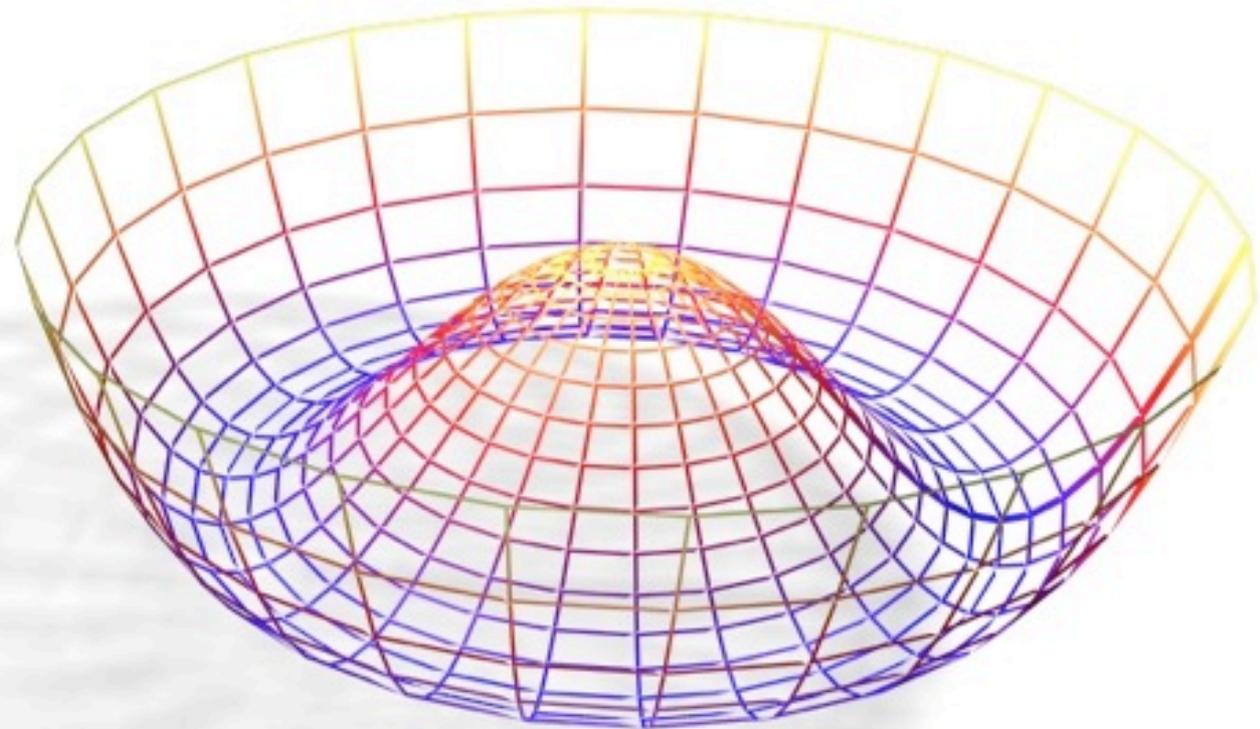




RooStats: high-level statistics tools in ROOT



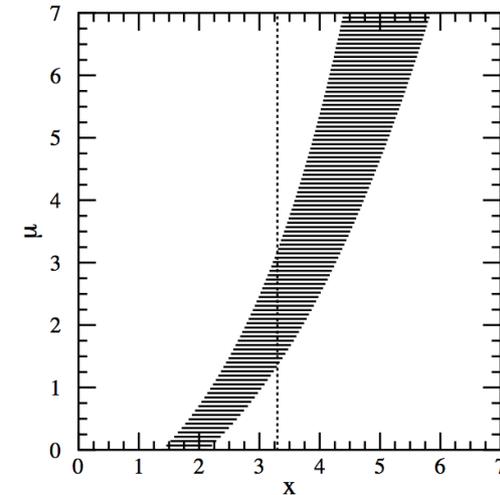
Kyle Cranmer,
New York University

on behalf of the RooStats developers

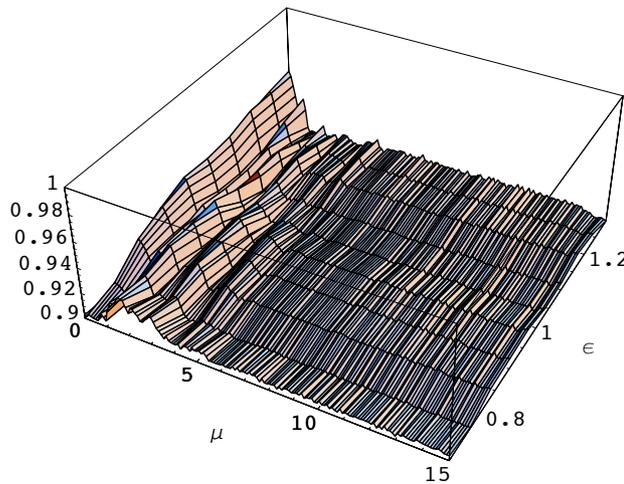
One Model, Many Methods

Essentially all statistical statements start with the basic probability density function $P(x|\mu, \nu)$

- ▶ *building a good model for the data is hard!*
 - *this is where we use our understanding of physics*
- ▶ there are several valid statistical methods
 - different consumers of results want different things
- ▶ want to re-use same model for the data in many different ways



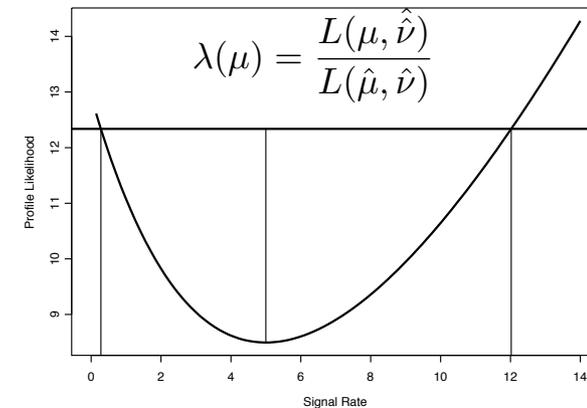
Neyman Construction



Coverage Studies

$$P(\mu|x) \propto \int P(x|\mu, \nu) \pi(\mu, \nu) d\nu$$

Bayes Theorem



Profile Likelihood
(MINUIT/MINOS)

New ATLAS-only email list: atlas-phys-stat-root@cern.ch

- ▶ initial focus RooFit/RooStats, but open to general ROOT questions

Next ROOT production release mid Dec. (5.28)

- ▶ Main efforts: clean-up, validation, bug fixes, and requests
- ▶ Several fixes for coding problems identified by **Coverity** code checker
- ▶ One more week devoted cleaning up tutorials and documentation

Aiming to have tutorial in Jan. to overlap with PhyStat

- ▶ ideal date: Fri., Jan 21 (but there is also a TMVA workshop)
- ▶ may also be able to do a tutorial on Fri., Jan 14

Citation: "The RooStats project", <http://arxiv.org/abs/1009.1003> Proceedings of the ACAT2010 Conference

Will be able to **support** an expert (post-doc or advanced graduate student) to be at CERN in exchange for fraction of time spent in RooFit/RooStats development, validation, maintenance, and support within the collaboration. Please contact me if you are interested.

RooStats provides tools for high-level statistics questions in ROOT

- ▶ it builds on **RooFit** which provides basic building blocks for statistical questions

I will start with an overview of RooFit and then move to RooStats

RooFit

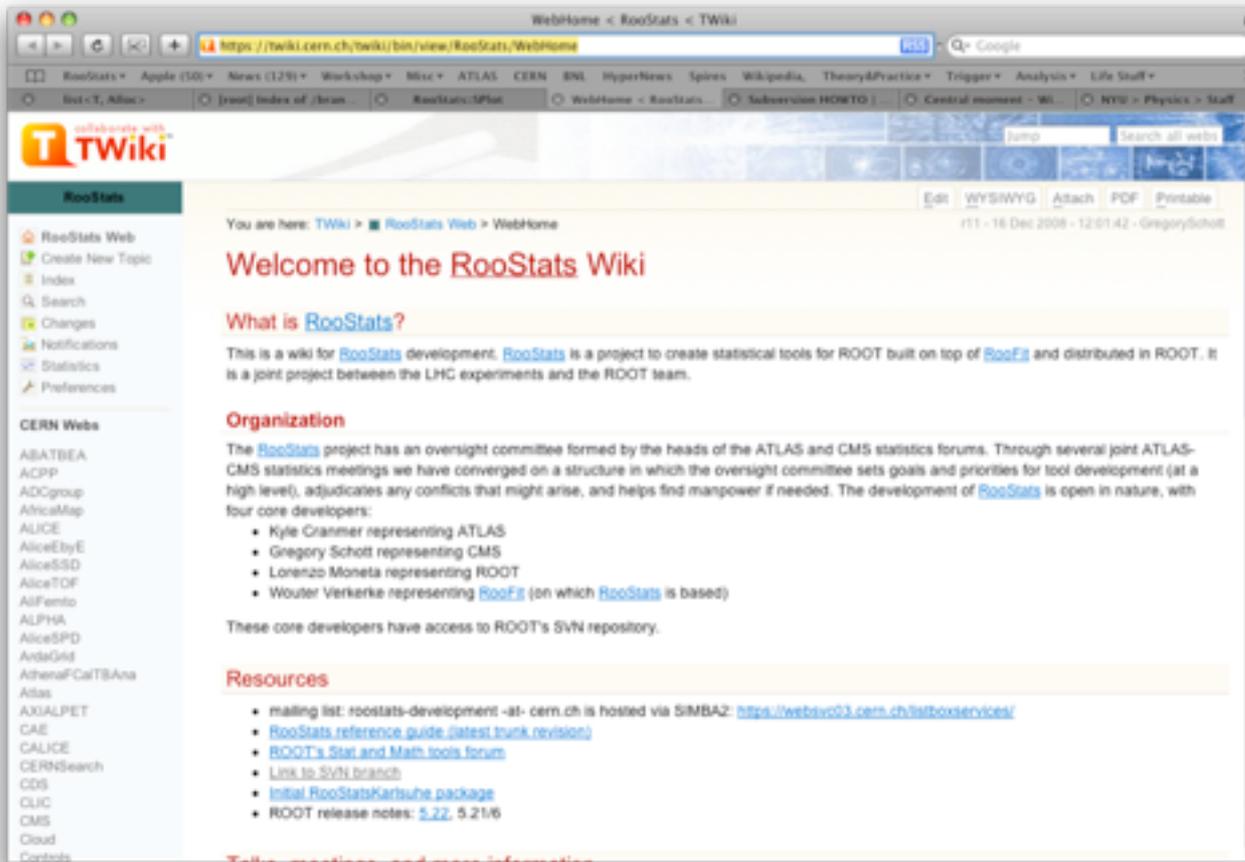
variables functions
probability density functions
binned & unbinned datasets
fitting toyMC generation
minimization integration

RooStats

hypothesis prior
hypothesis tests
confidence intervals (limits)
combinations
test statistic sampling distribution

Note, excellent slides from Wouter Verkerke on RooFit at SoS '08 (I will borrow from them)

<http://indico.in2p3.fr/materialDisplay.py?contribId=15&materialId=slides&confId=750>



<https://twiki.cern.ch/twiki/bin/view/RooStats/WebHome>

ScreenCast tutorials

<http://www.youtube.com/RooStats>

User's Guide

draft users guide

There is also a category in ROOT's Savannah bug tracking system

Core developers

- K. Cranmer (ATLAS)
- Gregory Schott (CMS)
- Wouter Verkerke (RooFit)
- Lorenzo Moneta (ROOT)
- open project, you are welcome to join. Contributions from
 - Max Baak, Kevin Belasco, Danilo Piparo, Giacinto Piacquadio, Maurizio Pierini, George H. Lewis, Alfio Lazzaro, Sven Kreiss, Mario Pelliccioni, Matthias Wolf
 - Included since ROOT v5.22
- Example macros in
 - `$ROOTSYS/tutorials/roostats`

Documentation

- code doc. via ROOT
- users manual currently 42 pages, linked from Wiki page

Contents

1 Introduction

- 1.1 Getting Started
- 1.2 Other Resources
- 1.3 Terminology used in this guide

2 Fundamental Interfaces in RooStats

- 2.1 RooRealVar, RooArgSet, RooAbsReal, & RooAbsPdf
- 2.2 RooWorkspace & Model Config
- 2.3 ConflInterval & IntervalCalculator
- 2.4 HypoTestResult & HypoTestCalculator
- 2.5 Test Statistic, Sampling Distribution

3 Parameter Estimation

4 Test Statistics and Sampling Distributions

- 4.1 TestStatistic interface and implementations

5 Hypothesis Testing

- 5.1 Single Channel Number Counting with Background Uncertainty
- 5.2 Profile Likelihood Ratio (the method of MINOS)
- 5.3 The Hybrid Calculator

6 Confidence Intervals

- 6.1 Profile Likelihood Ratio (the method of MINOS)
- 6.2 Neyman Construction
- 6.3 Feldman-Cousins
- 6.4 Neyman Construction with nuisance parameters
- 6.5 The "Profile Construction"
- 6.6 Markov Chain Monte Carlo
 - 6.6.1 MCMCCalculator
 - 6.6.2 ProposalFunction
 - 6.6.3 MetropolisHastings
 - 6.6.4 MCMCInterval
 - 6.6.5 MCMCIntervalPlot
 - 6.6.6 MarkovChain

7 Goodness of Fit

8 Coverage Studies

1.1 Getting Started

Since December 2008, RooStats has been distributed in the ROOT release since version 5.22 (December 2008). To use RooStats, you need a version of ROOT greater than 5.22, but you will probably want the most recent ROOT version since the project is developing quickly.

Option 1) Download the binaries for the latest ROOT release

You can download the most recent version of ROOT here: <http://root.cern.ch/>

Option 2) Check out and build the ROOT trunk

If you prefer to build ROOT from source,

```
svn co http://root.cern.ch/svn/root/trunk root
```

then build and install ROOT via (you may want different configure options)

```
configure --enable-roofit  
make  
make install
```

Option 3) Check out and build the RooStats branch

If you need a development or bug-fix that is not yet in a ROOT release, you can download the most recent version of the code from ROOT's subversion repository. To check it out, go to some temporary directory and type:

```
svn co https://root.cern.ch/svn/root/branches/dev/roostats root
```

then build and install ROOT via (you may want different configure options)

```
configure --enable-roofit  
make  
make install
```

For more information about building ROOT from source, see the ROOT webpage: <http://root.cern.ch/drupal/content/installing-root-source>.



Goal: Standardize interface for major statistical procedures so that they can work on an arbitrary RooFit model & dataset and handle many parameters of interest and nuisance parameters.

▶ **Status:** Done

- **ConfIntervalCalculator** & **HypoTestCalculator** interface for tools
- they return **ConfidenceInterval** and **HypoTestResult**

Goal: Implement most accepted techniques from Frequentist, Bayesian, and Likelihood-based approaches

▶ **Status:** Done / Ongoing

- **ProfileLikelihoodCalculator:** (Likelihood) the method of MINUIT/MINOS
- **FeldmanCousins:** (Frequentist) a specific version of Neyman Construction
- **MCMCCalculator:** (Bayesian) uses Metropolis-Hastings algorithm
- **HybridCalculator:** (Bayesian/Frequentist Hybrid) like what was used at LEP
- ...

Goal: Provide utilities to perform combined measurements

▶ **Status:** Partially done / Ongoing

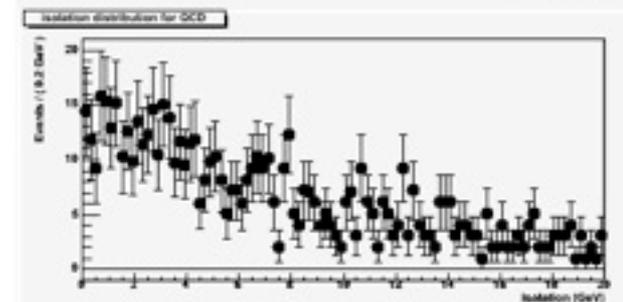
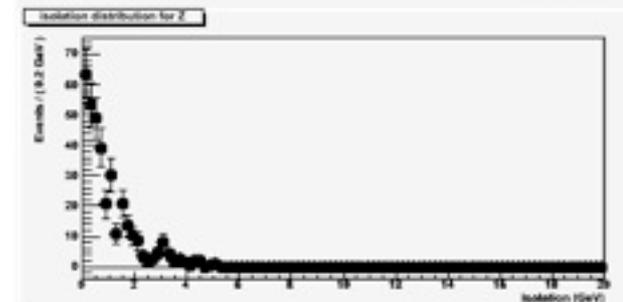
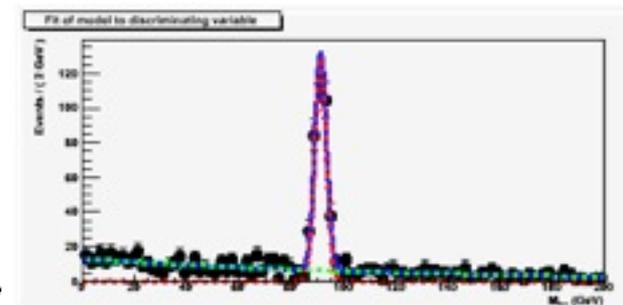
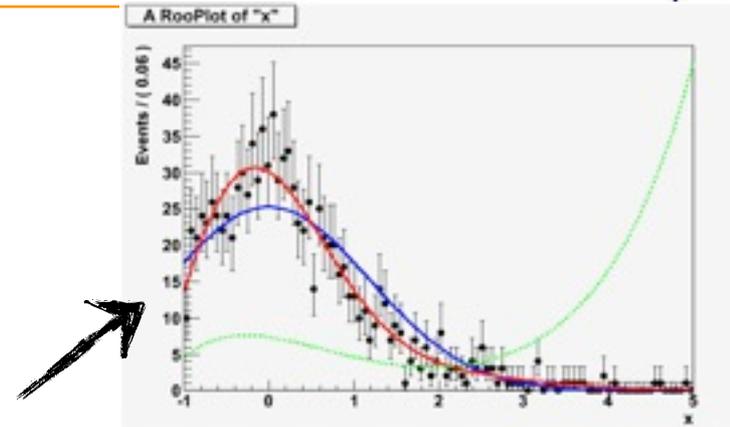
- **RooWorkspace** allows one to save arbitrary RooFit model (even with custom code) into a .root file. PDFs and DataSets have been extended to facilitate combinations.
- Next talk will show working Higgs & Top examples. Working to increase automation.



Goal: Provide utilities for common tasks.

► **Status:** Ongoing

- **BernsteinCorrection:** prompted by work in our statistics forum, automate procedure of correcting nominal model to data.
 - http://root.cern.ch/root/html/tutorials/roostats/rs_bernsteinCorrection.C.html
- **SPlot:** Working SPlot implementation that works for arbitrary models
 - rewritten from original code from BaBar
 - more general than TSPlot class
 - http://root.cern.ch/root/html/tutorials/roostats/rs301_splot.C.html
- **NumberCountingPdfFactory:** builds PDFs that describe a combination of number counting experiments with background uncertainty

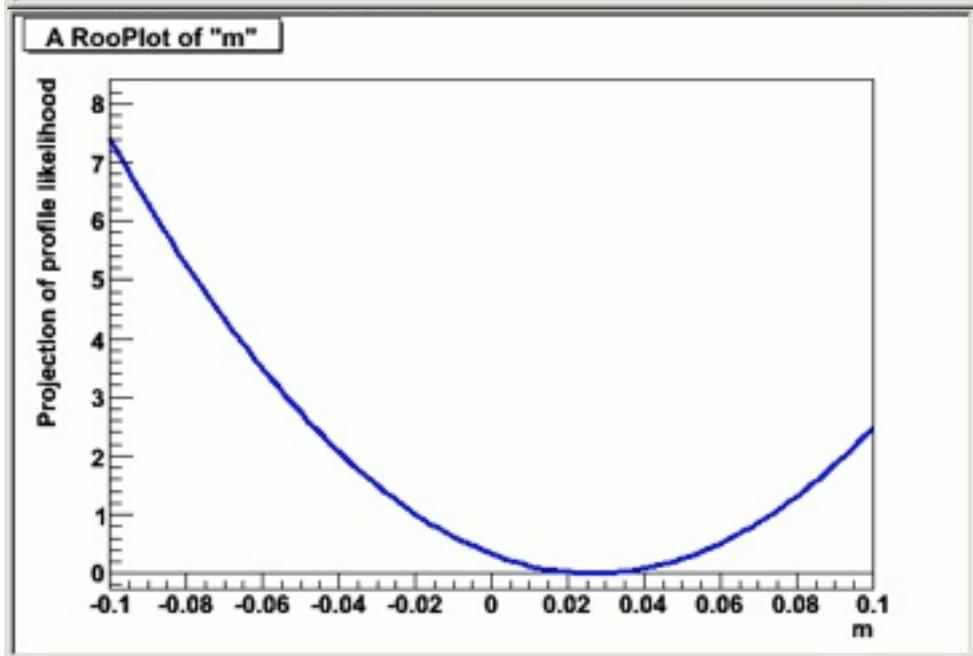
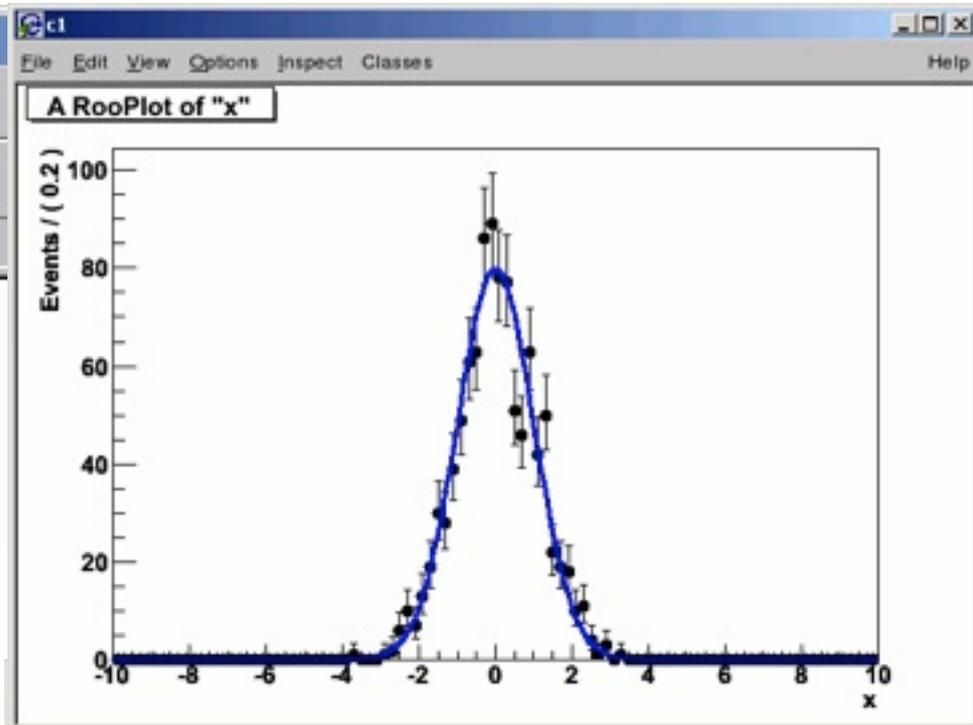
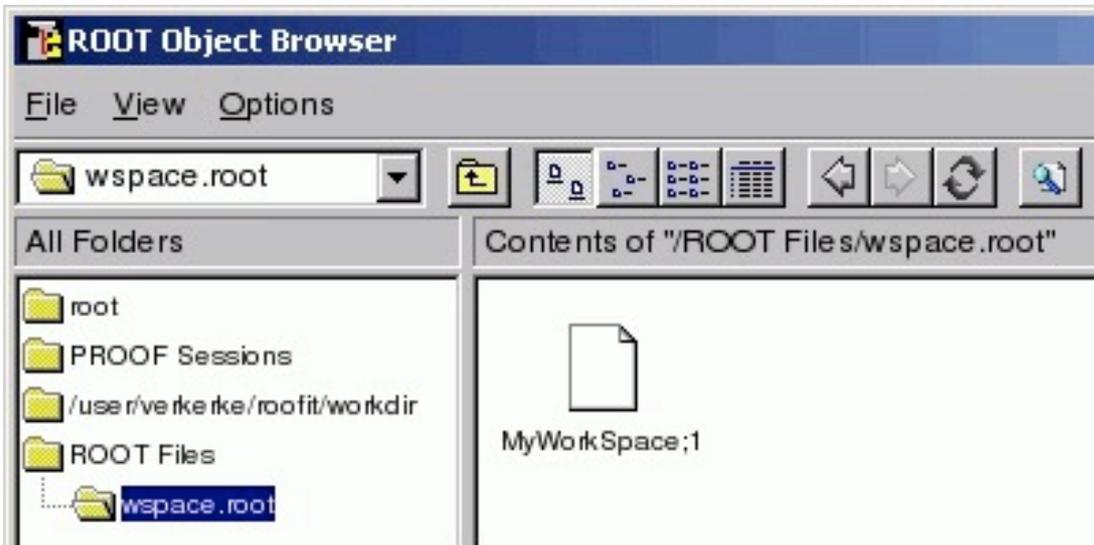


Goal: Provide utilities or examples requested by community

► **Status:** Ongoing

- **NumberCountingUtils:** provides standalone utilities for simple number counting with background uncertainty
- **Request:** something similar for limits. (more later)

Example of Digital Publishing



RooFit's Workspace now provides the ability to save in a ROOT file the full likelihood model, any priors you might want, and the minimal data necessary to reproduce likelihood function.

Can also evaluate integrals over x necessary for Neyman construction!

Need this for combinations, great potential for publishing results.

