

[TWiki](#) > [AtlasProtected Web](#) > [AtlasPhysics](#) > [SUSYWorkingGroup](#) > [SusyEtMissSubGroup](#) > [SUSYEtmissOneLepton](#) > [SusyFitter](#) > [HistFitterTutorial](#)
(2019-10-24, JeanetteLorenz)

HistFitterTutorial

Not yet
Certified as
ATLAS
Documentation

[Introduction](#)

[Part 0: Setting up HistFitter](#)

[Note on version](#)

[Setup in an ATLAS tutorial using cvmfs](#)

[Setup standalone](#)

[HistFitter usage](#)

[Part 1: Building & fitting RooFit PDF for a simple analysis](#)

[Simple counting experiment, input from user \(one bin, one region\) \(*\)](#)

[Running the fit](#)

[Visualizing the result](#)

[Checking the impact of systematic errors \(optional\)](#)

[Simple counting experiment, input from TTrees \(one bin, one region\) \(*\)](#)

[Running the fit](#)

[Understanding the configuration](#)

[Fit results](#)

[Part 2: More advanced fits](#)

[Types of fits and regions \(*\)](#)

[Setting the region type](#)

[Setting the fit types](#)

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

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

[Available features in the python code](#)

[Systematic uncertainties \(*\)](#)

[Varying modeling of systematic uncertainties](#)

[Exercise: the effect of a systematic type \(*\)](#)

[Type of systematic to choose](#)

[Binned shape fit with multiple channels \(*\)](#)

[Validation regions \(*\)](#)

[Exercise: add validation plots \(*\)](#)

[Data-driven background shape determination \(optional\)](#)

[Part 3: Tools for visualising fits results](#)

[Yields table](#)

[Systematics table](#)

[Validation pull plot](#)

[Part 4: Statistical interpretation - ROOT based vs HistFitter](#)

[Setting up a simple cut-and-count analysis in ROOT \(optional\)](#)

[The HistFitter equivalent](#)

[Fitting and plotting a workspace in ROOT \(optional\)](#)

[The HistFitter equivalent](#)

[Performing an hypothesis test in ROOT \(*\)](#)

[The HistFitter equivalent](#)

[Details on the output of -p](#)

[Free fit of the background + signal model](#)

[Hypothesis tests](#)

[Part 1: Setting up the asymptotic calculator and building the Asimov dataset:](#)

[Part 2: Executing the hypothesis tests](#)

[Setting an upper limit \(*\)](#)

[The HistFitter equivalent](#)

[Model-independent upper limits table \(*\)](#)

[Part 5: Making a complete exclusion contour plot \(*\)](#)

[Running the hypothesis tests](#)

[Creating and plotting the contour](#)

[Alternative \(old\) workflow](#)

[Creating a list file](#)

[Investigating the list file](#)

[Creating contour histograms](#)

[Creating contour plots](#)

[Automating the procedure](#)

[Analysis Check List](#)

[Useful links and Extra documentation](#)

[Troubleshooting](#)

[ROOT version](#)

[Setup with TAR file used for original tutorial while svn was not working](#)

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.

A second tutorial part covering more advanced features is available at: [HistFitterAdvancedTutorial](#)

Before starting, please have a look at the introductory HistFitter slides : [part 1](#) and [part 2](#) (from the [June 2016 ATLAS offline software tutorial](#)).

(Note: These are the slides typically presented at the start of a hand-on tutorial.)

Useful External Links:

[Statistics Forum Twiki](#)

[RooStats tutorial](#)

[Profile likelihood fits: guidelines and help](#)

Part 0: Setting up HistFitter

Note on version

NOTE: this tutorial was written for use with HistFitter v0.63.0. The latest recommended version can always be found at the [release notes page](#), and it is likely to have more features than advertised here. For production use we always recommended the latest advertised tag.

