Table of Contents

Table of Contents

HistFitterTutorial

This tutorial complements the **main HistFitter page** here.

Introduction

HistFitter is a high-level user-interface to perform binned likelihood fits and follow-up with their statistical interpretation. The user-interface and its underlying configuration manager are written in python, and are executing external computational software compiled in C++ such as HistFactory, RooStats and RooFit.

This tutorial is divided into five parts. Essentially, the first two parts demonstrate how to configure a fit, with or without shape analysis, using HistFitter. The third part demonstrates how to retrieve details of the fit results, like e.g. signal and background yields in the various channels. The fourth and fifth parts focus on statistics tools to interpret the fit results. The parts labeled (*) are the most important ones.

Part 0: Setting up HistFitter

To set up the package, log on to a prepared (see this link) machine and follow instructions below (replace USERNAME by your own user id). If you have not done so yet:

```
export ALRB_TutorialData=<data_dir_provided_for_tutorial>
setupATLAS
diagnostics
setMeUp <tutorial-version> (for example: =triumf-sep2014=)
```

Then proceed to check out the HistFitter package:

```
svn co svn+ssh://$USER@svn.cern.ch/reps/atlasphys/Physics/SUSY/Analyses/HistFitter/tags/HistFitte
cd HistFitterTutorial
```

Then run:

```
localSetupROOT
source setup.sh
cd src
make
cd ..
ln -s $ALRB_TutorialData/samples samples
```

Note: the recommended tag for the tutorial may not always be the same tag as recommended for physics analysis (currently these tags overlap).

Note: in case of need, the required input files are replicated here on lxplus:
`/eos/atlas/atlascerngroupdisk/phys-susy/histfitter/stronglepton/`
For more info about EOS, see this link.

Note: an earlier (now superceded) version of this tutorial was based on
`atlastest/Physics/SUSY/Analyses/HistFitter/tags/HistFitter-00-00-39-branch`.

Note: every time you login, navigate to your HistFitter directory (in this case `HistFitterTutorial`), and issue:

```
setupATLAS
localSetupROOT
source setup.sh
```

which will setup ROOT, the appropriate environment variables and software versions.

HistFitter usage

HistFitter is a python run script that takes a python configuration file as input. The configuration file sets up the pdf of the control, validation, and signal regions of your phyiscs analysis. (Details on what is contained in the configuration file follow later below.)

When running HistFitter without any configuration file, some help documentation is shown, demonstrating its command line usage. There are quite a few option. The most important ones, discussed in this tutorial, are: `-t` (create histograms from trees), `-w` (create workspaces), `-f` (perform a free fit), `-p` (run a hypothesis test), and `-l` (set an upper limit).

At any time, you can change HistFitters verbosity by setting the log level through `-L`, e.g. `-L WARNING`. All commandline options can be viewed by running

```
HistFitter.py --help
```

This gives:

```
 * * * Welcome to HistFitter * * *

usage: HistFitter.py [-h] [-L {VERBOSE,DEBUG,INFO,WARNING,ERROR,FATAL,ALWAYS}]
                     [-F {bkg,disc,excl}] [-t] [-w] [-x] [-f]
                     [--fitname FITNAME] [-m PARAM] [-n NUM_TOYS] [-s SEED]
                     [-a] [-i] [-l] [-p] [-z] [-g GRID_POINTS] [-r REGIONS]
                     [-d] [-D DRAW] [-b BACKGROUND] [-0] [-T] [-V] [-c CMD]
                     [-u USERARG] [-A] [-P]
                     configFile [configFile ...]

positional arguments:
  configFile            configuration file to execute

optional arguments:
  -h, --help            show this help message and exit
  -L {VERBOSE,DEBUG,INFO,WARNING,ERROR,FATAL,ALWAYS}, --log-level {VERBOSE,DEBUG,INFO,WARNING,ERR
                        set log level
  -F {bkg,disc,excl}, --fit-type {bkg,disc,excl}
                        type of fit to run
  -t, --create-histograms
                        re-create histograms from TTrees
  -w, --create-workspace
                        re-create workspace from histograms
  -x, --use-XML         write XML files by hand and call hist2workspace on
                        them, instead of directly writing workspaces
  -f, --fit             fit the workspace
  --fitname FITNAME     workspace name for fit (not specified takes 1st
                        available fitConfig)
  -m PARAM, --minos PARAM
                        run minos for asymmetric error calculation, optionally
                        give parameter names for which minos should be run,
                        space separated. For all params, use ALL
  -n NUM_TOYS, --num_toys NUM_TOYS
                        set the number of toys, <=0 means to use real data
  -s SEED, --seed SEED  set the random seed for toy generation
  -a, --use-asimov      use Asimov dataset for fitting and plotting
  -i, --interactive     remain in interactive mode after running
  -l, --limit-plot      make limit plot of workspace
  -p, --hypotest        run exclusion hypothesis test
  -z, --discovery-hypotest
                        run discovery hypothesis test
  -g GRID_POINTS, --grid_points GRID_POINTS
                        grid points to process (comma-seperated)
  -r REGIONS, --regions REGIONS
                        signal regions to process (comma-seperated)
  -d                    draw before/after plots
  -D DRAW, --draw DRAW  specify plots to draw, comma separated; choose from
                        ['allPlots', 'before', 'after', 'corrMatrix',
                        'sepComponents', 'likelihood']
  -b BACKGROUND, --background BACKGROUND
                        when doing hypotest, set background levels to values,
                        form of bkgParName,value
  -0, --no-empty        do not draw empty bins when drawing
  -T, --run-toys        run toys (default with mu)
  -V, --validation      include validation regions
  -c CMD, --cmd CMD     python commands to process (semi-colon-seperated)
  -u USERARG, --userArg USERARG
                        arbitrary user argument(s)
  -A, --use-archive-histfile
                        use backup histogram cache file
  -P, --run-profiling   Run a python profiler during main HistFitter execution
```

Have a look what the options `-t`, `-w`, `-f`, `-p` and `-l` mean. Finally, the option `-i` (which keeps HistFitter in interactive mode after running) is useful to execute additional python code. We will use it a few times throughout this tutorial.

Part 1: Building & fitting RooFit PDF for a simple analysis

This section of the tutorial starts with a very simple one-bin analysis example, that is then extended to a shape fit (one channel), and then to a simultaneous multi-channels shape fit. The extension from one-bin to shape fit shows the increased power for signal exclusion, while the extension to multi-channels is more to illustrate how to program more advanced fits in a situation closer to real life. The concepts of discovery vs exclusion fits, and of control vs validation regions are also illustrated.

Simple counting experiment, input from user (one bin, one region) (*)

We will start with the simplest example: a one-bin counting experiment where the inputs are given by the user. A setup for such an experiment is defined in the configuration file `analysis/tutorial/MyUserAnalysis.py`. We will run it first, and discuss it in more detail below.

Running the fit

Please run:

```
HistFitter.py -w -f analysis/tutorial/MyUserAnalysis.py
```

The command will create histograms from your input in `data/`, then create workspaces and store them in `results/MyUserAnalysis` (from the step `-w`). Next, it runs a fit, which also gets stored in the directory `results/MyUserAnalysis`. (The results get stored in a directory with the structure of `results/` and then a subdirectory with the name of your analysis, defined in the python configuration `configMgr.analysisName`.)

We have executed a fit in one signal region which has 7 observed events, a background expectation of 5.0 events, and a signal expectation of 5.0 as well. As a result, after the fit to data the signal strength (`mu_Sig`) is found and reduced to be 0.4 (+- 0.7), corresponding to 2 signal events.

The full fit result is:

```
  RooFitResult: minimized FCN value: -0.650537, estimated distance to minimum: 0.000108911
                covariance matrix quality: Full, accurate covariance matrix
                Status : MINIMIZE=0 HESSE=0

    Constant Parameter    Value
  --------------------  ------------
  binWidth_obs_x_UserRegion_cuts_0    1.0000e+00
  binWidth_obs_x_UserRegion_cuts_1    1.0000e+00
         nom_alpha_cor    0.0000e+00
         nom_alpha_ucb    0.0000e+00
         nom_alpha_ucs    0.0000e+00
  nom_gamma_stat_UserRegion_cuts_bin_0    1.0000e+00
           nominalLumi    1.0000e+00

    Floating Parameter  InitialValue    FinalValue +/-  Error     GblCorr.
  --------------------  ------------  --------------------------  --------
                  Lumi    1.0000e+00    9.9998e-01 +/-  3.88e-02  0.029970
             alpha_cor    0.0000e+00    6.1845e-03 +/-  9.93e-01  0.289918
             alpha_ucb    0.0000e+00   -1.1641e-02 +/-  9.93e-01  0.354760
             alpha_ucs    0.0000e+00   -1.0628e-03 +/-  9.93e-01  0.076783
  gamma_stat_UserRegion_cuts_bin_0    1.0000e+00    1.0001e+00 +/-  2.22e-01  0.509355
                mu_Sig    1.0000e+00    4.0478e-01 +/-  6.66e-01  0.609922
```

There are several parameters in the fit. Their meanings are as follows:

For example, the signal parameter (`mu_Sig`) we have already discussed. There is the luminosity uncertainty (`Lumi`), of 3.9%. Then there are two parameters that indicate uncorrelated systematic errors on the signal and background estimates: `alpha_ucb` and `alpha_ucs`. There's also a fully correlated systematic uncertainty between the signal and background estimates: `alpha_cor`. Finally, there's the parameter `gamma_stat_UserRegion_cuts_bin_0`, which indicates the statistical error on the signal and background estimates from limited MC statistics.

Visualizing the result

We can also visualize the fit result. Re-run the same command, but with the option `-D` used to draw several things:

```
HistFitter.py -w -f -D "before,after,corrMatrix" analysis/tutorial/MyUserAnalysis.py
```

Three figures have now been created in `results/MyUserAnalysis/`:

- `before` and `after` show the number of events before and after the fit. *Before* the fit we expect 10 background plus signal events. *After* the fit, the signal estimate has now been fit to be 2 events, and the total background plus signal events equals the number of observed events (is 7).
- The third plot, `corrMatrix` is a visual representation of the the correlation matrix of the fit parameters.

As one can see, the correlation of the systematic errors with the signal strength parameter are pretty large, and they are negative, meaning that if the fitted background estimate goes up, the fitted signal estimate goes down.

The sizes of the errors can be easily set at the top of `analysis/tutorial/MyUserAnalysis.py`. Please take a look at the numbers in this file. Changing the numbers of events and the sizes of the errors should be pretty intuitive.