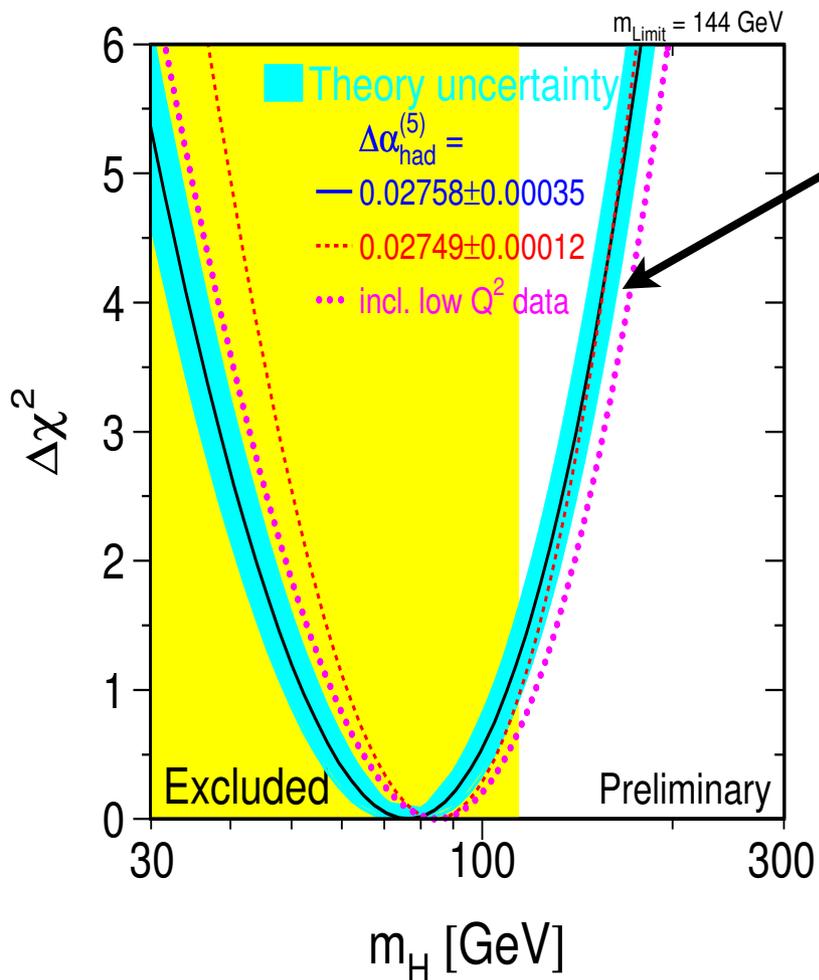
Examples of Published Likelihoods



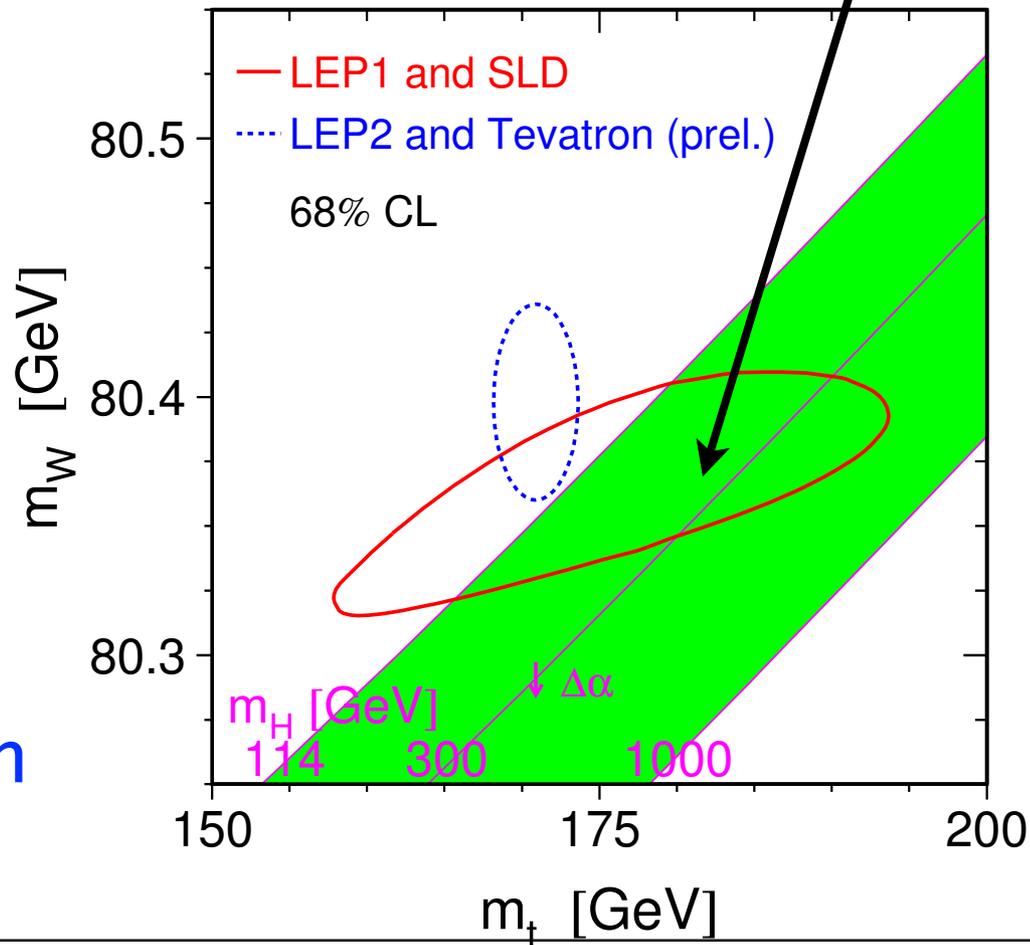
At previous PhyStats, we agreed to publish likelihood functions

You can find examples of published likelihoods in 1D

In 2-D you just get the contours



With the workspace, we can finally achieve this goal!

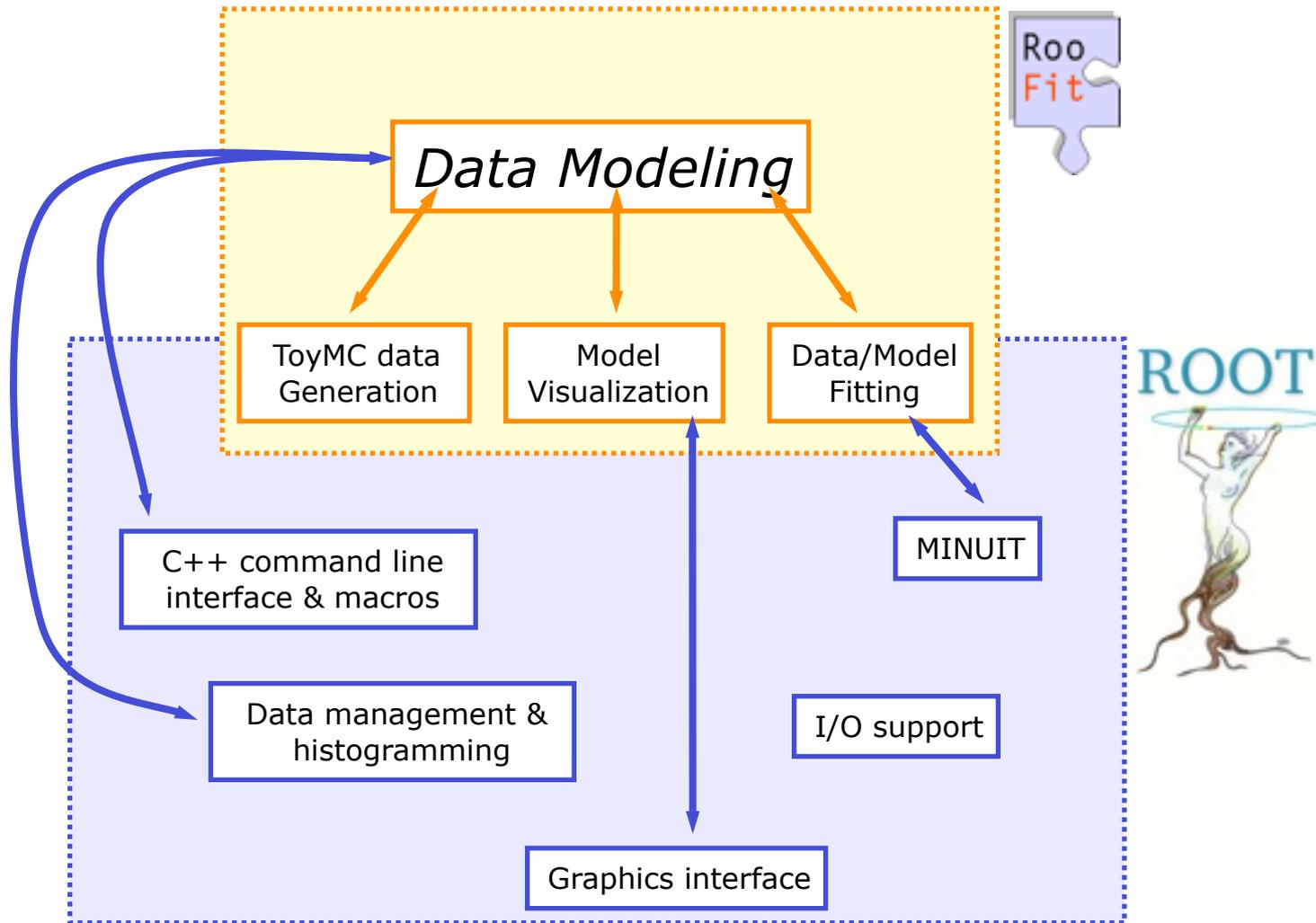


1 RooFit Introduction & Overview

- *Introduction*
- *Some basics statistics*
- *RooFit design philosophy*

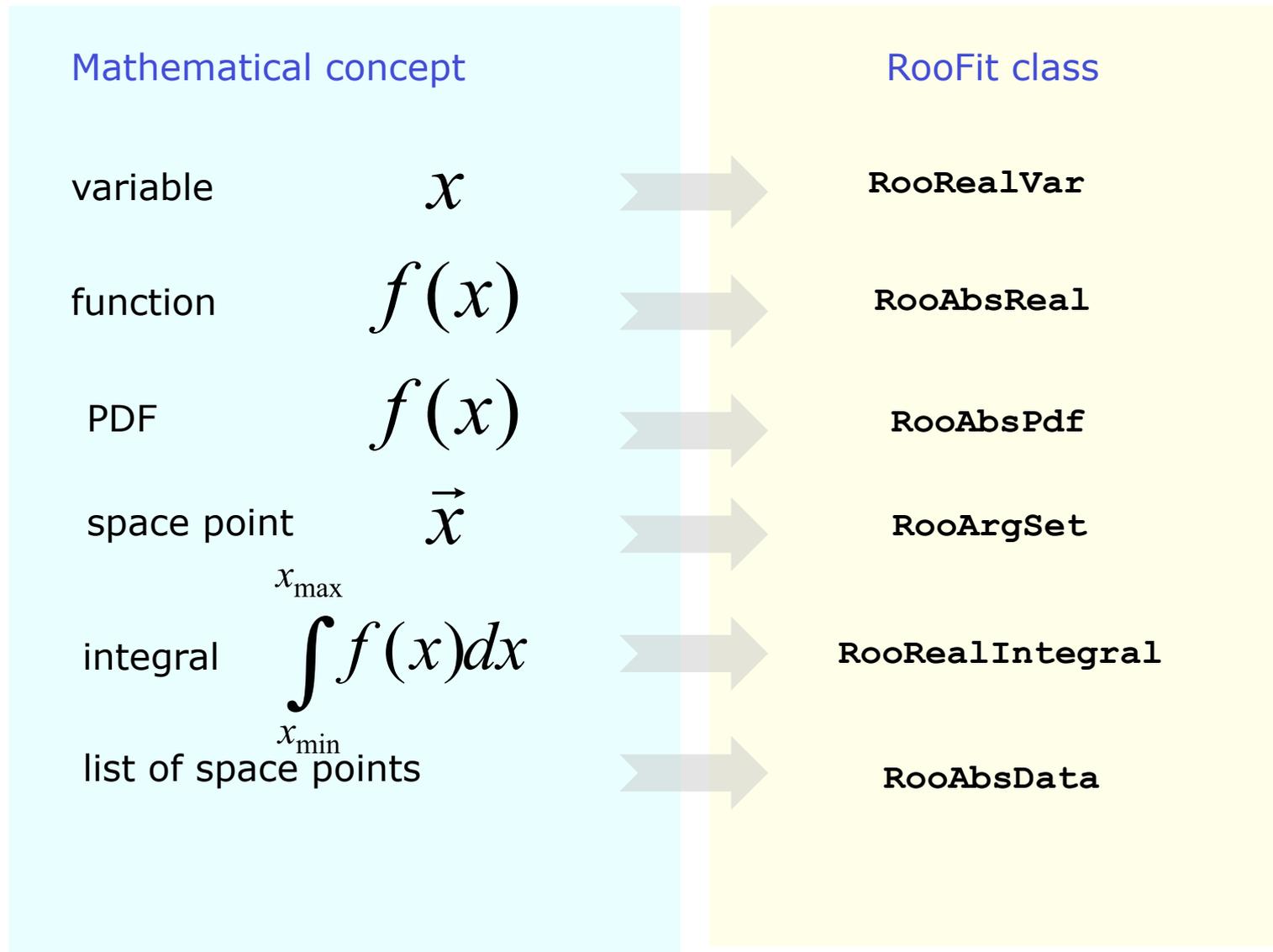
Introduction – Relation to ROOT

Extension to ROOT – (Almost) no overlap with existing functionality



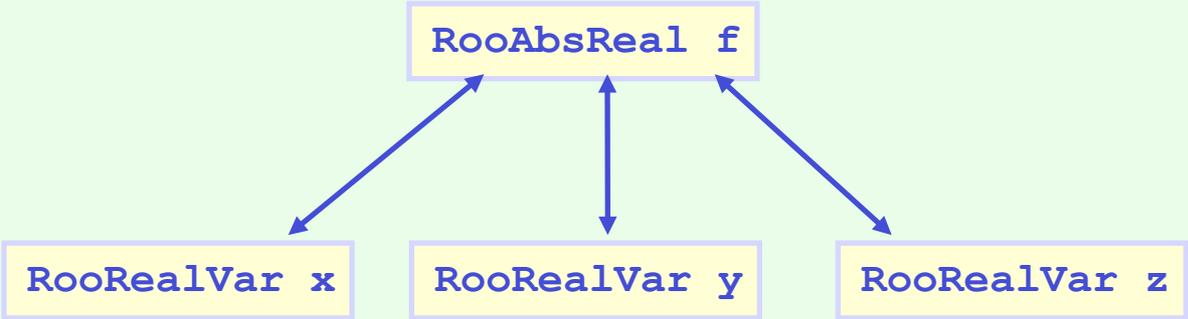
Roofit core design philosophy

- Mathematical objects are represented as C++ objects



Roofit core design philosophy

- Represent relations between variables and functions as client/server links between objects

Math	$f(x,y,z)$
Roofit diagram	 <pre>graph TD; f[RooAbsReal f] --> x[RooRealVar x]; f --> y[RooRealVar y]; f --> z[RooRealVar z];</pre>
Roofit code	<pre>RooRealVar x("x","x",5) ; RooRealVar y("y","y",5) ; RooRealVar z("z","z",5) ; RooBogusFunction f("f","f",x,y,z) ;</pre>

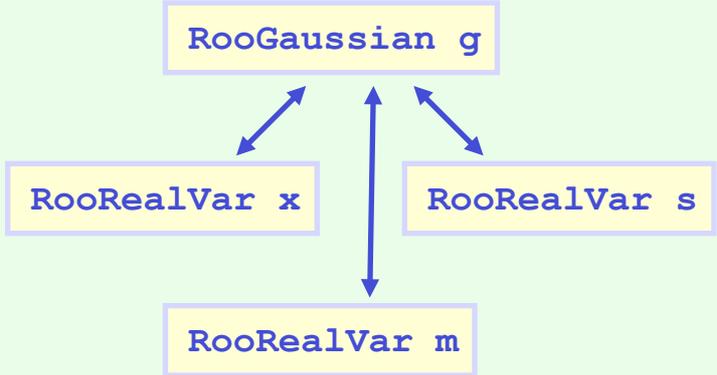
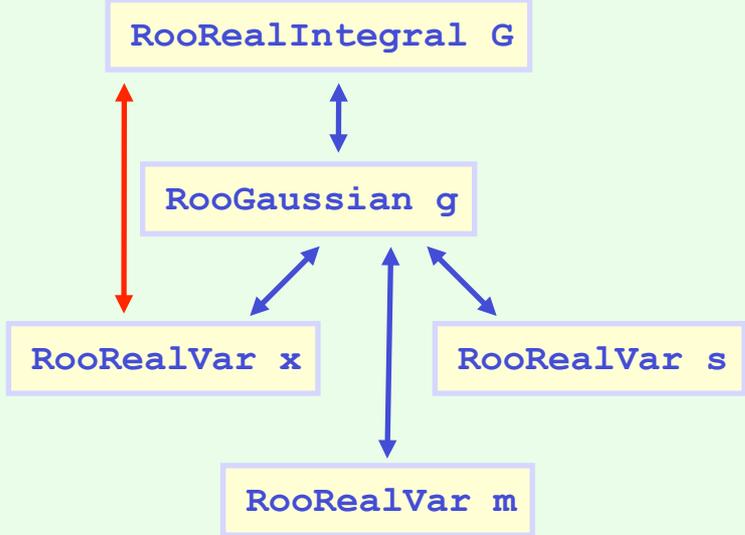
Roofit core design philosophy

- Composite functions → Composite objects

Math	$f(w,z)$ $g(x,y)$ \longrightarrow	$f(g(x,y),z) = f(x,y,z)$
Roofit diagram		
Roofit code	<pre> RooRealVar x("x","x",2) ; RooRealVar y("y","y",3) ; RooGooFunc g("g","g",x,y) ; RooRealVar w("w","w",0) ; RooRealVar z("z","z",5) ; RooFooFunc f("f","f",w,z) ; </pre>	<pre> RooRealVar x("x","x",2) ; RooRealVar y("y","y",3) ; RooGooFunc g("g","g",x,y) ; RooRealVar z("z","z",5) ; RooFooFunc f("f","f",g,z) ; </pre>

Roofit core design philosophy

- Represent integral as an object, instead of representing integration as an action

Math	$g(x, m, s)$	$\int_{x_{\min}}^{x_{\max}} g(x, m, s) dx = G(m, s, x_{\min}, x_{\max})$
Roofit diagram	 <pre> graph TD x[RooRealVar x] --> g[RooGaussian g] s[RooRealVar s] --> g m[RooRealVar m] --> g </pre>	 <pre> graph TD G[RooRealIntegral G] <--> g[RooGaussian g] x[RooRealVar x] --> g s[RooRealVar s] --> g m[RooRealVar m] --> g </pre>
Roofit code	<pre> RooRealVar x("x", "x", 2, -10, 10) RooRealVar s("s", "s", 3) ; RooRealVar m("m", "m", 0) ; RooGaussian g("g", "g", x, m, s) </pre>	<pre> RooAbsReal *G = g.createIntegral(x) ; </pre> <p style="text-align: right;">Wouter Verkerke, NIKHFF</p>

Object-oriented data modeling

- In RooFit every variable, data point, function, PDF represented in a C++ object
 - Objects classified by data/function type they represent, not by their role in a particular setup
 - All objects are **self documenting**
 - **Name** - Unique identifier of object
 - **Title** - More elaborate description of object

Objects representing a 'real' value.

```
RooRealVar mass("mass","Invariant mass",5.20,5.30) ;  
RooRealVar width("width","B0 mass width",0.00027,"GeV") ;  
RooRealVar mb0("mb0","B0 mass",5.2794,"GeV") ;
```

PDF object

```
RooGaussian b0sig("b0sig","B0 sig PDF",mass,mb0,width) ;
```

References to variables

2 Basic Functionality

- *Creating a p.d.f*
- *Basic fitting, plotting, event generation*
- *Some details on normalization, event generation*
- *Library of basic shapes (including non-parametric shapes)*

Basics – Creating and plotting a Gaussian p.d.f

Setup gaussian PDF and plot

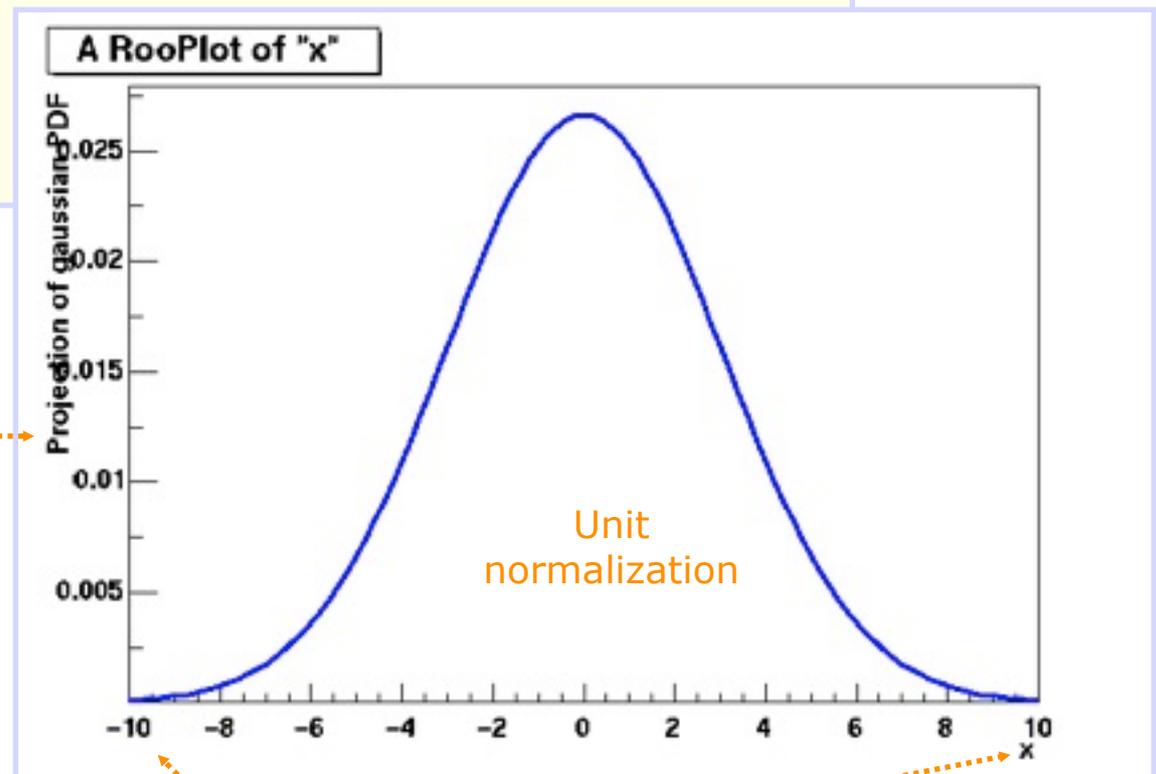
```
// Build Gaussian PDF
RoorealVar x("x","x",-10,10) ;
RoorealVar mean("mean","mean of gaussian",0,-10,10) ;
RoorealVar sigma("sigma","width of gaussian",3) ;

Roogaussian gauss("gauss","gaussian PDF",x,mean,sigma) ;

// Plot PDF
Rooplot* xframe = x.frame() ;
gauss.plotOn(xframe) ;
xframe->Draw() ;
```

Axis label from `gauss` title

A `Rooplot` is an empty frame capable of holding anything plotted versus its variable



Plot range taken from limits of `x`

Basics – Generating toy MC events

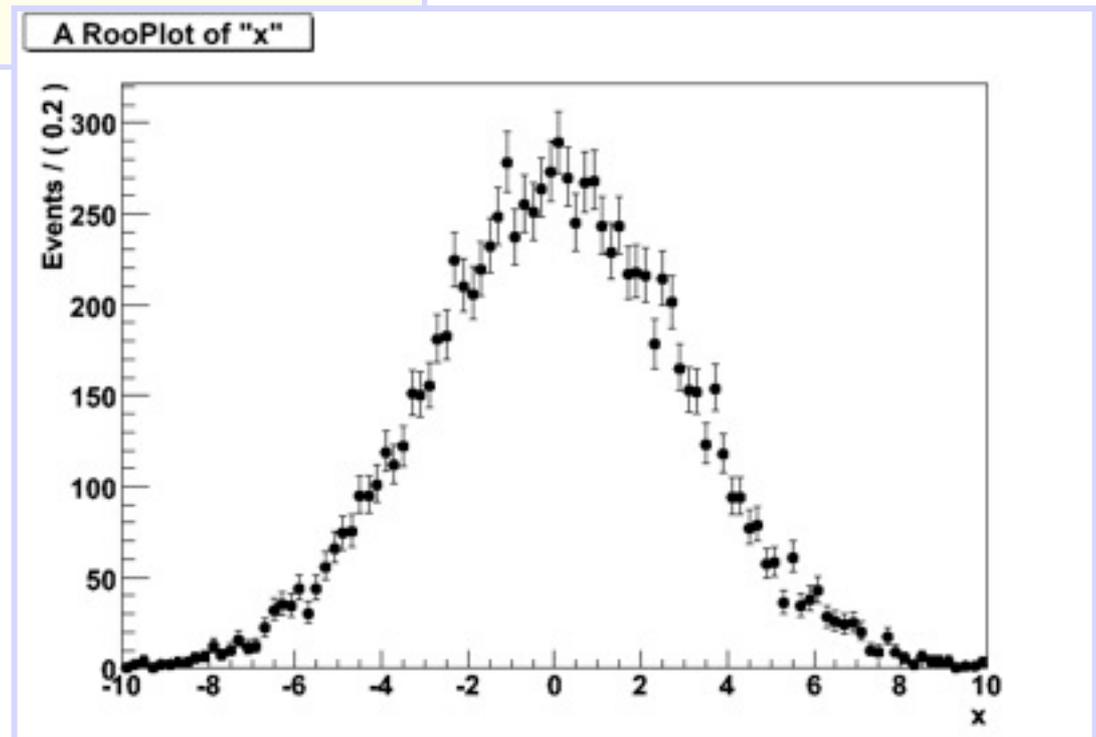
Generate 10000 events from Gaussian p.d.f and show distribution

```
// Generate a toy MC set
RooDataSet* data = gauss.generate(x,10000) ;

// Plot PDF
RooPlot* xframe = x.frame() ;
data->plotOn(xframe) ;
xframe->Draw() ;
```