NOTE: This tutorial was written for slc6. Please log in a slc6 machine via

```
ssh lxplus6.cern.ch -X -Y
```

Setup in an ATLAS tutorial using cvmfs

To set up the package, log on to a prepared (see the instructions of your local tutorial) machine and follow instructions below (replace USERNAME by your own user id). We will define an environment variable \$ALRB_TutorialData which will point to the location of the input data used for the tutorial (this depends on which tutorial and where you are running, you can copy the data over to a local area). For the moment we will assume you are working from lxplus, so you can do one of the following to define this variable (depending on which shell you are using):

Tutorial	zsh, bash shells	csh shell
CERN July 2018 (recommended)	export ALRB_TutorialData=/afs/cern.ch/atlas/project/PAT/tutorial/cern-may2015/	setenv ALRB_TutorialData /afs/cern.ch/atlas/project/PAT/tutorial/cern-may2015/
CERN May 2015	export ALRB_TutorialData=/afs/cern.ch/atlas/project/PAT/tutorial/cern-may2015/	setenv ALRB_TutorialData /afs/cern.ch/atlas/project/PAT/tutorial/cern-may2015/
CERN November 2014	export ALRB_TutorialData=/afs/cern.ch/atlas/project/PAT/tutorial/cern-nov2014/	setenv ALRB_TutorialData /afs/cern.ch/atlas/project/PAT/tutorial/cern-nov2014/

```
setupATLAS
export ALRB_TutorialData=<data_dir_provided_for_tutorial>
```

Then proceed to check out the HistFitter package. On lxplus:

```
git clone https://:gitlab.cern.ch:8443/HistFitter/HistFitter.git HistFitterTutorial
```

OR, on other machines, with an ssh key:

```
git clone ssh://gitlab.cern.ch:7999/HistFitter/HistFitter.git HistFitterTutorial
```

For info on how to generate an SSH key pair see the [GitLab Help](#)

Then:

```
cd HistFitterTutorial
```

Then proceed with the setup. On lxplus:

```
lssetup 'root 6.06.06-x86_64-slc6-gcc49-opt'
source setup.sh
```

Note that we have set here a specific Root version such that the numerical results below match with the numbers you get. In principle, HistFitter is supposed to work with every Root version.

[i] NOTE: If you are working with Root >= 6.18 the [MakeFile](#) as it comes along with the package works like it is. If you use Root < 6.18 you need to change the following in the src/MakeFile:

```
CXXFLAGS = -std=c++17 -fPIC -Wall
```

to:

```
CXXFLAGS = -std=c++1y -fPIC -Wall
```

Then continue with:

```
cd src
make
cd ..
```

You now have a working HistFitter installation! You can proceed working in this directory, but if you want to run inside a different test area that is also possible. Make sure you do this in whichever directory you will be running in:

```
mkdir samples
cd samples
ln -s $ALRB_TutorialData/HistFitter tutorial
cd ..
```

[i] Note: the recommended tag for the tutorial may not always be the same tag as recommended for physics analysis.

[i] Note: in case of need, the required input files are replicated here on lxplus:

/eos/atlas/atlasccrncgroupdisk/phys-susy/histfitter/stronglepton/

For more info about EOS, see [this link](#).

[i] Note: every time you login, navigate to your HistFitter directory (in this case [HistFitterTutorial](#)), and repeat the setup:

```
setupATLAS
lsetup 'root 6.06.06-x86_64-slc6-gcc49-opt'
source setup.sh
```

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

Setup standalone

[HistFitter](#) is currently working nicely on ubuntu systems (we have e.g. tested ubuntu 14.04). It's important that you have installed a decent Root version with the following options activated:

```
--enable-minuit2 --enable-roofit --enable-xml
```

(minuit2 and roofit are mandatory, xml nice to have)

Once you have installed Root and run the usual

```
source bin/thisroot.sh
```

please also run

```
source setup.sh
cd src
make
cd ..
```

Sourcing the setup script will make sure that the HistFitter libraries can be found.

Create the directory samples/tutorial under the HistFitter directory and download the input files in /afs/cern.ch/atlas/groups/susy/1lepton/samples/tutorial to the tutorial directory.

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 physics 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). Selected more advanced options are discussed in the [HistFitterAdvancedTutorial](#).

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

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

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`.)

Note: you could also run the two steps -w and -f in separate commands. This is true for any of the steps, e.g. "-t -w -f -p" could be split into 4 commands. If you run into a crash, it is easier to figure out the problem.