Alternatively adding the `-i` option, the plots will show up on your screen, and you will be left in python interactive mode:

```
HistFitter.py -w -f -D "before,after,corrMatrix" -i analysis/tutorial/MyUserAnalysis.py
```

To get out of interactive mode use `Control+d` command.

Checking the impact of systematic errors (optional)

Show Hide
As a little exercise, let's turn off the systematic errors on the background and signal estimates, and see the impact on the error on the fitted signal strength. One can turn off the errors by commenting out the following lines in the python configuration files:

```
bkgSample.buildStatErrors([nbkgErr],"UserRegion","cuts")
bkgSample.addSystematic(corb)
bkgSample.addSystematic(ucb)
```

and for the signal component:

```
sigSample.buildStatErrors([nsigErr],"UserRegion","cuts")
sigSample.addSystematic(cors)
sigSample.addSystematic(ucs)
```

When redoing the fit, what is the uncertainty on `mu_Sig` ?

(Answer: `0.53`)

Note that the luminosity uncertainty (treated as a special systematic uncertainty) can be turned off by commenting in the line:

```
meas.addParamSetting("Lumi",True,1)
```

Simple counting experiment, input from TTrees (one bin, one region) (*)

In this example, the input numbers of events are not set by hand, but are obtained from input `TTrees`. A simple example is defined in the configuration file `analysis/tutorial/MyOneBinExample.py`

Running the fit

Execute the following command:

```
HistFitter.py -t -w -f -F excl -D "before,after" -i analysis/tutorial/MyOneBinExample.py
```

Note that now we include the option `-t` when running the command. This ensures that HistFitter constructs appropriate cut strings and applies them on the trees in your data file. The histograms created are again stored in `data/`.

Understanding the configuration

We've defined a set of input files that we use when reading from trees:

```
bgdFiles = []
if configMgr.readFromTree:
    bgdFiles.append("samples/tutorial/SusyFitterTree_OneSoftEle_BG_v3.root")
    bgdFiles.append("samples/tutorial/SusyFitterTree_OneSoftMuo_BG_v3.root")
else:
    bgdFiles = [configMgr.histCacheFile]
    pass
configMgr.setFileList(bgdFiles)
```

Our signal region is then defined in the `configMgr` 's dictionary of cuts, called `cutsDict` :

```
configMgr.cutsDict["SR"] = "((lep1Pt < 20 && lep2Pt<10 && met>250 && mt>100 && jet1Pt>130 && jet2
```

Such cut strings are used in calls to `TTree::Project()` internally, so you can use any variable or function you have defined in ROOT before this point. The final discriminating variable used to define the SR is met/meff2Jet. Here `met==missing transverse momentum and =meff2Jet=` effective mass consisting of 2 jets, all leptons and `met`. You will see later that this is a good variable to distinguish this specifc signal from the backgrounds.

The signal region is defined later in the config file as

```
srBin = exclusionFitConfig.addChannel("cuts",["SR"],1,0.5,1.5)
```

Note the correspondence of `["SR"]` with the name we chose in our cuts dictionary before!

Throughout the code, you will see systematics defined with weights. Don't worry about that for now, their role will be explained later.

Fit results

We have executed an "exclusion fit" with `-F excl` (see next section) in a signal region with three observed events and a background expectation of ~4.5 events. As a result, the signal is found to be about 0 +/- 1.78, considering systematics on Alpgen KtScale and JES, as you can see on your screen:

```
                  Lumi    1.0000e+00    9.9944e-01 +/-  3.88e-02  0.000594
             alpha_JES    0.0000e+00   -1.9188e-01 +/-  8.74e-01  0.041096
      alpha_KtScaleTop    0.0000e+00    8.5289e-02 +/-  9.55e-01  0.033736
       alpha_KtScaleWZ    0.0000e+00    8.5656e-02 +/-  6.41e-01  0.023383
                mu_SIG    1.0000e+00    4.2723e-05 +/-  1.78e-01  0.009267
```

You can exit the interactive session by typing `Control d` and go to the next step.

Part 2: More advanced fits

After running some simple one-bin fits, we move now to slightly more complicated examples.

Types of fits and regions (*)

Depending on what you are trying to achieve with HistFitter (run a background fit? set a limit? calculate a p-value? calculate model-independent limits?), you will have to change your fit configuration. Each of these scenarios might or might not need a signal. These *fit types* we will cover in this section. Depending on which fit you will use, a region has a certain role.

Setting the region type

An analysis typically considers three different types of data regions (aka channels) for the fit:

- **Signal region** (SR): a signal-enriched region that drives the signal extraction but adds negligible constraints on the background parameters. Set this in your `HistFitter` configuration file as: `yourFitConfig.setSignalChannels([SR1,SR2,...])`
- **Control region** (CR): a background-enriched region with negligible signal contamination. Set as: `yourFitConfig.setBkgConstrainChannels([CR1,CR2,...])`
- **Validation region** (VR): a region in which the background and signal levels are predicted, but which are not used in the fit to constrain parameters. They only serve to validate (as the name suggests) the extrapolation from the control to the signal regions. Set as: `yourFitConfig.setValidationChannels([VR1,VR2,...])`

The fit is based on orthogonal signal and control regions. This way all the control and signal regions are statistically independent, can be modelled by separate PDFs, and thus be combined into one simultaneous fit.

The basic strategy is to share the background parameters of the PDF in all the different regions: CR, SR and VR. The background parameters are predominantly constrained by CRs, probably with large statistics, which in turn reduces the impact of their uncertainties in the SR. The results of a fit to the CR is used to derive (extrapolate) a background prediction in SR. The extrapolation to the validation regions are then used to verify the validity of the extrapolation using the so-called transfer factors.

Setting the fit types

Three different fit configurations are then used in the analysis:

- **Bkg-only fit**: Purpose is to estimate the backgrounds in the SRs and VRs, without assumptions on signal model. Only the CRs are used to constrain the fit parameters. Any potential signal contribution is neglected everywhere. This fit picks out only the regions that have been set as background control regions (see above). After this fit, extrapolation to signal and validation regions is possible. *In a background-only fit, use the SR as a validation region.*
- **Exclusion fit** (aka model-dependent signal fit): Purpose is to set limits on a specific model of BSM physics. Both CRs *and* SRs are used in the fit. The potential signal contribution is taken into account as predicted by the tested model in all the regions. In this configuration, validation regions should not be configured, as consequent hypothesis tests do not distinguish between control and validation regions.
- **Discovery fit** (aka model- *in* dependent signal fit): Purpose is to set model-independent limits on the number of BSM events in SR. Both CRs and *only one* SR are used in the fit. The signal is independently considered in each SR, but is neglected in the CRs. The background prediction can be conservative since any signal contribution in the CRs is attributed to background and thus yields a possible overestimation of the background in the SRs, but that should not happen if you designed your

CRs well. Dedicated SRs (and signal samples) need to be used, which are a single bin counting-experiments. ⚠ Very important is that you **do not input a signal model**. What you *MUST DO instead* is input a 'dummy' signal model that is a single bin histogram with the bin value at 1 (see further examples). Also in this configuration, validation regions should not be configured.

Your config file can be setup to directly choose one of these setups.
One can run a bkg-only fit as: `HistFitter.py -f -F bkg yourPythonConfigFile.py`. The option `-f` means fit, and `-F bkg` helps out to find the background-only fit in your config file. It sets the variable `myFitType` to `FitType.Background` , which can then be picked up in your configuration file to configure the details of your background-only fit setup.
One can run the exclusion fit as: `HistFitter.py -f -F excl yourPythonConfigFile.py`. The option `-F excl` sets the quantity `myFitType` to `FitType.Exclusion` . This can be picked up in your configuration file to configure the details of your exclusion fit setup, such as adding signal models to your regions.
One can run fit the discovery fit as: `HistFitter.py -F disc yourPythonConfigFile.py`. This option considers only the signal and control regions (see above). The option `-F disc` sets the quantity `myFitType` to `FitType.Discovery` .

Examples of these fit setups can be found in this configuration file of the SUSY soft lepton analysis.

Calculating p-values and upper limits for a signal in a one-bin experiment (*)

Having run a fit in which the signal strength is allowed to be free, we will now use this same signal region and uncertainties to proceed with an exclusion test. The signal model we used is one often used in the SUSY group: a 1-step simplified model with gluino mass of 425 GeV and LSP mass of 345 GeV (decay via a chargino). We will keep using the configuration file from the last section.

First, have a quick look again by eye that the signal does not match the data well.

```
 HistFitter.py -f -F excl -D "before,after" -i analysis/tutorial/MyOneBinExample.py
```

Note: we are *not* running the options `-t` and `-w` again, as there is no need to remake the workspaces from trees (sometimes a lengthy process when dealing with many regions and systematics). The amount of expected signal is displayed in red for this signal model. We can qualitatively see that it does not correspond to the observed data.

How does this translate into a quantitative exclusion? To find out, exit the session (`Control d`) and type: `HistFitter.py -F excl -l analysis/tutorial/MyOneBinExample.py`

The options `-p` and `-l` run a hypothesis test (using the signal model nominal expectation) and upper limit scan, respectively. Both options will be discussed in further detail in Part 4.

You can see on your screen a lot of output, and at the end the observed and expected CLs and CLb as a function of the signal strength (mu_SIG). The points where the p-value goes below 0.05 (red line) are excluded at 95% C.L. As also printed on your screen, you can see that these correspond to:

```
<INFO> HypoTestTool: The computed upper limit is: 0.669454 +/- 0
<INFO> HypoTestTool:  expected limit (median) 0.910282
```

The upper limit scan has determined this value by repeatedly running hypothesis tests until the p-value drops below a certain threshold (in our case 0.05, for the typical 95% confidence level). The number where it does is the expected upper limit.

Note: more information on calculating upper limits can be found in Part 4!

Improving the limits: configuring a simple shape fit (6 bins, one channel) (*)

Let us re-do the same exercise with a simple shape fit that uses the same samples and uncertainties as the above 1-bin example. However, we do not use the final discriminating variable `met/meff2Jet` to define the signal region, but instead we attempt to fit its shape.

We can bin our signal region in the variable by re-defining our signal region as:

```
srBin = exclusionFitConfig.addChannel("met/meff2Jet",["SR"],6,0.1,0.7)
srBin.useOverflowBin=True
srBin.useUnderflowBin=True
```

Hopefully the code speaks for itself: we bin the SR in `met/meff2Jet` in 6 bins, from 0.1 to 0.7. (This should also explain the mysterious 0.5 and 1.5 and "cuts" you might have noticed earlier in one-bin fits!)

