Table of Contents

RooStats Exercises......1
Exercise 1: Create a ModelConfig from the H->gg model used in the RooFit Exercise 2......2
Exercise 2: Profile Likelihood Calculator......2
Exercise 3: Using the Bayesian Calculators......5
Exercise 4: Create a Poisson Counting Model......8
Exercise 5: Compute a 95% Upper Limit with the Counting Model......10
Exercise 6: Compute the significance (Hypothesis Test)......11
Exercise 6b: Compute significance (p-value) as function of the signal mass......12
Exercise 7: Compute Limits using Hypothesis Test Inversion (CLs Limits)......14
Using the StandardHypoTestInvDemo.C tutorial......14
Exercise 8: Creating counting model (same as Ex. 4) using the HistFactory......15

RooStats Exercises

In these exercises we will learn to use the RooStats calculators to perform a statistical analysis on a model. We will use two types of models:

- H->gg model. This is the model we have used in the RooFit exercise 2. We need to create the a workspace with the `ModelConfig` class and the data in order to use it in RooStats.
- Poisson counting model: let's assume we observe `n` events in data while expecting `b` background event. We consider some systematic uncertainties in the background model value, `b`. We express this uncertainty as Gaussian.

Optionally we can use a log-normal or a a Poisson/Gamma distribution for expressing this uncertainty. We could also extend the model adding other systematic uncertainties.

For the H->gg we can compute:

- 68% CL 2-sided confidence interval from the (profiled-) likelihood ratio.
- 68% credible interval using the MCMCCalculator or the Bayesian Calculator (based on numerical integration).

On the Poisson model we can compute:

- The 95% Bayesian upper limit.
- The 95% frequentist upper limit using the CLs method
- The 95% interval using the Feldman-Cousins method

For running the RooStats calculators we read the workspace from the ROOT file. All the Roostats calculators take as input a data set and a ModelConfig object. It is therefore convenient to save the workspace (with model and data) in a ROOT file and then read it afterwards for running the calculators.

Since it is time consuming writing the code for running the calculators, we can run the standard Roostats tutorials (in $ROOTSYS/tutorials/roostats)

Tutorials we can use are:

- `StandardProfileLikelihoodDemo.C`
- `StandardBayesianNumericalDemo.C`
- `StandardBayesianMCMCDemo.C`
- `StandardHypoTestDemo.C`
- `StandardHypoTestInvDemo.C`

It is conveninet to copy these tutorials in your working directory so you can modify the, when needed.

```
cp $ROOTSYS/tutorials/roostats/StandardProfileLikelihoodDemo.C .
```

They can be run from the Root prompt by passing the workspace file name (e.g. *HggModel.root*), workspace name, model config name and data set name:

```
root [0] .L StandardProfileLikelihoodDemo.C
root [1] StandardProfileLikelihoodDemo("HggModel.root", "w", "ModelConfig", "obsData" )
```

Other tutorials go into more details to create various models and run the calculators. See them all at http://root.cern.ch/root/html/tutorials/roostats/index.html.

Exercise 1: Create a `ModelConfig` from the H->gg model used in the RooFit Exercise 2.

We show first how to create the `ModelConfig` object which we need to import into the workspace in order to be able to run later the RooStats calculators. We need to specify in the `ModelConfig`:

- the p.d.f of the model
- the parameters of interest (in this case the number of signal events)
- the nuisance parameters (the number of background events and the exponential shape parameters)
- the observables (in this case x, the reconstructed mass)

For convenience we fix the mass and the width of the Higgs to the best fit data value, by setting these parameter constants after the fit.

We will see later, that if we have a model where some external knowledge on a parameter is introduced, we would need to define the model global observables.

Note that we need to import also the data in the workspace. For CPU performance reasons (we have ~ 30000 events), we transform the data in a binned data set of 100 bins.

Below is how we create a `ModelConfig` for the Gaussian signal plus Exponential background model:

```
RooStats::ModelConfig mc("ModelConfig",&w);
mc.SetPdf(*w.pdf("model"));
mc.SetParametersOfInterest(*w.var("nsignal"));
mc.SetObservables(*w.var("x"));
// define set of nuisance parameters
w.defineSet("nuisParams","a1,a2,nbackground");

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


w.var("mass")->setConstant(true);
w.var("width")->setConstant(true);



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

// import the data
// for performance reason we transform the data in a binned data set
TH1 * h1 = data.createHistogram("x");
RooDataHist binData("obsData","obsData",*w.var("x"), h1);
w.import(binData);

// write workspace in the file
w.writeToFile("HggModel.root");
```

These lines must be included at the end of the macro we used for exercise 2.