Returned dataset is **unbinned** dataset (like a ROOT TTree with a RooRealVar as branch buffer)

Binning into histogram is performed in `data->plotOn()` call



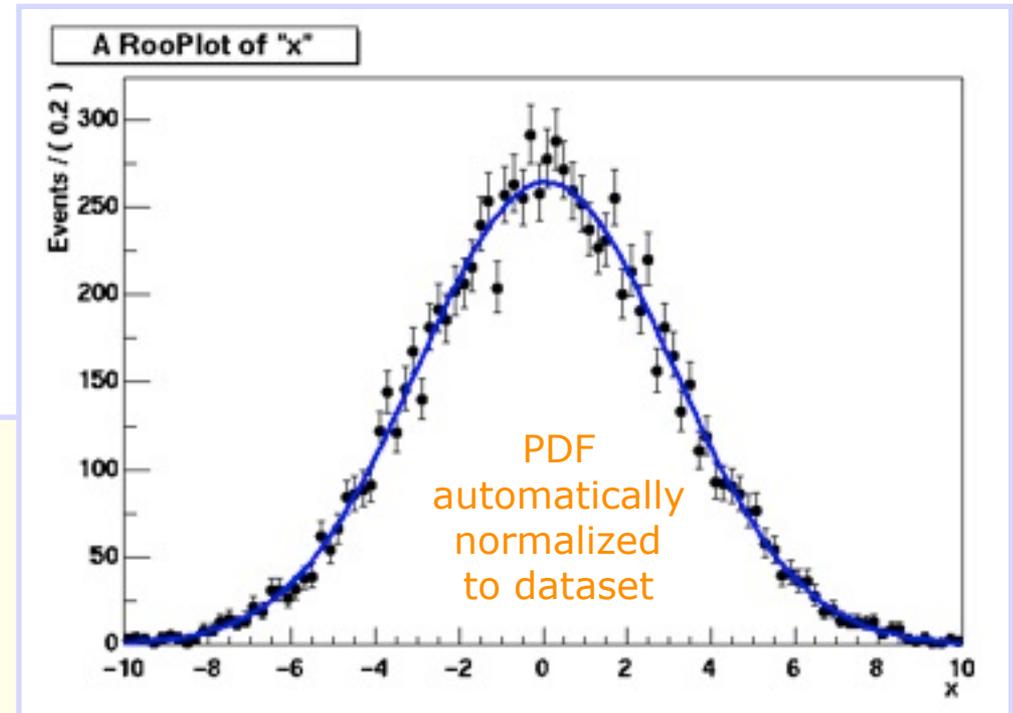
Basics – ML fit of p.d.f to *unbinned* data

demo1.cc

```
// ML fit of gauss to data
gauss.fitTo(*data) ;
(MINUIT printout omitted)

// Parameters if gauss now
// reflect fitted values
mean.Print()
RooRealVar::mean = 0.0172335 +/- 0.0299542
sigma.Print()
RooRealVar::sigma = 2.98094 +/- 0.0217306

// Plot fitted PDF and toy data overlaid
RooPlot* xframe2 = x.frame() ;
data->plotOn(xframe2) ;
gauss.plotOn(xframe2) ;
xframe2->Draw() ;
```

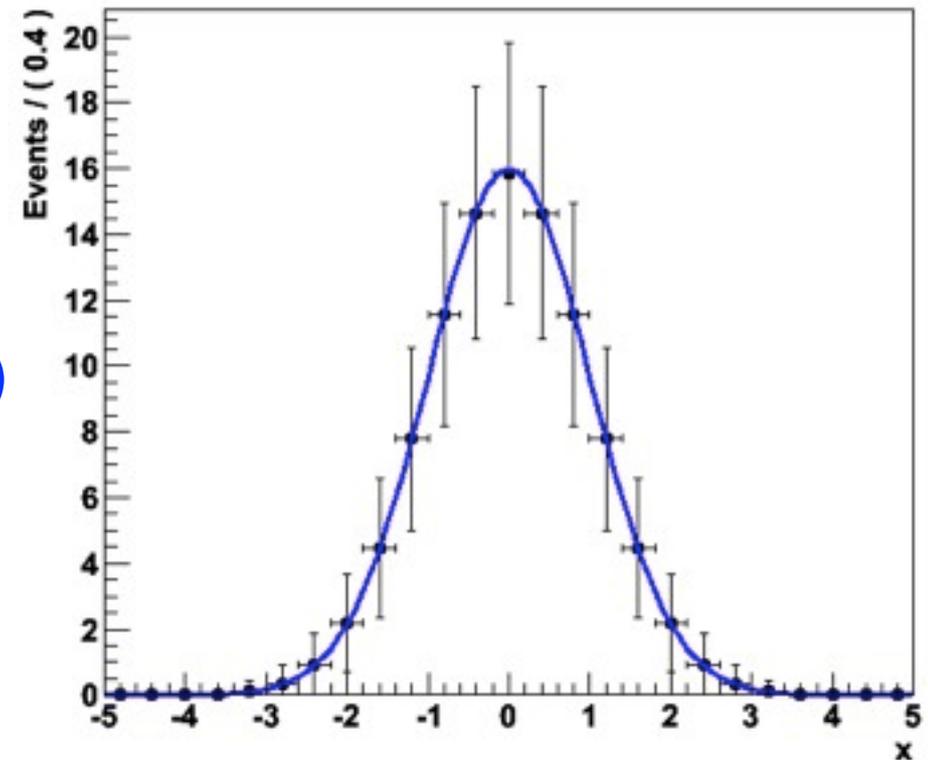


We now appreciate the role of the so-called “Asimov” dataset for estimating the median p-value for a given model

RooFit can now generate an Asimov dataset via the keyword **ExpectedData()**

```
RooWorkspace* wspace = new RooWorkspace("wspace");  
wspace->factory("Gaussian::g(x[-5,5],mu[0],sigma[1])");  
wspace->factory("ExtendPdf::model(g,norm[100])");  
RooAbsPdf* pdf = wspace->pdf("model");  
RooRealVar* x = wspace->var("x");  
  
RooAbsData* AsimovDataSet = pdf->generateBinned(*x, ExpectedData());  
  
RooPlot* frame = wspace->var("x")->frame();  
  
AsimovDataSet->plotOn(frame);  
pdf->plotOn(frame);  
  
frame->Draw();
```

A RooPlot of "x"



Basics – Observables and parameters of Gauss

- Class `RooGaussian` has *no intrinsic notion* or distinction between observables and parameters
- Distinction always implicit in use context with dataset
 - `x` = observable (as it is a variable in the dataset)
 - `mean, sigma` = parameters
- Choice of observables (for unit normalization) always passed to `gauss.getVal()`

```
gauss.getVal() ; // Not normalized (i.e. this is not a pdf)
gauss.getVal(x) ; // Guarantees Int[xmin, xmax] Gauss(x, m, s) dx == 1
gauss.getVal(s) ; // Guarantees Int[smin, smax] Gauss(x, m, s) ds == 1
```

How does it work – Normalization

- Flexible choice of normalization facilitated by explicit normalization step in RooFit p.d.f.s

`gauss.getVal(x)`

$$g(\mathbf{x}; m, s) = \frac{g(x, m, s)}{\int_{x_{\min}}^{x_{\max}} g(x, m, s) dx}$$

`gauss.getVal(s)`

$$g(\mathbf{s}; m, x) = \frac{g(x, m, s)}{\int_{s_{\min}}^{s_{\max}} g(x, m, s) ds}$$

- Supporting class `RooRealIntegral` responsible for calculation of any

$$\int_{\vec{x}_{\min}}^{\vec{x}_{\max}} g(\vec{x}; \vec{p}) d\vec{x}$$

- Negotiation with p.d.f on which (partial) integrals it can internally perform analytically
- Missing partes are supplemented with numerical integration
- Class `RooRealIntegral` can in principle integrate *everything*.

How does it work – Normalization

- A peak in the code of class **RooGaussian**

```
// Raw (unnormalized value) of Gaussian
Double_t RooGaussian::evaluate() const {
    Double_t arg= x - mean;
    return exp(-0.5*arg*arg/(sigma*sigma)) ;
}

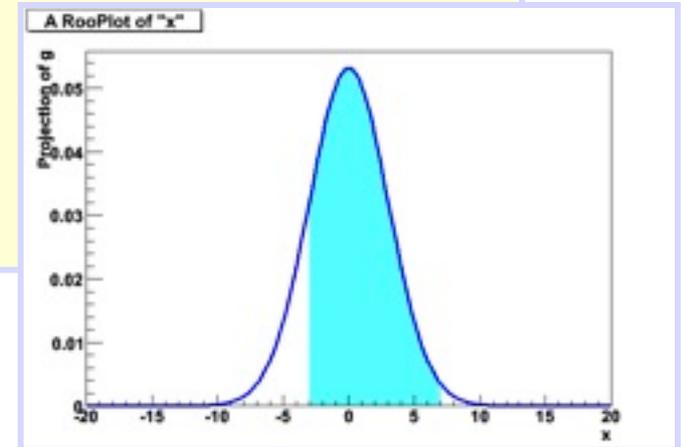
// Advertise that x can be integrated internally
Int_t RooGaussian::getAnalyticalIntegral(RooArgSet& allVars,
    RooArgSet& analVars, const char* /*rangeName*/) const {
    if (matchArgs(allVars,analVars,x)) return 1 ;
    return 0 ;
}

// Implementation of analytical integral over x
Double_t RooGaussian::analyticalIntegral(Int_t code,
    const char* rname) const {
    static const Double_t root2 = sqrt(2.) ;
    static const Double_t rootPiBy2 = sqrt(atan2(0.0,-1.0)/2.0) ;
    Double_t xscale = root2*sigma;
    return rootPiBy2*sigma*(RooMath::erf((x.max(rname)-mean)/xscale)
        -RooMath::erf((x.min(rname)-mean)/xscale)) ;
}
```

Basics – Integrals over p.d.f.s

- It is easy to create an object representing integral over a normalized p.d.f in a sub-range

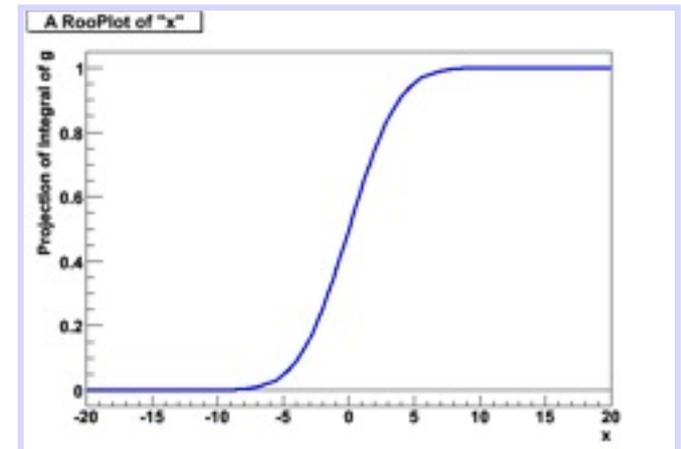
```
x.setRange("sig",-3,7) ;  
RooAbsReal* ig = g.createIntegral(x, NormSet(x), Range("sig")) ;  
cout << ig.getVal() ;  
0.832519  
mean=-1  
cout << ig.getVal() ;  
0.743677
```



- Similarly, one can also request the *cumulative distribution function*

$$C(x) = \int_{x_{\min}}^x F(x') dx'$$

```
RooAbsReal* cdf = gauss.createCdf(x) ;  
RooPlot* frame = x.frame() ;  
cdf->plotOn(frame)->Draw() ;
```



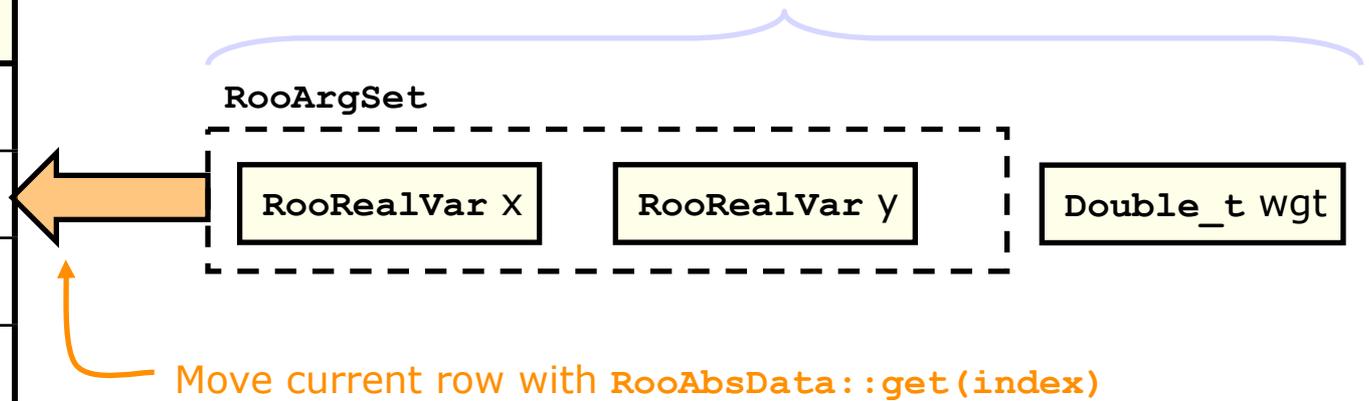
A bit more detail on RooFit datasets

- A dataset is a N-dimensional collection of points
 - With optional weights
 - No limit on number of dimensions
 - Observables continuous (`RooRealVar`) or discrete (`RooCategory`)
- Interface of each dataset is 'current' row
 - Set of RooFit value objects that represent coordinate of current event

(Internal ROOT `TTree`)

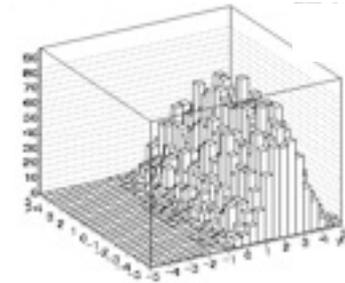
x	Y	wgt
1.0	6.6	1
3.5	11.1	1
2.7	2.2	1
5.2	1.1	1

Current coordinate return by `RooAbsData::get()`
Current weight returned by `RooAbsData::weight()`



Importing data sets

x	y	z
1	3	5
2	4	6
1	3	5
2	4	6



RooDataSet

RooDataHist

RooAbsData

- From ROOT trees

- `RooRealVar` variables are imported from /D /F /I tree branches
- `RooCategory` variables are imported from /I /b tree branches
- **Mapping** between `TTree` branches and dataset variables **by name**: e.g.
`RooRealVar x("x", "x", -10, 10)` imports `TTree` branch "x"

```
RooRealVar x("x", "x", -10, 10) ;  
RooRealVar c("c", "c", 0, 30) ;  
RooDataSet data("data", "data", inputTree, RooArgSet(x, c)) ;
```

- From ROOT `THx` histogram objects

```
RooDataHist bdata1("bdata", "bdata", RooArgList(x), histo1d) ;  
RooDataHist bdata2("bdata", "bdata", RooArgList(x, y), histo2d) ;
```

Start with a **model** for the data, eg. a probability density function for x written $P(x|\mu, \nu)$ that is parametrized by

- ▶ **parameters of interest:** $\mu : m_H, \sigma, \dots$
- ▶ **nuisance parameters:** $\nu : b, JES, \epsilon_b, \dots$

The **likelihood function** is given by

$$L(\mu, \nu) = \prod_{i \in \text{events}} P(x_i | \mu, \nu)$$

I will often refer to the **profile likelihood ratio:**

$$\lambda(\mu) = \frac{L(\mu, \hat{\nu})}{L(\hat{\mu}, \hat{\nu})}$$

Where $\hat{\nu}$ is the maximum likelihood estimator with μ fixed

Remember, $L(\mu, \nu)$ is not a probability.

An early edition to RooFit for the RooStats project was the profile likelihood ratio

$$\lambda(\mu) = \frac{L(\mu, \hat{\nu})}{L(\hat{\mu}, \hat{\nu})}$$

- MINOS error box and profile likelihood give same error for multi-dimensional likelihood
- profiling widens likelihood function, resulting in larger errors than one would get keeping the nuisance parameters fixed

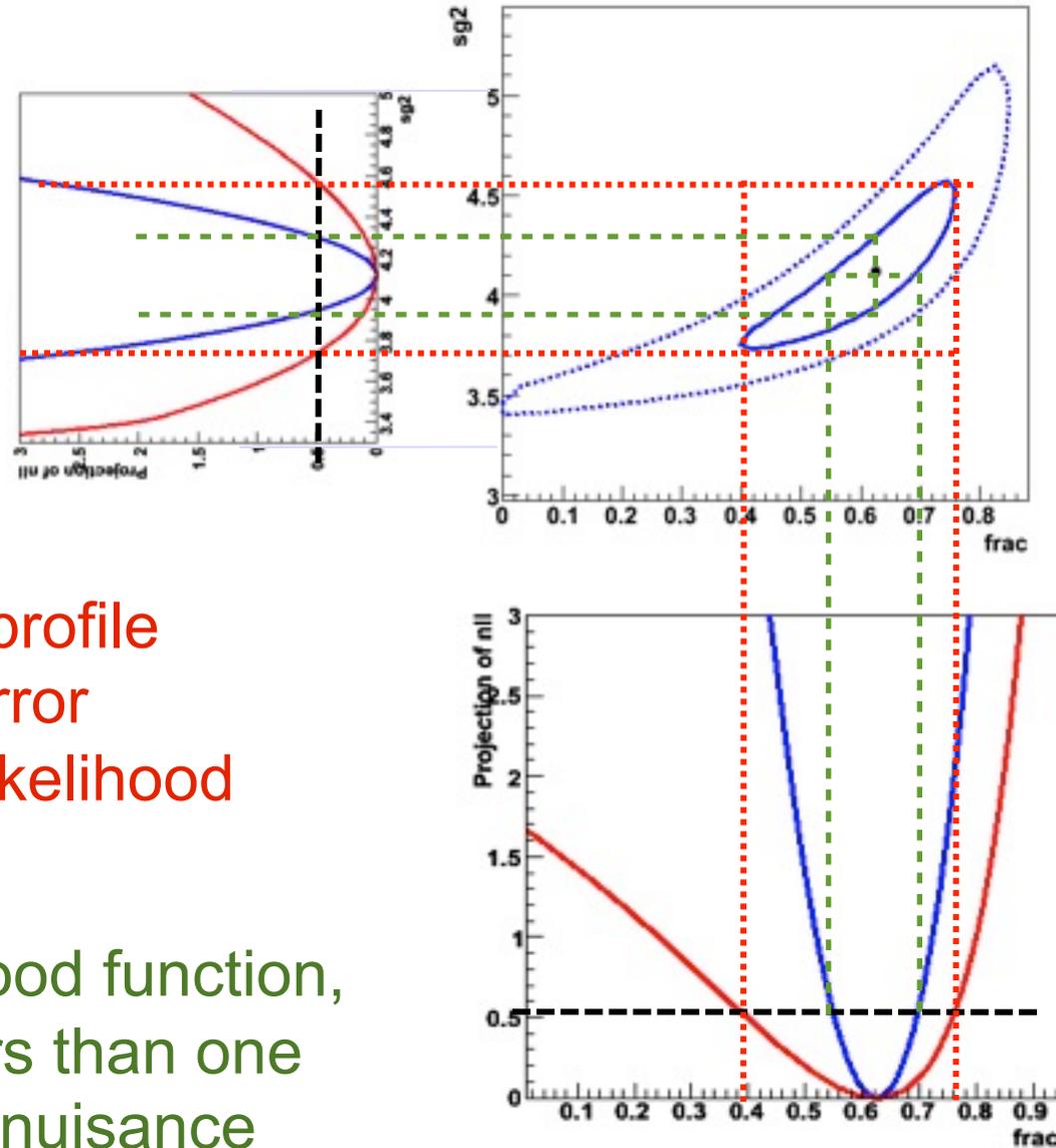


figure by Wouter Verkerke, NIKHEF

It is hard to remember the names of functions and their arguments

- ▶ start a root terminal, type the name of the class, then `::`, and hit `<tab>`.
 - that will show you all of the methods of the class (often too many).
- ▶ If you remember part of the name, you can tab complete
- ▶ once you find the right method, add `(`, and hit `<tab>` again to see list of arguments

```
root [1] RooGaussian::fitTo(<tab>
RooFitResult* fitTo(RooAbsData& data, RooCmdArg arg1 ....
RooFitResult* fitTo(RooAbsData& data, const RooLinkedList& cmdList)

root [1] RooStats::ConfInterval::IsInInterval( <tab>
Bool_t IsInInterval(const RooArgSet&) const
```

Note, RooFit methods usually start with a lower case letter, but ROOT coding convention is to start with an upper case letter

- ▶ expect upper case for methods inherited from core ROOT and all RooStats classes
- ▶ RooStats classes are in the namespace RooStats
 - either add `RooStats::` before class name or using namespace `RooStats;` to C++ file

You may want to code up a more complicated PDF

- ▶ RooFit provides a utility to create a skeleton code:

```
root [0] RooClassFactory::makePdf("MyCustomPdf","observable,mH,parameter")
```

- ▶ Put your code here (normalization is automatic)
 - numeric normalization is automatic or you can add analytical integration

```
MyCustomPdf::MyCustomPdf(const MyCustomPdf& other, const char* name) :  
    RooAbsPdf(other,name),  
    observable("observable",this,other.observable),  
    mH("mH",this,other.mH),  
    parameter("parameter",this,other.parameter)  
{  
}
```

```
Double_t MyCustomPdf::evaluate() const  
{  
    // ENTER EXPRESSION IN TERMS OF VARIABLE ARGUMENTS HERE  
    return 1.0 ;  
}
```

Put your code here →



Fundamental Interfaces for RooStats

Goal: Standardize interface for major statistical procedures so that they can work on an arbitrary RooFit model & dataset and handle many parameters of interest and nuisance parameters.

- ▶ RooFit already has an interface for PDFs of arbitrary complexity, but often one needs to specify some additional information to know how to use the PDF.
 - this motivated the class **ModelConfig**, which specifies:
 - what are the observables,
 - what are the parameters of interest,
 - is there an additional prior on the parameters (for Bayesian analysis)
 - is the model conditional on other observables (eg. event-by-event errors)
- ▶ We also need a way to package this information, the model, and all that it depends on so we can pass it too the tools. Similar requirements for combining multiple measurements.
 - this motivated the class **RooWorkspace**, which
 - owns the model and all that it depends upon
 - can read/write all information to a ROOT file
 - (interfaces to low-level factory... more later)



```
namespace RooStats {  
class ModelConfig : public TNamed {  
    // specify meaning of variables: observable, parameter, etc.  
    const RooArgSet * GetObservables() ;  
    const RooArgSet * GetParametersOfInterest() ;  
    const RooArgSet * GetNuisanceParameters() ;  
    const RooArgSet * GetConditionalObservables() ;  
  
    // specify the objective pdf and the Bayesian prior separately  
    RooAbsPdf * GetPdf() ;  
    RooAbsPdf * GetPriorPdf() ;
```

```
    // specify value of parameters for a particular hypothesis
```

```
    const RooArgSet * GetSnapshot () ;
```

```
    // get the associated workspace
```

```
    const RooWorkspace * GetWS() ;
```

```
    // corresponding Set methods
```

```
    virtual void SetPdf(RooAbsPdf& pdf) ;
```

```
    ...
```

```
class RooWorkspace : public TNamed {
```

```
    // import functions
```

```
    Bool_t import(const RooAbsArg& arg, ....) ;
```

```
    // use a low-level factory to create and edit objects in the workspace
```

```
    RooAbsArg* factory(const char* expr) ;
```

```
    // Accessor functions
```

```
    RooAbsPdf* pdf(const char* name) ;
```

```
    RooAbsData* data(const char* name) ;
```

```
    RooRealVar* var(const char* name) ;
```

```
    const RooArgSet* set(const char* name) ;
```

```
    // Write this workspace to a ROOT file
```

```
    writeToFile(const char* fileName, Bool_t recreate=kTRUE) ;
```

```
    // import class code for custom classes (eg. custom PDFs and functions)
```

```
    autoImportClassCode(Bool_t flag) ;
```

Two ways to create & import a model into the workspace

```
// Make observable and parameters
RooRealVar x("x","x", 150,100,200);
RooRealVar mu("mu","#mu", 150,130,170);
RooRealVar sigma("sigma","#sigma", 5,0,20);

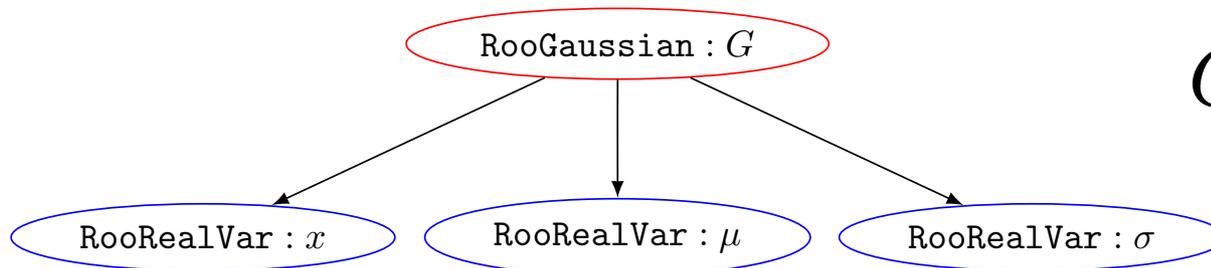
// make a simple model
RooGaussian G("G","G",x, mu, sigma);

// make graph to represent model (using GraphViz and latex formatting)
G.graphVizTree("GaussianModel.dot", ":",true, true);
```

or via

```
// use the workspace factory to do the same thing
RooWorkspace wspace("wspace");
wspace.factory("Gaussian::G(x[150,100,200],mu[150,130,170], sigma[5,0,20])");
wspace.pdf("G")->graphVizTree("GaussianModel_factory.dot", ":",true,true);
```

which both produce a graphical representation (in the form of Bell labs “graphviz” format)



$$G(x|\mu, \sigma)$$

An example with ModelConfig

Here we show use of the Workspace factory to create a model, and use of ModelConfig to specify what we will need for the statistical tools

Create a the pdf $G(x|\mu,1)$ and the variables x , μ , σ using the factory syntax

Create a new workspace

Create a new ModelConfig

```
// make a simple model via the workspace factory
RootWorkspace* wspace = new RootWorkspace();
wspace->factory("Gaussian::normal(x[-10,10],mu[-1,1],sigma[1]))");
wspace->defineSet("poi", "mu");
wspace->defineSet("obs", "x");

// specify components of model for statistical tools
ModelConfig* modelConfig = new ModelConfig("G(x|mu,1)");
modelConfig->SetWorkspace(*wspace);
modelConfig->SetPdf( *wspace->pdf("normal") );
modelConfig->SetParametersOfInterest( *wspace->set("poi") );
modelConfig->SetObservables( *wspace->set("obs") );

// create a toy dataset
RootDataSet* data = wspace->pdf("normal")->generate(*wspace->set("obs"),100);
```

Define parameter sets for observables and parameters of interest

Specify workspace that holds pdf, parameters of interest, observables, ...