As usual, create the trees and workspaces and perform the fit by running:
`HistFitter.py -t -w -f -F excl -D "before,after,corrMatrix" -i analysis/tutorial/MyShapeFitExample.py`

Have a look at the now-familiar plots, and you will notice that they have been binned in the variable you used.

We can now proceed to set a limit by using

```
HistFitter.py -F excl -l analysis/tutorial/MyShapeFitExample.py
...
<INFO> HypoTestTool: The computed upper limit is: 0.34073 +/- 0
<INFO> HypoTestTool:  expected limit (median) 0.318142
```

The expected limit is improved by a factor of 2x compared to the above single bin example! The single bin setup is also referred to as "cut & count" setup.

Available features in the python code

Most of the features in any HistFitter python configuration file are described in main HistFitter page here. In general the code should be largely self explainatory. Please ask questions to the instructors if you have any.

Systematic uncertainties (*)

Varying modeling of systematic uncertainties

One strength of HistFitter is its capability to model systematic uncertainties in a rather flexible way. To introduce this topic, look inside the file `analysis/tutorial/MyShapeFitExample.py` and notice the lines where systematic uncertainties are defined. You should see:

```
#topKtScale = Systematic("KtScaleTop",configMgr.weights,ktScaleTopHighWeights,ktScaleTopLowWeight
topKtScale = Systematic("KtScaleTop",configMgr.weights,ktScaleTopHighWeights,ktScaleTopLowWeights
#topKtScale = Systematic("KtScaleTop",configMgr.weights,ktScaleTopHighWeights,ktScaleTopLowWeight

jes = Systematic("JES","_NoSys","_JESup","_JESdown","tree","overallSys")
```

First notice that KtScale is a weight-based systematic defined by High/Low variations of weights from branches of the nominal input Tree. Alternatively, JES is a tree-based systematic: the High/Low variations are specified by pointing to different trees ("_NoSys","_JESup","_JESdown").

For more info on the setup of a configuration file, see here. For more info on the types of systematic uncertainties on can apply, see here. A more physical description of these features can also be found in Sec. 11 of ATL-COM-PHYS-2011-1743.

Exercise: the effect of a systematic type (*)

Try to change the treatment of the topKtScale systematic from overallSys , to histoSys to normHistoSys .

After each modification, re-run:
`HistFitter.py -t -w -F excl -x analysis/tutorial/MyShapeFitExample.py`

We have now used the option `-x` in the code. What this does is that rather than creating workspaces using HistFactory's python interface, HistFitter writes out an XML file and runs the binary `hist2workspace` with that as input. The workspaces written are identical, but writing out the XML files can help you understand the structure of the workspace you've created.

Have a look at this file: `config/MyShapeFitExample/Exclusion_SR_metmeff2Jet.xml`

where you can see:

```
  <Sample Name="Top" HistoName="hTopNom_SR_obs_metmeff2Jet" InputFile="data/MyShapeFitExample.roo
    <HistoSys Name="KtScaleTop" HistoNameHigh="hTopKtScaleTopHigh_SR_obs_metmeff2Jet" HistoNameLo
```

This specifies the actual histograms used to derive the nuisance parameters of the KtScale systematic for the Top sample in the fit. We can open our data file and draw any histogram in it (here using 2 colours to differentiate the two histograms):

```
root -l data/MyShapeFitExample.root

hTopKtScaleTopHigh_SR_obs_metmeff2Jet.SetLineColor(2)
hTopKtScaleTopLow_SR_obs_metmeff2Jet.SetLineColor(4)
hTopKtScaleTopLow_SR_obs_metmeff2Jet.Draw()
hTopKtScaleTopHigh_SR_obs_metmeff2Jet.Draw("same")
hTopNom_SR_obs_metmeff2Jet.Draw("same")
```

Note that the systematic objects themselves have no effect until they are explicitely added to the fit configuration, like this:

```
exclusionFitConfig.getSample("Top").addSystematic(topKtScale)
exclusionFitConfig.getSample("WZ").addSystematic(wzKtScale)
exclusionFitConfig.addSystematic(jes)
```

> **Tip:** in real-world production code, histograms are usually created from trees which can be quite time-consuming. Write your code with some simple controlling if-statements that turn off both the creation of a systematic and its `addSystematic()` call. There is no need to create a systematic you won't ever be using in any of your fits!

Binned shape fit with multiple channels (*)

This example is a simplified version of the "soft lepton" fits documented in ATLAS-CONF-2012-041 and ATL-COM-PHYS-2011-1743, based on the configuration file python/MyConfigExample.py.

Starting from input TTrees, it builds histograms on the fly to perform a simultaneous fit of the jet multiplicity in two control regions: for ttbar (TR) and W+jets (WR). Only three systematic uncertainties are considered: one tree-based systematic (JES), one weight-based systematic (Alpgen KT scale), and the MC stat uncertainty.

First run the background only fit:

```
HistFitter.py -t -w -f -D "before,after" -i
analysis/tutorial/MyConfigExample.py
```

You can see the fitted jet multiplicity distributions in the two control regions. To proceed with an exclusion fit run:

```
HistFitter.py -t -w -f -F excl -D "before,after"
--fitname="Sig_SM_GG_onestepCC_425_385_345" -i
analysis/tutorial/MyConfigExample.py
```

and to calculate the upper limit also:

```
HistFitter.py -l -F excl analysis/tutorial/MyConfigExample.py
```

As you can see, the exclusion power is also improved by the addition of the control region, compared to the simple shape fit with one channel. The control regions are designed to be non-sensitive to signal, but they do adjust the background estimation in the signal regions (take a look at the `mu_Top, mu_W` normalization factors for the backgrounds). This multi-channels fit includes many more uncertainties and starts to get closer to a real-life example.

Validation regions (*)

Another very interesting feature of the multi-channel example is its usage of validation regions. Open again the file `analysis/tutorial/MyConfigExample.py`. Enable the validation regions by setting `doValidation=True` and execute:

```
HistFitter.py -t -w -f -D "before,after"
analysis/tutorial/MyConfigExample.py
```

You can now not only see on the screen the fitted distributions, but also several validation distributions that were not used to constrain the fit but where the fit results are projected. You can see that the data/MC agreement in the validation regions is improved after the fit.

Keep in mind: validation regions should *only* be used in a background fit! We want to test an extrapolated fit result, and not in an exclusion test constrain the backgrounds and signal also in these regions.

Exercise: add validation plots (*)

Edit `analysis/tutorial/MyShapeFitExample.py` in order to add validation plots. Define a validation by relaxing the cuts on the variables `met` and `mt`, but remember to keep it orthogonal to the signal region. Try experimenting with the variable the validation region is binned in, as well as binning. Does the fit improve the data/MC agreement as well in this case?

Data-driven background shape determination

Show Hide
⚠ NOTE: This is an outdated/broken feature, so some aspects may not be working (as other examples). There are other possibilities for data-driven estimates input into HistFitter, discussed above and below.

All previous examples have relied on background estimations from MC. Many analyses actually use data-driven estimations (ABCD method for instance). Such methods can also be implemented using what is known as a ShapeFactor in HistFactory jargon. This essentially builds a sample as a set of bin-by-bin normalization factors in each bin. The example you will be using emulates the DataDriven example from HistFactory as a user-defined HistFitter analysis. This is a simple, two-channel analysis, with two bins in each channel and a transfer factor of one between the channels. The signal sample is assumed to only contribute in the signal region and the data are varied accordingly, while the remaining background samples are taken to be flat. You will be using the `analysis/tutorial/MyUserAnalysis_ShapeFactor.py` configuration file.

The PDF can be built in the normal way for a user-defined analysis (i.e. not from trees) and you can run the fit

```
cd $HISTFITTER/
HistFitter.py -w -f -D "before,after" -i analysis/tutorial/MyUserAnalysis_ShapeFactor.py
```

Note that the plots before fit seem to have huge disagreements, this is because the data-driven background has not been estimated yet. If you look at the plots after the fit, the agreement should be good. One other thing worth noting are the values returned by the fit result. Pay particular attention to the gamma_DDShape_bin_{0,1} parameters, these give the estimated yield in each bin and the errors on the yields. A parameter has also been added to account for the uncertainty on the extrapolation of the data-driven sample from the control region to the signal region, here called alpha_BG2Xtrap.

Now lets inspect the file. On opening you will see that it is rather similar to the other user-defined examples seen previously. However, the event counts are now lists (one number per bin) and there are counts for the CR and SR. Three systematics have also been defined (all with a 5% size): the first gives the uncertainty on the cross section of background 1; the second gives the uncertainty on the signal cross section; and the third is the uncertainty on the extrapolation of background 2 (the data-driven background). These are all implemented as 'userOverallSys', as would be expected for systematics affecting the overall cross section of samples.

Further down, however, you will notice some differences. Firstly there are two buildHisto methods for each of the bkgSample and dataSample. These define the different event counts in each region. For example

```
bkgSample = Sample("Bkg",kGreen-9)
bkgSample.setNormByTheory(True)
bkgSample.buildHisto(nBkgCR,"CR","cuts")
bkgSample.buildHisto(nBkgSR,"SR","cuts")
bkgSample.addSystematic(bg1xsec)
```

The data-driven background is defined by adding a ShapeFactor

```
ddSample = Sample("DataDriven",kGreen+2)
ddSample.addShapeFactor("DDShape")
```

For such samples a flat histogram will be build with a single event in each bin to base the bin-by-bin scaling against in the fit. In this way the fit result corresponds to the scaling of the single events, and so to the expected yields.

Note that the signal region is treated in a different way for this analysis, so that the extrapolation uncertainty is only applied to the DataDriven (background 2) sample

```
chanSR.getSample("DataDriven").addSystematic(xtrap)
```

and that the signal sample is only applied in this region

```
chanSR.addSample(sigSample)
```

As usual, try changing some of the parameters of the fit. What happens if some signal contamination is considered in the control region? How does the extrapolation uncertainty affect the result? How might one extend this to a more complex fit that has a realistic transfer factor?

Part 3: Tools for visualising fits results

When using fits in your analysis, you will have to present your results in an easy-to-understand way. Here, we cover three important things:

- *Yields tables*: after a fit, you will want to show the yields in one or more regions defined in your analysis, including uncertainties
- *Systematics tables*: the impact of the various systematic uncertainties has to be shown
- *Pull plots*: a graphic representing the agreement between observed data and background estimates; usually calculated for the validation regions

HistFitter provides scripts to perform all three things.