Note: If you don't want to pollute the HistFitter directory, it is possible to work in a separate directory. Just comment out `configMgr.writeXML` in the file, and specify the full path:

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

(If you want to write XML, you will otherwise need to copy the *.dtd file to make sure -w can still run.)

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 (μ_{Sig}) is found and reduced to be 0.4 (± 0.7), corresponding to 2 signal events.

The full fit result is:

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

For example, the signal parameter (μ_{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: α_{ucb} and α_{ucs} . There's also a fully correlated systematic uncertainty between the signal and background estimates: α_{cor} . Finally, there's the parameter $\gamma_{\text{stat_UserRegion_cuts_bin_0}}$, 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 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)

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)

```

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-independent 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 -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.

Start HistFitter in interactive mode and run the following:

```
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 -p -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 (μ_{SIG}). This also gets stored in the file `results/upperlimit_cls_poi_SM_GG_onestepCC_425_385_345_Asym_CLs_grid_ts3.root.eps`. The points where the p-value goes below 0.05 (red line) are excluded at 95% C.L. The CLs and upper limit are also printed on the screen:

```
<INFO> HypoTestTool: The computed upper limit is: 0.669437 +/- 0
<INFO> HypoTestTool: expected limit (median) 0.910312
```

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. An example of this is presented in the configuration file `analysis/tutorial/MyShapeFitExample.py`.

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: 0.330398 +/- 0
<INFO> HypoTestTool: expected limit (median) 0.315543
```

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 explanatory. 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,ktScaleTopLowWeights,"weight","overallSys")
topKtScale = Systematic("KtScaleTop",configMgr.weights,ktScaleTopHighWeights,ktScaleTopLowWeights,"weight","histoSys")
#topKtScale = Systematic("KtScaleTop",configMgr.weights,ktScaleTopHighWeights,ktScaleTopLowWeights,"weight","normHistoSys")
```

and

```
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 one 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 HistFitter'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_SM_GG_onestepCC_425_385_345_SR_metmeff2Jet.xml

where you can see:

```
<Sample Name="Top" HistoName="hTopNom_SR_obs_metmeff2Jet" InputFile="data/MyShapeFitExample.root" NormalizeByTheory="False">
  <HistoSys Name="KtScaleTop" HistoNameHigh="hTopKtScaleTopHigh_SR_obs_metmeff2Jet" HistoNameLow="hTopKtScaleTopLow_SR_obs_metmeff2Jet" />
```

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 explicitly 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!

Type of systematic to choose

Looking for a description on all available systematic uncertainty types and which one is supposed to use when? See the [flowchart and description](#) in the advanced tutorial.

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

[MyConfigExample.py](#)

Before proceeding, make sure that the parameter doValidation=False in analysis/tutorial/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 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
```

You'll notice the new argument -F, which is set to excl. This argument sets the *fit type*. The rationale for such an argument is easy: in a real-world analysis, you don't want to change your analysis file by hand all the time in order to modify the definition and/or role a region can play. Instead, we can rely on the fit type specified on the command line and adapt our behaviour accordingly. In the code, you will see lines such as these:

```
if myFitType==FitType.Discovery:
```

If we run a background fit, we just define our background regions and add samples (including optional validation regions). For an exclusion fit, we need a signal to test. Finally, for a so-called discovery fit (used for e.g. model-independent upper limits), we need a freely floating dummy signal.

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" -i 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 (optional)

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

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_combined_NormalMeasurement_model_afterFit.root -o MyYieldsTable.tex
```

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:

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 SR1sl2j -o systable_SR1sl2j.tex
```

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_a
```

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.

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).

The HistFitter equivalent

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.

The HistFitter equivalent

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. 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 $\mu < \mu_{\text{hat}}$)
- **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 reproducible. (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:      0.136027   sigma: 1.09834
| CLsexp:    0.0782798   sigma: 1.41674
| CLsu1S (+1s)  0.235372   sigma: 0.721269
| CLsd1S (-1s)  0.0210993   sigma: 2.03156
| CLsu2S (+2s)  0.523149   sigma: -0.0580596
| CLsd2S (-2s)  0.00748652   sigma: 2.43303
```

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:      0.136365   sigma: 1.0968
| CLsexp:    0.0838288   sigma: 1.37977
| CLsu1S (+1s)  0.276979   sigma: 0.591841
| CLsd1S (-1s)  0.020027   sigma: 2.05319
| CLsu2S (+2s)  0.620974   sigma: -0.30804
| CLsd2S (-2s)  0.0042269   sigma: 2.63339
```

- 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 2>&1 | tee out.log
```

(This also writes a log file out.log which we need in [HistFitterTutorial#Details on the output of p](#) below.)

