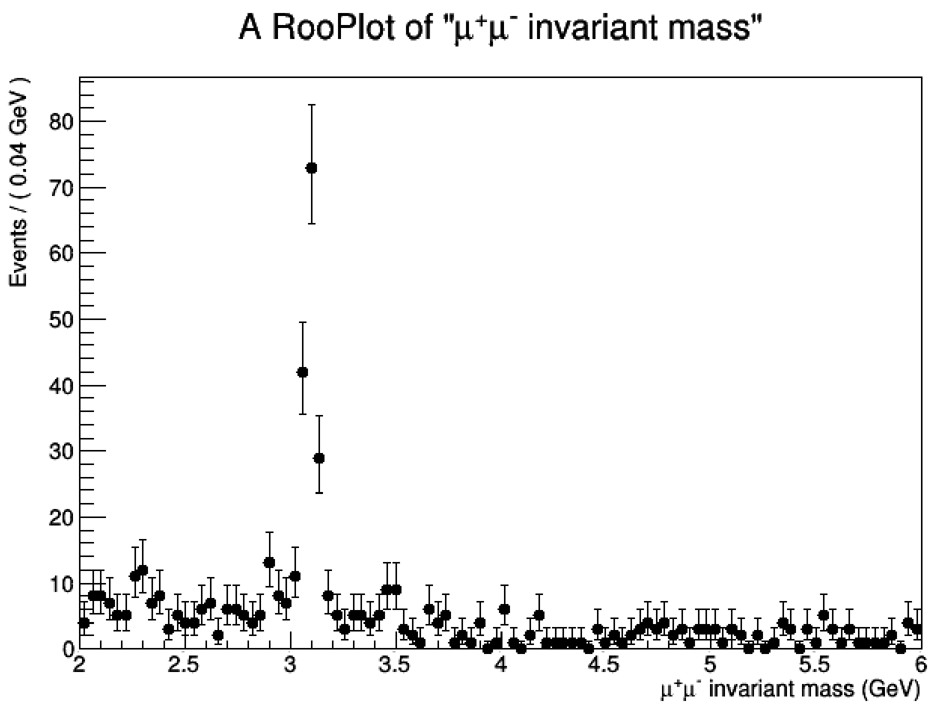
Table of Contents

Exercises for the INFN school of Statistics 2022

The goal of these exercises is to get acquainted with few of the statistical analysis tools (mostly used in HEP). Introductory slides will present a general introduction on the main features of RooFit and RooStats, and will give some additional information on the hands-on session.

This exercises use the CMS public data from the 2010 proton-proton running. In particular, the invariant mass spectrum of muon pairs from data is provided. The purpose of the session is to approach the discovery of a "new" particle in the context of the observation of the (2S) resonance in the CMS data. No specific knowledge of the CMS apparatus is needed. This is what the distribution of the dimuon invariant mass spectrum looks like with about 2% of the total 2010 statistics:



The J/ is of course very visible around the 3.1 GeV mass point. We expect to see an excess around 3.65 GeV for the (2S). Is it there? Let's investigate!

```
Do notice that usually experiments do not proceed in this way towards a discovery. In gene
For the purpose of describing the statistical tools in the time frame of this class, we wi
```

Summary

This exercise session introduces methods and tools for tackling very common problems in statistics, in the HEP context. The main topics are finding a confidence interval for a parameter of interest, estimating the statistical significance of an excess of events with respect to the background expectations, and accounting for systematics uncertainties. We cover the most popular techniques: profile likelihood, Bayesian. Participants will learn the main features of the RooFit and RooStats software frameworks for statistics and data modeling. We overview the popular cases of a hunting a signal peak over a predictable falling background.

Reference materials

- Statistics in Theory - a lecture by Bob Cousins
- Statistical methods in LHC data analysis - Luca Lista
- RooFit reference slides - by Wouter Verkerke, one of RooFit authors

- RooFit tutorials - a set of working macros that showcase all major features of RooFit
- RooStats Manual - concise, contains clear summary of statistics concepts and definitions
- RooStats tutorial - by Kyle Cranmer, one of the RooStats developers
- RooStats tutorials - a set of working macros that showcase all major features of RooStats

Computing environment

All exercises should run on any ROOT installation containing also the RooFit libraries (RooStats is included in the RooFit installation). On your laptop you obtain this by passing the *--enable-roofit* on the *configure* step of the ROOT installation. If you have a debian based installation, you just need to install the libRooFit library. All the code here uses PyROOT, which is a python interface to ROOT libraries, so a python installation is also required.

Alternatively, if you have a CERN account, most of the central ROOT installations contain RooFit. This is true for the linux environment in many other lab computing pools.