Most of the information from this section can also be found in `$HISTFITTER/scripts/README_TABLESCRIPTS`. The scripts described in this section are also in the directory `$HISTFITTER/scripts/`.

Yields table

After running the example from the 1-lepton analysis above (with `doValidation=True`):
`HistFitter.py -t -w -f analysis/tutorial/MyConfigExample.py`
one naturally would like to see the fitted yields in a region. Here we discuss a general script, `YieldsTable.py` to produce these yields in a LaTeX table.

After setting up HistFitter, the scripts `YieldsTable.py` should be available in your `$PATH`. Run it from the command line and it will present its options:

```
Usage:
YieldsTable.py [-o outputFileName] [-c channels] [-w workspace_afterFit] [-s samples] [-b]

Minimal set of inputs [-c channels] [-w workspace_afterFit] [-s samples]
*** Options are:
-c <channels>: single channel (region) string or comma separated list accepted (OBLIGATORY)
-w <workspaceFileName>: single name accepted only (OBLIGATORY) ;   if multiple channels/regions g
-s <sample>: single unique sample name or comma separated list accepted (OBLIGATORY)
-o <outputFileName>: sets the output table file name
-B: run blinded; replace nObs(SR) by -1
-a: use Asimov dataset (off by default)
-b: shows also the error on samples Before the fit (off by default)
-S: also show the sum of all regions (off by default)
-y: take symmetrized average of minos errors
-C: full table caption
-L: full table label
-u: arbitrary string propagated to the latex table caption
-t: arbitrary string defining the latex table name

For example:
YieldsTable.py -c SR7jTEl,SR7jTMu -s WZ,Top -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_je
YieldsTable.py -c SR7jTEl,SR7jTMu -w  /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_lepton
YieldsTable.py -c SR3jTEl,SR3jTMu,SR4jTEl,SR4jTMu -s WZ,Top -w /afs/cern.ch/user/c/cote/susy0/use
YieldsTable.py -c S2eeT,S2mmT,S2emT,S4eeT,S4mmT,S4emT -w /afs/cern.ch/user/c/cote/susy0/users/cot
YieldsTable.py -c S2eeT,S2mmT,S2emT,S4eeT,S4mmT,S4emT -w /afs/cern.ch/user/c/cote/susy0/users/cot
YieldsTable.py -c S2eeT,S2mmT,S2emT,S4eeT,S4mmT,S4emT -w /afs/cern.ch/user/c/cote/susy0/users/cot
YieldsTable.py -c S2eeT,S2mmT,S2emT,S4eeT,S4mmT,S4emT -w /afs/cern.ch/user/c/cote/susy0/users/cot
```

In the minimal case, to produce a yields table, you need to provide the script with a workspace file, the sample names you want to see as a comma-separated string and the regions/channels where you want the yields to be calculated. All regions must be in this workspace. See the output above for more options.

For example, try running on the workspace file produced by running on the `MyConfigExample.py` file:

```
cd $HISTFITTER
YieldsTable.py -s Top,WZ,BG,QCD -c SLWR_nJet,SLTR_nJet -w results/MyConfigExample/BkgOnly_combine
```

Open the output file and compare expected and fitted yields. Run the file through LaTeX and you will see a nice table with all the results from the fit you just performed!

As a small exercise, now rerun the yields table bu also including one or more validation regions, for example the signal region `SS_metmeff2Jet`.

Systematics table

Here we discuss a general script, `SysTable.py` to produce the propagated systematic errors in a LaTex table. After setting up HistFitter, the `SysTable.py` should be available just like the script for yields. Again we can run it to see its options:

```
Usage:
SysTable.py [-c channels] [-w workspace_afterFit] [-o outputFileName] [-o outputFileName] [-s sam

Minimal set of inputs [-c channels] [-w workspace_afterFit]
*** Options are:
-c <channels>: single channel (region) string or comma separated list accepted (OBLIGATORY)
-w <workspaceFileName>: single name accepted only (OBLIGATORY) ;   if multiple channels/regions g
-s <sample>: single unique sample name or comma separated list accepted (sample systematics will
-m <method>: switch between method 1 (extrapolation) and method 2 (refitting with 1 parameter con
-o <outputFileName>: sets the output table file name, name defined by regions if none provided
-b: shows the error on samples Before the fit (by default After fit is shown)
-%: also show the individual errors as percentage of the total systematic error (off by default)
-y: take symmetrized average of minos errors

For example:
SysTable.py -w MyName_combined_BasicMeasurement_model_afterFit.root  -c SR7jTEl_meffInc,SR7jTMu_m
SysTable.py -w MyName_combined_BasicMeasurement_model_afterFit.root  -c SR7jTEl,SR7jTMu -s Top,WZ
SysTable.py -c SR3Lhigh_disc_cuts -s '[topZ,topW,ttbarHiggs,singleTopZ],[diBosonWZ,diBosonPowhegZ

Method-1: set all parameters constant, except for the one you're interested in, calculate the err
Method-2: set the parameter you're interested in constant, redo the fit with all other parameters
```

The minimal set of inputs consists of a workspace name and the regions/channels as a comma-separated string where you want to see the systematics. In the SUSY group, Method-1 is the default method.

For example, one can now run:

```
cd $HISTFITTER
SysTable.py -w results/MyConfigExample/BkgOnly_combined_NormalMeasurement_model_afterFit.root -c
```

Again LaTeX the output file to see a nice table.

Validation pull plot

In order to validate the extrapolation to a signal region, the usual method is to define *validation regions* in between each SR and its CRs. The extrapolated backgrounds can then be compared to regions where no signal is expected, in order to validate the extrapolation method.

To demonstrate such a plot, we rely again on the `MyConfigExample.py` example. Rerun the workspace production and bkg-only fit again with validation regions turned on (`doValidation=True`):
`HistFitter.py -t -w -f  analysis/tutorial/MyConfigExample.py`

Then rerun the yields table script with all validation regions enabled:

```
YieldsTable.py -s Top,WZ,BG,QCD -c
SLWR_nJet,SLTR_nJet,SR1sl2j_cuts,SLVR2_nJet,SS_metmeff2Jet,SSloose_metmeff2Jet
-w
results/MyConfigExample/BkgOnly_combined_NormalMeasurement_model_afterFit.root
-o MyYieldsTable.tex
```

This will produce not only the `.tex` file but also a `.pickle` file that we will use now to make the pull plot. To run the pull plot script execute:

```
python analysis/tutorial/examplePullPlot.py
```

This will produce the plot (`histpull_SS_metmeff2Jet.png`) showing the various CRs and VRs, which you can see for example by:

```
display histpull_SS_metmeff2Jet.png &
```

This script calls the `makePullPlot()` function from `python/pullPlotUtils.py`. If you want to see how to adjust this script for your analysis, please read some details by clicking on Show next.

Show Hide
The relevant function is the following:

```
makePullPlot(pickleFilename, regionList, samples, renamedRegions, region, doBlind)
```

Its arguments are these:

- `pickleFilename` is an output pickle from the yields table script. (Note that the example pull plot script first execute that its exists by running the script.)
- `regionList` is a list of regions to be used in the plot (the bins) and is a list of strings. The example uses `makeRegionList()` to construct this.
- `renamedRegions` is a dictionary of region names to be changed. Its keys are the region names, the values the new names. The example uses `renameRegions()` to build the dict.
- `region` is the name of the SR you're making the pull plot for. It is not used in the code anywhere but only in the plot title label.
- `doBlind` is a boolean that determines whether to show the data point for the any region that includes `SR` in the name (not the case for our example signal region).

Finally, note that you can redefine the pull plot module's color functions to define your own style of plot:

```
pullPlotUtils.getRegionColor = getRegionColor
pullPlotUtils.getSampleColor = getSampleColor
```

Tip: if your analysis has SR-dependent lists of validation regions (i.e. you need different regionLists per SR), it's advisable to modify `makeRegionList()` with an argument rather than keeping hard-coded lists in. This switch is used in e.g. the 0-lepton analysis, where VRZ and CRZ could each be used to constrain the Z background. That is the motivation for having a function that returns a list defined rather than just a list.

Part 4: Statistical interpretation - ROOT based vs HistFitter

This second part of the tutorial focusses on the analysis and statistical interpretation that can be done once the analysis configuration has been defined (as done in part one).

Useful examples included in HistFitter, and discussed below, are:

- Configuring and constructing a workspace using RooStats HistFactory machinery (optional).
- The fit of the pdf to data, and plots of before- and after-fit results (optional, as already discussed).
- Doing an exclusion or discovery hypothesis test.
- Setting a signal model-dependent or model-independent upper limit.
- Making an exclusion contour plot in 2D (or any-dimensional) parameter space of a theory model (here: mSUGRA).

The underlying tools to do these examples are fully based on RooFit and RooStats code. Recall that HistFitter is a wrapper around the RooFit and RooStats functionality, with some useful (user-friendly) added features.

The sections below illustrate alternatively:

- How to use the RooFit and/or RooStats code for a specific task. In particular, this demonstrates the basic HistFactory code that HistFitter is based upon.
- And then how easily the same can be done directly with the HistFitter package.

For the exercises, basic RooFit knowledge is not required.

Setting up a simple cut-and-count analysis in ROOT (optional)

This part discusses how to setup a single bin analysis with HistFactory, and discusses the xml configuration files. (These xml were mentioned before, but are not produced standard anymore. However they can come in handy when debugging your analysis).

Show Hide
This example creates a root file with a RooFit workspace containing the pdf and dataset of a simple counting experiment.

```
cd $HISTFITTER/analysis/simplechannel/
```

The simple counting experiment needs several inputs. First, the number of observed events in data and the event signal and background expectation values for the bin of interest, and second, the (relative) uncertainties on the background and/or signal estimates. Using the RooStats `HistFactory` machinery, the first are given by histograms, and the latter are configured in xml configuration files.

Take a quick look at the macro `makeCountingExperiment.C`. The data, background, and signal histograms are created in the ROOT macro `makeCountingExperiment.C`, and are stored in the root file `data/counting_exp_data.root`. As can be seen in the macro, the observed number of events in data and the expected signal and background expectation events are set to `10, 1, 5` events respectively.

Secondly, in the macro `makeCountingExperiment.C` the command line `hist2workspace config/example.xml` is executed.