... and we generate a toy dataset with 100 measurements of the observables (x) (note, the data is NOT part of the ModelConfig)

Using the Workspace concept

- Up to now, to share with colleagues need to distribute *both* a data file and a ROOT macro that builds the RooFit p.d.f
- Now add the **Workspace** – Persistent container for both data and functions

*Create the
workspace
container object*

*Use standard
ROOT I/O
to store wspace*

```
RooAbsPdf& g ; // from preceding example  
RooAbsData& d ; // from preceding example
```

```
RooWorkspace w("w","my workspace") ;  
w.import(g) ;  
w.import(d) ;
```

```
w.writeToFile("myresult.root") ;
```

- Both data and p.d.f. are now stored in file!

A look at the workspace

- What is in the workspace?

w.Print() ;

RooWorkspace(w) my workspace contents

variables

(x,m,s)  RooRealVar* x = w.get("x") ;

p.d.f.s

RooGaussian::g[x=x mean=m sigma=s] = 0  RooAbsPdf* g = w.pdf("g") ;

datasets

RooDataSet::d(x)  RooAbsData* d = w.data("d") ;

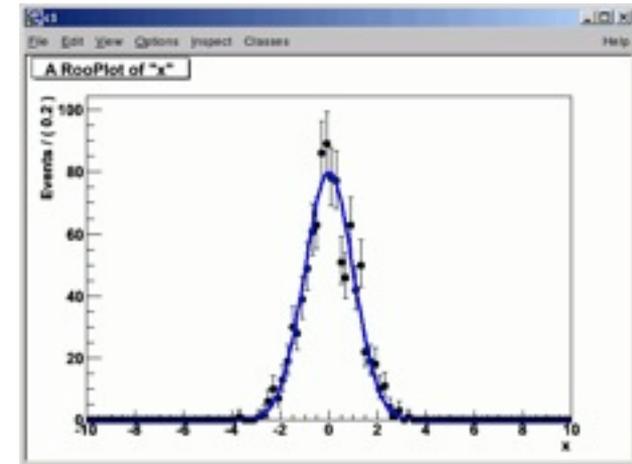
Using persisted p.d.f.s.

- Using both model & p.d.f from file

```
TFile f("myresults.root");  
RooWorkspace* w = f.Get("w");
```

*Make plot
of data
and p.d.f*

```
RooPlot* xframe = w->var("x")->frame();  
w->data("d")->plotOn(xframe);  
w->pdf("g")->plotOn(xframe);
```

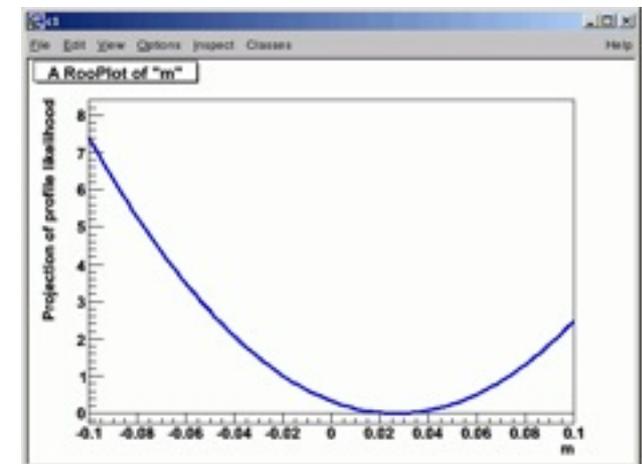


*Construct
likelihood
& profile LH*

```
RooNLLVar nll("nll","nll",*w->pdf("g"),*w->data("d"));  
RooProfileLL pll("pll","pll", nll,*w->var("m"));
```

*Draw
profile LH*

```
RooPlot* mframe = w->var("m")->frame(-1,1);  
pll.plotOn(mframe);  
mframe->Draw()
```



- Note that above code is independent of actual p.d.f in file → e.g. full Higgs combination would work with identical code**

We try to keep a clean separation between the tools and the results

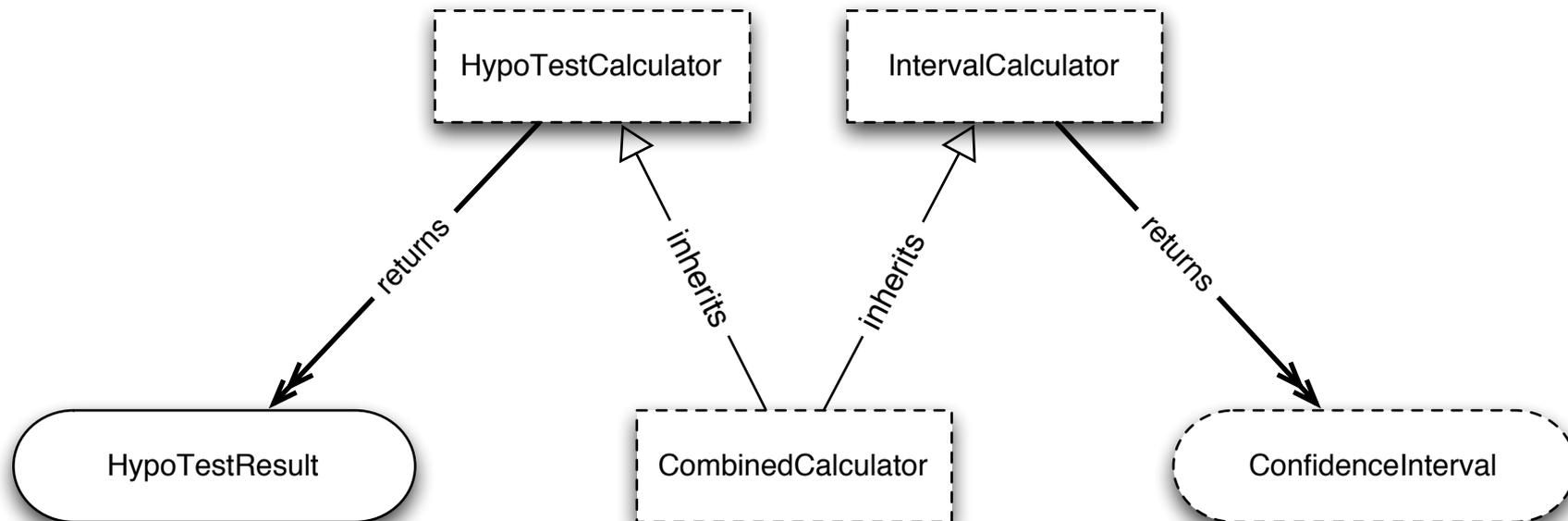
- results in general can be saved into a ROOT file

Two main classes of tools: Hypothesis Tests & Confidence Intervals

- each has an abstract interface for tools and results

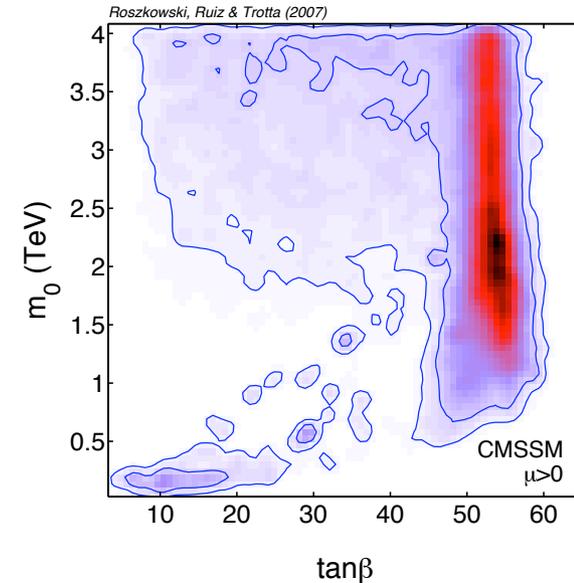
Let's think about the interfaces,

- what do they need to know?
- what do they provide?



ConfInterval is the interface for confidence intervals in RooStats.

- ▶ There are many types of intervals: from a simple range $[a,b]$ in 1-D to a complicated, disconnected region in multiple dimensions.
 - So the common interface is simply to ask the interval if a given point **IsInInterval**.
 - The Interval also knows what **ConfidenceLevel** it was constructed at and the space of parameters for which it was constructed.
- ▶ Any tool inheriting from **IntervalCalculator** can return a **ConfInterval**.



```
namespace RooStats {
  class ConfInterval : public TNamed {

    // check if given point is in the interval
    virtual Bool_t IsInInterval(const RooArgSet&) const = 0;

    // used to set confidence level. Keep pure virtual
    virtual void SetConfidenceLevel(Double_t cl) = 0;

    // return confidence level
    virtual Double_t ConfidenceLevel() const = 0;

    // return list of parameters of interest defining this interval (return a ↵
    // new cloned list)
    virtual RooArgSet* GetParameters() const = 0;

  }
}
```



IntervalCalculator is the interface for a tools which produce **ConfIntervals**.

- ▶ After configuring the calculator, one only needs to ask **GetInterval**, which will return a **ConfInterval** pointer (the user takes ownership of new interval).
- ▶ assumes that any interval calculator can be configured by specifying the:
 - single model configuration (one model over a space of parameters of interest),
 - data set,
 - confidence level

```
namespace RooStats {
  class IntervalCalculator {

    // Main interface to get a ConfInterval, pure virtual
    virtual ConfInterval* GetInterval() const = 0;

    // Get the size of the test (eg. rate of Type I error)
    virtual Double_t Size() const = 0;

    // Get the Confidence level for the test
    virtual Double_t ConfidenceLevel() const = 0;

    // Set the DataSet ( add to the the workspace if not already there ?)
    virtual void SetData(RooAbsData&) = 0;

    // Set the Model
    virtual void SetModel(const ModelConfig & /* model */) = 0;

    // set the size of the test (rate of Type I error) ( e.g. 0.05 for a 95% ←
    // Confidence Interval)
    virtual void SetTestSize(Double_t size) = 0;

    // set the confidence level for the interval (e.g. 0.95 for a 95% ←
    // Confidence Interval)
    virtual void SetConfidenceLevel(Double_t cl) = 0;

  };
}
```

HypoTestResult is the class that holds the result of a Hypothesis Test

- ▶ **Very simple class:**
 - stores p-value for the null and alternate hypotheses as calculated by a **HypoTestCalculator**
 - equivalently, Gaussian significance, CLb and CLs+b
 - (note can calculate CLs = CLs+b / CLb, which is used within physics, but is not a probability)
- ▶ **Note, HypoTestResult is a concrete class, not an interface.**

```
namespace RooStats {
  class HypoTestResult : public TNamed {

    // Return p-value for null hypothesis
    virtual Double_t NullPValue() const {return fNullPValue;}

    // Return p-value for alternate hypothesis
    virtual Double_t AlternatePValue() const {return fAlternatePValue;}

    // Convert NullPValue into a "confidence level"
    virtual Double_t CLb() const {return 1.-NullPValue();}

    // Convert AlternatePValue into a "confidence level"
    virtual Double_t CLsplusb() const {return AlternatePValue();}

    // CLs is simply CLs+b/CLb (not a method, but a quantity)
    virtual Double_t CLs() const ;

    // familiar name for the Null p-value in terms of 1-sided Gaussian ←
    // significance
    virtual Double_t Significance() const;

  };
}
```



HypoTestCalculator is the interface tools that produce HypoTestResults

- ▶ The interface currently assumes that any interval calculator can be configured by specifying:
 - a model configuration for the null,
 - a model configuration for the alternate (often a totally different PDF),
 - a data set,
- ▶ After configuring the calculator, one simply calls **GetHypoTest**, which will return a HypoTestResult pointer (the user takes ownership of new object).

```
namespace RooStats {
  class HypoTestCalculator {

    // main interface to get a HypoTestResult, pure virtual
    virtual HypoTestResult* GetHypoTest() const = 0;

    // Set the model for the null hypothesis
    virtual void SetNullModel(const ModelConfig& model) = 0;

    // Set the model for the alternate hypothesis
    virtual void SetAlternateModel(const ModelConfig& model) = 0;

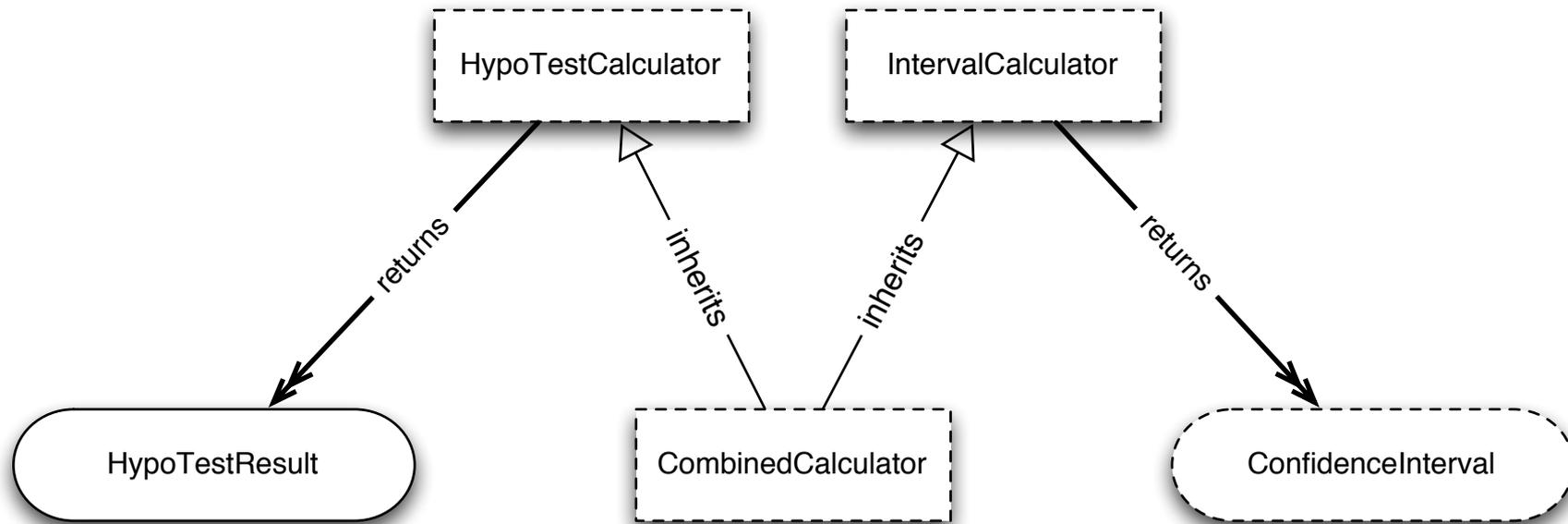
    // Set the DataSet
    virtual void SetData(RooAbsData& data) = 0;

    // Set a common model for both the null and alternate
    virtual void SetCommonModel(const ModelConfig& model) ;

  }
}
```

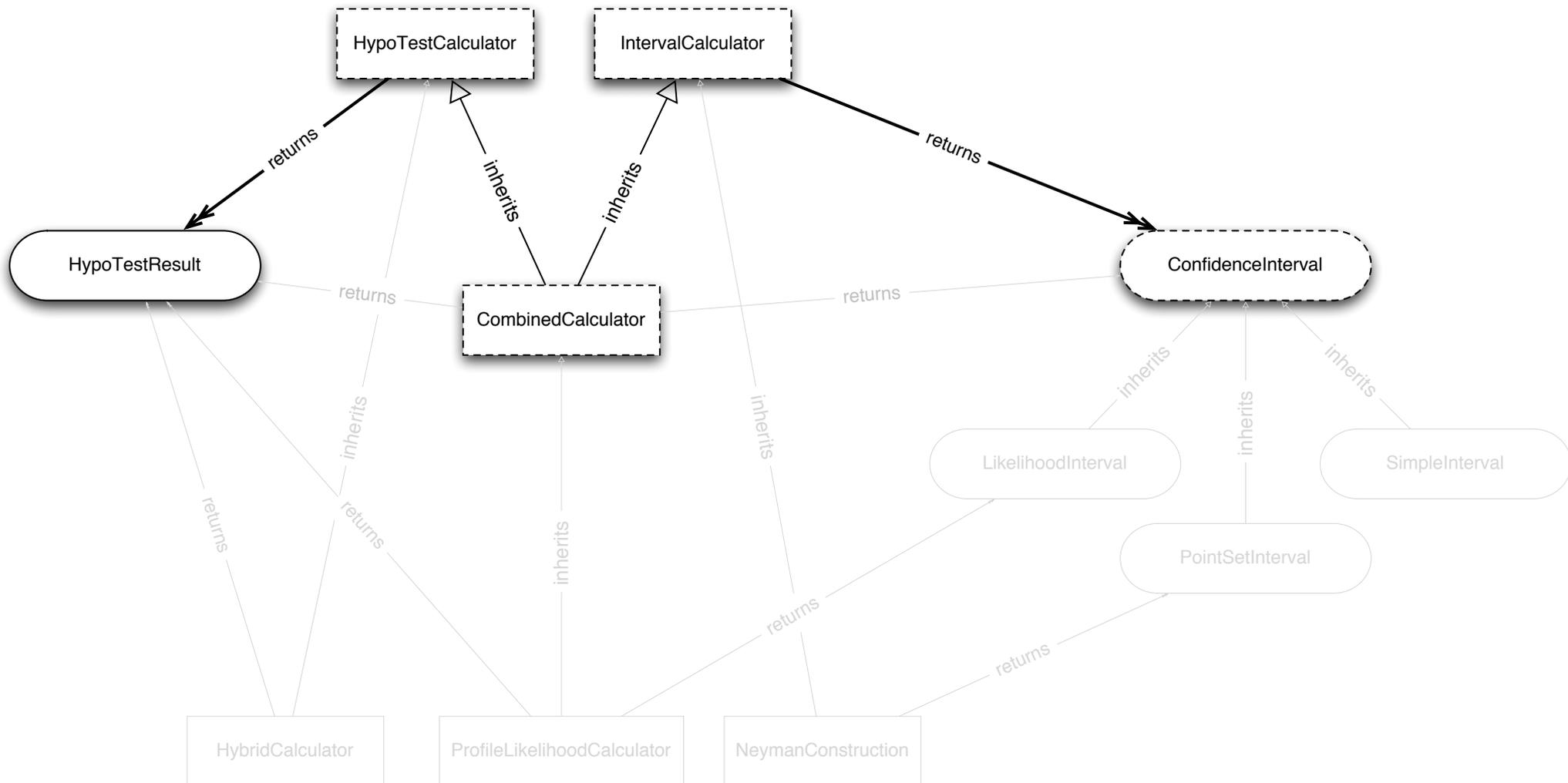
CombinedCalculator is the interface tools that can do both hypothesis tests and produce confidence intervals

- ▶ In this case, the null and alternate models share the same pdf, and are specified by specific settings for the parameters of interest



We have tried to unify the way that one specifies the statistical problem, with RooAbsData and ModelConfig

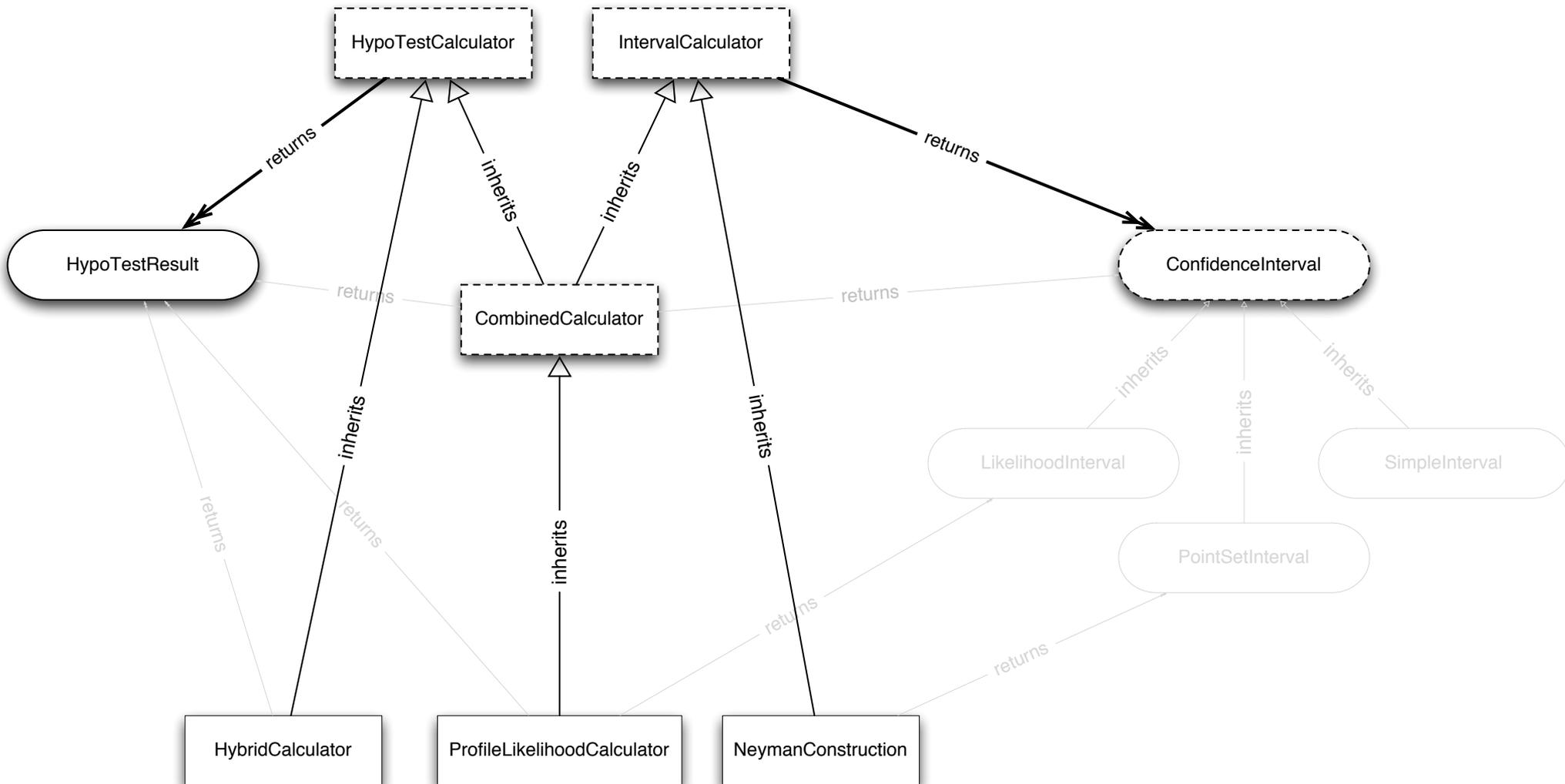
- any additional configuration of the tools should be seen as configuration of the method itself, and several of the tools are highly configurable



HybridCalculator is an example of a **HypoTestCalculator**, it returns a **HypoTestResult**

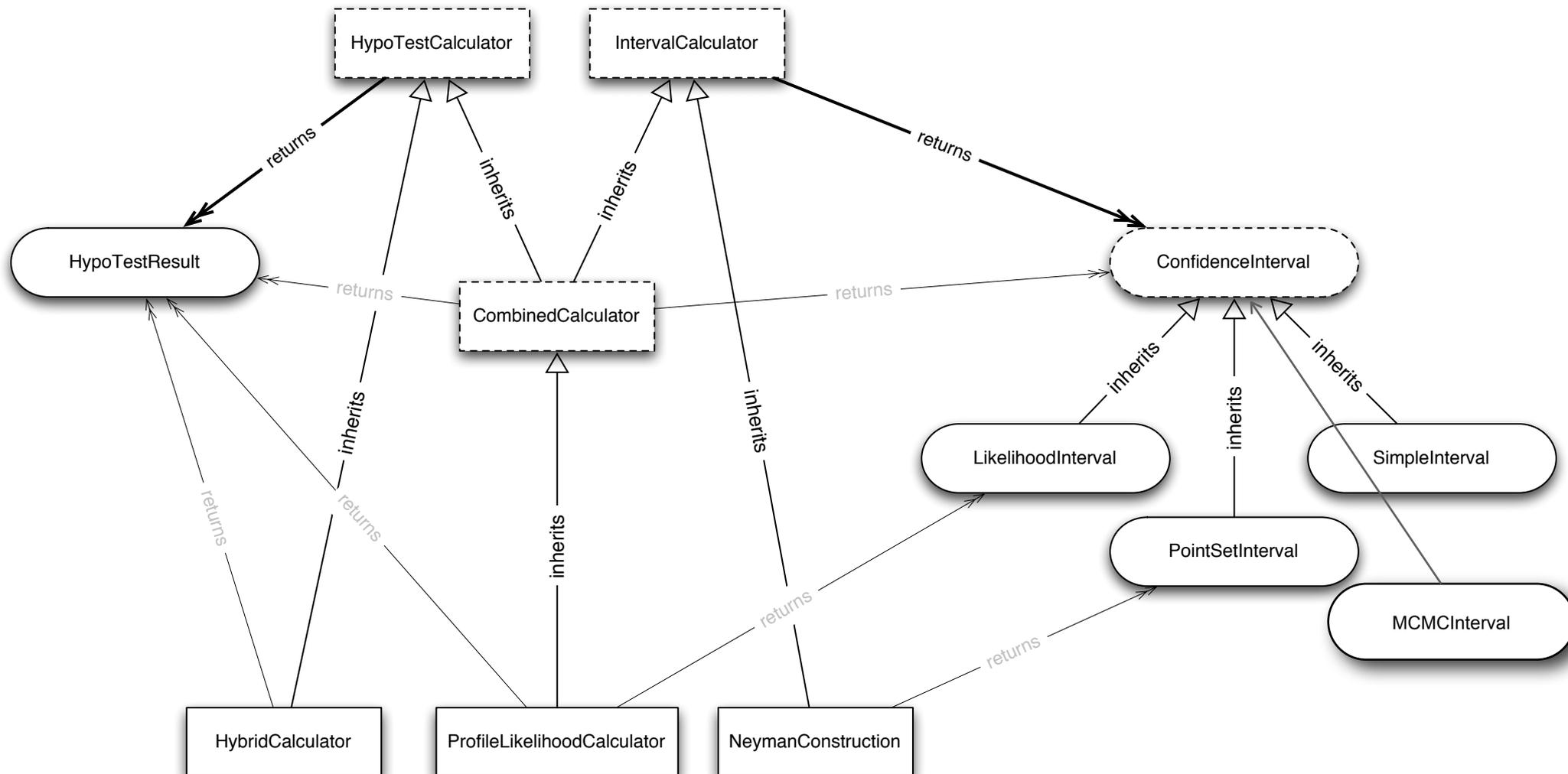
ProfileLikelihoodCalculator is an example of a **CombinedCalculator**, it can return either a **HypoTestResult** or a **ConfInterval**