PyROOT programs can be ran using the following command from the shell prompt:

```
python3 macro.py
```

In order to have all ROOT libraries available in PyROOT, the following line should be added at the begin of every program (`macro.py`, in the previous command line example):

```
import ROOT
```

Exercise #0: Fit a lower statistics data sample using RooFit

First of all, you will need a copy of the data ROOT file containing the invariant mass information from dimuon events. We will start with a data sample which corresponds to about 20% of the statistics available on 2010:

```
wget https://twiki.cern.ch/twiki/pub/Main/INFNStatRooStats2022/DataSet_lowstat.root
```

on MacOS, you may want to use

```
curl -o DataSet_lowstat.root https://twiki.cern.ch/twiki/pub/Main/INFNStatRooStats2022/Dat
```

Let's create a skeleton for a PyROOT program. Let's call it, for instance, `exercise_0.py`, and load the ROOT libraries:

```
#Import the ROOT libraries
import ROOT
```

- Open the ROOT file containing the data, and extract the RooDataSet object. A RooDataSet is essentially an ntuple (similar to a TTree in simple ROOT) containing your N-dimensional observables (in this case it has one dimension, it contains the dimuon invariant mass measurement for the selected events):

```
fInput = ROOT.TFile("DataSet_lowstat.root")
dataset = fInput.Get("data")
```

- Define the observables of the problem. In our case, this is a one-dimensional problem, and the observable is the invariant mass of the dimuon system. To handle this, RooFit uses the class `RooRealVar`. Note that at the constructor you have several options. In the following, either a range will be specified, or an initial value plus a range.

```
#The observable
mass = ROOT.RooRealVar("mass","#mu^{+}#mu^{-} invariant mass",2.,6.,"GeV")
```

- Modelization of the observable distribution. Let's start from the most striking feature: the J/ peak. We will describe this signal using a Crystal Ball function (see Wikipedia), which is essentially a Gaussian core with a polynomial left tail to account for energy losses. First, we have to define the parameters of the function.

```
#The Jpsi signal parametrization: we'll use a Crystal Ball
meanJpsi = ROOT.RooRealVar("meanJpsi","The mean of the Jpsi Crystal Ball",3.1,2.8,3.2)
sigmaJpsi = ROOT.RooRealVar("sigmaJpsi","The width of the Jpsi Crystal Ball",0.3,0.0001,1.
alphaJpsi = ROOT.RooRealVar("alphaJpsi","The alpha of the Jpsi Crystal Ball",1.5,-5.,5.)
nJpsi = ROOT.RooRealVar("nJpsi","The alpha of the Jpsi Crystal Ball",1.5,0.5,5.)
```

- Define a PDF for the signal model. The Crystal Ball distribution, in this case, is provided by the `RooCBshape` class:

```
CBJpsi = ROOT.RooCBShape("CBJpsi","The Jpsi Crystall Ball",mass,meanJpsi,sigmaJpsi,alphaJp
```

- Now we have to define a model for the excess around 3.65 GeV. Since there aren't many events involved, the functional form to describe this peak cannot be easily determined by fitting the data. Since the lineshape is mostly driven by experimental resolution, we will assume that this peak has a similar shape as the J/ . So we'll describe the (2S) with another Crystal Ball, whose parameters are in common with the J/ , but for a shifted peak value. In this way, we use the high statistics of the J/ peak to better constrain our signal modelization

```
#The psi(2S) signal parametrization: width will be similar to Jpsi, but with shifted mass
meanpsi2S = ROOT.RooRealVar("meanpsi2S","The mean of the psi(2S) Crystal Ball",3.7,3.65,3.
CBpsi2S = ROOT.RooCBShape("CBpsi2S","The psi(2S) Crystal Ball",mass,meanpsi2S,sigmaJpsi,al
```

- Now also define the background. A third order polynomial is enough here. We'll use Chebychev polynomials (see Wikipedia):

```
#Background parametrization: just a polynomial
a1 = ROOT.RooRealVar("a1","The a1 of background",-0.7,-2.,2.)
a2 = ROOT.RooRealVar("a2","The a2 of background",0.3,-2.,2.)
a3 = ROOT.RooRealVar("a3","The a3 of background",-0.03,-2.,2.)
backgroundPDF = ROOT.RooChebychev("backgroundPDF","The background PDF",mass,ROOT.RooArgLis
```

- Now, define the yields for the different contributions to the spectrum, again using the `RooRealVar` class. Consider first the J/ and background yields:

```
#Define the yields
NJpsi = ROOT.RooRealVar("NJpsi","The Jpsi signal events",1500.,0.1,10000.)
Nbkg = ROOT.RooRealVar("Nbkg","The bkg events",5000.,0.1,50000.)
```