Exercise 2: Profile Likelihood Calculator

In this exercise we will see how to run the RooStats Profile Likelihood calculator on the Gaussian signal plus Exponential background model. We will create a ROOT macro, doing the following:

- Read the workspace from the file and get a pointer to the `ModelConfig` and data set objects.
- Create the Profile likelihood calculator class

- Run the calculator by getting the interval at the corresponding desired confidence level
- Plot the interval and the profile likelihood function

Hint running StandardProfileLikelihoodDemo.C code Hide

- Edit the macro, and find the line containing `SetConfidenceLevel`.
- Change from 0.95 to 0.68 and save the macro
- Run the macro as following

```
root [0] .L StandardProfileLikelihoodDemo.C
root [1] StandardProfileLikelihoodDemo("HggModel.root", "w", "ModelConfig", "obsData" )
```

Hint for writing RooStats code Hide
Here is the code to read the workspace from the file. Remember to add at the beginning of the macro `using namespace RooStats;`.

```
using namespace RooStats;

void SimplePL(
   const char* infile =  "HggModel.root",
   const char* workspaceName = "w",
   const char* modelConfigName = "ModelConfig",
   const char* dataName = "obsData" )
{

  // 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 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);
```

We create now the `ProfileLikelihoodCalculator class passing the =ModelConfig` object and a data set object (`RooAbsData`):

```
ProfileLikelihoodCalculator pl(*data,*mc);
```

Now the calculator has all information and can compute the interval. We need to set first the desired confidence level and then call `GetInterval( )`:

```
pl.SetConfidenceLevel(0.683); // 68% interval
LikelihoodInterval* interval = pl.GetInterval();
```

We can then retrieve the actual interval lower/upper value for the desired parameter (e.g. the parameter of interest):

```
// find the iterval on the first Parameter of Interest
RooRealVar* firstPOI = (RooRealVar*) mc->GetParametersOfInterest()->first();

double lowerLimit = interval->LowerLimit(*firstPOI);
double upperLimit = interval->UpperLimit(*firstPOI);


cout << "\n68% interval on " <<firstPOI->GetName()<<" is : ["<<
```

```
    lowerLimit << ", "<<
    upperLimit <<"] "<<endl;
```

At the end we can plot the interval and the profiled likelihood function in a ROOT Canvas using the `LikelihoodIntervalPlot` class:

```
LikelihoodIntervalPlot * plot = new LikelihoodIntervalPlot(interval);
plot->SetRange(0,150);  // change range to have interval visible (default range is variable range
plot->Draw();
```

Note that for some complex model, which require time to evaluate, it can be convenient to use less points and also use the "tf1" option

```
plot->SetPoints(50);
plot->Draw("tf1");
```

Solution Hide

```
using namespace RooStats;

void SimplePL(
   const char* infile =  "HggModel.root",
   const char* workspaceName = "w",
   const char* modelConfigName = "ModelConfig",
   const char* dataName = "obsData" )
{
  /////////////////////////////////////////////////////////////
  // 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 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);


  ProfileLikelihoodCalculator pl(*data,*mc);
  pl.SetConfidenceLevel(0.683); // 68% interval
  LikelihoodInterval* interval = pl.GetInterval();

   // find the iterval on the first Parameter of Interest
  RooRealVar* firstPOI = (RooRealVar*) mc->GetParametersOfInterest()->first();

  double lowerLimit = interval->LowerLimit(*firstPOI);
  double upperLimit = interval->UpperLimit(*firstPOI);


  cout << "\n68% interval on " <<firstPOI->GetName()<<" is : ["<<      lowerLimit << ", "<<      up
  //plot->SetNPoints(50);  // do not use too many points, it could become very slow for some mode
  plot->Draw("tf1");  // use option TF1 if too slow (plot.Draw("tf1")

}
```

- Profile Likelihood result on H->gg model:

```
68% interval  on nsignal is : [201.512, 348.135]
```

Profile Likelihood scan for the number of signal events

Exercise 3: Using the Bayesian Calculators

In this exercise we will learn to use the RooStats Bayesian calculators. We have the choice between using the MCMC calculator or the numerical one. We will start using the MCMC and optionally we will show how to use the numerical Bayesian calculator. We will use the same model used before for the Profile Likelihood calculator.

What we need to do is very similar to the previous exercise. We run the macro `StandardBayesianMCMCDemo.C` or `StandardBayesianNumericalDemo.C`. What the macro does is:

- Read the model from the file as in the previous exercise
- Create the `MCMCCalculator` class
- Configure the calculator setting the proposal function, number of iterations, etc..
- Call the `GetInterval` function to compute the desired interval. The returned `MCMCInterval` class will contained the resulting Markov-Chain.
- Plot the result using the `MCMCIntervalPlot` class