NeymanConstruction is an example of an **IntervalCalculator**, it returns a **ConfInterval**



The **ProfileLikelihoodCalculator** returns a **LikelihoodInterval**, which is a concrete implementation of **ConfInterval** (based on contours of likelihood function)

The **NeymanConstruction** returns a **PointSetInterval**, which is a different concrete implementation (based on scanning points in the parameter space)





HypoTestCalculators

- ▶ **HybridCalculator**
 - hybrid Bayes-frequentist calculation (marginalize nuisance parameters)
- ▶ **ProfileLikelihoodCalculator**
 - the method of MINUIT/MINOS, based on Wilks's theorem

IntervalCalculators

- ▶ **ProfileLikelihoodCalculator**
 - the method of MINUIT/MINOS, based on Wilks's theorem
- ▶ **NeymanConstruction**
 - general purpose Neyman Construction class, highly configurable: choice of TestStatistic, TestStatSampler (defines ensemble/conditioning), integration boundary (upper, lower, central limits), and parameter points to scan
- ▶ **FeldmanCousins**
 - specific configuration of NeymanConstruction for Feldman-Cousins (generalized for nuisance parameters)
- ▶ **MCMCCalculator**
 - Bayesian Markov Chain Monte Carlo (Metropolis Hastings), proposal function is highly customizable
- ▶ **BayesianCalculator**
 - Bayesian posterior calculated via numeric integration routines, currently only supports one parameter
- ▶ **HypoTestInverter**
 - adapter any HypoTestCalculator and forms an IntervalCalculator



An Example Confidence Interval

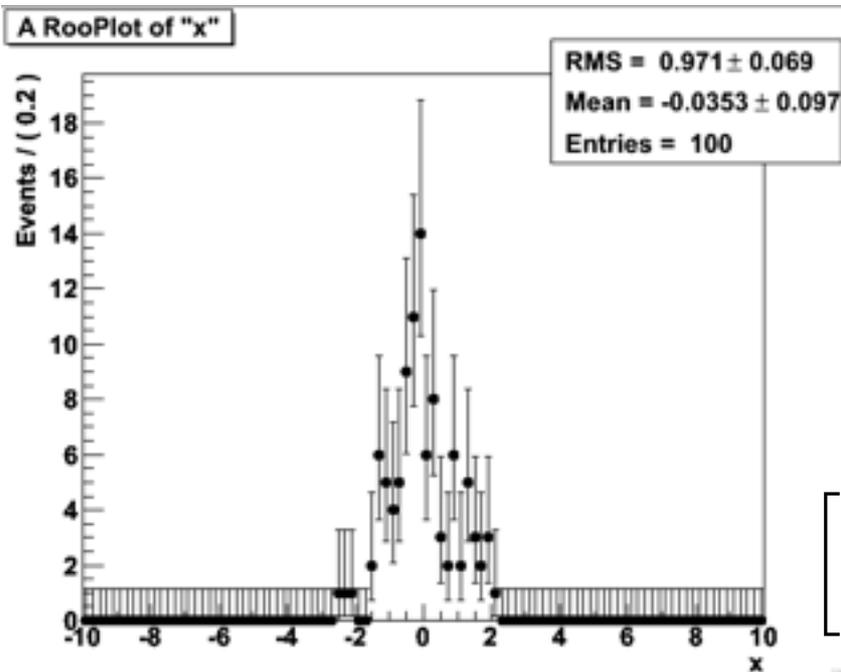
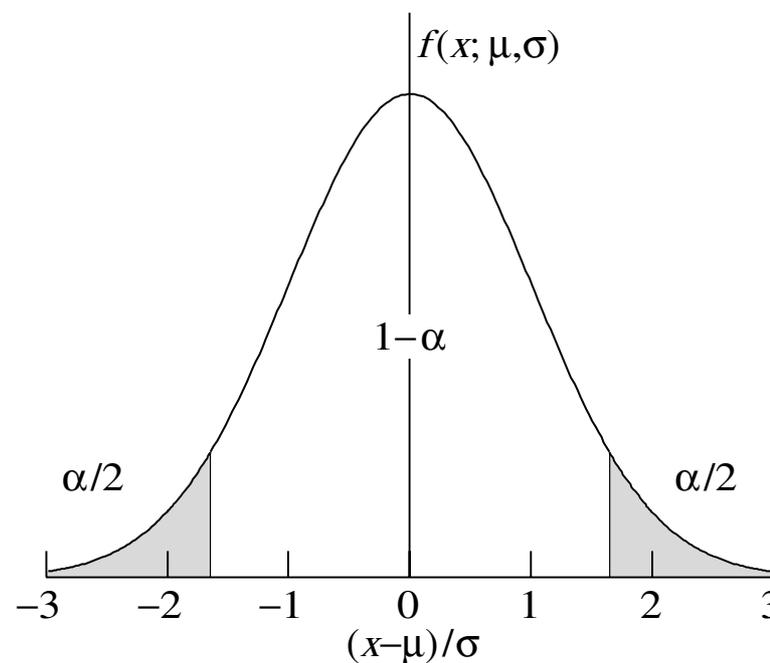
The example problem

Let's consider an example where we know the answer:

- ▶ 95% CL interval on the mean of a Gaussian with $\sigma=1$
- ▶ generate a toy dataset with $N = 100$
- ▶ Look up in PDG what we should expect
- ▶ $\sigma = 1, \delta = 1.96, N = 100$

Table 32.1: Area of the tails α outside $\pm\delta$ from the mean of a Gaussian distribution.

α	δ	α	δ
0.3173	1σ	0.2	1.28σ
4.55×10^{-2}	2σ	0.1	1.64σ
2.7×10^{-3}	3σ	0.05	1.96σ
6.3×10^{-5}	4σ	0.01	2.58σ
5.7×10^{-7}	5σ	0.001	3.29σ
2.0×10^{-9}	6σ	10^{-4}	3.89σ



$$\left[\bar{x} - \frac{\delta}{\sqrt{N}}, \bar{x} + \frac{\delta}{\sqrt{N}} \right] \xrightarrow[\bar{x} = -0.035]{\text{expected interval}} [-0.23, 0.16]$$

Here we show use of the Workspace factory to create a model, and use of ModelConfig to specify what we will need for the statistical tools

Create a the pdf $G(x|\mu,1)$ and the variables x , μ , σ using the factory syntax

Create a new workspace

Create a new ModelConfig

Define parameter sets for observables and parameters of interest

```
// make a simple model via the workspace factory
RootWorkspace* wspace = new RootWorkspace();
wspace->factory("Gaussian::normal(x[-10,10],mu[-1,1],sigma[1]))");
wspace->defineSet("poi","mu");
wspace->defineSet("obs","x");

// specify components of model for statistical tools
ModelConfig* modelConfig = new ModelConfig("G(x|mu,1)");
modelConfig->SetWorkspace(*wspace);
modelConfig->SetPdf( *wspace->pdf("normal") );
modelConfig->SetParametersOfInterest( *wspace->set("poi") );
modelConfig->SetObservables( *wspace->set("obs") );

// create a toy dataset
RootDataSet* data = wspace->pdf("normal")->generate(*wspace->set("obs"),100);
```

Specify workspace that holds pdf, parameters of interest, observables, ...

... and we generate a toy dataset with 100 measurements of the observables (x)

Once one has the model and the data, creating the interval is quite easy!

Create a **ProfileLikelihoodCalculator**, constructor needs data and the model config

Want a 95% CL

```
// set confidence level
double confidenceLevel = 0.95;

// example use profile likelihood calculator
ProfileLikelihoodCalculator plc(*data, *modelConfig);
plc.SetConfidenceLevel( confidenceLevel);
LikelihoodInterval* plInt = plc.GetInterval();
```

Obtain the resulting interval

(Note, here we know the return type is a **LikelihoodInterval**, in general one could use the interface **ConfInterval**. Note, you take ownership of plInt.)

Use the interval

```
cout << "plc interval is [ " <<
plInt->LowerLimit(*mu) << ", " <<
plInt->UpperLimit(*mu) << "]" << endl;
mu->setVal(0);
cout << "is mu=0 in the interval? " << plInt->IsInInterval(*mu) << endl;
```

```
expected interval is [-0.231277, 0.160716]
plc interval is      [-0.231277, 0.160716]
is mu=0 in the interval? 1
```

!

Example using Feldman-Cousins

```
// example use of Feldman-Cousins
FeldmanCousins fc(*data, *modelConfig);
fc.SetConfidenceLevel( confidenceLevel);

// special options
fc.SetNBins(200); // number of points to test per parameter
fc.UseAdaptiveSampling(true); // make it go faster

PointSetInterval* interval = fc.GetInterval();

std::cout << "fc interval is [" <<
  interval->LowerLimit(*mu) << " , " <<
  interval->UpperLimit(*mu) << "]" << endl;
```

```
expected interval is [-0.231277, 0.160716]
plc interval is      [-0.231277, 0.160716]
fc interval is       [-0.215    , 0.165    ]
Real time 0:00:36, CP time 34.900
```

By default, the FeldmanCousins calculator only samples 10 points per parameter, (pretty fast)

- ▶ modify this with SetNBins

The default number of samples is $50/(\text{type I error rate})$

- ▶ For 95% CL, 1000 toys/point
- ▶ for 200 parameter points, that's 200,000 generations of datasets and MINUIT minimizations! About 15 min.

The AdaptiveSampling algorithm will use fewer toys for obvious points and more near boundary

- ▶ reduces it to 2 min!
- ▶ will explain algorithm later on

Note F-C interval is consistent within step-size for the upper limit, off by one step in lower limit... likely a statistical fluctuation in ToyMC.

```
// example use of BayesianCalculator
// now we also need to specify a prior in the ModelConfig
wspace->factory("Uniform::prior(mu)");
modelConfig->SetPriorPdf(*wspace->pdf("prior"));
```

```
// example usage of BayesianCalculator
BayesianCalculator bc(*data, *modelConfig);
bc.SetConfidenceLevel( confidenceLevel);
SimpleInterval* bcInt = bc.GetInterval();
```

The BayesianCalculator requires an additional ingredient beyond what is currently in our ModelConfig

▶ it needs a prior for mu

Once that is specified in the ModelConfig, use of the BayesianCalculator is the same as the other tools

▶ currently this tool is restricted to one parameter, waiting for some additional support in underlying RooFit integration infrastructure.

```
expected interval is [-0.231277, 0.160716]
plc interval is      [-0.231277, 0.160716]
fc interval is       [-0.215    , 0.165    ]
bc interval is       [-0.232274, 0.159713]
```

Note bayesian interval is based on numerical integration instead of ToyMC. Accuracy of answer depends on accuracy specified in numerical integration, can be configured.

```
// example use of MCMCInterval
MCMCCalculator mc(*data, *modelConfig);
mc.SetConfidenceLevel( confidenceLevel);

// special options
mc.SetNumBins(1000); // bins used internally for representing posterior
mc.SetNumBurnInSteps(500); // first N steps to be ignored as burn-in
mc.SetNumIters(100000);

MCMCInterval* mcInt = mc.GetInterval();
```

```
expected interval is [-0.231277, 0.160716]
plc interval is      [-0.231277, 0.160716]
fc interval is       [-0.215    , 0.165    ]
bc interval is       [-0.232274, 0.159713]
mc interval is       [-0.227    , 0.157    ]
```

The MCMCCalculator requires the same ingredients as the Bayesian Calculator, but uses Markov Chain Monte Carlo to calculate the posterior

This class works is very configurable, and will be described in more detail later.

- here we set the binning resolution for the parameter of interest, the number of “burn in” steps, the number of iterations to run the MCMC



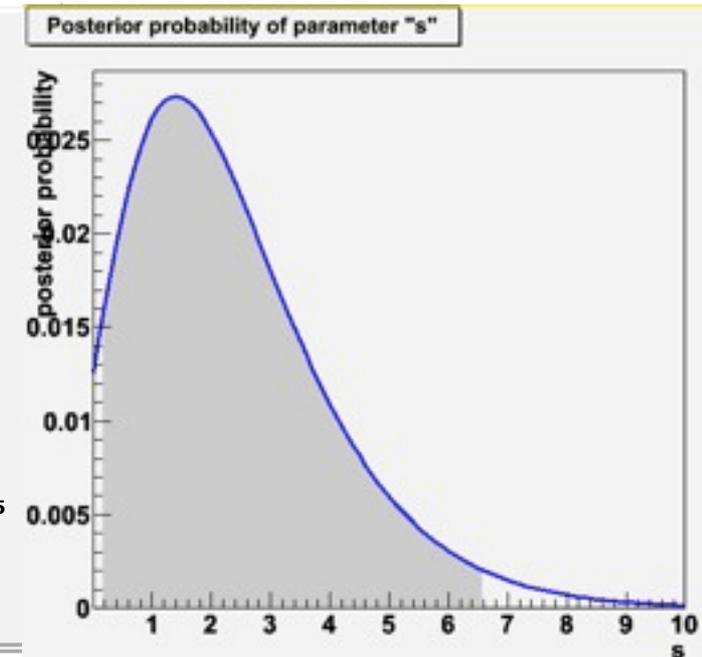
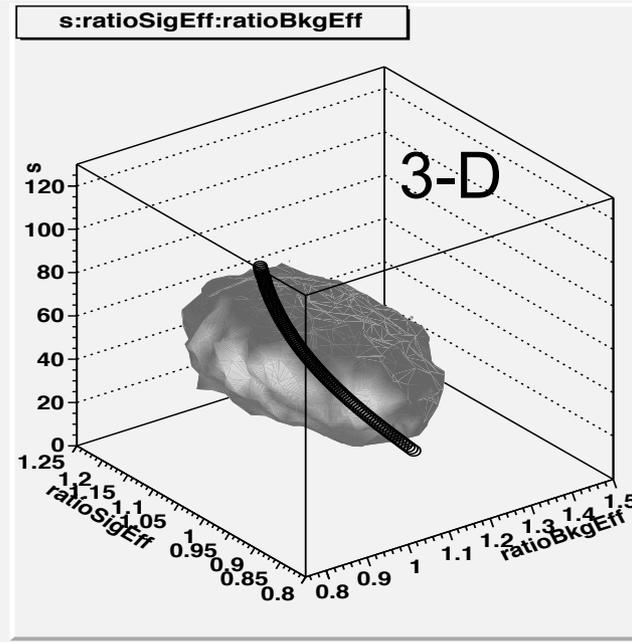
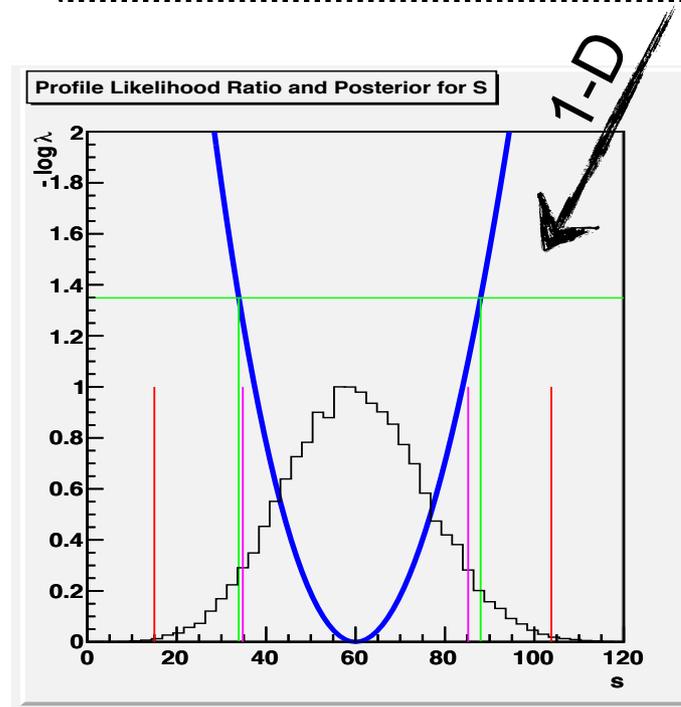
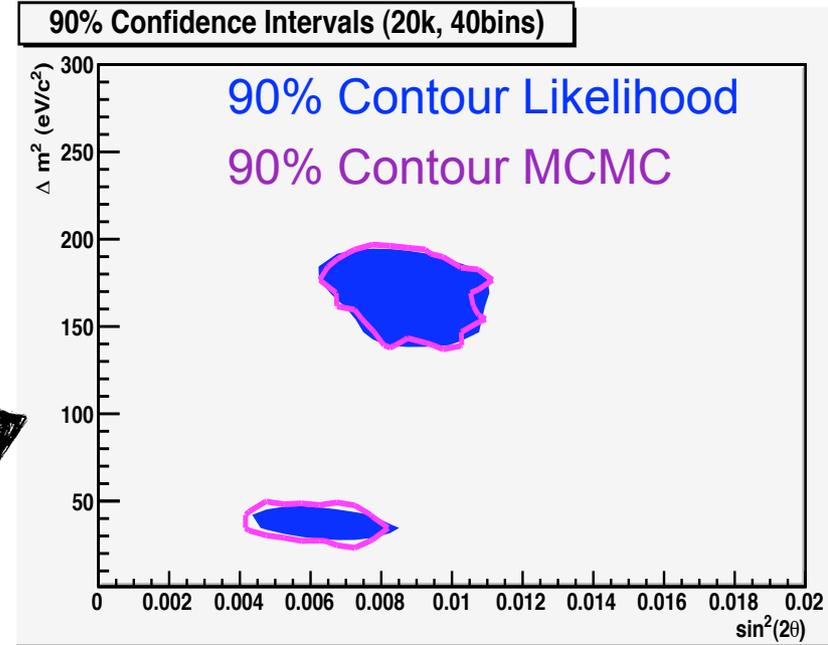
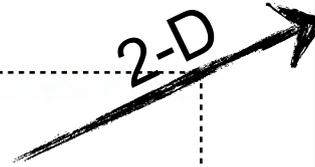
Visualizing the Results (under development)

We are making progress on visualization

- Different ways of visualizing in 1,2,3-Dim.
- natural axis is different for different objects (likelihood, posterior, etc)
- visualization tools are separated from results classes

```
LikelihoodIntervalPlot plot(plInt);
plot.Draw();

MCMCIntervalPlot* mcPlot = new MCMCIntervalPlot(*mcInt);
```





An Example Hypothesis Test

Let's consider an example that has been studied recently: the “on/off” problem

- it consists of two number counting measurements:
 - a “main measurement” N that are signal candidates
 - a sideband measurement, where one observes b_{est} events and expects that to be Poisson-distributed around the true, unknown b (our nuisance parameter)

$$P(N, b_{\text{est}} | s, b) = \underbrace{\text{Pois}(N | s + b)}_{\text{total model}} \underbrace{\text{Pois}(b_{\text{est}} | b)}_{\text{sideband}}$$

total model main measurement sideband

This problem has a known frequentist solution based on the Binomial distribution

- The resulting significance is called Z_{Bi} and it is known analytically
- RooStats has a little utility for this problem called NumberCountingUtils
- In this toy example, let's say that $b_{\text{est}} = 100$ and $N = 144$

```
double tau = 1; // bkgEstimate directly estimates bkg in main measurement
```

```
double Z_Bi = RooStats::NumberCountingUtils::BinomialWithTauObsZ(nObs, bkgEst, tau);  
std::cout << "Z_Bi significance estimation: " << Z_Bi << std::endl;
```

```
Z_Bi significance estimation: 2.75905
```

Creating the null & alternate model

```
// use a RooWorkspace to store the pdf, etc...  
RooWorkspace* myWS = new RooWorkspace("myWS");
```

```
// Observable  
myWS->factory("x[0,0,1]");
```

```
// Pdf in observable,  
myWS->factory("Uniform::sigPdf(x)");  
myWS->factory("Uniform::bkgPdf(x)");  
myWS->factory("SUM::model(S[50,0,100]*sigPdf,B[100, 0, 200]*bkgPdf)");  
// Pdf of a sideband measurement, that estimated 100 background events  
myWS->factory("Poisson::sideband(bkgEstimate[100], B)");  
myWS->factory("PROD::modelWithConstraint(model, sideband)");
```

Here we could choose a dummy variable x with a uniform distribution for signal and background to model the number counting problem

- ▶ strange for number counting, but more obvious to generalize if sig, bkg have different shapes

In contrast, we imagine a sideband measurement for the background done by your colleague estimated $b_{\text{est}}=100$

- ▶ Total model is product of the two

Here we both the alternate & null need a ModelConfig instance

- ▶ only difference is the value of the parameter S
- ▶ these are saved as "parameter snapshots"
- ▶ both ModelConfigs will be imported into workspace

```
// create model config for alternate  
ModelConfig* alternateModel = new ModelConfig("alternate");  
alternateModel->SetWorkspace( *myWS );  
alternateModel->SetPdf( *myWS->pdf("modelWithConstraint") );  
alternateModel->SetPriorPdf( *myWS->pdf("priorNuisance") );  
alternateModel->SetParametersOfInterest( *myWS->set("POI") );  
alternateModel->SetNuisanceParameters( *myWS->set("nuisance") );  
alternateModel->SetObservables( *myWS->set("observables") );  
myWS->loadSnapshot( "alternate" );  
alternateModel->SetSnapshot( *myWS->set("POI") );
```

```
// create model config for null  
ModelConfig* nullModel = new ModelConfig(*alternateModel);  
myWS->loadSnapshot( "null" );  
nullModel->SetSnapshot( *myWS->set("POI") );
```

Example using ProfileLikelihoodCalculator

```
// Open the file and import the workspace & ModelConfigs
TFile* file = new TFile(fileName);
RooWorkspace* myWS = (RooWorkspace*) file->Get("myWS");
ModelConfig* alternateModel = (ModelConfig*) myWS->obj("alternateModel");
ModelConfig* nullModel = (ModelConfig*) myWS->obj("nullModel");
RooAbsData* data = myWS->data("data");
```

Here we imagine a new ROOT session, where we open a file with the workspace in it, and retrieve everything we need from inside

Setting up the ProfileLikelihoodCalculator is the same as before, except

- ▶ we must specify the parameter values for null and alternate
- ▶ **ToDo:** automatically set via ModelConfig

```
// Set up the ProfileLikelihoodCalculator
ProfileLikelihoodCalculator plc(*data, *alternateModel);
plc.SetAlternateParameters( *alternateModel->GetSnapshot() ); // should be automatic
plc.SetNullParameters( *nullModel->GetSnapshot() ); // should be automatic
```

```
// use the calculator
HypoTestResult* plcResult=plc.GetHypoTest();
double plcSignificance = plcResult->Significance();
```

Using the calculator as a HypoTestCalculator is easy!

```
Z_Bi significance estimation: 2.75905
profile significance estimation: 2.82453
```

The profile likelihood is an approximation and a different method, not expected to give the same result



<http://root.cern.ch/root/html/tutorials/roostats/HybridInstructional.C.html>

```
HybridCalculator hc1(*data, sb_model, b_model);
ToyMCSampler *toymcs1 = (ToyMCSampler*)hc1.GetTestStatSampler();
toymcs1->SetNEventsPerToy(1); // because the model is in number counting form
toymcs1->SetTestStatistic(&binCount); // set the test statistic
hc1.SetToys(20000,1000);
hc1.ForcePriorNuisanceAlt(*w->pdf("py"));
hc1.ForcePriorNuisanceNull(*w->pdf("py"));

-----
HypoTestResult *r1 = hc1.GetHypoTest();
```



Other Fundamental Interfaces

Most of the hard-work for the Neyman-Construction is going to be in **generating the sampling distribution for the test statistic**

- Same is true for several of our tools: coverage studies, the current HybridCalculator

Therefore we are adding a **TestStatSampler** interface that returns a **SamplingDistribution**