- For the (2S), we can actually determine the cross section, instead of just the number of events. This is a simple transformation that gets us to the actual parameter of interest. A function of variables can be expressed in RooFit using the `RooFormulaVar` class:

```
#Now define the number of psi(2S) events as a product of crss section*efficiency*luminosit
#Let's assume we measured the trigger, reconstruction and identification efficiency for di
#Lowstat sample has 0.64 pb-1
#Fullstat sample has 37 pb-1
eff_psi = ROOT.RooRealVar("eff_psi","The psi efficiency",0.75,0.00001,1.)
lumi_psi  = ROOT.RooRealVar("lumi_psi","The CMS luminosity",0.64,0.00001,50.,"pb-1")
cross_psi = ROOT.RooRealVar("cross_psi","The psi xsec",3.,0.,40.,"pb")

#Now define the number of psi events
Npsi = ROOT.RooFormulaVar("Npsi","@0*@1*@2",ROOT.RooArgList(eff_psi,lumi_psi,cross_psi))
```

```
#Important! We cannot fit simultaneously the efficiency, the luminosity and the cross sect
#We need to fix two of them, so we'll keep our POI floating
#One can also add an additional PDF to give predictive power on the other two parameters (
eff_psi.setConstant(1)
lumi_psi.setConstant(1)
```

- The total PDF is a linear combination of the different components with weights that are proportional to `NJpsi`, `Npsi` and `Nbkg`. This is implemented in RooFit in the class `RooAddPdf`. Note that the lists of PDF and yields are passed via the `RooArgList` class:

```
#Compose the total PDF
totPDF = ROOT.RooAddPdf("totPDF","The total PDF",ROOT.RooArgList(CBJpsi,CBpsi2S,background
```

- As we have the dataset ready, we can fit it to the PDF model in order to determine the desired parameters:

```
#Do the actual fit
totPDF.fitTo(dataset, ROOT.RooFit.Extended(1))

#Print values of the parameters (that now reflect fitted values and errors)
print("##############")
meanpsi2S.Print()
NJpsi.Print()
Npsi.Print()
print("##############")
```

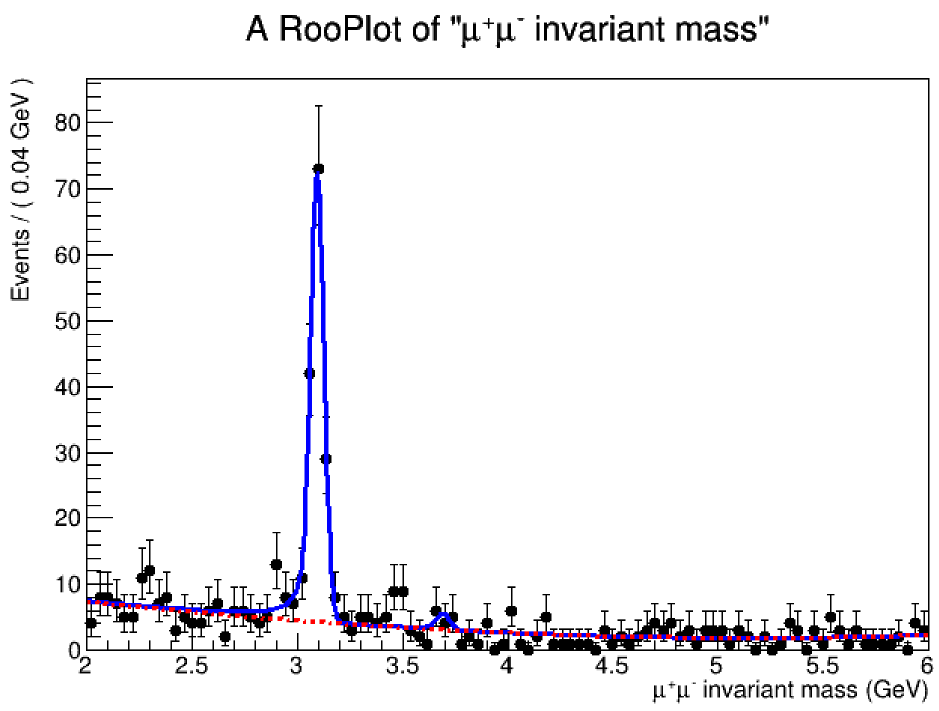
- The result of the fit is printed as usual Minuit output, but you can also produce a plot of the fit PDF and its signal and background components, separately. Note that you need to use the class `RooPlot`, you can't directly call the usual ROOT `Draw()` method for RooFit objects:

```
#Now plot the data and the fit result
xframe = mass.frame()
dataset.plotOn(xframe)
totPDF.plotOn(xframe)

#One can also plot the single components of the total PDF, like the background component
totPDF.plotOn(xframe, ROOT.RooFit.Components("backgroundPDF"), ROOT.RooFit.LineStyle(ROOT.

#Draw the results
c1 = ROOT.TCanvas()
xframe.Draw()
c1.SaveAs("exercise_0.png")
```

- In order to make the PDF model and the dataset available for use in next exercises, you can save it into a `RooWorkspace` object that can be written directly into a ROOT file:

```
#Now save the data and the PDF into a Workspace, for later use for statistical analysis
ws = ROOT.RooWorkspace("ws")
getattr(ws,'import')(dataset)
getattr(ws,'import')(totPDF)

fOutput = ROOT.TFile("Workspace_mumufit.root","RECREATE")
ws.Write()
fOutput.Write()
fOutput.Close()
```

- Now execute the python program. The result should look like the following:



The entire exercise is available here:

- exercise_0.py

Exercise #1: using toy MCs to test fit robustness

- First, create a skeleton python and call it `exercise_1.py`. Load ROOT:

```
import ROOT
```

- Import the workspace from the file produced in the previous exercise:

```
#Open the rootfile and get the workspace from the exercise_0
fInput = ROOT.TFile("Workspace_mumufit.root")
ws = fInput.Get("ws")
ws.Print()
```

- The MC study machinery needs to be fed the observable and the PDF, so we'll extract them from the RooWorkspace. Important: what we saved in the Workspace is the PDF (with all the parameters) in its last status, which is just after the fit in exercise 0:

```
#Get the observable and PDF out of the Workspace
mass = ws.var("mass")
totPDF = ws.pdf("totPDF")
```

- Initialize RooMCStudy. It requires to know the PDF and the observable (for the toy MC generation). You can pass additional arguments. Here, we add the request Extended(1), which means that in every single toy MC experiment the total number of events generated will be a Poissonian fluctuation of the expected total number of events (which is the fit result of the previous exercise). Also, we want to save the fit result of each experiment for later use.

```
#Initialize RooMCStudy
mc_study = ROOT.RooMCStudy(totPDF, ROOT.RooArgSet(mass), ROOT.RooFit.Extended(1), ROOT.Roo
```

- Now we need to generate N experiments, fit them, and save the result. It is done with this line:

```
#Generate 1000 experiments and fit each one, each fluctuating in Nevents = NJpsi + Npsi + N
mc_study.generateAndFit(1000)
```

- Now we can study the behavior of our parameters along the experiments. RooFit provides some automatic tools to keep track of all the fit results and plot them. Let's consider our parameter of interest, `cross_psi`, and verify its statistical properties, like its central value across the experiments, the error, and the pull:

```
#Now let's see the results. For example, the study of the cross section variable
cross_psi = ws.var("cross_psi")
frame_cross_par = mc_study.plotParam(cross_psi, ROOT.RooFit.Bins(40))
frame_cross_err = mc_study.plotError(cross_psi, ROOT.RooFit.Bins(40), ROOT.RooFit.FrameRan
frame_cross_pul = mc_study.plotPull(cross_psi, ROOT.RooFit.Bins(40), ROOT.RooFit.FitGauss(
```

- We may also want to check that the NLL has a healthy distribution across the experiments:

```
#Also, let's see the distribution of the NLL for all the fits
frame_nll = mc_study.plotNLL(ROOT.RooFit.Bins(40))
```

- Now let's plot all this:

```
#Now plot the whole thing
ROOT.gStyle.SetOptStat(0)

mcstudy_Canvas = ROOT.TCanvas("mcstudy_Canvas")
mcstudy_Canvas.Divide(2,2)

mcstudy_Canvas.cd(1)
frame_cross_par.Draw()

mcstudy_Canvas.cd(2)
frame_cross_err.Draw()

mcstudy_Canvas.cd(3)
frame_cross_pul.Draw()

mcstudy_Canvas.cd(4)
frame_nll.Draw()

mcstudy_Canvas.SaveAs("exercise_1.png")
```

- The `cross_psi` variable may have some strongly non-Gaussian behavior, as it involves a low statistics object that may be affected by strong fluctuations. As a comparison, let's also save the behavior of a variable that involves the J/ . After all, this is a very large peak with relatively high statistics, and we expect that variables involved with this peak will show a Gaussian behavior:

```
#Now plot some less constroversial variable
meanJpsi = ws.var("meanJpsi")
frame_mjpsi_par = mc_study.plotParam(meanJpsi, ROOT.RooFit.Bins(40))
frame_mjpsi_pul = mc_study.plotPull(meanJpsi, ROOT.RooFit.Bins(40), ROOT.RooFit.FitGauss(1

mcstudy_Canvas_jpsi = ROOT.TCanvas("mcstudy_Canvas_jpsi","",2000,500)   #2000,500 is just
mcstudy_Canvas_jpsi.Divide(2,1)

mcstudy_Canvas_jpsi.cd(1)
frame_mjpsi_par.Draw()

mcstudy_Canvas_jpsi.cd(2)
frame_mjpsi_pul.Draw()

mcstudy_Canvas_jpsi.SaveAs("exercise_1_mjpsi.png")
```

- RooMCStudy actually needs to be deleted manually from memory in pyROOT, or you'll get some errors (doesn't affect the result)

```
#this is just to help pyroot clean the memory
del mc_study
```

- You are now ready to run the exercise. What conclusions can you make about the `cross_psi` parameter?