The measurement uncertainties are set in the file `config/example_channel.xml`, which is called from the top configuration file `config/example.xml`.

Take a look at `config/example.xml` :

```
<!DOCTYPE Combination  SYSTEM "HistFactorySchema.dtd">

<Combination OutputFilePrefix="./results/example">

  <Input>./config/example_channel.xml</Input>

  <Measurement Name="GaussExample" Lumi="1." LumiRelErr="0.1" BinLow="0" BinHigh="2" Mode="comb"
    <POI>SigXsecOverSM</POI>
    <ParamSetting Const="True">Lumi alpha_syst1</ParamSetting>
    <!-- don't need <ConstraintTerm> default is Gaussian-->
  </Measurement>
</Combination>
```

and at `config/example_channel.xml` :

```
<!DOCTYPE Channel  SYSTEM 'HistFactorySchema.dtd'>

<Channel Name="channel1" InputFile="./data/counting_exp_data.root" HistoName="" >
  <Data HistoName="data" HistoPath="" />
  <Sample Name="signal" HistoPath="" HistoName="signal">
    <OverallSys Name="syst1" High="1.2" Low="0.8"/>
    <NormFactor Name="SigXsecOverSM" Val="1" Low="0." High="30." Const="True" />
  </Sample>
  <Sample Name="background" HistoPath="" NormalizeByTheory="True" HistoName="background">
    <OverallSys Name="syst2" Low="0.9" High="1.1"/>
  </Sample>
</Channel>
```

Note the following in the second xml configuration file:

- This second file defines our 'channel', labelled "channel1", which in this case is our (single-bin) counting experiment.
- The root file with input histograms is defined, as is the name of the data histogram.
- There is a background and signal component, labelled "background" and "signal" respectively.
- The relative signal and background uncertainties are 20 and 10 percent respectively. Asymmetric uncertainties can be given. The two nuisance parameters describing these uncertainties are labelled `alpha_syst1` and `alpha_syst2`, where the prefix "alpha_" is automatically added.
- The signal component is scaled with the signal strength parameter `SigXsecOverSM`.
- For the background `NormalizeByTheory` is set to `"True"`, meaning that the background estimate is a MC-based estimate, and a luminosity uncertainty will be added.

In the top-level xml configuration file, note that:

- The luminosity uncertainty is set to 10 percent. The corresponding nuisance parameter is this uncertainty is called `Lumi`.
- The signal and luminosity uncertainties are actually turned off, by setting their nuisance parameters to constant in the fit.
- The "Parameter Of Interest" (POI) is set to be the signal strength parameter `SigXsecOverSM`. This classification is important for doing limit setting using the RooStats tools later.

The extra purpsose of this xml file is to combine the underlying channels into a combined measurement. Since only one underlying xml file is provided as input, in this example the combination also consists of only one (and the same) channel.

There is some extra xml code in the top-level xml configuration file that's been commented out. For example:

```
  <Measurement Name="GammaExample" Lumi="1." LumiRelErr="0.1" BinLow="0" BinHigh="2" Mode="comb"
    <POI>SigXsecOverSM</POI>
    <ParamSetting Const="True">Lumi alpha_syst1</ParamSetting>
    <ConstraintTerm Type="Gamma" RelativeUncertainty=".3">syst2</ConstraintTerm>
  </Measurement>
```

and

```
  <Measurement Name="LogNormExample" Lumi="1." LumiRelErr="0.1" BinLow="0" BinHigh="2" Mode="comb
    <POI>SigXsecOverSM</POI>
    <ParamSetting Const="True">Lumi alpha_syst1</ParamSetting>
    <ConstraintTerm Type="LogNormal" RelativeUncertainty=".3">syst2</ConstraintTerm>
  </Measurement>
```

Turning on these two bits of xml code changes the Gaussian constraint (the default) on the background (syst2), into either a Gamma or LogNormal constraints, both with a relative uncertainty of 0.3.

Now run:

```
root -b -q makeCountingExperiment.C
```

This produces a lot of output on the screen.

In this process, two workspaces containing pdfs and datasets are made: one being the individual counting experiment, which we had labelled channel1, and a combined workspace of all the input channels. In this example, the 'combined' workspace consists solely of channel1.

For both workspaces, the pdf is fit to the dataset. One can see the fit output on the screen. Note that the two floating fit parameters are the signal strength parameter and `alpha_syst2`. (This fit option can be turned off with: ExportOnly="True", in config/example.xml) We'll explore and reproduce this fit in more detail later.

When done, quickly open:

```
root -l data/counting_exp_data.root
```

to confirm that the data, signal, and background histograms have been created. Close root.

Now do:

```
ls -l results/
```

You should see something like:

```
-rw-r--r-- 1 mbaak zp  7398 Apr 24 23:58 example_GaussExample.root
-rw-r--r-- 1 mbaak zp 19230 Apr 24 23:58 example_channel1_GaussExample_model.root
-rw-r--r-- 1 mbaak zp 10980 Apr 24 23:58 example_channel1_GaussExample_profileLR.eps
-rw-r--r-- 1 mbaak zp 20479 Apr 24 23:58 example_combined_GaussExample_model.root
-rw-r--r-- 1 mbaak zp 10980 Apr 24 23:58 example_combined_GaussExample_profileLR.eps
-rw-r--r-- 1 mbaak zp  1090 Apr 24 23:58 example_results.table
```

The two root files, "combined" and "channel1", contain the two produced workspaces.

Let's plot:

```
display results/example_channel1_GaussExample_profileLR.eps
```

which shows the the likelihood and profile likelihood curves as a function of the signal strength. (Question: which curve is which?) Note that the likelihood minimum is found at 5 signal events, as expected.

Exercises:

- Make the luminosity and `alpha_syst1` floating fit parameters, and check how different the corresponding likelihood curves look.
- Make two workspaces where the constraint on the `syst2` uncertainty is either a Gamma or Lognormal function (keep `Lumi` and `alpha_syst1` fixed), and check how different the corresponding likelihood curves look compared with the Gamma constraint.
- Create a second channel xml file, labelled "channel2", with different background and signal uncertainties, and combine this with the first channel xml file in the top level xml file, by adding a second Input line. See how the combined likelihood curve is now thinner than those of the two individual channels.

We can now use the created workspaces for fitting by hand, doing hypothesis tests, and setting upper limits. We will do so in the following sections.

The HistFitter equivalent

Show Hide
Constructing the workspace of a simple counting-experiment in one bin directly with HistFitter is straight-forward, and is as simple as running

```
cd $HISTFITTER/
HistFitter.py -x -w analysis/tutorial/MyUserAnalysis.py
```

Before running this, take a look at `analysis/tutorial/MyUserAnalysis.py`.

- At the top of this configuration file, one can see how the observed and expected number of events in counting experiment are set, including the errors on the signal and background components. In this example, there are uncorrelated errors on the background and signal components, and also one correlated uncertainty. There is also a total MC statistics uncertainty on the sum of the signal and background.
- A bit below, the total background, signal, and data sample are defined. Note how the systematic uncertainties are added as objects to the background and signal samples. The signal estimate comes from a MC estimate ("normalized by theory") and, as such, a luminosity uncertainty is included as well.
- Further below still, the samples are added to the analysis object, and the signal strength is defined as the parameter of interest. Then the "channel" is defined, consisting of a "cut" analysis of 1 bin, over the observable range from 0.5 to 1.5.

Now run the command.

What can be seen in the output is that:

- After some initialization, the input histograms are created and stored under `data/MyUserAnalysis.root`.
- Then the xml configuration files of the channel are constructed on the fly, and are stored in the directory `config/`. These files follow almost the same structure as in the example above.
- Finally, histfactory command is executed: `hist2workspace config/MyUserAnalysis/SPlusB.xml`, and the resulting workspace is stored under `results/MyUserAnalysis/SPlusB_combined_NormalMeasurement_model.root`.

The automated fitting of the workspace and plotting of the likelihood curve has been turned off in the xml file construction.

Fitting and plotting a workspace in ROOT (optional)

This example shows how to open a RooFit workspace in ROOT, how to perform basic operations to the pdf and datasets it contains.

Show Hide
This example shows how to open a RooFit workspace in ROOT, how to perform basic operations to the pdf and datasets it contains. The input workspace used is the one created in the simple-counting experiment example (above), but it can be any workspace as created with HistFitter.

```
cd $HISTFITTER/macros/Examples/fittest/
```

and run:

```
root -l fittest.C
```

Study the output on the screen, quit ROOT, and then take another look at the macro `fittest.C`.

- The top of the macro simply opens the input root file, lists its contents, and retrieves the workspace "channel1".
- Then the contents of the workspace are printed. When executing the macro, this looks as follows:

```
RooWorkspace(channel1) channel1 workspace contents

variables
---------
(Lumi,SigXsecOverSM,alpha_syst1,alpha_syst2,binWidth_obs_x_channel1_0,binWidth_obs_x_channel1_1,n

p.d.f.s
-------
RooGaussian::alpha_syst1Constraint[ x=alpha_syst1 mean=nom_alpha_syst1 sigma=1 ] = 1
RooGaussian::alpha_syst2Constraint[ x=alpha_syst2 mean=nom_alpha_syst2 sigma=1 ] = 1
RooRealSumPdf::channel1_model[ binWidth_obs_x_channel1_0 * L_x_signal_channel1_overallSyst_x_Exp
RooGaussian::lumiConstraint[ x=Lumi mean=nominalLumi sigma=0.1 ] = 1
RooProdPdf::model_channel1[ lumiConstraint * alpha_syst1Constraint * alpha_syst2Constraint * chan

functions
--------
RooProduct::L_x_background_channel1_overallSyst_x_Exp[ Lumi * background_channel1_overallSyst_x_E
RooProduct::L_x_signal_channel1_overallSyst_x_Exp[ Lumi * signal_channel1_overallSyst_x_Exp ] = 1
RooStats::HistFactory::FlexibleInterpVar::background_channel1_epsilon[ paramList=(alpha_syst2) ]
RooHistFunc::background_channel1_nominal[ depList=(obs_x_channel1) ] = 5
RooProduct::background_channel1_overallSyst_x_Exp[ background_channel1_nominal * background_chann
RooStats::HistFactory::FlexibleInterpVar::signal_channel1_epsilon[ paramList=(alpha_syst1) ] = 1
RooHistFunc::signal_channel1_nominal[ depList=(obs_x_channel1) ] = 1
RooProduct::signal_channel1_overallNorm_x_sigma_epsilon[ SigXsecOverSM * signal_channel1_epsilon
RooProduct::signal_channel1_overallSyst_x_Exp[ signal_channel1_nominal * signal_channel1_overallN

datasets
--------
RooDataSet::asimovData(obs_x_channel1)
RooDataSet::obsData(obs_x_channel1)

named sets
----------
ModelConfig_GlobalObservables:(nom_alpha_syst2)
ModelConfig_NuisParams:(alpha_syst2)
ModelConfig_Observables:(obs_x_channel1)
```

```
ModelConfig_POI:(SigXsecOverSM)
coefList:(binWidth_obs_x_channel1_0,binWidth_obs_x_channel1_1)
constraintTerms:(lumiConstraint,alpha_syst1Constraint,alpha_syst2Constraint)
globalObservables:(nom_alpha_syst2)
likelihoodTerms:(channel1_model)
obsAndWeight:(weightVar,obs_x_channel1)
observablesSet:(obs_x_channel1)
shapeList:(L_x_signal_channel1_overallSyst_x_Exp,L_x_background_channel1_overallSyst_x_Exp)

generic objects
---------------
RooStats::ModelConfig::ModelConfig
```

