfig * bModel = (ModelConfig*) sbModel->Clone();
  bModel->SetName("B Model");
  poi->setVal(0);
  bModel->SetSnapshot( *poi  );

  // create the AsymptoticCalculator from data,alt model, null model
  AsymptoticCalculator  ac(*data, *sbModel, *bModel);
  ac.SetOneSidedDiscovery(true);  // for one-side discovery test
  //ac.SetPrintLevel(-1);  // to suppress print level

  // run the calculator
  HypoTestResult * asResult = ac.GetHypoTest();
  asResult->Print();


  return;  // comment this line if you want to run the FrequentistCalculator
  std::cout << "\n\nRun now FrequentistCalculator.....\n";   FrequentistCalculator   fc(*data, *s
  // use one-sided profile likelihood
  profll.SetOneSidedDiscovery(true);

  // configure  ToyMCSampler and set the test statistics
  ToyMCSampler *toymcs = (ToyMCSampler*)fc.GetTestStatSampler();
  toymcs->SetTestStatistic(&profll);

  if (!sbModel->GetPdf()->canBeExtended())
     toymcs->SetNEventsPerToy(1);

  // run the test
  HypoTestResult * fqResult = fc.GetHypoTest();
  fqResult->Print();

  // plot test statistic distributions
  HypoTestPlot * plot = new HypoTestPlot(*fqResult);
  plot->SetLogYaxis(true);
  plot->Draw();

}
```

- Result of the significance test on the Gaussian Signal plus Background model (nsig=50, nbckg=1000):

```
Results HypoTestAsymptotic_result:
 - Null p-value = 0.00118838
 - Significance = 3.0386
 - CL_b: 0.00118838
 - CL_s+b: 0.502432
 - CL_s: 422.787
```

Run the Frequentist calculator

Hint Hide
For the Frequentist calculator we need to :

- set the test statistic we want to use
- set the number of toys for the null and alternate model

```
FrequentistCalculator   fc(*data, *sbModel, *bModel);
fc.SetToys(2000,500);    // 2000 for null (B) and 500 for alt (S+B)

// create the test statistics
ProfileLikelihoodTestStat profll(*sbModel->GetPdf());
// use one-sided profile likelihood
profll.SetOneSidedDiscovery(true);

// configure  ToyMCSampler and set the test statistics
ToyMCSampler *toymcs = (ToyMCSampler*)fc.GetTestStatSampler();
toymcs->SetTestStatistic(&profll);

if (!sbModel->GetPdf()->canBeExtended())
   toymcs->SetNEventsPerToy(1);
```

We run now the hypothesis test

```
HypoTestResult * fqResult = fc.GetHypoTest();
fqResult->Print();
```

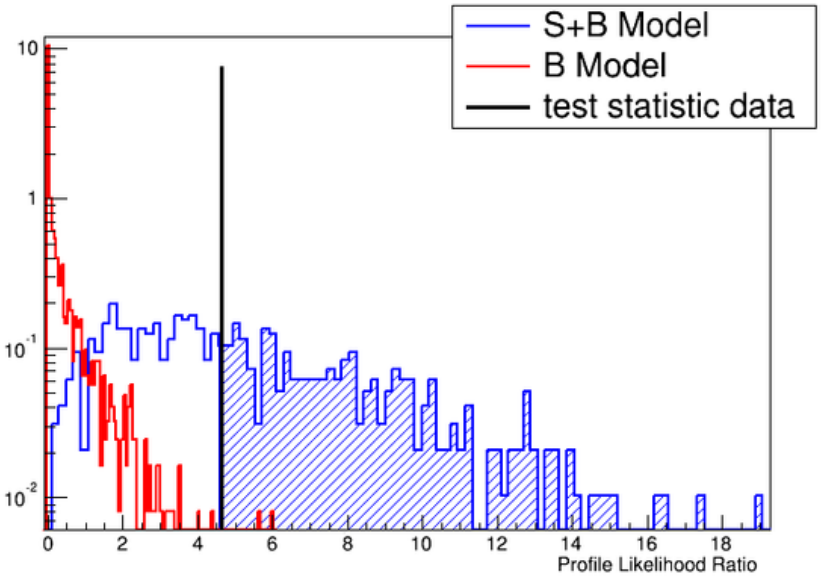
And we can also plot the test statistic distributions obtained from the pseudo-experiments for the two models. We use the `HypTestPlot` class for this.

```
// plot test statistic distributions
HypoTestPlot * plot = new HypoTestPlot(*fqResult);
plot->SetLogYaxis();
plot->Draw();
```

Frequentist calculator Solution Hide
Use the macro SimpleHypoTest.C remove the `return` statement in the middle and run it again

- Result of the significance test on the Gaussian Signal plus Background model (nsig=50, nbckg=1000) using the Frequentist calculator:

```
Results HypoTestCalculator_result:
 - Null p-value = 0.001 +/- 0.000706753
 - Significance = 3.09023 +/- 0.2099 sigma
 - Number of Alt toys: 500
 - Number of Null toys: 2000
 - Test statistic evaluated on data: 4.61654
 - CL_b: 0.001 +/- 0.000706753
 - CL_s+b: 0.496 +/- 0.02236
 - CL_s: 496 +/- 351.262
```




Test statistics distribution for the B model (red) and S+B model obtained with the Frequentist calculator

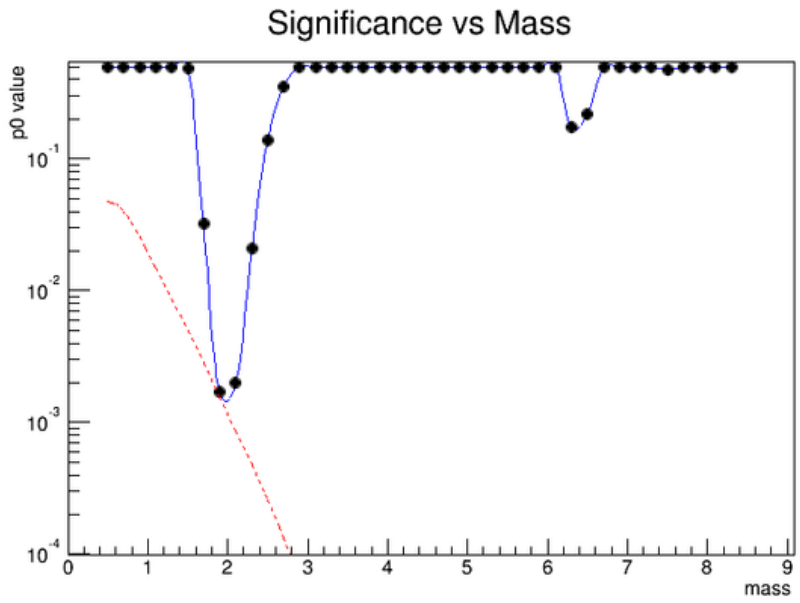
## Exercise 17: Compute significance (p-value) as function of the signal mass

As an optional exercise to lear more about computing significance in RooStats, we will study the significance (or equivalent the p-value) we obtain in our Gaussian signal plus exponential background model as function of the signal mass hypothesis. For doing this we perform the test of hypothesis, using the AsymptoticCalculator, for several mass points. We can then plot the obtained p-value for the null hypothesis (`p0`). We will also estimate the expected significance for our given S+B model as function of its mass. The expected significance can be obtained using a dedicated function in the `AsymptoticCalculator`: `AsymptoticCalculator::GetExpectedPValues` using as input the observed p-values for the null and the alternate models. See the Asymptotic formulae paper for the details.

Solution Hide

```
using namespace RooStats;
using namespace RooFit;

void p0Plot() {

  const char* infile =  "GausExpModel.root";
  const char* workspaceName = "w";
  const char* modelConfigName = "ModelConfig";
  const char* dataName = "data";

  ///////////////////////////////////////////////////////////////
  // First part is just to access the workspace file
  ///////////////////////////////////////////////////////////////

  // Check if example input file exists
  TFile *file = TFile::Open(infile);

  // get the workspace out of the file
  RooWorkspace* w = (RooWorkspace*) file->Get(workspaceName);


  // get the modelConfig out of the file
  RooStats::ModelConfig* mc = (RooStats::ModelConfig*) w->obj(modelConfigName);

  // get the modelConfig out of the file
```

```
  RooAbsData* data = w->data(dataName);

  // get the modelConfig (S+B) out of the file
  // and create the B model from the S+B model
  ModelConfig*  sbModel = (RooStats::ModelConfig*) w->obj(modelConfigName);
  sbModel->SetName("S+B Model");
  RooRealVar* poi = (RooRealVar*) sbModel->GetParametersOfInterest()->first();
  poi->setVal(50);  // set POI snapshot in S+B model for expected significance
  sbModel->SetSnapshot(*poi);
  ModelConfig * bModel = (ModelConfig*) sbModel->Clone();
  bModel->SetName("B Model");
  poi->setVal(0);
  bModel->SetSnapshot( *poi  );

  vector<double> masses;
  vector<double> p0values;
  vector<double> p0valuesExpected;

  double massMin = 0.5;
  double massMax = 8.5;

  // loop on the mass values

  for( double mass=massMin; mass<=massMax; mass += (massMax-massMin)/40.0 )    {

     AsymptoticCalculator *  ac  = new AsymptoticCalculator(*data, *sbModel, *bModel);
     ac->SetOneSidedDiscovery(true);  // for one-side discovery test
     AsymptoticCalculator::SetPrintLevel(-1);


     HypoTestResult* asymCalcResult = ac->GetHypoTest();

     asymCalcResult->Print();

     masses.push_back( mass );
     p0values.push_back( asymCalcResult->NullPValue() );

     double expectedP0 = AsymptoticCalculator::GetExpectedPValues(  asymCalcResult->NullPValue(),
     p0valuesExpected.push_back( expectedP0 );
     std::cout << "expected p0 = " << expectedP0 << std::endl;    }   TGraph * graph1  = new TGrap
  graph1->Draw("APC");
  graph2->SetLineStyle(2);
  graph2->Draw("C");
  graph1->GetXaxis()->SetTitle("mass");
  graph1->GetYaxis()->SetTitle("p0 value");
  graph1->SetTitle("Significance vs Mass");
  graph1->SetMinimum(graph2->GetMinimum());
  graph1->SetLineColor(kBlue);
  graph2->SetLineColor(kRed);
  gPad->SetLogy(true);


}
```




p value obtained as function of the signal mass in the Gaussian plus exponential background model

---

This topic: Main > RootGridKaTutorial2013
Topic revision: r10 - 2014-01-17 - LorenzoMoneta