The entire exercise is available below:

- exercise_1.py

Exercise #2: compute the excess significance

- First of all, create a skeleton python program and call it, for instance, `exercise_2.py`. Load ROOT:

```
import ROOT
```

- Import the workspace from the file produced in the previous exercise:

```
#Open the rootfile and get the workspace from the exercise_0
fInput = ROOT.TFile("Workspace_mumufit.root")
ws = fInput.Get("ws")

ws.Print()
```

- You can set as constant some of the parameters of the model. As an example, we set the J/ mean of the Crystal Ball function. A parameter that is constant is of course unchanged, for example, in a likelihood scan. The more parameters are constant, the higher your statistical sensitivity, but this is a strong statement: setting a parameter as constant means that its uncertainty is negligible in this problem. Note how you can access variables by name using the `RooWorkspace` class:

```
#You can set constant parameters that are known
#If you leave them floating, the fit procedure will determine their uncertainty
ws.var("meanJpsi").setConstant(1)
```

- RooStats needs to configure the PDF model using the class `ModelConfig`:

```
#Set the RooModelConfig and let it know what the content of the workspace is about
model = ROOT.RooStats.ModelConfig()
model.SetWorkspace(ws)
model.SetPdf("totPDF")
```

- Let's access the parameter `cross_psi` from the workspace and take a "snapshot" (i.e.: a clone). We want to set `cross_psi=0` in order to determine the significance, measured from the p-value corresponding to the background-only hypothesis.

```
#Here we explicitly set the value of the parameters for the null hypothesis
#We want no signal contribution, so cross_psi = 0
cross_psi = ws.var("cross_psi")
poi = ROOT.RooArgSet(cross_psi)
nullParams = poi.snapshot()
nullParams.setRealValue("cross_psi",0.)
```

- We use the class ProfileLikelihoodCalculator to implement the profile-likelihood method and determine the significance with the class HypoTestResult:

```
#Build the profile likelihood calculator
plc = ROOT.RooStats.ProfileLikelihoodCalculator(ws.data("data"), model)
plc.SetParameters(poi)
plc.SetNullParameters(nullParams)
```

- Now we just need to print out the result:

```
#We get a HypoTestResult out of the calculator, and we can query it.
htr = plc.GetHypoTest()

print("-------------------------------------------------")
print("The p-value for the null is ", htr.NullPValue())
print("Corresponding to a signifcance of ", htr.Significance())
print("-------------------------------------------------")

#PyROOT sometimes fails cleaning memory, this helps
del plc
```

Run this exercise. Is the peak aound 3.65 GeV significant?

The complete exercise is available below:

- exercise_2.py

Exercise #3: compute upper limits to the signal yield using frequentist and Bayesian methods

- Create a new python file, call it, for instance, `exercise_3.py`, and load the ROOT libraries. Then, import the workspace created in exercise #0. Also, set a few parameters as constant. You would normally not do this, but otherwise the processing time for calculator becomes too long for this hands-on session:

```
import ROOT

#Open the rootfile and get the workspace from the exercise_0
fInput = ROOT.TFile("Workspace_mumufit.root")
ws = fInput.Get("ws")
ws.Print()

#You can set constant parameters that are known
#If you leave them floating, the fit procedure will determine their uncertainty
#Right now we will fix all the nuisance parameters just to speed up the computing time
ws.var("meanJpsi").setConstant(1)
ws.var("sigmaJpsi").setConstant(1)
ws.var("alphaJpsi").setConstant(1)
ws.var("nJpsi").setConstant(1)
ws.var("NJpsi").setConstant(1)
ws.var("meanpsi2S").setConstant(1)
ws.var("Nbkg").setConstant(1)
ws.var("a1").setConstant(1)
ws.var("a2").setConstant(1)
ws.var("a3").setConstant(1)
```

- Now we need to have two PDF models: one with both signal and background, one with background only, which was the only model needed in the previous exercise. Again, this is just to setup the proper `ModelConfig` for RooStats.

```
#Configure the model, we need both the S+B and the B only models
sbModel = ROOT.RooStats.ModelConfig()
sbModel.SetWorkspace(ws)
sbModel.SetPdf("totPDF")
sbModel.SetName("S+B Model")
poi = ROOT.RooArgSet(ws.var("cross_psi"))
poi.find("cross_psi").setRange(0.,40.)  #this is mostly for plotting
sbModel.SetParametersOfInterest(poi)

bModel = sbModel.Clone()
bModel.SetPdf("totPDF")
bModel.SetName( sbModel.GetName() + "_with_poi_0")
poi.find("cross_psi").setVal(0)
bModel.SetSnapshot(poi)
```