For example, one can see the fit variables we've already encountered, `Lumi, SigXsecOverSM, alpha_syst1, alpha_syst2`, but also other nominal-value variables that are are constant in the fit, for example `nominalLumi`, which is set to 1 (i.e. 100 percent). One interesting variable is `obs_x_channel1`, which is the "observable" that describes our counting-bin of interest. For example, this could be the effective mass when studying a cut-and-count experiment in the effective mass distribution.

Also, one can see the complete pdf, `model_channel1`, which is a product of a Poisson function `channel1_model` and Gaussian constraint terms on the luminosity and background and signal efficiencies, `lumiConstraint, alpha_syst1Constraint,` and `alpha_syst2Constraint` respectively.

The functions are used to build the pdfs, typically to propagate the impact of the uncertainties into the background and signal estimates.

Then there are two datasets: one with the observed number of events `obsData`, and the other with the nominal expected number of events `asimovData`, as taken from the nominal background and signal estimates. Note that the Asimov dataset can have a non-integer number of events.

The named sets and generic objects are used as info for the RooStats limit setting machinery. For example, one can again see the POI (parameter of interest.)

- Then comes a useful trick. The lines:

```
w->exportToCint();
using namespace channel1;
```

make all object in the workspace `channel1` directly accessible in CINT.

- For example, one can now easily access variables, or use the pdf to fit datasets:

```
model_channel1.fitTo(asimovData);

// make these floating fit parameters
alpha_syst1.setConstant(false);
Lumi.setConstant(false);

// then fit the observed dataset
RooFitResult* result = model_channel1.fitTo(obsData,Save());
```

Question: do the fitted signal strengths agree with what one naively would expect?

Note that the fit result can be retrieved as an object from the fit and can be studied in more detail later. Or be stored, for example.

- The bottom of the macro shows how one can make a simple plot of a dataset, with the pdf on top of it.

Possible exercises:

- How does the error on the fitted signal strength change when setting the nuisance parameters `Lumi, alpha_syst1, alpha_syst2` to constant (i.e. no uncertainties) or when floating them in the fit to data?
- Copy-paste the top of the macro, upto `CUT`, into an interactive session of ROOT, and take a look (i.e. Print()) at some of the objects in the workspace, for example `SigXsecOverSM, asimovData, channel1_model`.

The HistFitter equivalent

Show Hide
The HistFitter equivalent of fitting and plotting a workspace has been preprogrammed, and by default produces some very nice plots before and after the fit.

Continuing with the simple cut-and-count workspace we have created in the previous "HistFitter equivalent" section, a fit of this workspace is simply done as:

```
cd $HISTFITTER/
HistFitter.py -f -D "before,after" analysis/tutorial/MyUserAnalysis.py
```

The -f is for fitting, the -d is for plots, the -i makes the python job interactive. By default, that observed dataset is fit. Run with the option -a to fit the Asimov dataset instead. Quit python by pressing Ctrl-D.

In the output of this command, after the usual initialization, what can be seen is the following:

- First, the errors on the fit parameters are reset to their default values. This is useful for plotting the size of errors on the pdf in the distribution of the observable **before the fit**.
- Then, the dataset and pdf are plotted in the (defined) observable, before the fit.
- The dataset is then fit. One can see the Minuit fit output on the screen.
- Then, the data and pdf are plot **after the fit**.

The before- and after-fit plots are stored in the `results/` directly. Their names start with `can_`, and have `beforeFit` and `afterFit` in their names. Take a look a the plots. Don't they look great? What can be seen is the data distribution, that signal and background components, and (normally) the total uncertainty on the signal plus background prediction.

Performing an hypothesis test in ROOT (*)

This example shows how to do a hypothesis test on a workspace using the RooStats statistics machinery, and to obtain the p-value of such a test. Hypothesis tests are typically done for each grid point in a SUSY 2D-parameter phase-space grid to draw an exclusion contour line.

The input workspace used is one created with the simple-counting experiment example (above), but can be any workspace created by the HistFactory machinery. The number of observed and expected background and signal events are set to 7, 5, and 5 respectively.

```
cd $HISTFITTER/macros/Examples/p-values/
```

There's the macro: `pvalue.C`. Let's take a look at it first:

```
void
pvalue()
{
  int seed=1;             // 0 = cpu clock, so random
  const char* fileprefix = "example";
  int  calculatorType=0; // 2=asymptotic approximation limit. 0=frequentist limit
  int  testStatType=3;   // one-sided test profile statistic (ATLAS standard)
  int  ntoys=5000;
  bool doUL = true;      // true = exclusion, false = discovery

  // open the workspace
  gSystem->Load("libSusyFitter.so");
  TFile *file = TFile::Open("example_channel1_GaussExample_model.root");
  RooWorkspace* w = (RooWorkspace *)file->Get("channel1");

  // set random seed for toy generation
  RooRandom::randomGenerator()->SetSeed(seed);

  // do hypothesis test and get p-value
  LimitResult result = RooStats::get_Pvalue( w, doUL, ntoys, calculatorType, testStatType );
  result.Summary();

  file->Close();
}
```

After opening the workspace, doing the hypothesis test and getting the p-value is done with only one line:

```
  LimitResult result = RooStats::get_Pvalue( w, doUL, ntoys, calculatorType, testStatType );
```

There are only a couple of important settings one can change:

- `doUL`, set to true or false, for either the exclusion or discovery hypothesis test.

- calculatortype = 0 Freq calculator (= limit using toy experiments, using a fully Frequentist approach)
- type = 1 Hybrid calculator (= limit using toy experiments, using a Bayesian-Frequentist hybrid approach)
- type = 2 Asymptotic calculator (= limit using asymptotic formula)
- type = 3 Asymptotic calculator using nominal Asimov data sets (not using fitted parameter values but nominal ones)
- **The calculators typically used in ATLAS are 0, or 2.**

- When choosing the Frequentist or Hybrid calculator, one needs to specify the number of toy experiments.

- testStatType = 0 LEP
- = 1 Tevatron
- = 2 Profile Likelihood
- = 3 Profile Likelihood one sided (i.e. = 0 if mu < mu_hat)
- **The one-sided profile likelihood test statistic is a default used at the LHC.**

- The random seed is set to 1 here, such that results are reproducable. (0 is the CPU clock).

Now let's run the script:

```
root -b -q pvalue.C
```

This will take approximately a minute to run.

In the output one can see that there are 5000 toys generated for the null hypothesis, and 2500 toys for the alternate hypothesis. For exclusion, these are the signal model hypothesis (where the signal strength parameter has been set to 1) and background-only hypothesis (with the signal strength set to zero) respectively. Realize that this is one place where the earlier-defined "parameter of interest" is used by the RooStats machinery.

The default setting is signal model "exclusion", so the printed p-values of interest are the CLs ones. This results in:

```
 | CLs:                 0.32334      sigma: 0.458378
 | CLsexp:              0.179896      sigma: 0.91576
 | CLsu1S (+1s)         0.404644      sigma: 0.241345
 | CLsd1S (-1s)         0.0710259      sigma: 1.46819
 | CLsu2S (+2s)         0.997391      sigma: -2.79321
 | CLsd2S (-2s)         0      sigma: 7.4
```

Here, `CLs` is the CLs p-value corresponding to the observed number of events. `CLsexp` corresponds to the expected p-value of the background-only hypothesis. The four other CLs values are also expectation p-value, varying one or two sigma around `CLsexp`.

Exercises:

- Now run the script again but switch to the asymptotic calculator? How well do the CLs number agree with the outcome of the toy experiments?

Answer:

```
 | CLs:                 0.302309      sigma: 0.517772
 | CLsexp:              0.161811      sigma: 0.987044
 | CLsu1S (+1s)         0.409991      sigma: 0.227569
 | CLsd1S (-1s)         0.051809      sigma: 1.62756
 | CLsu2S (+2s)         0.74298      sigma: -0.652561
 | CLsd2S (-2s)         0.0148638      sigma: 2.1737
```

- Do the same exercise, but now switch to the **discovery** hypothesis test. Note that here the p-value of interest is called: `p0`, or the `Null p-value`. How well do the Frequentist calculator and asymptotic approximations agree here?

The HistFitter equivalent

We continue with the workspace created from the `analysis/tutorial/MyUserAnalysis.py` HistFitter configuration file. To be sure, re-run:
`HistFitter.py -w -f -D "before,after" -i analysis/tutorial/MyUserAnalysis.py`

Doing an hypothesis test on a set of workspaces previously created by HistFitter is as simple as doing:

```
cd $HISTFITTER/
HistFitter.py -p analysis/tutorial/MyUserAnalysis.py
```

By default, all signal models in the configuration file are processed one-by-one. The command line option `-s` sets the random seed for toy generation (the default is CPU clock: 0).

The hypothesis test configuration can by found at the top of the configuation file:

```
# Setting the parameters of the hypothesis test
configMgr.doExclusion=True # True=exclusion, False=discovery
#configMgr.nTOYs=5000       # number of toys when doing frequentist calculator
configMgr.calculatorType=2 # 2=asymptotic calculator, 0=frequentist calculator
configMgr.testStatType=3   # 3=one-sided profile likelihood test statistic (LHC default)
configMgr.nPoints=20       # number of values scanned of signal-strength for upper-limit determin
```

In this example, the RooStats asymptotic calculator is used.