Since the nuisance parameters are scanned in all their range, we need to limited them in a restricted range to integrate them. We can use for example a range defined by 10 sogma of their fitted value. We can add this lines in the tutorial macro:

```
// define reasanable range for nuisance parameters ( +/- 10 sigma around fitted values) to facili
RooArgList nuisPar(*mc->GetNuisanceParameters());
for (int i = 0; i < nuisPar.getSize(); ++i) {
   RooRealVar & par = (RooRealVar&) nuisPar[i];
   par.setRange(par.getVal()-10*par.getError(), par.getVal()+10*par.getError());
}
```

Hint for running the macro Hide
The macro computes by default a 95% upper limit. You need to change again the confidence level and in addition comment the line

```
mcmc.SetLeftSideTailFraction(0);
```

Then you need to change the line setting the maximum of the POI (number of signal event). Set it to 1000.

```
firstPOI->setMax(1000);
```

root [0] .L StandardBayesianMCMCDemo.C root [1] StandardBayesianMCMCDemo("HggModel.root", "w", "ModelConfig", "obsData")

Hint for writing the code Hide
Assuming we have already the code for reading a model from the previous exercise, we create the `MCMCCalculator class passing the =ModelConfig` object and a data set object (`RooAbsData`) exactly as we did for the `ProfileLikelihoodCalculator`:

```
MCMCCalculator mcmc(*data,*mc);
mcmc.SetConfidenceLevel(0.683); // 683% interval
```

Configure the calculator.

- We set the proposal function and the number of iterations. As proposal we use the `SequentialProposal` function. This function samples each parameter independently using a gaussian pdf with a sigma given by a fraction of the parameter range. We use in this case a value of 0.1 for the fraction. It is important to check that the acceptance of the proposal is not too low. In that case we are wasting iterations.

```
// Define proposal function. This proposal function seems fairly robust
SequentialProposal sp(0.1);
mcmc.SetProposalFunction(sp);
```

Set then the number of iterations. We use a not too large number of iterations to avoid waiting too long in this tutorial example.

```
// set number of iterations and initial burning steps
mcmc.SetNumIters(100000);         // Metropolis-Hastings algorithm iterations
mcmc.SetNumBurnInSteps(1000);       // first N steps to be ignored as burn-in
```

Define the interval we want to compute. We will compute a central interval but we can compute

- a shortest interval (default and the only one defined in a multi-dimensional posterior)
- central interval (equal probability on both the left and right side)
- one sided (lower or upper limits)

```
mcmc.SetLeftSideTailFraction(0.5); // 0.5 for central Bayesian interval  and 0 for upper limit
```

Run the calculator (obtain the Markov chain and retrieve the actual interval lower/upper value for the desired parameter (e.g. the parameter of interest) by integrating the posterior function:

```
MCMCInterval* interval = mcmc.GetInterval();

// print out the iterval on the first Parameter of Interest
RooRealVar* firstPOI = (RooRealVar*) mc->GetParametersOfInterest()->first();
cout << "\n68% interval on " <<firstPOI->GetName()<<" is : ["<<
             interval->LowerLimit(*firstPOI) << ", "<<
             interval->UpperLimit(*firstPOI) <<"] "<< endl;
```

At the end we plot the interval and the posterior function in a ROOT Canvas using the `MCMCIntervalPlot` class:

```
// make a plot of posterior function
new TCanvas("IntervalPlot");
```

```
MCMCIntervalPlot plot(*interval);
plot.Draw();
```

Solution Hide

```
using namespace RooFit;
using namespace RooStats;

void SimpleMCMC( 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 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);

  RooRealVar * nsig = w->var("nsig");
  if (nsig) nsig->setRange(0,200);

  MCMCCalculator mcmc(*data,*mc);
  mcmc.SetConfidenceLevel(0.683); // 68.3% interval

  // Define proposal function. This proposal function seems fairly robust
  SequentialProposal sp(0.1);
  mcmc.SetProposalFunction(sp);

  // set number of iterations and initial burning steps
  mcmc.SetNumIters(100000);         // Metropolis-Hastings algorithm iterations
  mcmc.SetNumBurnInSteps(1000);       // first N steps to be ignored as burn-in

  // default is the shortest interval
  // here we use central interval
  mcmc.SetLeftSideTailFraction(0.5); // for central Bayesian interval
  //mcmc.SetLeftSideTailFraction(0); // for one-sided Bayesian interval

  // run the calculator
  MCMCInterval* interval = mcmc.GetInterval();

  // print out the iterval on the first Parameter of Interest
  RooRealVar* firstPOI = (RooRealVar*) mc->GetParametersOfInterest()->first();
  cout << "\n68% interval on " <<firstPOI->GetName()<<" is : ["<<     interval->LowerLimit(*first


  // make a plot of posterior function
  new TCanvas("IntervalPlot");
  MCMCIntervalPlot plot(*interval);
  plot.Draw();

}
```

- MCMC Bayesian result :

```
Metropolis-Hastings progress: ........................................................................
[#1] INFO:Eval -- Proposal acceptance rate: 9.2037%
[#1] INFO:Eval -- Number of steps in chain: 92037
68% interval on nsig is :  [210, 350]
```