- Part 1: **compute a CLs, modified frequentist upper limit**. The computation is based on toy Monte Carlos that allow to compute confidence levels for s+b and b hypotheses. This requires the interplay of different RooStats classes: `FrequentistCalculator`, which is initialized taking as input the two PDF models (background only, signal plus background), `HypoTestInverter`, which, for a given calculator (a `FrequentistCalculator` in this case) computes the confidence interval and returns it as a `HypoTestInverterResult` object. In the computation, the profile-likelihood test statistics (via the `ProfileLikelihoodTestStat` class) is passed to the "sampler" (`ToyMCSampler`) present in the "calculator" object.

```
#First example is with a frequentist approach
fc = ROOT.RooStats.FrequentistCalculator(ws.data("data"), bModel, sbModel)
fc.SetToys(2500,1500)

#Create hypotest inverter passing the desired calculator
calc = ROOT.RooStats.HypoTestInverter(fc)

#set confidence level (e.g. 95% upper limits)
calc.SetConfidenceLevel(0.95)

#use CLs
calc.UseCLs(1)

#reduce the noise
calc.SetVerbose(0)

#Configure ToyMC Sampler
toymcs = calc.GetHypoTestCalculator().GetTestStatSampler()

#Use profile likelihood as test statistics
profll = ROOT.RooStats.ProfileLikelihoodTestStat(sbModel.GetPdf())

#for CLs (bounded intervals) use one-sided profile likelihood
profll.SetOneSided(1)

#set the test statistic to use for toys
toymcs.SetTestStatistic(profll)

npoints = 8 #Number of points to scan
# min and max for the scan (better to choose smaller intervals)
poimin = poi.find("cross_psi").getMin()
poimax = poi.find("cross_psi").getMax()

print("Doing a fixed scan  in interval : ", poimin, " , ", poimax)
calc.SetFixedScan(npoints,poimin,poimax);

result = calc.GetInterval() #This is a HypoTestInveter class object
upperLimit = result.UpperLimit()
```

- Part 2: **compute a Bayesian upper limit**: this part is simple, since no `HypoTestInverter` is needed. The upper limit is just given by the posterior Bayesian PDF obtained by the `BayesianCalculator`. The Bayesian calculator returns a `SimpleInterval`

```
#Example using the BayesianCalculator
#Now we also need to specify a prior in the ModelConfig
#To be quicker, we'll use the PDF factory facility of RooWorkspace
#Careful! For simplicity, we are using a flat prior, but this doesn't mean it's the best c
ws.factory("Uniform::prior(cross_psi)")
sbModel.SetPriorPdf(ws.pdf("prior"))

#Construct the bayesian calculator
bc = ROOT.RooStats.BayesianCalculator(ws.data("data"), sbModel)
bc.SetConfidenceLevel(0.95)
bc.SetLeftSideTailFraction(0.) # for upper limit

bcInterval = bc.GetInterval()
```

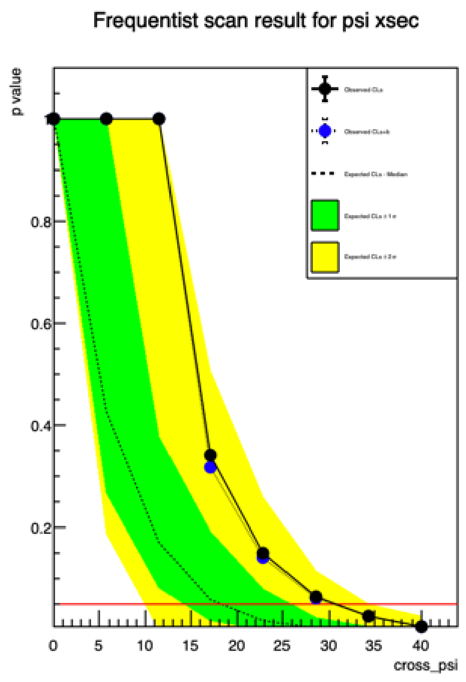
- Now print the results of the two methods:

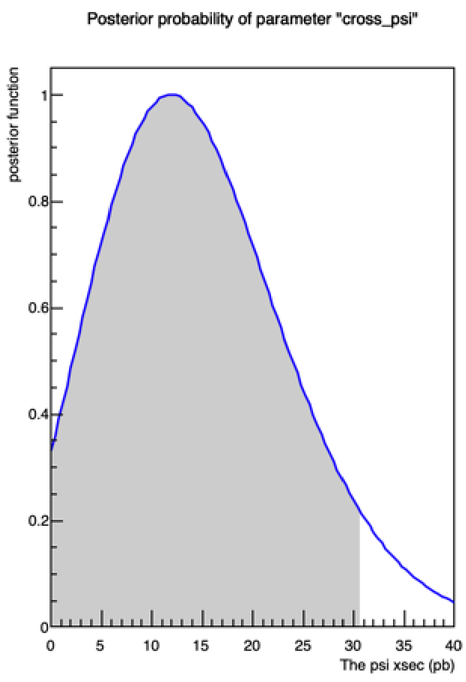
```
#Now let's print the result of the two methods
#First the CLs
print("################")
print("The observed CLs upper limit is: ", upperLimit)

#Compute expected limit
print("Expected upper limits, using the B (alternate) model : ")
print(" expected limit (median) ", result.GetExpectedUpperLimit(0))
print(" expected limit (-1 sig) ", result.GetExpectedUpperLimit(-1))
print(" expected limit (+1 sig) ", result.GetExpectedUpperLimit(1))
print("################")

#Now let's see what the bayesian limit is
print("Bayesian upper limit on cross_psi = ", bcInterval.UpperLimit())
```