When running the command, one sees the same type of output as in the section above.

The hypothesis test results of all signal models processed (only one in our example) are stored under `results/MyUserAnalysis_Output_hypotest.root` for use later. When using the asymptotic calculator, however, no plot of the generated test statistics is stored.

As an exercise one can again compare running with toys and with asymptotic calculator.

Setting an upper limit (*)

This example shows how to set an exclusion upper limit on a signal model contained in a workspace, using the RooStats statistics machinery. An exercise demonstrates how to set an upper limit on the number of possible signal events in a simple cut-and-count experiment.

Note that setting an upper limit simply means: performing a set of exclusion hypothesis tests with varying values of the parameter of interest -- being the signal strength in our case -- and then interpolating for which parameter of interest value we have 95% exclusion. For this reason, setting an upper limit is also called **hypotest inversion**.

The input workspace used is one created with the simple-counting experiment example (above), but can be any workspace created by the HistFactory machinery. The number of observed and expected background and signal events are set to 7, 5, and 5 respectively. All systematic uncertainties have been turned on, including the cross-section uncertainty on the signal expectation. Remember that there's a 10% uncertainty on the background efficiency, a 20% uncertainty on the signal expectation, and a 10% luminosity uncertainty.

```
cd $HISTFITTER/macros/Examples/upperlimit/
```

There's only one (simple) macro: upperlimit.C. Take a look at it first.

The macro looks very much like the p-value macro in the previous section. After opening the workspace, determining the upper limit on the signal model is done with only one line:

```
  // determine the upper limit and make a plot
  RooStats::HypoTestInverterResult* hypo = RooStats::MakeUpperLimitPlot(fileprefix,w,calculatorTy
```

There are two more options:

- The same options apply as in the p-value macro, with one additional option: `useCLs`, which is either true or false. This option simply means that the CLs value is used to determine the 95\% upper limit, instead of CLs+b. As CLs is the LHC default, we will leave this to true.
- Another option specifies the number of hypothesis tests done to determine the upper limit (`npoints`, which is the number of values at which `mu_SIG` is set). We will leave this to 20.

Now let's run the script:

```
root -b -q upperlimit.C
```

This will only take a few seconds.

The output shows that several fits are being performed by the asymptotic calculator. It ends with:

```
<INFO> HypoTestTool: The computed upper limit is: 1.55912 +/- 0
<INFO> HypoTestTool:  expected limit (median) 1.32589
<INFO> HypoTestTool:  expected limit (-1 sig) 0.856831
<INFO> HypoTestTool:  expected limit (+1 sig) 2.15693
<INFO> HypoTestTool:  expected limit (-2 sig) 0.599806
```

```
<INFO> HypoTestTool:  expected limit (+2 sig) 3.53856
```

Meaning that the upper limit on the signal model, at 95% confidence level, is at a signal strength of 1.56. (Remember that the signal expectation is 5 events.) This upper limit includes the uncertainties on the predicted signal expectation. The expected upper limit for a background-only result is also given, at 1.33 times the signal strength, including the 1 and 2 sigma variations on this number.

There are two output files produced.

Take a look at the post script file `upperlimit_cls_poi_example_Asym_CLs_grid_ts3.root.eps`. It shows the CLs, CLb, and CLs+b p-values as a function of different signal strength values. Also shown, in green and yellow bands, are the 1 and 2 sigma variations around the background-only expected CLs values. The 95% upper limit on the signal model is where the CLs curve crosses the horizontal 5% line in red. Note that there are 20 evaluations along the signal strength axis. The axis range is automagically determined.

There is also a root file stored called `example_Asym_CLs_grid_ts3`. It contains a RooStats summary object of the hypothesis test inversion. Open the file in ROOT. One can now reproduce the upper limit plot by doing:

```
TFile *_file0 = TFile::Open("example_Asym_CLs_grid_ts3.root");
.ls;
gSystem->Load("libSusyFitter.so");
RooStats::AnalyzeHypoTestInverterResult(result_SigXsecOverSM, 2, 3, true, 20);
```

Exercise:

- To determine the signal model independent upper limit on the number of signal events in our cut-and-count example, one can create a dummy signal model, with no signal uncertainties and a signal expectation of 1 event. That way any upper limit determined on the signal strength parameter is exactly the upper limit on a possible number of signal events in the bin. A quick alternative here is to simply turn off signal and luminosity uncertainties in the workspace before determining the upper limit. Do this (hint: it is one line of code that you need to change in `upperlimit.C`). What is the exclusion upper limit on the (model-independent) number of signal events? How many events better is this than the number obtained including the uncertainties? (Answer: 0.19 events.)

The HistFitter equivalent

We continue again with the workspace created earlier from the `analysis/tutorial/MyUserAnalysis.py` HistFitter configuration file.

Setting upper limits on a set of workspaces previously created by HistFitter is as simple as doing:

```
cd $HISTFITTER/
HistFitter.py -l analysis/tutorial/MyUserAnalysis.py
```

By default, all signal models in the configuration file are processed one-by-one. The command line option: -s sets the random seed for toy generation (the default is CPU clock: 0).

Again, the hypothesis test configuration can by found at the top of the configuation file:

```
# Setting the parameters of the hypothesis test
configMgr.doExclusion=True # True=exclusion, False=discovery
#configMgr.nTOYs=5000      # number of toys when doing frequentist calculator
configMgr.calculatorType=2 # 2=asymptotic calculator, 0=frequentist calculator
configMgr.testStatType=3   # 3=one-sided profile likelihood test statistic (LHC default)
configMgr.nPoints=20       # number of values scanned of signal-strength for upper-limit determin
```

In this example, the RooStats asymptotic calculator is used.

When running the command, one sees the same type of output as in the section above. The observed CLs upper limit is found to be 2.2496 times the signal strength.

Again, a nice plot of the upper limit set is stored under `results/upperlimit_cls_poi_Sig_Asym_CLs_grid_ts3.root.eps`. The hypothesis inversion result is stored under `results/MyUserAnalysis_Output_upperlimit.root` for use later.

Model-independent upper limits table (*)

Here we discuss a general script, `UpperLimitTable.py` to produce the model-independent upper limits in a LaTex table. After setting up HistFitter, the `UpperLimitTable.py` can be run similarly to the yields table and systematic table scripts above. Also this script will present its options:

```
Usage:
UpperLimitTable.py [-c channels] [-w workspace] [-l lumi] [-n nTOYS] [-a asymptotic/Asimov] [-o o
Minimal set of inputs [-c channels] [-w workspace] [-l lumi]
UpperLimitTable.py needs the workspace file _before_ the fit, so not XXX_afterFit.root
Every channel (=SR) needs to have its own workspace file, with the same naming scheme only replac

*** Options are:
-c <channels>: single channel string (=SR) or comma separated list accepted (OBLIGATORY)
-w <workspaceFileName>: single name accepted only (OBLIGATORY) ;   if multiple channels given in
-l <lumi>: same unit as used for creating the workspace by HistFitter (OBLIGATORY)
-n <nTOYS>: sets number of TOYs (default = 3000)
-a : use asimov dataset, ie asymptotic calculation insted of toys (default is toys)
-p <poiNames>: single POI name string (mu_<SRname>) or comma separated list accepted, only needed
-o <outputFileName>: sets the output table file name
-i stays in interactive session after executing the script (default off)

For example:
UpperLimitTable.py -c SR4jTEl -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_leptons/res
UpperLimitTable.py -c SR4jTEl,SR4jTMu -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_lep
UpperLimitTable.py -c SR4jTEl,SR4jTMu -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_lep
UpperLimitTable.py -c SR4jTEl -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_leptons/res
UpperLimitTable.py -c SR4jTEl -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_leptons/res
UpperLimitTable.py -c SR4jTEl -w /afs/cern.ch/user/k/koutsman/HistFitterUser/MET_jets_leptons/res
```

The minimal set of inputs consists of a workspace name, the luminosity in `fb` and the region/channel.

The workspace created for the upper limit calculation must contain only one signal region and no(!) validation regions. Hence, if you have multiple signal regions, you must create a separate workspace for each signal region. The upper limit can only be calculated in a single bin signal region (cut-and-count) case, so not possible for a shape-fit in the signal region (no problem if the control region is a shape-fit). If the analysis is not a single bin, then this doesn't really work because the relative contribution to the different bins will depend on the model in question.

⚠ **Very important** is hence that you *DO NOT INPUT A SIGNAL MODEL* (as the limit would not be model-_IN_dependent then). Instead, you should input a 'dummy' signal model that is a single bin histogram with the bin value at 1. This is exactly what the discovery fit in the configuration file in this section does.

An example config file is called `MyUpperLimitAnalysis_SS.py`, where SS points to the fact that it is only the `SS` signal region that we are interested in. This config file is based on the previously used `MyUserAnalysis.py`, do a `diff` between the two to see the difference:

```
diff -u analysis/tutorial/MyUserAnalysis.py analysis/tutorial/MyUpperLimitAnalysis_SS.py
```

As you see, we only kept one systematic which defines the total systematic uncertainty on the background estimate in the signal region. For your own analysis you can get this number by running the `YieldsTable.py` script, as described above. We also turned off any uncertainty on the signal model, both systematic and coming from MC statistics, as it is just a 'dummy' model with `nsig = 1.`. The background estimation is now 5 events, while we observe 7, so we see an excess.

Now run it (without -t):

```
HistFitter.py -w -f -D "before,after" -i analysis/tutorial/MyUpperLimitAnalysis_SS.py
```

As you see, the fit finds back the excess `mu_SS   1.9985e+00 +/-  2.58e+00`. That is why we set the dummy model to be equal to 1, so the fitted signal strength is equivalent to the excess in events. Now run the upper limit script with 1000 toys on the combined workspace before the fit (hence without the `_afterFit.root`):

```
UpperLimitTable.py -c SS -w results/MyUpperLimitAnalysis_SS/SPlusB_combined_NormalMeasurement_mod
```

The upper limit calculation takes some time, especially if you are working with toys, so be patient. The calculation is similar to the calculation being performed if running `-l`, but a bit simplified. The result should come out as:

```
 - Null p-value = 0.208 +/- 0.012835
 - Significance = 0.81338 +/- 0.0447864 sigma
 - Number of Alt toys: 500
 - Number of Null toys: 1000
 - Test statistic evaluated on data: 0.293789
 - CL_b: 0.208 +/- 0.012835
 - CL_s+b: 0.35 +/- 0.0213307
 - CL_s: 1.68269 +/- 0.145939
```

and is also saved in the `LaTeX` table called `UpperLimitTable_SS_nToys1000.tex`. As well, the hypotest result is saved in the root file `htiResult_poi_mu_SS_ntoys_1000_calctype_0_npoints_20.root`.