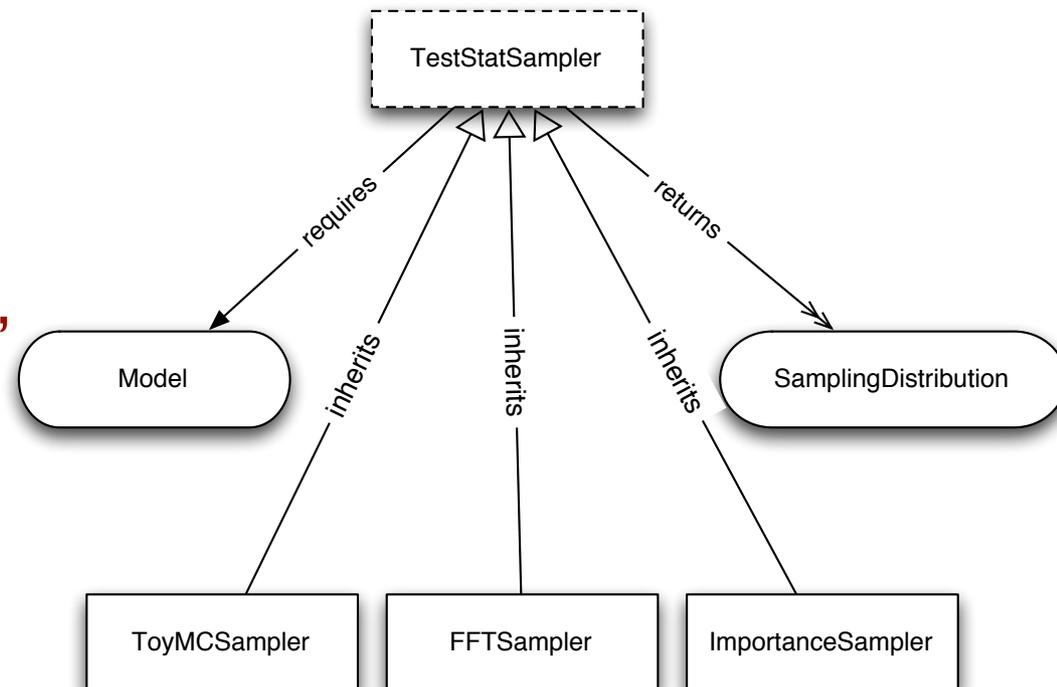
- **SamplingDistribution** (should support merging for use on clusters)

ToyMCSampler will be used also by HybridCalculator

- avoid duplicating code
- progress towards PROOF, batch, and MPI parallelization

ToyMC is not the only approach

- eg. analytic approach with FFTs and importance sampling



Most of the hard-work for the Neyman-Construction is going to be in **generating the sampling distribution for the test statistic**

- Same is true for several of our tools: coverage studies, the current HybridCalculator

Therefore we are adding a **TestStatSampler** interface that returns a **SamplingDistribution**

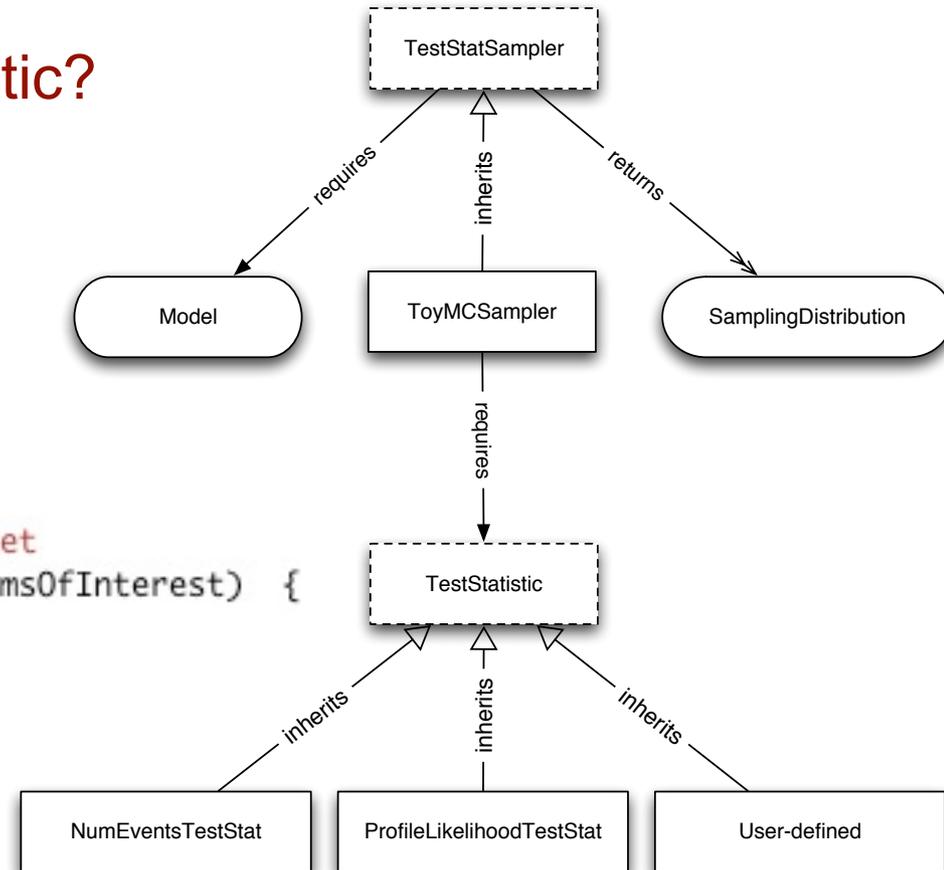
- How do you define a new test statistic?

Make it easy for user to extend:

make a new class that inherits from TestStatistic and implement this method:

```
// Main interface to evaluate the test statistic on a dataset  
virtual Double_t Evaluate(RooAbsData& data, RooArgSet& paramsOfInterest) {
```

First example test statistic is ProfileLikelihood



Three common test statistics

We express cross-section as $\mu = \sigma/\sigma_{SM}$ for convenience.

Effect of systematics is parametrized by one or more “nuisance parameters” denoted ν .

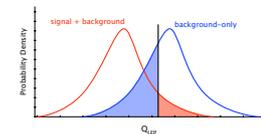
- best fit point is: $\hat{\mu}, \hat{\nu}$
- best fit of nuisance parameters with μ fixed is $\hat{\nu}$ (aka “profiled”)

In principle, s+b and b-only models can have different parametrizations

RooStats has the three common test statistics used in the field (and more)

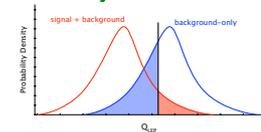
- simple likelihood ratio (used at LEP, nuisance parameters fixed)

$$Q_{LEP} = L_{s+b}(\mu = 1) / L_b(\mu = 0)$$



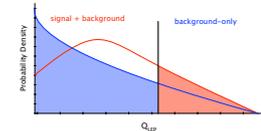
- ratio of profiled likelihoods (used commonly at Tevatron)

$$Q_{TEV} = L_{s+b}(\mu = 1, \hat{\nu}) / L_b(\mu = 0, \hat{\nu}')$$



- profile likelihood ratio (related to Wilks's theorem)

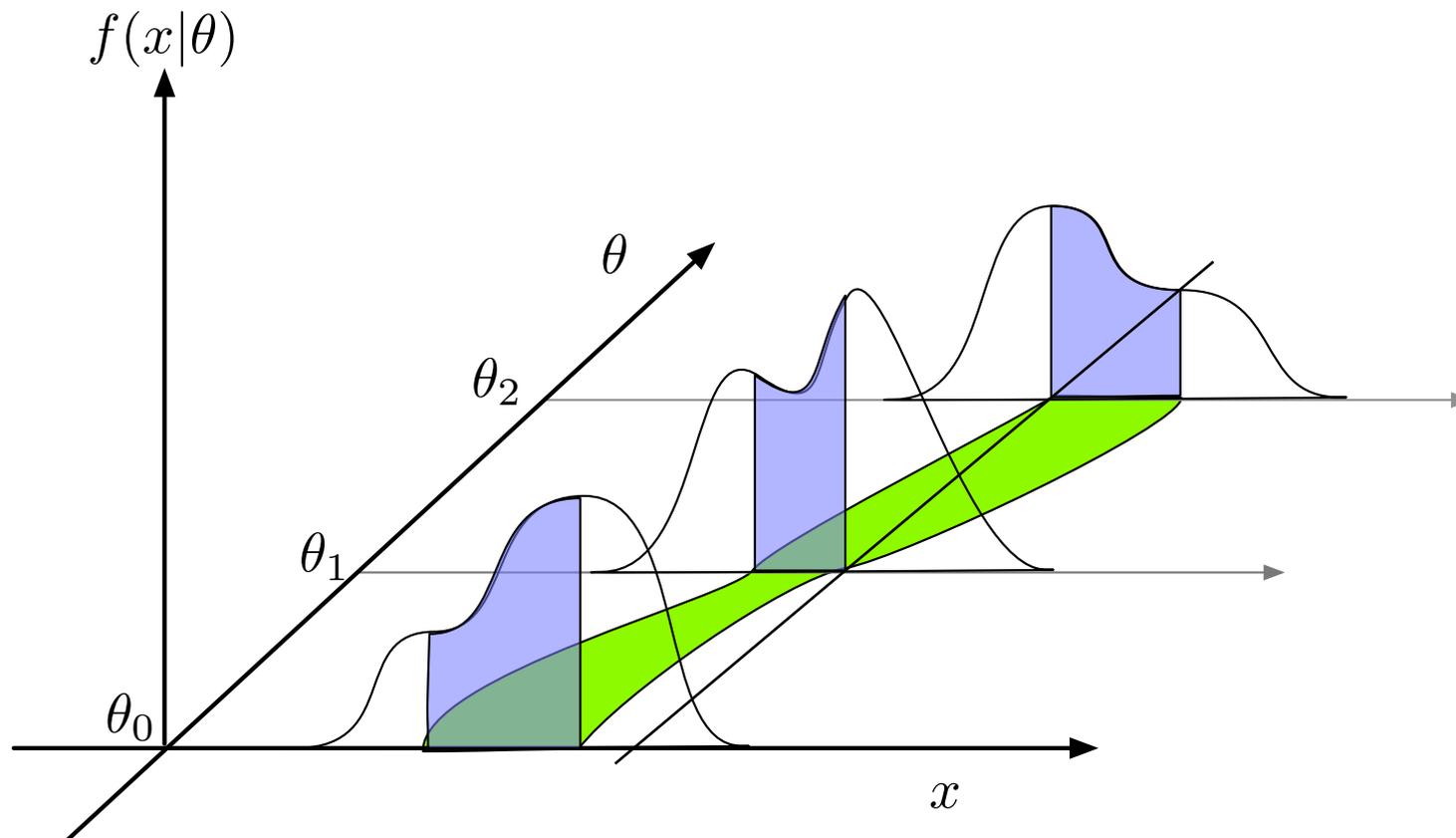
$$\lambda(\mu) = L_{s+b}(\mu, \hat{\nu}) / L_{s+b}(\hat{\mu}, \hat{\nu})$$





More about the Neyman Construction Calculator

- ▶ Treat each point in parameter space θ independently
- ▶ For each point, need distribution of some test statistic x
- ▶ Choose an **ordering rule** that selects a specific $1 - \alpha$ region
- ▶ Confidence Interval is **set** of parameter points where data in acceptance region (eg. intersects confidence belt)



Feldman & Cousins “Unified Approach” looks like this:

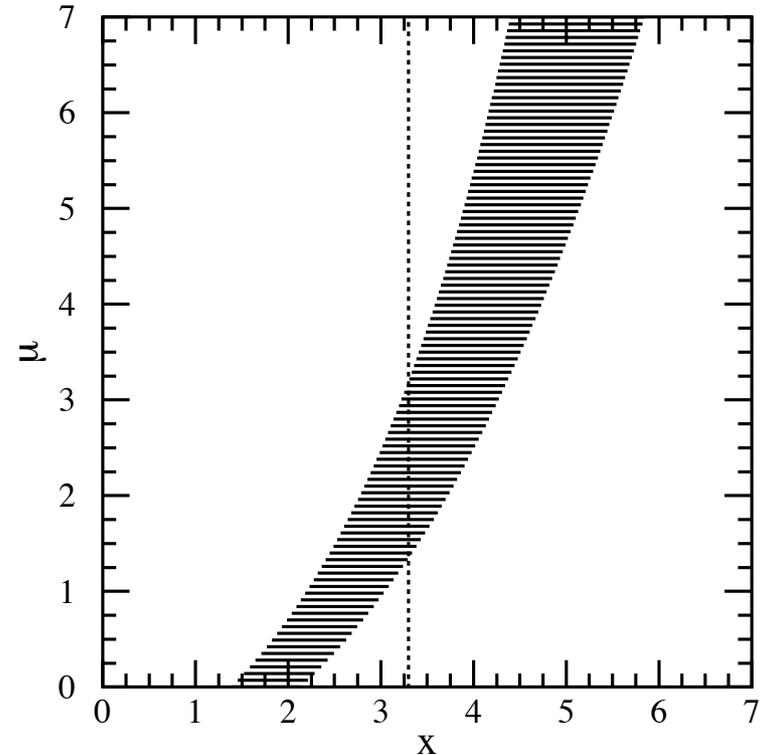
Neyman Construction

- For each μ : find region R_μ with probability $1 - \alpha$
- Confidence Interval includes all μ consistent with observation at x_0

Ordering Rule specifies what region

F-C ordering rule is the Likelihood Ratio

$$R_\mu = \left\{ x \mid \frac{L(x|\mu)}{L(x|\mu_{\text{best}})} > k_\alpha \right\}$$



The F-C ordering rule follows naturally from Neyman-Pearson Lemma

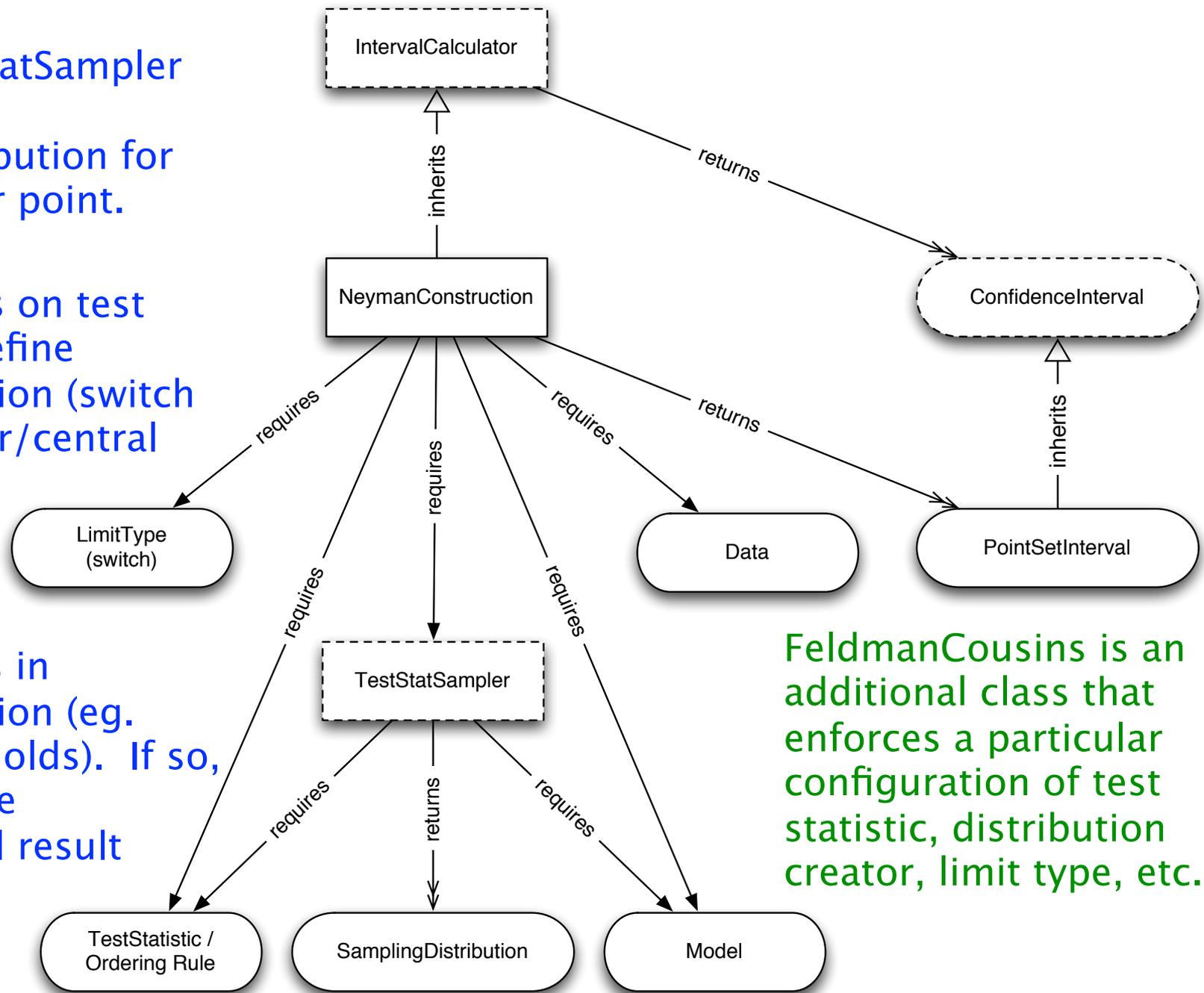
Neyman Construction Implementation



It uses a TestStatSampler to generate a SamplingDistribution for each parameter point.

Find thresholds on test statistic that define acceptance region (switch for upper/lower/central limits)

Check if data is in acceptance region (eg. between thresholds). If so, add point to the PointSetInterval result



FeldmanCousins is an additional class that enforces a particular configuration of test statistic, distribution creator, limit type, etc.

Consider a simple and well studied example. Intervals are tabulated in Feldman & Cousins original paper

http://root.cern.ch/root/html/tutorials/roostats/rs401c_FeldmanCousins.C.html

```
// make a simple model
RooRealVar x("x", "", 1,0,50);
RooRealVar mu("mu", "", 2.5,0, 15); // with a limit on mu>=0
RooConstVar b("b", "", 3.);
RooAddition mean("mean", "", RooArgList(mu,b));
RooPoisson pois("pois", "", x, mean);
RooArgSet parameters(mu);

// create a toy dataset
RooDataSet* data = pois.generate(RooArgSet(x), nEventsData);

////////// show use of Feldman-Cousins
RooStats::FeldmanCousins fc;
// set the distribution creator, which encodes the test statistic
fc.SetPdf(pois);
fc.SetParameters(parameters);
fc.SetTestSize(.05); // set size of test
fc.SetData(*data);
fc.UseAdaptiveSampling(true);
fc.FluctuateNumDataEntries(false);
fc.SetNBins(100); // number of points to test per parameter

// use the Feldman-Cousins tool
ConfInterval* interval = fc.GetInterval();

ConfidenceBelt* belt = fc.GetConfidenceBelt();
```

Model is simply

$$Pois(x|\mu + b)$$

where b is known

- ▶ Consider b=3

Generated data has

- ▶ x=7

Usage of FeldmanCousins utility is very easy

- ▶ same interface as Profile Likelihood interval calculator

FeldmanCousins utility gives for $x=7$, $b=3$ gives an interval:

- ▶ $\mu \in [0.8, 9.5]$ with default settings (step size of 0.15)
- ▶ takes ~19 seconds to test 100 points at 90% confidence
- ▶ compare to original paper $\mu \in [0.9, 9.53]$

TABLE IV. 90% C.L. intervals for the Poisson signal mean μ , for total events observed n_0 , for known mean background b ranging from 0 to 5.

$n_0 \backslash b$	0.0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	5.0
0	0.00, 2.44	0.00, 1.94	0.00, 1.61	0.00, 1.33	0.00, 1.26	0.00, 1.18	0.00, 1.08	0.00, 1.06	0.00, 1.01	0.00, 0.98
1	0.11, 4.36	0.00, 3.86	0.00, 3.36	0.00, 2.91	0.00, 2.53	0.00, 2.19	0.00, 1.88	0.00, 1.59	0.00, 1.39	0.00, 1.22
2	0.53, 5.91	0.03, 5.41	0.00, 4.91	0.00, 4.41	0.00, 3.91	0.00, 3.45	0.00, 3.04	0.00, 2.67	0.00, 2.33	0.00, 1.73
3	1.10, 7.42	0.60, 6.92	0.10, 6.42	0.00, 5.92	0.00, 5.42	0.00, 4.92	0.00, 4.42	0.00, 3.95	0.00, 3.53	0.00, 2.78
4	1.47, 8.60	1.17, 8.10	0.74, 7.60	0.24, 7.10	0.00, 6.60	0.00, 6.10	0.00, 5.60	0.00, 5.10	0.00, 4.60	0.00, 3.60
5	1.84, 9.99	1.53, 9.49	1.25, 8.99	0.93, 8.49	0.43, 7.99	0.00, 7.49	0.00, 6.99	0.00, 6.49	0.00, 5.99	0.00, 4.99
6	2.21, 11.47	1.90, 10.97	1.61, 10.47	1.33, 9.97	1.08, 9.47	0.65, 8.97	0.15, 8.47	0.00, 7.97	0.00, 7.47	0.00, 6.47
7	3.56, 12.53	3.06, 12.03	2.56, 11.53	2.09, 11.03	1.59, 10.53	1.18, 10.03	0.89, 9.53	0.39, 9.03	0.00, 8.53	0.00, 7.53
8	3.96, 13.99	3.46, 13.49	2.96, 12.99	2.51, 12.49	2.14, 11.99	1.81, 11.49	1.51, 10.99	1.06, 10.49	0.66, 9.99	0.00, 8.99
9	4.36, 15.30	3.86, 14.80	3.36, 14.30	2.91, 13.80	2.53, 13.30	2.19, 12.80	1.88, 12.30	1.59, 11.80	1.33, 11.30	0.43, 10.30
10	5.50, 16.50	5.00, 16.00	4.50, 15.50	4.00, 15.00	3.50, 14.50	3.04, 14.00	2.63, 13.50	2.27, 13.00	1.94, 12.50	1.19, 11.50
11	5.91, 17.81	5.41, 17.31	4.91, 16.81	4.41, 16.31	3.91, 15.81	3.45, 15.31	3.04, 14.81	2.67, 14.31	2.33, 13.81	1.73, 12.81
12	7.01, 19.00	6.51, 18.50	6.01, 18.00	5.51, 17.50	5.01, 17.00	4.51, 16.50	4.01, 16.00	3.54, 15.50	3.12, 15.00	2.38, 14.00
13	7.42, 20.05	6.92, 19.55	6.42, 19.05	5.92, 18.55	5.42, 18.05	4.92, 17.55	4.42, 17.05	3.95, 16.55	3.53, 16.05	2.78, 15.05
14	8.50, 21.50	8.00, 21.00	7.50, 20.50	7.00, 20.00	6.50, 19.50	6.00, 19.00	5.50, 18.50	5.00, 18.00	4.50, 17.50	3.59, 16.50
15	9.48, 22.52	8.98, 22.02	8.48, 21.52	7.98, 21.02	7.48, 20.52	6.98, 20.02	6.48, 19.52	5.98, 19.02	5.48, 18.52	4.48, 17.52
16	9.99, 23.99	9.49, 23.49	8.99, 22.99	8.49, 22.49	7.99, 21.99	7.49, 21.49	6.99, 20.99	6.49, 20.49	5.99, 19.99	4.99, 18.99
17	11.04, 25.02	10.54, 24.52	10.04, 24.02	9.54, 23.52	9.04, 23.02	8.54, 22.52	8.04, 22.02	7.54, 21.52	7.04, 21.02	6.04, 20.02
18	11.47, 26.16	10.97, 25.66	10.47, 25.16	9.97, 24.66	9.47, 24.16	8.97, 23.66	8.47, 23.16	7.97, 22.66	7.47, 22.16	6.47, 21.16
19	12.51, 27.51	12.01, 27.01	11.51, 26.51	11.01, 26.01	10.51, 25.51	10.01, 25.01	9.51, 24.51	9.01, 24.01	8.51, 23.51	7.51, 22.51
20	13.55, 28.52	13.05, 28.02	12.55, 27.52	12.05, 27.02	11.55, 26.52	11.05, 26.02	10.55, 25.52	10.05, 25.02	9.55, 24.52	8.55, 23.52

FeldmanCousins utility gives for $x=7$, $b=3$ gives an interval:

- ▶ $\mu \in [0.4, 10.7]$ with default settings (step size of 0.15)
- ▶ takes ~30 seconds to test 100 points at 95% confidence
- ▶ compare to original paper $\mu \in [0.3, 10.8]$

TABLE VI. 95% C.L. intervals for the Poisson signal mean μ , for total events observed n_0 , for known mean background b ranging from 0 to 5.