Posterior function for the number of signal events obtained using the `MCMCCalculator`

Exercise 4: Create a Poisson Counting Model

We create now a counting model based on Poisson Statistics. Let's suppose we observe `nobs` events when we expect `nexp`, where `nexp = s+b` (`s` is the number of signal events and `b` is the number of background events. The expected distribution for `nexp` is a Poisson : `Poisson( nobs | s+b)`. `s` is the parameter we want to estimate or set a limit (the parameter of interest), while `b` is the nuisance parameter. `b` is not known exactly, but has uncertainty `sigmab`. The nominal value of `b` is = b0=. To express this uncertainty we add in the model a Gaussian constraint. We can interpret this term as having an additional measurement `b0` with an uncertainty `sigmab`: i.e. we have a likelihood for that measurement `=Gaussian( b0 | b, sigma)`. In case of Bayesian statistics we can interpret the constraint as a prior knowledge on the parameter b.

- Make first the RooFit workspace using its factory syntax. Let's assume `nobs=3`, `b0=1`, `sigmab=0.2`.
- Create the `ModelConfig` object and import in the workspace. We need to add in the `ModelConfig` also `b0` as a "global observable=. The reason is that `b0` needs to be treated as an auxiliary observable in case of frequentist statistics and varied when tossing pseudo-experiments.

Hint Hide
We make first the RooFit workspace using its factory syntax. We have `nobs=3`, `b0=1`, `sigmab=0.2`.

```
int nobs = 3;    // number of observed events
double b0 = 1; // number of background events
double sigmab = 0.2;    // relative uncertainty in b

RooWorkspace w("w");

// make Poisson model * Gaussian constraint
w.factory("sum:nexp(s[3,0,15],b[1,0,10])");
// Poisson of (n | s+b)
w.factory("Poisson:pdf(nobs[0,50],nexp)");
// Gaussian constraint to express uncertainty in b
w.factory("Gaussian:constraint(b0[1,0,10],b,sigmab[0.2])");
```

```
// the total model p.d.f will be the product of the two
w.factory("PROD:model(pdf,constraint)");
```

We need also to define `b0`, the global observable, as a constant variable (this is needed when we fit the model). However, `b0` often needs to have a range. w.var("b0")->setConstant(true);

After creating the model p.d.f we create the `ModelConfig` object and we set `b0` as the "global observable=

```
ModelConfig mc("ModelConfig",&w);
mc.SetPdf(*w.pdf("model"));
mc.SetParametersOfInterest(*w.var("s"));
mc.SetObservables(*w.var("nobs"));
mc.SetNuisanceParameters(*w.var("b"));
// need now to set the global observable
mc.SetGlobalObservables(*w.var("b0"));
// this is needed for the hypothesis tests
mc.SetSnapshot(*w.var("s"));

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

We generate then an hypothetical observed data set. Since we have a counting model, the data set will contain only one event and the observable `nobs` will have our desired value (i.e. 3). Remember to import the data set in the workspace and then save the workspace in a ROOT file.

```
// make data set with the namber of observed events
RooDataSet data("data","", *w.var("nobs"));
w.var("nobs")->setVal(3);
data.add(*w.var("nobs") );
// import data set in workspace and save it in a file
w.import(data);
w.writeToFile("ComtingModel.root", true);
```

Code Solution Hide

```
using namespace RooFit;
using namespace RooStats;

void CountingModel(  int nobs = 3,           // number of observed events
                     double b = 1,           // number of background events
                     double sigmab = 0.2 )   // relative uncertainty in b
{
   RooWorkspace w("w");

// make Poisson model * Gaussian constraint
   w.factory("sum:nexp(s[3,0,15],b[1,0,10])");
// Poisson of (n | s+b)
   w.factory("Poisson:pdf(nobs[0,50],nexp)");
   w.factory("Gaussian:constraint(b0[0,10],b,sigmab[1])");
   w.factory("PROD:model(pdf,constraint)");


   w.var("b0")->setVal(b);
   w.var("b0")->setConstant(true); // needed for being treated as global observables
   w.var("sigmab")->setVal(sigmab*b);


   ModelConfig mc("ModelConfig",&w);
   mc.SetPdf(*w.pdf("model"));
   mc.SetParametersOfInterest(*w.var("s"));
   mc.SetObservables(*w.var("nobs"));
   mc.SetNuisanceParameters(*w.var("b"));

   // these are needed for the hypothesis tests
   mc.SetSnapshot(*w.var("s"));
```

```
   mc.SetGlobalObservables(*w.var("b0"));

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

   // make data set with the namber of observed events
   RooDataSet data("data","", *w.var("nobs"));
   w.var("nobs")->setVal(3);
   data.add(*w.var("nobs") );
   // import data set in workspace and save it in a file
   w.import(data);

   w.Print();