Part 5: Making a complete exclusion contour plot (*)

When you have a large class of models, typically the result is visualized in a 2D plot with a contour line. These lines are made in a conceptually simple way: run repeated hypothesis test in a plane, and interpolate to get that line where the CLs value is equal to 0.05. We'll create a simple contour plot ourselves by running over a list of signal points in mSUGRA and visualizing the result.

There are several steps to making a final contour plot:

1. run repeated hypothesis tests
2. merge all the output root files into one (if stored in a seperate file)
3. transform this set of hypothesis tests into a plain-text file
4. create a TH2D from the ascii data in this list file
5. plot the TH2D to draw a contour line at the requested CLs level

The counting experiment is based on the HistFitter configuration file:

```
analysis/tutorial/MySimpleChannelConfig.py
```

It described a simple counting experiment in one bin, with:

- a JES uncertainty on the total background estimate and the signal estimate,
- a cross-section uncertainty on the signal estimate, and
- a luminosity uncertainty on the signal estimate.

The signal estimates are given for 173 grid points in an msugra plane.

The histograms for the background, signal, and error estimates are given in: `data/MySimpleChannelAnalysis2.root`.

Running the hypothesis tests

⚠ **Warning**, running the following two HistFitter commands can take a while and requires some disk space. To skip this, start below at the plotting step.

We have already run the `-t` step for you, so you only need to construct the workspaces, but first copy the histograms file to the data/ directory (as expected by HistFitter):

```
cp samples/tutorial/MySimpleChannelAnalysis.root data/
```

To make the workspaces (containing the pdf and dataset) for each grid point, simply do:

```
HistFitter.py -w analysis/tutorial/MySimpleChannelConfig.py
```

To make an exclusion contour plot, an hypothesis test need to be run over each grid point. The setting for the hypothesis test are set at the top of `analysis/tutorial/MySimpleChannelConfig.py`.

```
## setting the parameters of the hypothesis test
#configMgr.nTOYs=5000
configMgr.calculatorType=2 # 2=asymptotic calculator, 0=frequentist calculator
configMgr.testStatType=3   # 3=one-sided profile likelihood test statistic (LHC default)
configMgr.nPoints=20       # number of values scanned of signal-strength for upper-limit determin
```

To save time, in this case, the asymptotic calculator is used together with the one-sided profile likelihood test statistic. (The recommendation for most SUSY analyses is to use the asymptotic calculator, but to validate it with toy experiments for several points.)

Run the hypothesis tests by doing (-p):

```
cd $HISTFITTER
HistFitter.py -p analysis/tutorial/MySimpleChannelConfig.py
```

This produces root files with hypothesis test results for each grid point:

```
results/MySimpleChannelAnalysis_fixSigXSecDown_hypotest.root
results/MySimpleChannelAnalysis_fixSigXSecNominal_hypotest.root
results/MySimpleChannelAnalysis_fixSigXSecUp_hypotest.root
```

where as the names suggest, for each grid point the nominal signal x-section and x-section=+/-1 sigma hypo tests are performed, to be able to draw the uncertainty band.

Tip: in a real analysis, you can split the running of 173 points over multiple calls to HistFitter! You could then submit these to e.g. a cluster/batch-system to compute the p-values. In that scenario, you will have to use ROOT's `hadd` to merge all the output hypothesis tests (`*hypotest.root`) files into one.

Creating a list file

We should now have ROOT files with a set of results. This step assumes that the nominal file is `results/MySimpleChannelAnalysis_fixSigXSecNominal_hypotest.root` To convert the contents of this file with hypo test results into a readable TTree, do:

```
cd $HISTFITTER/macros/Examples/contourplot
root -b -q makelistfiles.C
```

where the `makelistfiles.C` points to the root file with hypothesis test results. Take a look at this macro. Note that hypothesis test objects picked up follow the naming convention:

```
const char* format     = "hypo_SU_%f_%f_0_10";
// interpret %f's above respectively as two variables of interest (separated by ':')
const char* interpretation = "m0:m12";
```

where the two `%f` strings are interpreted as m0 and m12 respectively (two mass scales of the underlying signal model).

This command produces a text file with p-values and mass scales (amongst others) for all grid points as a list:

```
MySimpleChannelAnalysis_fixSigXSecNominal_hypotest__1_harvest_list
```

Now we need to repeat this step for the two files `results/MySimpleChannelAnalysis_fixSigXSecDown_hypotest.root` and `results/MySimpleChannelAnalysis_fixSigXSecUp_hypotest.root`. For this open `makelistfiles.C` in a text editor and modify the line

```
const char* inputfile  = "$HISTFITTER/results/MySimpleChannelAnalysis_fixSigXSecNominal_hypotest.
```

to point to one of the other files. Repeat running

```
root -b -q makelistfiles.C
```

Repeat the two steps again for the remaining file.

You should now have three files:

```
MySimpleChannelAnalysis_fixSigXSecNominal_hypotest__1_harvest_list
MySimpleChannelAnalysis_fixSigXSecUp_hypotest__1_harvest_list
MySimpleChannelAnalysis_fixSigXSecDown_hypotest__1_harvest_list
```

Investigating the list file

There's a nice feature that this list can always be read directly in as a TTree in ROOT by doing :

```
root -l summary_harvest_tree_description.h
```

which gives you a TTree named "tree" with the results. The file gets written out by HistFitter automatically, and defines a colon-seperated string to read in an ascii file into a TH2D. (ROOT has an example macro that looks very similar to this, in case you're unfamiliar with reading in data in this way.)

You could now plot the CLs value in this way:

```
tree->Print();
tree->Draw("CLs:m12:m0");
```

Take a look at some of the other observables in this tree. Next, we will move on to plot the CLs in a more customary way: a line in a plane.

Creating contour histograms

Contour histograms based on the stored p-values are based on these list files. Run:

```
root -b -q makecontourhists.C
```

The p-values are converted into one-sided significances, from which the interpolated histograms are constructed. Take a look at this script. The underlying script used here is to make the histograms is: `contourmacros/m0_vs_m12_nofloat.C` In this examle, the histograms are stored under `MySimpleChannelAnalysis2_fixSigXSecNominal_hypotest__1_harvest_list.root`.

Open this file in ROOT, and plot, for example, the significance of the observed CLs values:

```
root -l
TFile *_file0 = TFile::Open("MySimpleChannelAnalysis_fixSigXSecNominal_hypotest__1_harvest_list.r
.ls
sigp1clsf->Draw("colz") ;
```

In same file, several other histograms are defined, corresponding to e.g. the +2 sigma (contains "p2") or -1 sigma (contains "m1") bands. These get used when plotting making a contour plot in its full glory with all uncertainty bands.

We need to repeat `root -b -q makecontourhists.C` for the other two list files as well. For this open `makecontourhists.C` in a text editor and modify the line

```
 const char* ehistfile = m0_vs_m12_nofloat("MySimpleChannelAnalysis_fixSigXSecNominal_hypotest__1
```

to point to one of the other files, `MySimpleChannelAnalysis_fixSigXSecUp_hypotest__1_harvest_list` or `MySimpleChannelAnalysis_fixSigXSecDown_hypotest__1_harvest_list`. Rerun `root`

`-b -q makecontourhists.C`. Repeat the last two steps for the remaining file. We have now three file, containing histograms for the nominal signal cross sections, and for the down or up variation of the theoretical uncertainties on the signal by 1 sigma:

```
MySimpleChannelAnalysis_fixSigXSecNominal_hypotest__1_harvest_list.root
MySimpleChannelAnalysis_fixSigXSecDown_hypotest__1_harvest_list.root
MySimpleChannelAnalysis_fixSigXSecUp_hypotest__1_harvest_list.root
```

Creating contour plots

To make final plots, as based on these histograms, we first need to copy some cosmetics files:

```
cp $HISTFITTER/samples/tutorial/contour/* contourmacros/
```

Then run:

```
root -b -q makecontourplots.C
```

Take a look at this macro. You will find there are a few options you can set, for example, the luminosity or text in the plot. The underlying script to make the plots is:
`contourmacros/SUSY_m0_vs_m12_all_withBand_cls.C`.
The contour lines are set at an exclusion value of 95% CL. Clearly, a dedicated analysis will probably have to modify this macro to its own liking.

The final plots are stored under `plots/`. For example, take a look at:
`plots/atlascls_m0m12_wband1_showcms0_MySimpleChannelAnalysis2_fixSigXSecNominal_`

One can see the observed and expected CLs contour lines, which happen to be identical in this example, and the 1 sigma variations around the expected CLs contour lines.

Automating the procedure

In a real-world analysis, you will like create a wrapper script that will execute all these steps for you in succession for each grid that want to create a contour plot for. You would then just execute this wrapper script to save yourself a lot of hassle.

Analysis Check List

A useful checklist for problems you can run into and things to take into consideration when doing an analysis can be found at HistFitterChecklist.

Useful links and Extra documentation

- HistFitter e-group
- HistFitter Tutorial
- Introductory HistFitter slides from Dan Short, April 2012
- In-depth description of published results based on HistFitter at SUSY ETmiss Meeting by David Cote (04.2012)
- Talk at SUSY Workshop by David Cote (11.2011)
- Talk at SUSY EtMiss Meeting by Dan Short (12.2011)
- Talk at SUSY EtMiss Meeting by Max Baak (02.2012)
- Wiki page explaining the meaning of HistFactory arguments
- Mathematical description of the PDF produced by HistFactory
- SUSY shape fit meetings in Indico
- atlas-phys-stat-root e-group
- Higgs tutorial
- Exotics tutorial
- Higgs combination

Troubleshooting

Don't hesitate to send your `HistFitter` related questions or feedback on the tutorial to our mailing list:

`atlas-phys-susy-histfitter_at_cern.ch`

Major updates:
-- Max Baak & David Cote (CERN) - 25-Sep-2012
-- Max Baak & David Cote (CERN) - 24-Apr-2012
-- JeanetteLorenz - 21 Jun 2014
-- Main.Alex Koutsman - 12 Sep 2014

Responsible: MaxArjenBaak
Last reviewed by: **Never reviewed**

This topic: AtlasProtected > HistFitterTutorial
Topic revision: r86 - 06 Oct 2014 - MaxArjenBaak