$n_0 \backslash b$	0.0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	5.0
0	0.00, 3.09	0.00, 2.63	0.00, 2.33	0.00, 2.05	0.00, 1.78	0.00, 1.78	0.00, 1.63	0.00, 1.63	0.00, 1.57	0.00, 1.54
1	0.05, 5.14	0.00, 4.64	0.00, 4.14	0.00, 3.69	0.00, 3.30	0.00, 2.95	0.00, 2.63	0.00, 2.33	0.00, 2.08	0.00, 1.88
2	0.36, 6.72	0.00, 6.22	0.00, 5.72	0.00, 5.22	0.00, 4.72	0.00, 4.25	0.00, 3.84	0.00, 3.46	0.00, 3.11	0.00, 2.49
3	0.82, 8.25	0.32, 7.75	0.00, 7.25	0.00, 6.75	0.00, 6.25	0.00, 5.75	0.00, 5.25	0.00, 4.78	0.00, 4.35	0.00, 3.58
4	1.37, 9.76	0.87, 9.26	0.37, 8.76	0.00, 8.26	0.00, 7.76	0.00, 7.26	0.00, 6.76	0.00, 6.26	0.00, 5.76	0.00, 4.84
5	1.84, 11.26	1.47, 10.76	0.97, 10.26	0.47, 9.76	0.00, 9.26	0.00, 8.76	0.00, 8.26	0.00, 7.76	0.00, 7.26	0.00, 6.26
6	2.21, 12.75	1.90, 12.25	1.61, 11.75	1.11, 11.25	0.61, 10.75	0.11, 10.25	0.00, 9.75	0.00, 9.25	0.00, 8.75	0.00, 7.75
7	2.58, 13.81	2.27, 13.31	1.97, 12.81	1.69, 12.31	1.29, 11.81	0.79, 11.31	0.29, 10.81	0.00, 10.31	0.00, 9.81	0.00, 8.81
8	2.94, 15.29	2.63, 14.79	2.33, 14.29	2.05, 13.79	1.78, 13.29	1.48, 12.79	0.98, 12.29	0.48, 11.79	0.00, 11.29	0.00, 10.29
9	4.36, 16.77	3.86, 16.27	3.36, 15.77	2.91, 15.27	2.46, 14.77	1.96, 14.27	1.62, 13.77	1.20, 13.27	0.70, 12.77	0.00, 11.77
10	4.75, 17.82	4.25, 17.32	3.75, 16.82	3.30, 16.32	2.92, 15.82	2.57, 15.32	2.25, 14.82	1.82, 14.32	1.43, 13.82	0.43, 12.82
11	5.14, 19.29	4.64, 18.79	4.14, 18.29	3.69, 17.79	3.30, 17.29	2.95, 16.79	2.63, 16.29	2.33, 15.79	2.04, 15.29	1.17, 14.29
12	6.32, 20.34	5.82, 19.84	5.32, 19.34	4.82, 18.84	4.32, 18.34	3.85, 17.84	3.44, 17.34	3.06, 16.84	2.69, 16.34	1.88, 15.34
13	6.72, 21.80	6.22, 21.30	5.72, 20.80	5.22, 20.30	4.72, 19.80	4.25, 19.30	3.84, 18.80	3.46, 18.30	3.11, 17.80	2.47, 16.80
14	7.84, 22.94	7.34, 22.44	6.84, 21.94	6.34, 21.44	5.84, 20.94	5.34, 20.44	4.84, 19.94	4.37, 19.44	3.94, 18.94	3.10, 17.94
15	8.25, 24.31	7.75, 23.81	7.25, 23.31	6.75, 22.81	6.25, 22.31	5.75, 21.81	5.25, 21.31	4.78, 20.81	4.35, 20.31	3.58, 19.31
16	9.34, 25.40	8.84, 24.90	8.34, 24.40	7.84, 23.90	7.34, 23.40	6.84, 22.90	6.34, 22.40	5.84, 21.90	5.34, 21.40	4.43, 20.40
17	9.76, 26.81	9.26, 26.31	8.76, 25.81	8.26, 25.31	7.76, 24.81	7.26, 24.31	6.76, 23.81	6.26, 23.31	5.76, 22.81	4.84, 21.81
18	10.84, 27.84	10.34, 27.34	9.84, 26.84	9.34, 26.34	8.84, 25.84	8.34, 25.34	7.84, 24.84	7.34, 24.34	6.84, 23.84	5.84, 22.84
19	11.26, 29.31	10.76, 28.81	10.26, 28.31	9.76, 27.81	9.26, 27.31	8.76, 26.81	8.26, 26.31	7.76, 25.81	7.26, 25.31	6.26, 24.31
20	12.33, 30.33	11.83, 29.83	11.33, 29.33	10.83, 28.83	10.33, 28.33	9.83, 27.83	9.33, 27.33	8.83, 26.83	8.33, 26.33	7.33, 25.33

The NeymanConstruction provides printouts as it scans the parameter space

On point 70 of 200 points in the parameter scan

Total number of toy MC pseudo-experiments to construct the acceptance region for this parameter point

name and value of parameters at this point

```
NeymanConstruction: Prog: 70/200 total MC = 78 this test stat = 3.63743  
mu=-0.305 [-1e+30, 2.14545] in interval = 0
```

...

```
NeymanConstruction: Prog: 78/200 total MC = 702 this test stat = 1.79968  
mu=-0.225 [-1e+30, 1.99669] in interval = 1
```

...

```
NeymanConstruction: Prog: 80/200 total MC = 78 this test stat = 1.44024  
mu=-0.205 [-1e+30, 2.01067] in interval = 1
```

value of test statistic for the observed data at this point

Is point in interval
1 = yes
0 = no

Limits of test statistic defining acceptance region (1e+30 = infinity)

Note, the test stat was well inside the acceptance region for this point, so the adaptive sampling algorithm used very few toy MC experiments to decide

the test stat was close to the boundary of acceptance region for this point, so the adaptive sampling algorithm used many more MC experiments to refine the boundary

How do you extend the Neyman Construction to include nuisance parameters?

▸ **Bayesian hybrid approach**

- eg. marginalize and only consider parameters of interest in construction (“FC²H”) -- requires a new test statistic (in progress)

[14] F. Tegenfeldt, J. Conrad, “On Bayesian treatment of systematic uncertainties in confidence interval calculations,” Nucl. Instrum. Meth. **A539**, 407-413 (2005). [physics/0408039].

[19] G. C. Hill, “Comment on ‘Including systematic uncertainties in confidence interval construction for Poisson statistics’,” Phys. Rev. **D67**, 118101 (2003). [physics/0302057].

▸ **Frequentist approach**

- generalize the ordering rule

- eg. use profile likelihood ratio, Cranmer, PhyStat03 & Punzi, PhyStat05

1. do the “full construction”: eg, construction over params of interest and nuisance parameters

- Projection of intervals can cause overcoverage

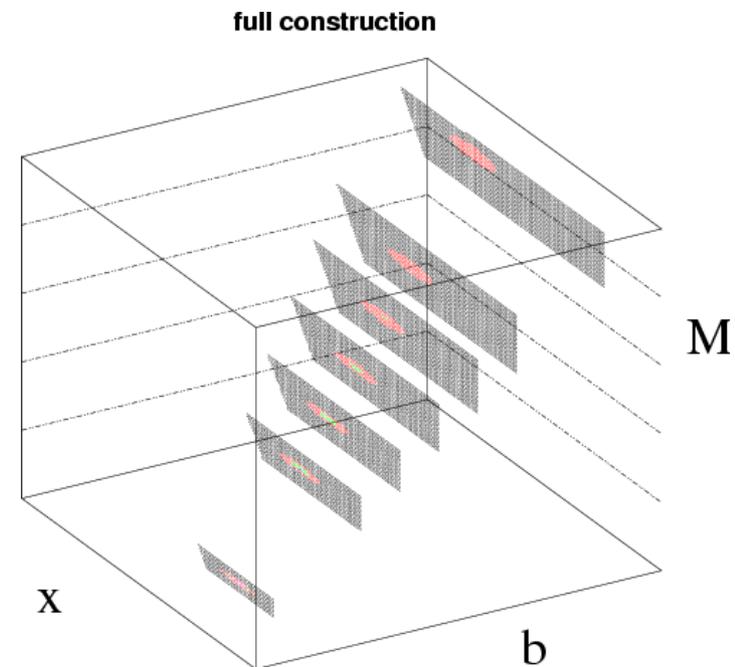
2. do the “profile construction”: eg. only consider sub-space of conditional M.L.E. of the nuisance parameters given data & parameters of interest (eg. the “hat hat” variables).

- Only approximate coverage, but good in practice

3. same as above, but calibrate the threshold on the ordering rule to ensure coverage

[22] K. S. Cranmer, “Frequentist hypothesis testing with background uncertainty,” [physics/0310108].

[23] G. Punzi, “Ordering algorithms and confidence intervals in the presence of nuisance parameters,” [physics/0511202].



ideal shape of conf. region

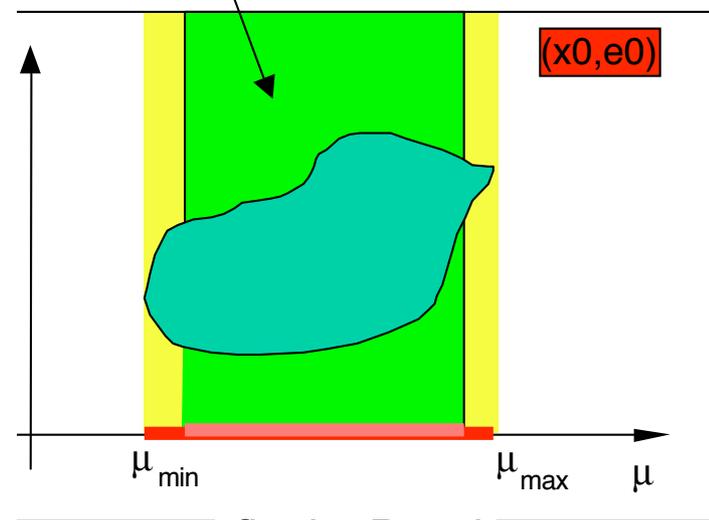
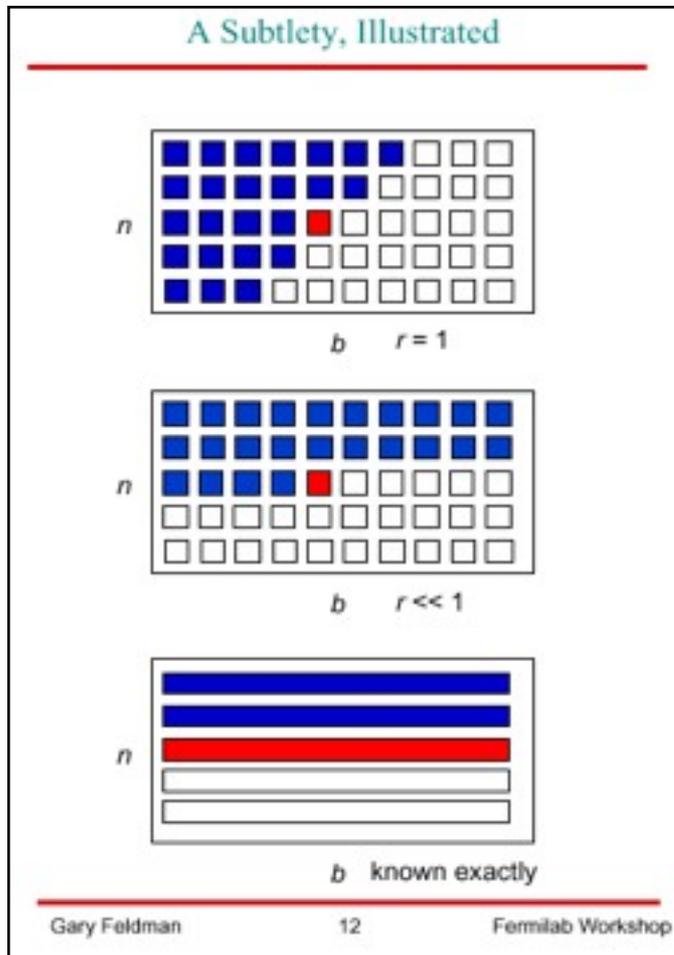
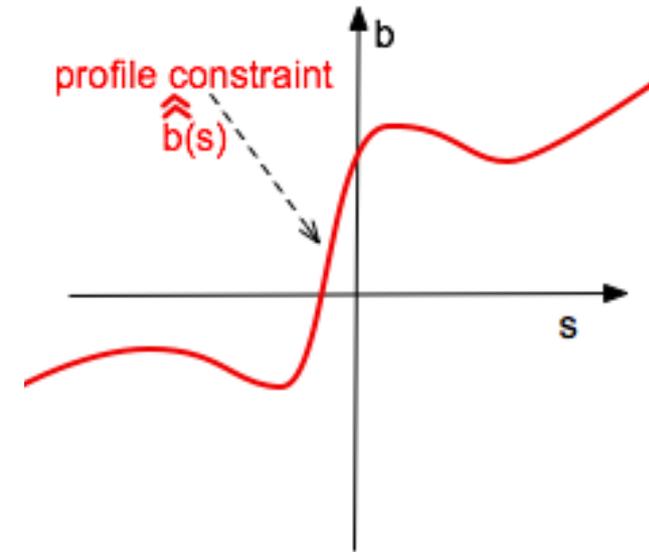


fig. by Punzi

Gary Feldman presented an approximate Neyman Construction, based on the same ordering rule, but only on a subspace of the parameters based on profiling



The profile construction means that one does not need to scan each nuisance parameter

- ▶ easier computationally

This approximation does not guarantee exact coverage, but

- ▶ tests indicate impressive performance

This is **default** implementation of FeldmanCousins calculator if the model has nuisance parameters

- ▶ Note that when using profile likelihood ratio, we are testing considering the same sub-space.

At the 2000 Stat. Conference at FermiLab, Feldman showed that Unified Method with Nuisance Parameters is in Kendall's Theory (chapter on Likelihood ratio tests & test efficiency)

Variable	Meaning
θ_r	physics parameters
θ_s	nuisance parameters
$\hat{\theta}_r, \hat{\theta}_s$	unconditionally maximize $L(x \hat{\theta}_r, \hat{\theta}_s)$
$\hat{\theta}_s$	conditionally maximize $L(x \theta_{r0}, \hat{\theta}_s)$

$$\begin{aligned} & (H_0 : \theta_r = \theta_{r0}) \\ & (H_1 : \theta_r \neq \theta_{r0}) \end{aligned}$$

Now consider the Likelihood Ratio

$$l = \frac{L(x|\theta_{r0}, \hat{\theta}_s)}{L(x|\hat{\theta}_r, \hat{\theta}_s)}$$

Intuitively l is a reasonable test statistic for H_0 : it is the maximum likelihood under H_0 as a fraction of its largest possible value, and large values of l signify that H_0 is reasonably acceptable.



Under development (as of May 2010):

- ▶ finalized ConfidenceBelt class
 - will support read/write, merging, refining, plotting, etc.
- ▶ general performance improvements & code optimization
- ▶ ability to run parameter points in parallel using PROOF
- ▶ improved visualization of the result
- ▶ easier control over parameter points being scanned
 - scan within a range, logarithmic scanning for selected variables, ...
- ▶ additional choices for test statistic
 - marginalized likelihood ratio, mean of dataset, N events in cut, ...
- ▶ additional choices for test stat sampler
 - importance sampling, analytic convolution via FFT (for some problems)



More about the MCMC Calculator

Markov Chain Monte Carlo (MCMC) is a nice technique which will produce a sampling of a parameter space which is proportional to a posterior

- ▶ it works well in high dimensional problems
- ▶ Metropolis–Hastings Algorithm: generates a sequence of points $\{\vec{\alpha}^{(t)}\}$
 - Given the likelihood function $L(\vec{\alpha})$ & prior $P(\vec{\alpha})$, the posterior is proportional to $L(\vec{\alpha}) \cdot P(\vec{\alpha})$
 - propose a point $\vec{\alpha}'$ to be added to the chain according to a proposal density $Q(\vec{\alpha}'|\vec{\alpha})$ that depends only on current point $\vec{\alpha}$
 - if posterior is higher at $\vec{\alpha}'$ than at $\vec{\alpha}$, then add new point to chain
 - else: add $\vec{\alpha}'$ to the chain with probability

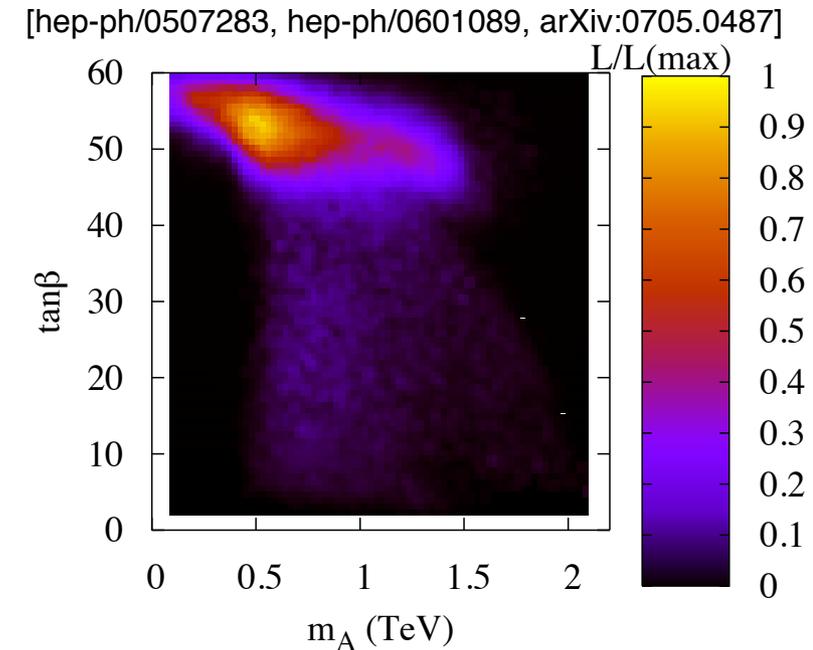
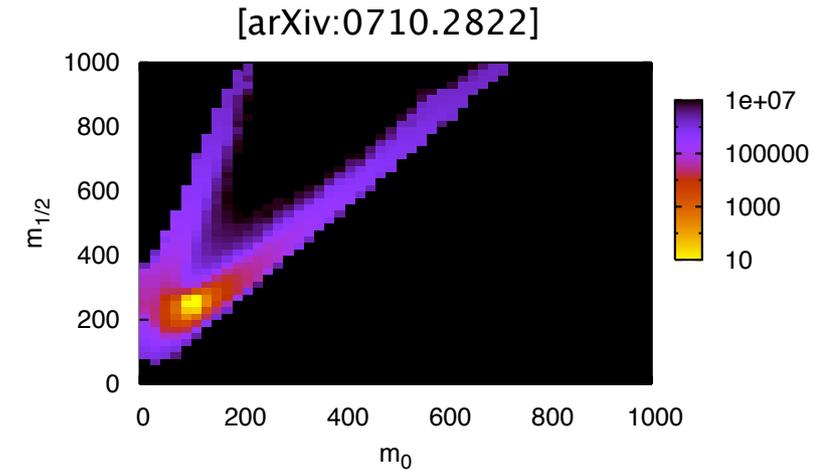
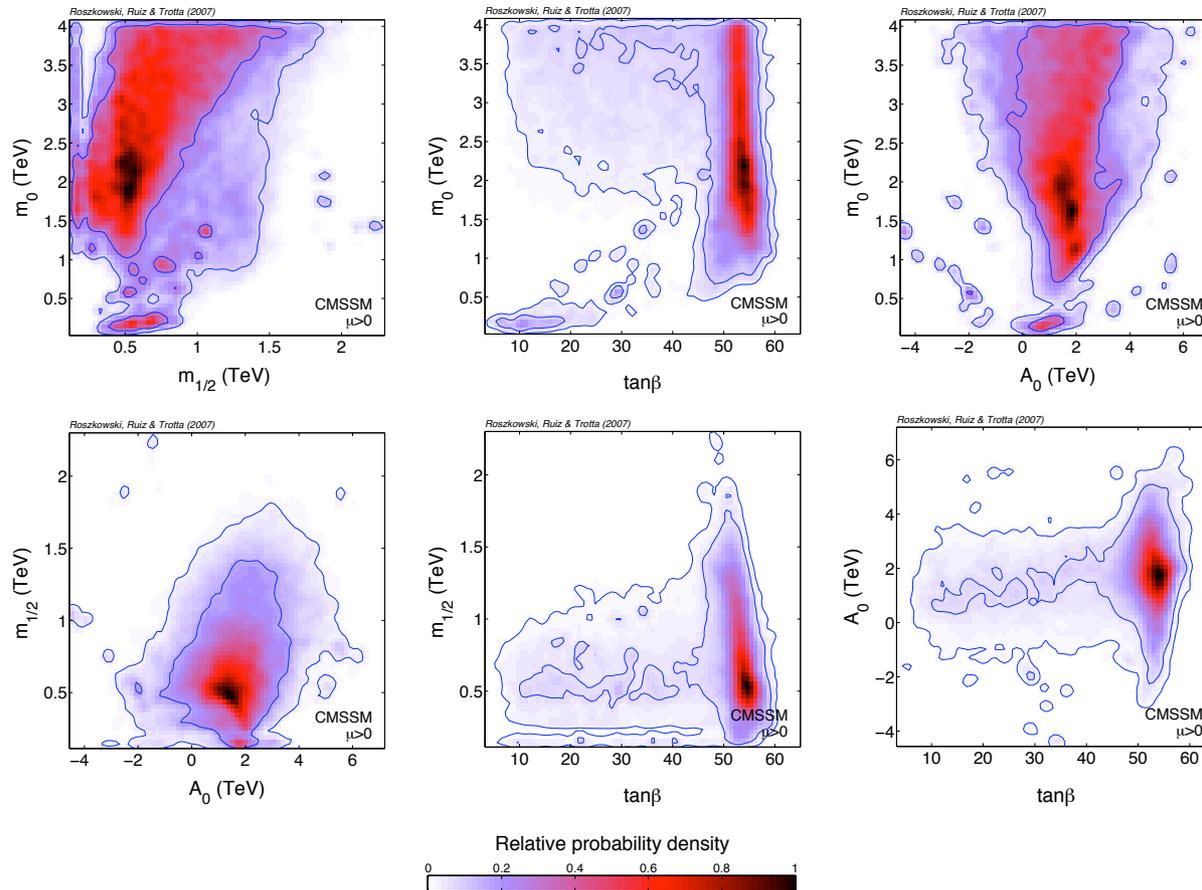
$$\rho = \frac{L(\vec{\alpha}') \cdot P(\vec{\alpha}') \cdot Q(\vec{\alpha}|\vec{\alpha}')}{L(\vec{\alpha}) \cdot P(\vec{\alpha}) \cdot Q(\vec{\alpha}'|\vec{\alpha})}$$

- (appending original point $\vec{\alpha}$ with complementary probability)
- ▶ RooStats works with any $L(\vec{\alpha}), P(\vec{\alpha})$
- ▶ Since last week: can use any RooFit PDF as proposal function $Q(\vec{\alpha}'|\vec{\alpha})$

Work done primarily by Kevin Belasco, a Princeton undergraduate I'm working with.

Use of Markov Chain Monte Carlo

Markov Chain Monte Carlo is what is used to make these types of “weather forecasts” that scan SUSY parameter space based on existing measurements for WMAP, precision electroweak, etc.



MCMCCalculator implements the interface for IntervalCalculator

- ▶ it runs Metropolis-Hastings sampling, but that is of general use, so that is its own class.
- ▶ The MetropolisHastings sampler will return a MarkovChain
- ▶ The MetropolisHastings algorithm needs a ProposalFunction, which is an abstract interface
 - We can support arbitrary RooFit PDFs as proposal functions
 - Provide ProposerHelper tool to make it easier to make them
- ▶ The MCMCInterval will use the chain to make the smallest interval it can
- ▶ The MCMCIntervalPlot class for visualizing results

To do:

- ▶ run parallel chains and support merging
- ▶ provide convergence tests and auto-detect burn-in



6.6.1 MCMCCalculator

MCMCCalculator runs the Metropolis-Hastings algorithm (section 6.6.3) with the parameters of your model, a $-\log(\text{Likelihood})$ function constructed from the model and data set, and a ProposalFunction (section 6.6.2). This generates a Markov chain posterior sampling, which is used to create an MCMCInterval (section 6.6.4).

The most basic usage of an MCMCCalculator is automatically set up with the simplified 3-arg constructor (or 4-arg with RooWorkspace). This automatic configuration package consists of a UniformProposal, 10,000 Metropolis-Hastings iterations, 40 burn-in steps, and 50 bins for each RooRealVar; it also determines the interval by kernel-estimation, turns on sparse histogram mode, and finds a 95% confidence interval (see sections 6.6.3 and 6.6.4 to learn about these options). These are reasonable settings designed to minimize configuration steps and hassle for beginning users, making the code very simple:

```
// data is a RooAbsData, model is a RooAbsPdf,  
// and parametersOfInterest is a RooArgSet  
MCMCCalculator mc(data, model, parametersOfInterest);  
ConfInterval* mcmcInterval = mc.GetInterval();
```

All other MCMCCalculator constructors are designed for maximum user control and thus do not have any automatic settings. You can customize the configuration (and override any automatic settings) through the mutator methods provided by MCMCCalculator. It may be easiest to use the default no-args constructor and perform all configuration with these methods.

```
// A simple ProposalFunction  
ProposalFunction* pf = new UniformProposal();  
  
MCMCCalculator mc;  
mc.SetData(data);  
mc.SetPdf(model);  
mc.SetParameters(parametersOfInterest);  
mc.SetProposalFunction(*pf);  
mc.SetNumIters(100000);           // Metropolis-Hastings algorithm iterations  
mc.SetNumBurnInSteps(50);        // first N steps to be ignored as burn-in  
mc.SetNumBins(50);               // bins to use for RooRealVars in histograms  
mc.SetTestSize(.1);              // 90% confidence level  
mc.SetUseKeys(true);             // Use kernel estimation to determine interval  
ConfInterval* mcmcInterval = mc.GetInterval();
```

6.6.2 ProposalFunction

The ProposalFunction interface generalizes the task of proposing points in some distribution for some set of variables, presumably for use with the Metropolis-Hastings algorithm.

PdfProposal is the most powerful and general ProposalFunction derived class. It proposes points in the distribution of any RooAbsPdf you pass it to serve as its proposal density function. It also provides a generalized means of updating PDF parameters based on the current values of PDF observables. This is useful for centering the proposal density function around the current point when proposing others or for advanced control over the widths of peaks or anything else in the proposal function. It also has a cacheing mechanism (off by default) which almost always significantly speeds up proposals.