   TString fileName = "CountingModel.root";

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

}
```

Suppose you want to define the uncertainty in the background as log-normal. How do you do this ?

Suppose instead the background is estimated from another counting experiments. How do you model this ? (This model is often call on/off model)

You can look at this past RooStats exercise (RooStatsTutorialsAugust2012) for these solutions.

Exercise 5: Compute a 95% Upper Limit with the Counting Model

Using the previously defined Counting model we can compute a 95% upper limit on the signal parameter `s`. We could use :

- The Bayesian Calculator or the MCMC Calculator
- The Profile Likelihood Calculator

Suppose we use the Bayesian calculator:

Hint Hide
In this case we can re-use the macro `StandardBayesNumericalDemo.C`. We just need to change these lines, setting the tip elf interval and give as input the right ROOT file containing the workspace.

```
//bayesianCalc.SetConfidenceLevel(0.68); // 68% interval
bayesianCalc.SetConfidenceLevel(0.95); // 95% interval

// set the type of interval
//bayesianCalc.SetLeftSideTailFraction(0.5); // for central interval
bayesianCalc.SetLeftSideTailFraction(0.); // for upper limit
```

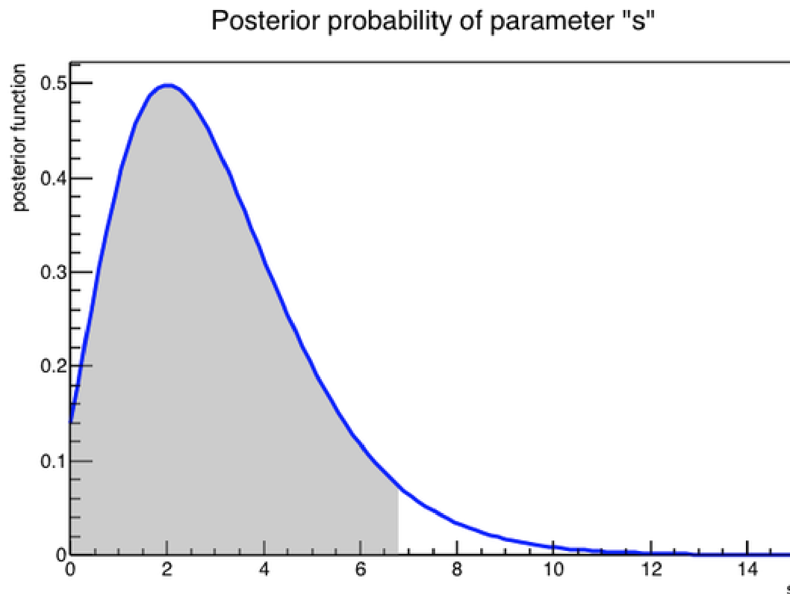
This is the result we will obtain on the Bayesian calculator:

Solution Hide

- Numerical Bayesian 95% Upper limit result on Counting Poisson model

```
[#1] INFO:Eval -- BayesianCalculator::GetInterval - found a valid interval : [0 , 6.80149 ]
```

```
95% interval on s is : [0, 6.80149]
```



Posterior function for the counting model (nobs = 3, b = 1) obtained using the `BayesianCalculator`

Exercise 6: Compute the significance (Hypothesis Test)

The aim of this exercise is to compute the significance of observing a signal in the Hgg model. 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).

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 (type 0)
- Hybrid Calculator (type 1)
- Asymptotic calculator (type 2)

The type refers to the index we specify in the `StandardHypoTestDemo.C` macro.

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, but we will use the one-side Profile likelihood test statistics (test_statistics_type = 3).

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

So we will do the exercise running the =StandardHypoTestDemo.C macro. We run as following

```
root [0] .L StandardHypoTestDemo.C
root [1] StandardHypoTestDemo("HggModel.root", "w", "ModelConfig","", "obsData",2,3 )
```

Asymptotic calculator Solution Hide

- Result of the significance test on H->gg model:

```
Results HypoTestAsymptotic_result:
 - Null p-value = 8.57802e-05
 - Significance = 3.75759
 - CL_b: 8.57802e-05
 - CL_s+b: 0.494872
 - CL_s: 5769.07
```

Exercise 6b: 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 H->gg 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, const char * workspaceName = "w", const char * modelConfigName =


  /////////////////////////////////////////////////////////////
  // 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(200);  // 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 = 112;
   double massMax = 158;

   // loop on the mass values

   int npoints = 30;
   for( double mass=massMin; mass<=massMax; mass += (massMax-massMin)/double(npoints) )    {

      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

Exercise 7: Compute Limits using Hypothesis Test Inversion (`CLs` Limits).

In our last exercise we will compute an upper limit using the frequentist hypothesis test inversion method. We need to perform the hypothesis test for different parameter of interest points and compute the corresponding p-values. Since we are interesting in computing a limit, the test null hypothesis, that we want to disprove, is the in this case the S+B model, while the alternate hypothesis is the B only model. It is important to remember this, when we construct the hypothesis test calculator.

For doing the inversion `RooStats` provides an helper class, `HypoTestInverter`, which is constructed using one of the HypoTest calculator (Frequentist, Hybrid or Asymptotic). To compute the limit we need to do:

- Create/read a workspace with S+B and B only model. If the B model does not exist we can create as in the previous exercise (setting the poi to zero).
- Create an HypoTest calculator but using as null the S+B model and as alt the B only model
- Configure the calculator if needed (set test statistics, number of toys, etc..)
- Create the `HypoTestInverter` class and configure it (e.g. to set the number and range of points to scan, to use `CLs` for computing the limit, etc)
- Run the Inverter using `GetInterval` which returns an `HypoTestInverterResult` object.
- Look at the result (e.g. limit, expected limits and bands)
- Plot the result of the scan and the test statistic distribution obtained at every point.

Using the `StandardHypoTestInvDemo.C` tutorial

This is a macro in $ROOTSYS/tutorials/roostats which can run the hypothesis test inversion with several possible options. We can find the tutorial code also here. Suppose we want to compute the 95% `CLs` limit on the counting model:

```
root [0] .L $ROOTSYS/tutorials/roostats/StandardHypoTestInvDemo.C+
root [1] StandardHypoTestInvDemo("GausExpModel.root", "w", "ModelConfig", "", "data", 2, 3, true,
```

The macro will print out the observed limit, expected limit and +/1,2 sigma bands/ It will also produce a p-value (`CLs` in this case) scan plot.

Next we can run the Frequentist calculation (it will take considerably more time, but we use only 1000 toys):

```
root [1] StandardHypoTestInvDemo("GausExpModel.root", "w", "ModelConfig", "", "data", 0, 3, true,
```

Since we are using toys to estimate the test statistics distribution, the macro will produce the sampling distribution plots for each scanned point.

Note that, since we are running on the number counting model, we need to set to true the last option

For reference, here is the list of input arguments to the macro:

```
StandardHypoTestInvDemo(const char * infile = 0,
                        const char * wsName = "combined",
                        const char * modelSBName = "ModelConfig",
                        const char * modelBName = "",
                        const char * dataName = "obsData",
                        int calculatorType = 0,
                        int testStatType = 0,
                        bool useCLs = true ,
                        int npoints = 6,
                        double poimin = 0,
                        double poimax = 5,
```

```
int ntoys=1000,
bool useNumberCounting = false,
const char * nuisPriorName = 0){
```

Where we have for the calculatorType:

- type = 0 Freq calculator
- type = 1 Hybrid calculator
- type = 2 Asymptotic calculator

- testStatType = 0 LEP
- = 1 Tevatron
- = 2 Profile Likelihood
- = 3 Profile Likelihood one sided (i.e. = 0 if mu < mu_hat)

- useCLs : scan for CLs (otherwise for CLs+b)

- npoints: number of points to scan , for autoscan set npoints = -1

- poimin,poimax: min/max value to scan in case of fixed scans (if min > max, try to find automatically)

- ntoys: number of toys to use

- useNumberCounting: set to true when using number counting events

- nuisPriorName: name of prior for the nnuisance. This is often expressed as constraint term in the global model

Exercise 8: Creating counting model (same as Ex. 4) using the HistFactory

We create the same model as the one created in Exercise 4, but using this time the HistFactory. Compute then the CLs 95% upper limit on the signal rate parameter `mu`.

Hint on making the model Hide
To create the model we need to create first the input histograms for the observed events, the expected signal and background.

```
// create first input histograms
int nobs = 3; double b = 1; double errb = 0.2;

// observed histogram
TH1D * hobs = new TH1D( hobs","hobs",1,0,1);
hobs->SetBinContent(1,nobs);

//signal histogram (assume expected one is 1)
TH1D * hs = new TH1D("hs","signal  histo",1,0,1);
hs->SetBinContent(1,1);


TH1D * hb = new TH1D("hb","bkg  histo",1,0,1);
hb->SetBinContent(1,b);
```

Then we create the HistFactory class, `Measurement`. We can set global parameter values like the luminosity

```
HistFactory::Measurement meas("CountingModel","CountingModel");
meas.SetPOI("mu");
```

```
meas.SetLumi(1.0);
meas.SetLumiRelErr(0.1);  // not relevant
// this does not make lumi varying
meas.AddConstantParam("Lumi");
```

And then the `Channel` and `Sample` classes. We add a systematic uncertainty on the background sample

```
HistFactory::Channel channel("SignalRegion");
channel.SetData(hobs);

HistFactory::Sample signal("signal");
signal.AddNormFactor("mu",1,0,30);
signal.SetHisto(h1_s);
channel.AddSample(signal);

HistFactory::Sample backg("background");
backg.SetHisto(h1_b);
backg.AddOverallSys("b_unc",1.-errb, 1+errb);  // b uncertainty
channel.AddSample(backg);

meas.AddChannel(channel);
```

We could eventually add also other systematics uncertainties or change the constraint type of the systematics. See the HIstFactory User Guide or the reference documentation for the needed information.

At the end we create the workspace model and write it in a file.