- Plot the results of the two methods:

```
#Plot now the result of the scan

#First the CLs
freq_plot = ROOT.RooStats.HypoTestInverterPlot("HTI_Result_Plot","Frequentist scan result
#Then the Bayesian posterior
bc_plot = bc.GetPosteriorPlot()

#Plot in a new canvas with style
dataCanvas = ROOT.TCanvas("dataCanvas")
dataCanvas.Divide(2,1)
dataCanvas.SetLogy(0)
dataCanvas.cd(1)
freq_plot.Draw("2CL")
dataCanvas.cd(2)
bc_plot.Draw()
dataCanvas.SaveAs("exercise_3.png")
```

- Pay attention at the actual numbers printed (remember: the meaning of the upper limit is very different in the two approaches!)
- The plot will look like the following:





The complete exercise is available below:

- exercise_3.py

Exercise #4: measure the cross section with full statistics

- First of all, obtain the full statistics sample for the 2010 CMS data:

```
wget https://twiki.cern.ch/twiki/pub/Main/INFNStatRooStats2022/DataSet.root
```

- Return to the program `exercise_0.py`, and modify it to read the new file. Rerun the program, this will create a new workspace containing the data and the fit for the full statistics
- Remember: change the luminosity to the value of the full 2010 CMS statistics: 37 pb-1
- Then rerun exercise_2.py and notice the significance of the excess with full statistics. Congratulations, you're in the '70s!
- Create a new program `exercise_4.py` that imports the workspace, as in the previous exercises. As before, declare a few parameters constants to speed up the computing (you would probably not do that normally):

```
import ROOT

#Open the rootfile and get the workspace from the exercise_0
fInput = ROOT.TFile("Workspace_mumufit.root")
ws = fInput.Get("ws")
ws.Print()

#You can set constant parameters that are known
#If you leave them floating, the fit procedure will determine their uncertainty
ws.var("meanJpsi").setConstant(1)
ws.var("sigmaJpsi").setConstant(1)
ws.var("alphaJpsi").setConstant(1)
ws.var("nJpsi").setConstant(1)
ws.var("NJpsi").setConstant(1)
ws.var("meanpsi2S").setConstant(1)
ws.var("Nbkg").setConstant(1)
ws.var("a1").setConstant(1)
ws.var("a2").setConstant(1)
ws.var("a3").setConstant(1)

#Let the model know what is the parameter of interest
cross_psi = ws.var("cross_psi")
cross_psi.setRange(4., 16.)  #this is mostly for plotting reasons
poi = ROOT.RooArgSet(cross_psi)
```

- Now, set the `ModelConfig` and `cross_psi` as parameter of interest (poi). Also, now we want 68 percent intervals:

```
#Configure the model
model = ROOT.RooStats.ModelConfig()
model.SetWorkspace(ws)
model.SetPdf("totPDF")
model.SetParametersOfInterest(poi)

#Set confidence level
confidenceLevel = 0.68
```

- The computation of the 68 percent confidence interval is performed using the `ProfileLikelihoodCalculator`:

```
#Build the profile likelihood calculator
plc = ROOT.RooStats.ProfileLikelihoodCalculator(ws.data("data"), model)
plc.SetParameters(poi)
plc.SetConfidenceLevel(confidenceLevel)

#Get the interval
pl_Interval = plc.GetInterval()
```

- As alternative method, a probability interval can be computed using a Bayesian method. While using the Bayesian calculator would be perfectly correct, in this example with high statistics the integration part of the method would require a really long time. On the other hand, one can use the RooStats Markov Chain MC implementation to speed things up:

```
#Now let's determine the Bayesian probability interval
#We could use the standard Bayesian Calculator, but this would be very slow for the integr
#So we profit of the Markov-Chain MC capabilities of RooStats to speed things up

mcmc = ROOT.RooStats.MCMCCalculator(ws.data("data") , model)
mcmc.SetConfidenceLevel(confidenceLevel)
mcmc.SetNumIters(20000)           #Metropolis-Hastings algorithm iterations
mcmc.SetNumBurnInSteps(100)       #first N steps to be ignored as burn-in
mcmc.SetLeftSideTailFraction(0.5) #for central interval

MCMC_interval = mcmc.GetInterval()
```