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

The hypothesis test configuration can be found at the top of the configuration 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-side
```

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.

Details on the output of -p

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,calculatorType,testStatType,ntoys,useCLs,npoints);
```

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: 1.55944 +/- 0
<INFO> HypoTestTool: expected limit (median) 1.32593
<INFO> HypoTestTool: expected limit (-1 sig) 0.855869
<INFO> HypoTestTool: expected limit (+1 sig) 2.15729
<INFO> HypoTestTool: expected limit (-2 sig) 0.600953
<INFO> HypoTestTool: expected limit (+2 sig) 3.53821
```

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 <number> sets the random seed for toy generation (the default is CPU clock: 0).

Again, the hypothesis test configuration can be found at the top of the configuration 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-side
```

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 2.24688 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.

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.

<!--For an example config file, take a look at: `/HistFitterUser/MET_jets_leptons/python/MyDiscoveryAnalysis_SR3jTEI.py`.

In this example we input the total background and the uncertainty on it by hand (as calculated by the yields table script), as well as a 'dummy' signal.-->

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 $\mu_{SS} 2.0328e+00 \pm 2.60e+00$. 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_model.root -l 4.713 -n 1000
```

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:

```
Results HypoTestCalculator_result:
- Null p-value = 0.225 +/- 0.0132051
- Significance = 0.755415 +/- 0.0440299 sigma
- Number of Alt toys: 500
- Number of Null toys: 1000
- Test statistic evaluated on data: 0.293789
- CL_b: 0.225 +/- 0.0132051
- CL_s+b: 0.324 +/- 0.0209296
- CL_s: 1.44 +/- 0.125679
```

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 separate 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) config
```

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 and plotting the contour

After running the hypothesis test, to create and plot the exclusion contour, follow the instructions in this [README](#) or, alternatively, the ones for the old workflow below.

Alternative (old) workflow

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

ROOT version

In principle, the package setup.sh should work out of the box and nothing needs to be done. But if for some reason you need to setup your enviroment by yourself, the recommended tag for ROOT is v5-32-02 (or any higher patch version of v5-32) and can be obtained with:

```
svn co http://root.cern.ch/svn/root/tags/v5-32-02 root
```

Follow the instructions in README/INSTALL for the installation of ROOT.

A (temporary) installation at CERN can be found under:

```
/afs/cern.ch/atlas/offline/external/FullChainTest/tier0/test/mbaak/root/root-v5.32.02/
```

To use this ROOT version, do:

```
cd /afs/cern.ch/atlas/offline/external/FullChainTest/tier0/test/mbaak/root/root-v5.32.02/
source bin/thisroot.sh
cd -
```

Setup with TAR file used for original tutorial while svn was not working

```
cp /afs/cern.ch/user/c/cote/public/HistFitterTutorial.tar.gz .tar xvfz [[HistFitterTutorial]][HistFitterTutorial]].tar.gz
cd [[HistFitterTutorial]][HistFitterTutorial]]
source setup.shcd src
make clean
make
cd ..
cp /afs/cern.ch/user/d/dshort/public/configWriter.py python
```

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

-- [JeanetteLorenz](#) - 18 March 2015

-- Chiara.Rizzi & Sarah.Williams - April 2018

<!--Please add the name of someone who is responsible for this page so that he/she can be contacted if changes are needed.

The creator's name will be added by default, but this can be replaced if appropriate.

Put the name first, without dashes.-->

Responsible: [SophioPatarai](#)

<!--Once this page has been reviewed, please add the name and the date e.g. [StephenHaywood](#) - 31 Oct 2006 -->

Last reviewed by: **Never reviewed**

Topic revision: r122 - 2019-10-24 - [JeanetteLorenz](#)

Copyright &© 2008-2019 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.
Ideas, requests, problems regarding TWiki? [Send feedback](#)