```
RooWorkspace * w = HistFactory::MakeModelAndMeasurementFast(meas);
w->writeToFile("CountingModel_HF.root");
```

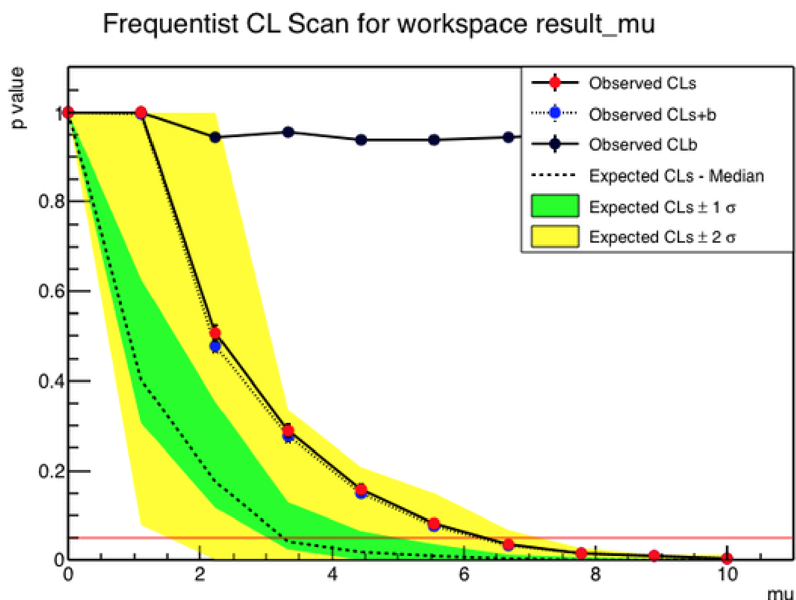
After having written the workspace in the file we can run the RooStats tutorial `StandardHypoTestInvDemo.C`, using for example the FrequentistCalculator and the AsymptoticCalculator to compute the CLs 95% limit:

```
root[0] .x StandardHypoTestInvDemo("CountingModel.root","w","ModelConfig","","data",0,3, true, 10
```

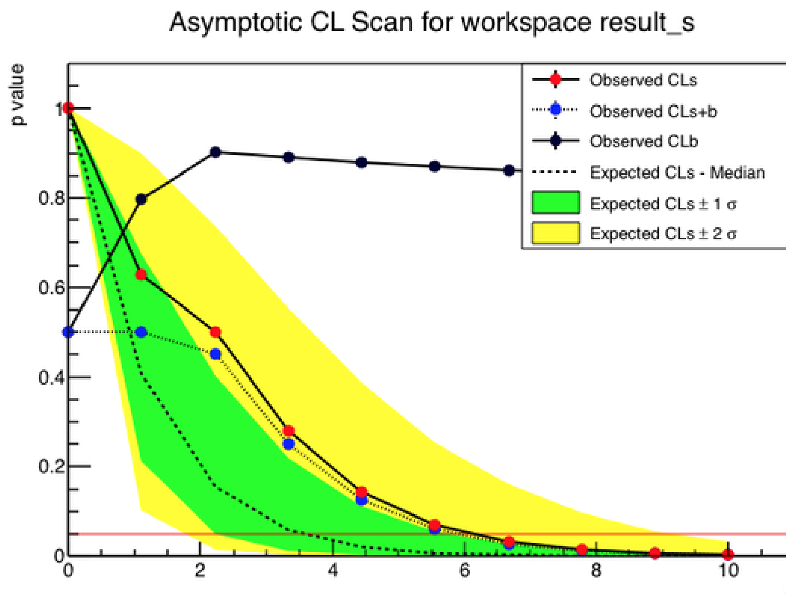
Solution Hide

```
using namespace RooFit;
using namespace RooStats;
using namespace RooStats::HistFactory;


// make a counting model using the histfactory

TString fileName = "CountingModel_HF.root";

void CountingModel_HF( int nobs=3, double b = 1, double errb = 0.2) {


   TH1D * hobs = new TH1D("hobs","hobs",1,0,1);
   hobs->SetBinContent(1,nobs);


   TH1D * h1_s = new TH1D("hs","signal  histo",1,0,1);
   h1_s->SetBinContent(1,1);


   TH1D * h1_b = new TH1D("hb","bkg  histo",1,0,1);
   h1_b->SetBinContent(1,b);