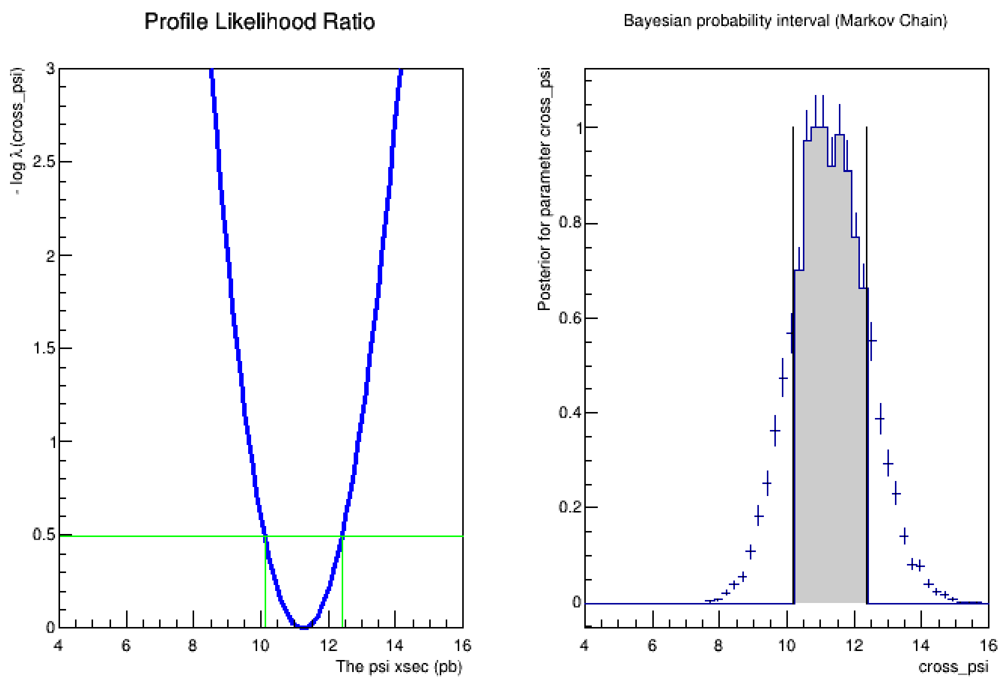
- Finally, plots for both methods can be produced:

```
#Let's make a plot
dataCanvas = ROOT.TCanvas("dataCanvas")
dataCanvas.Divide(2,1)

dataCanvas.cd(1)
plot_Interval = ROOT.RooStats.LikelihoodIntervalPlot(pl_Interval)
plot_Interval.SetTitle("Profile Likelihood Ratio")
plot_Interval.SetMaximum(3.)
plot_Interval.Draw()

dataCanvas.cd(2)
plot_MCMC = ROOT.RooStats.MCMCIntervalPlot(MCMC_interval)
plot_MCMC.SetTitle("Bayesian probability interval (Markov Chain)")
plot_MCMC.Draw()

dataCanvas.SaveAs("exercise_4.png")
```

- And one can print the intervals determined by the two methods:

```
#Now print the interval for mH for the two methods
print("PLC interval is [", pl_Interval.LowerLimit(cross_psi), ", ", pl_Interval.UpperLimit

print("Bayesian interval is [", MCMC_interval.LowerLimit(cross_psi), ", ", MCMC_interval.U

#PyROOT sometimes fails cleaning memory, this helps
del plc
```

- The plot will look like this:



The complete exercise is available here:

- exercise_4.py

Exercise #5: incorporate systematic uncertainties

Let's imagine that we have a 10% uncertainty on the signal efficiency. This is usually determined externally from the analysis workflow we have just described. We want to include this additional information into the determination of the cross section confidence/probability interval. First of all, let's copy `exercise_4.py` into a new file, so that we can modify it to include the systematic uncertainty

```
cp exercise_4.py exercise_5.py
```

Now, let's open `exercise_5.py`.

- Firstly, we want to build a PDF to describe the nature of the uncertainty for the signal efficiency parameter. We will assume a Gaussian behavior (this is not necessarily a sound choice, but it's simple enough for this example!), so we'll use a Gaussian PDF where the observable is a signal efficiency modifier, with no bias (i.e. fixed mean value fixed to 1) and a width of 1.1 (corresponding to a 10%

uncertainty). Add this after the import of the workspace:

```
#Create a modifier to account for signal efficiency uncertainty
ws.factory("Gaussian::effConstrain(gSigEff[1.],ratioSigEff[1.,0.,3],0.1)")   #Gaussian wit
```

The signal efficiency modifier is an additional parameter (fixed to 1 in case of no systematic uncertainty) that rescales the signal efficiency when multiplied to it.

- Now modify the parametric definition of the number of (2S) events to include the signal efficiency modifier in the product, and redefine the total PDF with this new definition. Then, multiply the total PDF by the Gaussian function describing the behavior of the efficiency modifier:

```
ws.factory("SUM::totPDF_withscaling( prod(cross_psi,lumi_psi,eff_psi,ratioSigEff)*CBpsi2S
ws.factory("PROD::totPDF_withconstrain(totPDF_withscaling, effConstrain)")
```

- Now reprint the workspace, to see that the new modifier and the new definition of the total PDF have been included

```
print("##################################")
print("Now printing the workspace with the additional constraints")
print("##################################")
print(ws.Print())
```

- Now modify in the ModelConfig setting the PDF to be used, from `totPDF` to `totPDF_withconstrain`
- Remember to change the name of the output plot to `exercise_5.png`

How does the additional systematic uncertainty change the result?

The complete exercise is available here:

- exercise_5.py

-- MarioPelliccioni - 2022-05-02

This topic: Main > INFNStatRooStats2022
Topic revision: r5 - 2022-05-16 - MarioPelliccioni