   HistFactory::Measurement meas("CountingModel","CountingModel");
   meas.SetPOI("mu");
```

```
   meas.SetLumi(1.0);
   meas.SetLumiRelErr(0.001);   // not relevant
   // this does not make lumi varying
   meas.AddConstantParam("Lumi");

   HistFactory::Channel channel("SignalRegion");
   channel.SetData(hobs);

   HistFactory::Sample signal("signal");
   signal.AddNormFactor("mu",1,0,30);
   //signal.AddOverallSys("sig_unc",0.99, 1.01);
   signal.SetHisto(h1_s);
   channel.AddSample(signal);

   HistFactory::Sample backg("background");
   //backg.AddNormFactor("taub",b,0,10);
   backg.SetHisto(h1_b);
   backg.AddOverallSys("b_unc",1.-errb, 1+errb);   // b uncertainty
   channel.AddSample(backg);


   meas.AddChannel(channel);



   // make the model
   RooWorkspace * w = HistFactory::MakeModelAndMeasurementFast(meas);

   // to change interpolation mode to piecewise linear (default is type=4, polynomial interp + ex
   HistFactory::FlexibleInterpVar * fl = (HistFactory::FlexibleInterpVar *) w->function("backgrou
   if  (fl) fl->setAllInterpCodes(0);
   else printf("ERROR changing interp code \n");

   w->writeToFile(fileName);

   printf("written model to file for nobs = %d, b = %f \n",nobs, b);
}
```



Scan result for the Poisson counting model using the Frequentist calculator



Scan result for the Poisson counting model using the Asymptotic calculator

Suppose you want to extend the model to include an auxillary Poisson measurement (an additional channel) for the background. The number of expected events in the background region are given by `tau*b`, where `tau` is a normalization parameter. This model is called On/Off model. How do you create this model with the HistFactory ?

Solution for On/Off model Hide

This is the code solution for the On/Off model. We assume that in the signal region we observe 3 events, in the background region we observe 10 events and `tau`, the scale factor is `10`.

```
using namespace RooFit;
using namespace RooStats;
using namespace RooStats::HistFactory;


// make a on/off model using the histfactory

TString fileName = "OnOffModel_HF.root";

void OnOffModel_HF( int nOn=3, int nOff = 10, double tau = 10) {


   TH1D * hobsA = new TH1D("hobsON","hobsON",1,0,1);
   hobsA->SetBinContent(1,nOn);

   TH1D * hobsB = new TH1D("hobsOFF","hobsOFF",1,0,1);
   hobsB->SetBinContent(1,nOff);

   TH1D * h1 = new TH1D("h1","unit histo",1,0,1);
   h1->SetBinContent(1,1);
   TH1 * h1_bA = (TH1*) h1->Clone("h1_bA");
   TH1 * h1_bB = (TH1*) h1->Clone("h1_bB");

   HistFactory::Measurement meas("OnOffModel","OnOffModel");
   meas.SetPOI("mu");

   meas.SetLumi(1.0);
   meas.SetLumiRelErr(0.001);  // not relevant
   // this does not make lumi varying
   meas.AddConstantParam("Lumi");

   printf("make channel A\n");
```

```
    HistFactory::Channel channelA("Signalregion");
    channelA.SetData(hobsA);

    HistFactory::Sample signal("signal");
    signal.AddNormFactor("mu",1,0,100);
    //signal.AddOverallSys("sig_unc",0.99, 1.01);
    signal.SetHisto(h1);
    channelA.AddSample(signal);

    HistFactory::Sample backgA("bA");
    backgA.AddNormFactor("bA",1,0,10);
    backgA.SetHisto(h1_bA);
    channelA.AddSample(backgA);

    printf("make channel B\n");
    HistFactory::Channel channelB("Bregion");
    channelB.SetData(hobsB);

    // assume no signal in Off region

    HistFactory::Sample backgB("bB");
    backgB.AddNormFactor("bA",1,0,100);
    backgB.AddNormFactor("tau",tau,1,1.E6);
    backgB.SetHisto(h1_bB);
    channelB.AddSample(backgB);


    printf("adding channels....\n");
    meas.AddChannel(channelA);
    meas.AddChannel(channelB);

    // we need to set tau as a constant parameters
    meas.AddConstantParam("tau");


    meas.Print();

          // make the model
    RooWorkspace * w = HistFactory::MakeModelAndMeasurementFast(meas);

    w->writeToFile(fileName);

    printf("written model to file for nOn = %d, nOff = %d, tau = %f \n",nOn, nOff, tau);
}
```

-- LorenzoMoneta - 2015-03-23

This topic: RooStats > RooStatsExercisesMarch2015
Topic revision: r5 - 2015-03-26 - LorenzoMoneta

Copyright &© 2008-2021 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.
or Ideas, requests, problems regarding TWiki? use Discourse or Send feedback