Here's a PdfProposal construction example that uses a covariance matrix from the RooFitResult. Be careful that your RooArgLists have the same order of variables as the covariance matrix uses:

```
// Assume we have x, y, and z as RooRealVar* parameters of our model

// Create clones to serve as mean variables
RooRealVar* mu_x = (RooRealVar*)x->clone("mu_x");
RooRealVar* mu_y = (RooRealVar*)y->clone("mu_y");
RooRealVar* mu_z = (RooRealVar*)z->clone("mu_z");

// Fit model to data to get a covariance matrix
// (you can also just construct your own custom covariance matrix)
TMatrixDSym covFit = model->fitTo(*data)->covarianceMatrix();

// Make a PDF to be our proposal density function
RooMultiVarGaussian mvg("mvg", "mvg", RooArgList(*x, *y, *z), // Careful!
                        RooArgList(*mu_x, *mu_y, *mu_z), covFit);
PdfProposal pf(mvg);

// Optional mappings to center the proposal function around the current point
pf.AddMapping(*mu_x, *x);
pf.AddMapping(*mu_y, *y);
pf.AddMapping(*mu_z, *z);

pf.SetCacheSize(100); // when we must generate proposal points, generate 100
```



Since PdfProposal functions are powerful but annoying to build, we created ProposalHelper to make it easier. It will build a multi-variate Gaussian proposal function and has some handy options for doing so. Here's how to create exactly the same proposal function as in the example above. Note that using RooFitResult::floatParsFinal() to set the RooArgList of variables ensures the right order.

```
RooFitResult* fit = model->fitTo(*data);

// Easy ProposalFunction construction with ProposalHelper
ProposalHelper ph;
ph.SetVariables(fit->floatParsFinal());
ph.SetCovMatrix(fit->covarianceMatrix());
ph.SetUpdateProposalParameters(kTRUE); // auto-create mean vars and add mappings
ph.SetCacheSize(100);
ProposalFunction* pf = ph.GetProposalFunction();
```

ProposalHelper can also create a PdfProposal with a "Bank of Clues" (cite paper) component. This will add a PDF with a kernel placed at each "clue" point to the proposal density function. This will increase the frequency of proposals in the clue regions which can be especially useful for helping the Metropolis-Hastings algorithm find small and/or distant regions of interest (no free lunch, of course, you need to know these regions beforehand to pick the clue points). Just pass a RooDataSet with (possibly weighted) entries for each clue

ProposalHelper can also create a PdfProposal with a "Bank of Clues" (cite paper) component. This will add a PDF with a kernel placed at each "clue" point to the proposal density function. This will increase the frequency of proposals in the clue regions which can be especially useful for helping the Metropolis-Hastings algorithm find small and/or distant regions of interest (no free lunch, of course, you need to know these regions beforehand to pick the clue points). Just pass a RooDataSet with (possibly weighted) entries for each clue point. You can also choose what fraction of the total proposal function integral comes from the bank of clues PDF.

```
// assume bankOfClues is a RooDataSet with weighted "clues" as entries

RooArgSet vars(x,y);

ProposalHelper ph;
ph.SetVariables(vars);
ph.SetClues(bankOfClues);           // use bankOfClues to make a clues PDF
ph.SetCluesFraction(0.15);         // clues PDF accounts for 15% of PDF integral
ph.SetUpdateProposalParameters(kTRUE); // auto-create mean vars and add mappings
ph.SetCacheSize(100);
ProposalFunction* pf = ph.GetProposalFunction();
```

Using the covariance matrix from a RooFitResult is not required. If you do not set the covariance matrix, ProposalHelper constructs a pretty good default for you – a diagonal matrix with sigmas set to some fraction of the range of each corresponding RooRealVar. You can set this fraction yourself (the default is 1/6th).

To help Metropolis-Hastings find small and/or distant regions of interest that you do not know beforehand, you can set ProposalHelper to add a fraction of uniform proposal density to the proposal function. Use the ProposalHelper::SetUniformFraction() method to choose what fraction the uniform PDF makes up of the entire proposal function integral.

UniformProposal is a specialized implementation of a ProposalFunction that proposes points in a uniform distribution over the range of the variables. Its low overhead as compared to a PdfProposal using a RooUniform PDF and guaranteed symmetry makes it much faster at proposing in a purely uniform distribution. UniformProposal does not need a caching mechanism.

6.6.3 MetropolisHastings

A MetropolisHastings object runs the Metropolis-Hastings algorithm to construct a Markov chain posterior sampling of a function. At each step in the algorithm, a new point is proposed (section 6.6.2) and possibly "visited" based on its likelihood relative to the current point. Even when the proposal density function is not symmetric, MetropolisHastings maintains detailed balance when constructing the Markov chain by counterbalancing the relative likelihood between the two points with the relative proposal density. That is, given the current point x , proposed point x' , likelihood function L , and proposal density function Q , we visit x' iff

$$\frac{L(x') Q(x|x')}{L(x) Q(x'|x)} \geq \text{Rand}[0,1]$$

MetropolisHastings supports ordinary and log-scale functions. This is particularly useful for handling either regular likelihood or \pm log-likelihood functions. You must tell MetropolisHastings the type and sign of function the you have supplied (if you supply a regular function, make sure that it is never 0). Then set a ProposalFunction, parameters to propose for, and a number of algorithm iterations. Call ConstructChain() to get the Markov chain.

```
RooAbsReal* function = new RooGaussian("gauss", "gauss" x, mu, sigma);
RooArgSet vars(x);

// make our MetropolisHastings object
MetropolisHastings mh;
mh.SetFunction(*function);          // function to sample
mh.SetType(MetropolisHastings::kRegular);
mh.SetSign(MetropolisHastings::kPositive);
mh.SetProposalFunction(proposalFunction);
mh.SetParameters(vars);
mh.SetNumIters(10000);
MarkovChain* chain = mh.ConstructChain();
```

Here's how to do a similar task using a negative log-likelihood function instead:

```
RooAbsReal* nll = pdf->createNLL(*data);
RooArgSet* vars = nll->getParameters(*data);
RemoveConstantParameters(vars); // to be safe

MetropolisHastings mh;
mh.SetFunction(*nll);          // function to sample
mh.SetType(MetropolisHastings::kLog);
mh.SetSign(MetropolisHastings::kNegative);
mh.SetProposalFunction(proposalFunction);
mh.SetParameters(*vars);
mh.SetNumIters(10000);
MarkovChain* chain = mh.ConstructChain();
```

6.6.4 MCMCInterval

MCMCInterval is a ConflInterval that determines the confidence interval on your parameters from a MarkovChain (section 6.6.6) generated by Monte Carlo. To determine the confidence interval, MCMCInterval integrates the posterior where it is the tallest until it finds the correct cutoff height C to give the target confidence level P . That is, to find

$$\int_{f(\mathbf{x}) \geq C} f(\mathbf{x}) d^n x = P$$

MCMCInterval has a few methods for representing the posterior to do this integration. The default is simply as a histogram, so this integral turns into a summation of bin heights. If you have no more than 3 parameters and 100 bins, a standard histogram will be fine. However, for higher dimensions or bin numbers, it is faster and less memory intensive to use a sparse histogram data structure (aside: in a regular histogram, 4 variables with 100 bins each requires ~ 4 GB of contiguous memory, a tall order). By default, the histogram method adds bins to the interval until at least the desired confidence level has been reached (use `MCMCInterval::SetHistStrict()` to change this).

Another posterior representation option is kernel-estimation (often termed "keys") using a `RooNDKeysPdf`, which has more theoretical validity because it takes the arbitrariness out of choosing a histogram binning. The kernel-estimation method usually takes longer than the histogram method because it typically requires several integrations to find the right cutoff such that $|P_{calculated} - P_{target}| < \epsilon$ ($\epsilon = 0.01$ by default).

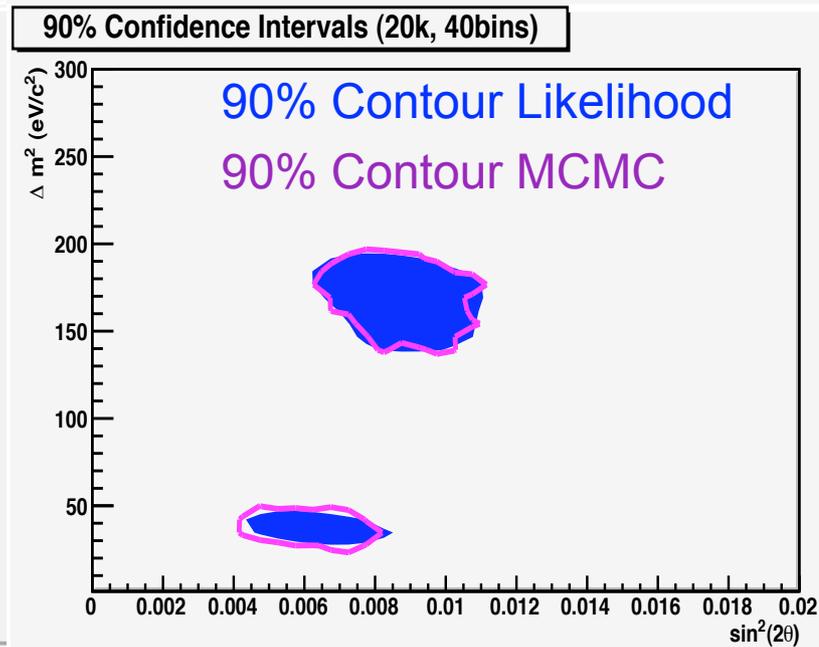
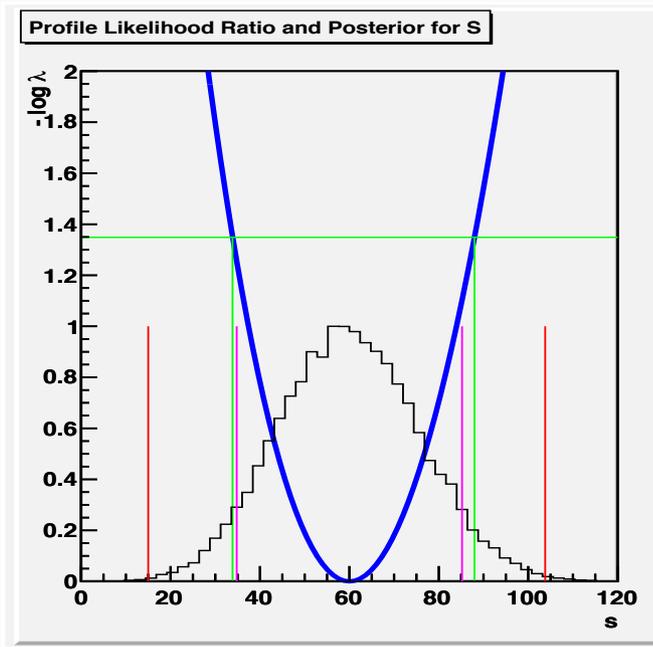
To try to remove the arbitrariness of the starting point in the Markov chain, which was rather random when it was generated by `MetropolisHastings`, a certain number of "burn-in" steps can be ignored from the beginning of the chain. Generally it is a good idea to use burn-in, but the number of steps to discard depends on the function you are sampling and your proposal function, so it is off by default. Usually you will tell the `MCMCCalculator` (section 6.6.1) the number of burn-in steps to use by calling `MCMCCalculator::SetNumBurnInSteps()`, since it configures the `MCMCInterval`. For future versions, automatic burn-in step calculations are being considered.

6.6.5 MCMCIntervalPlot

The MCMCIntervalPlot class helps you to visualize the interval and Markov chain. The function MCMCIntervalPlot::Draw() will draw the interval as determined by the type of posterior representation the MCMCInterval was configured for (i.e. histogram or keys PDF). To specifically ask for a certain interval determination to be drawn, use DrawHistInterval() or DrawKeysPdfInterval().

```
MCMCInterval* interval = (MCMCInterval*)mcmcCalc.GetInterval(); // must cast
MCMCIntervalPlot mcPlot(*interval);

// Draw posterior
TCanvas* c = new TCanvas("c");
mcPlot.SetLineColor(kOrange); // optional
mcPlot.SetLineWidth(2);      // optional
mcPlot.Draw();
```



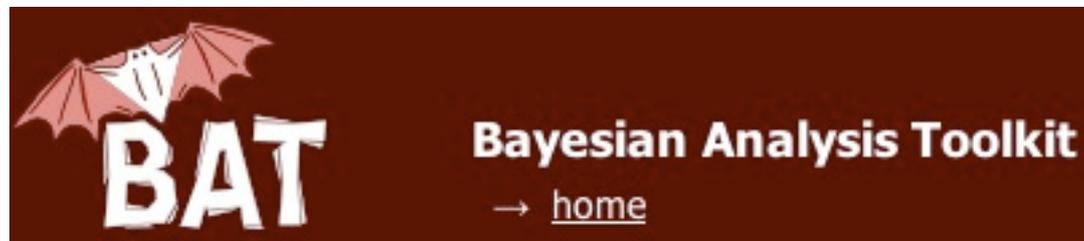
6.6.6 MarkovChain

A MarkovChain stores a series of weighted steps through N-dimensional space in which each step depended only on the one before it. Each step in the chain also has a $-\log(\text{likelihood})$ value associated with it. The class supplies some simple methods to access each step in the chain in order:

```
MCMCInterval* interval = (MCMCInterval*)mcmcCalc.GetInterval(); // must cast
const MarkovChain* chain = interval->GetChain();

// Print the contents of the chain
for (Int_t i = 0; i < chain->Size(); i++) {
    cout << "Entry # " << i << endl;
    const RooArgSet* entry = chain->Get(i);
    entry->Print("v");
    cout << "weight = " << chain->Weight() << endl; // weight of current entry
    cout << "NLL = " << chain->NLL() << endl; // NLL value of current entry
}
```

We have a good relationship with developers of BAT which also implements the Metropolis-Hastings algorithm



<http://www.mppmu.mpg.de/bat/>

Gregory Schott has developed an adaptor so that BAT can use a RooFit/RooStats model and data stored in a RooWorkspace.

- ▶ This is useful for cross-checks of the RooStats MCMC
- ▶ Also developing interface so that BAT is implementation of a second MCMC tool

Given BAT, why have MCMC in RooStats at all?

- ▶ Primary goal of RooStats is to have Bayesian, Frequentist, and Likelihood-based statistical formalism working with a common interface.
 - ie. we need an MCMC algorithm that inherits from **IntervalCalculator** and returns a **ConfidenceInterval**.
- ▶ Also, we found several places that we can make optimizations or utilize other functionality within RooStats to improve our MCMC implementation
- ▶ Finally, good training and nice to have cross-checks

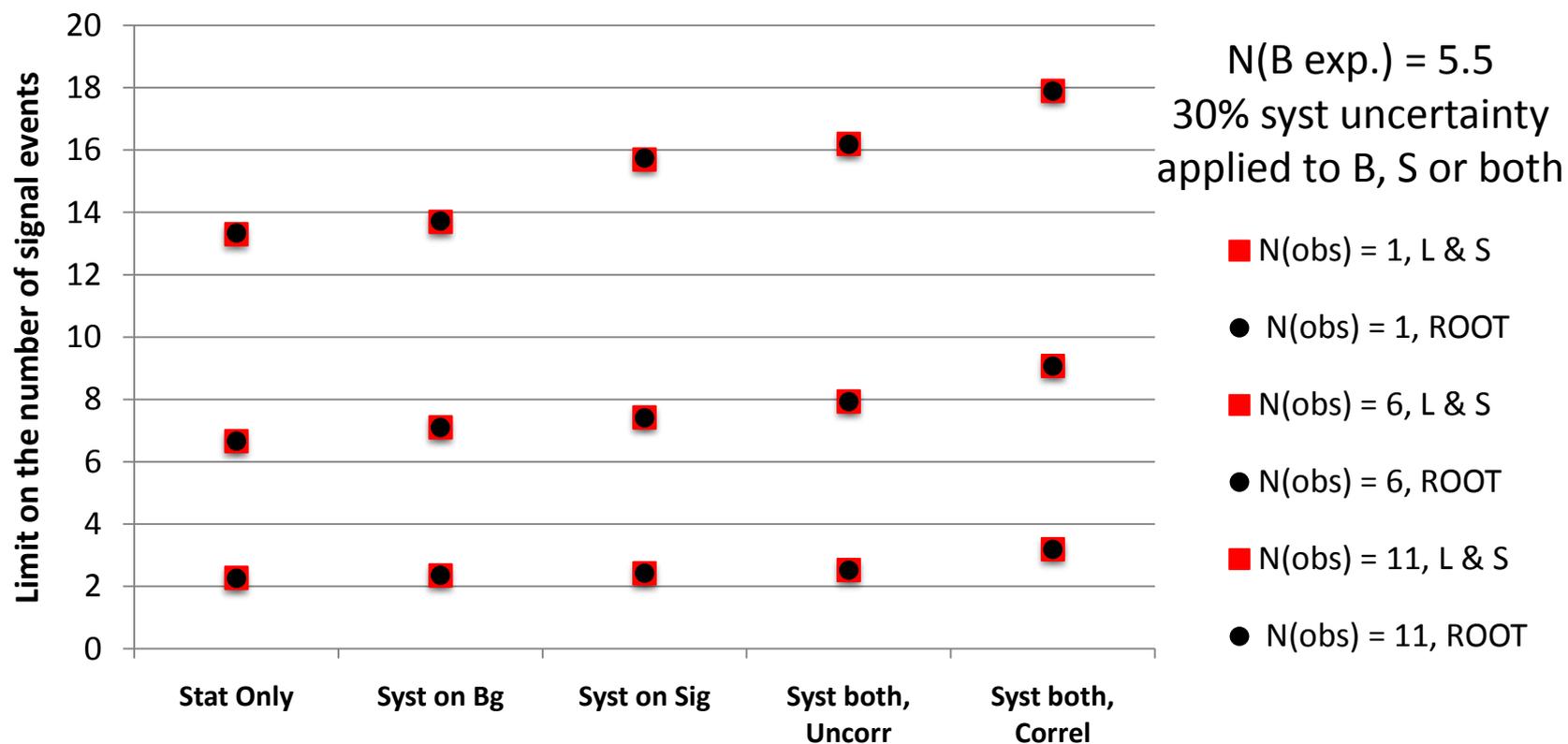


Validation



Profile Likelihood method

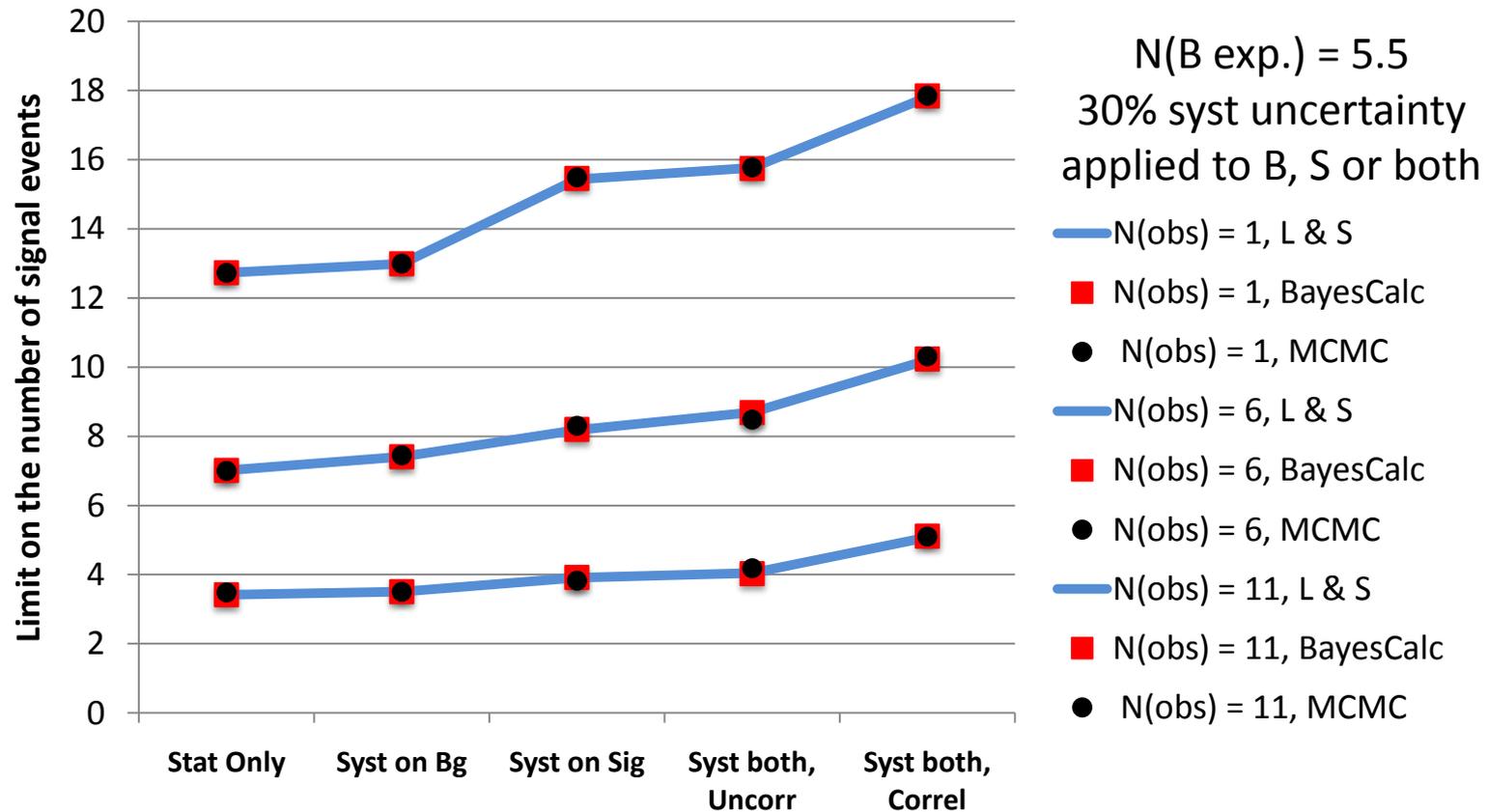
Comparison for simple counting experiments with different $n(\text{obs})$ and syst. uncertainties





Bayesian

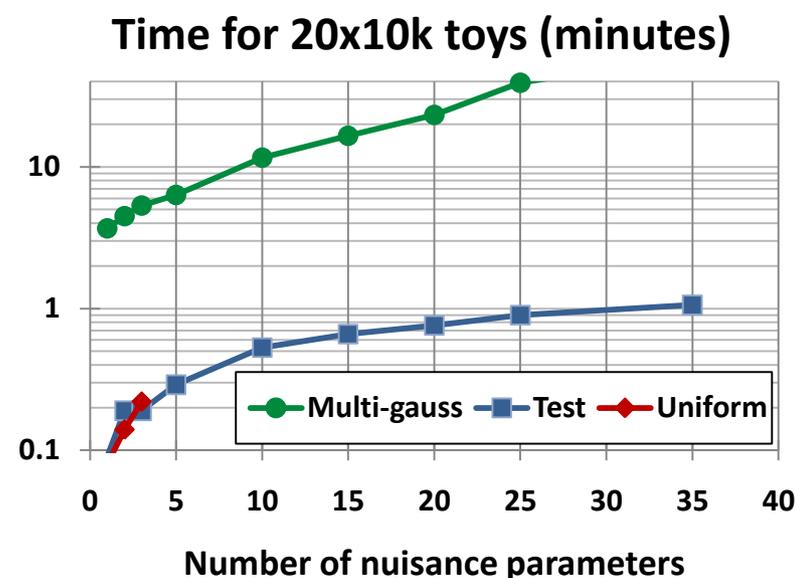
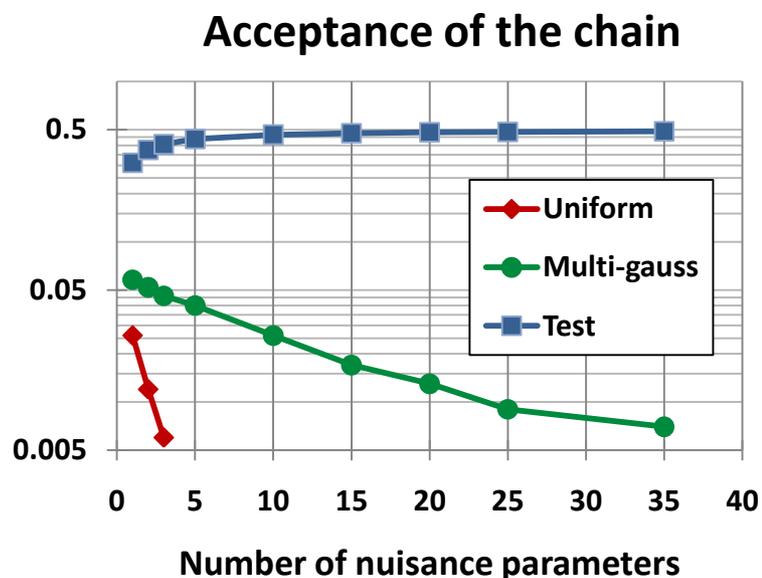
On simple experiments, good agreement between L&S, BayesianCalculator and MCMCCalculator





MCMC Proposals

- Evaluated the performance of the various proposal with H->WW model but including only N nuisance parameters.



- Due to the higher correlation between points with the test proposal, the uncertainty increases with the number of nuisances (20x10k toys: 0.09%→1.4% for 1→35 nuisances)



More about the Hybrid Calculator



At LEP the Cousins-Highland Method was used for Systematics

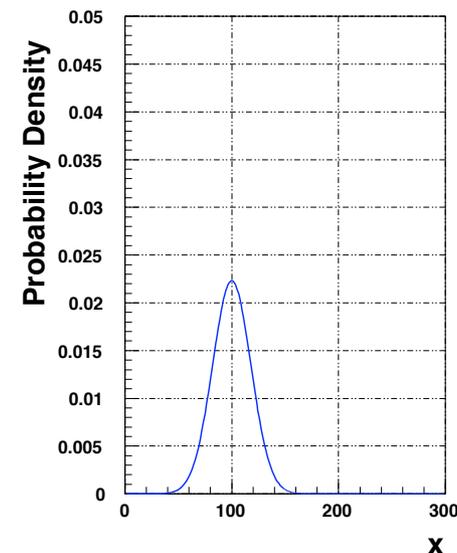
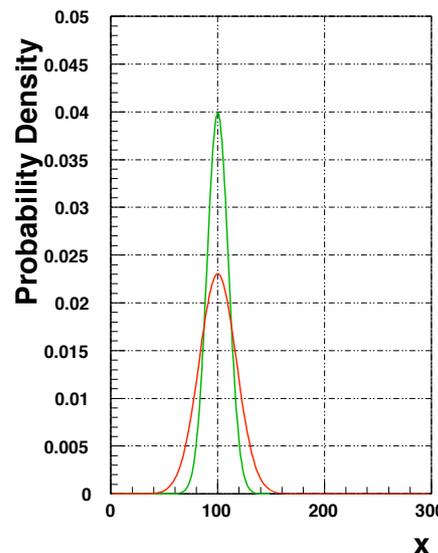
[12] R. D. Cousins, V. L. Highland, “Incorporating systematic uncertainties into an upper limit,” Nucl. Instrum. Meth. **A320**, 331-335 (1992).

The Cousins-Highland method
integrates-out b

$$L(x|H_0, M) = \int_b L(x|b)L(b|M)$$

But it uses a Bayesian notion $L(b)$

$$L(b|M) = \frac{L(M|b) L(b)}{L(M)}$$



That’s why we call it the HybridCalculator, because of Bayesian ingredient

- More consistent to think of $L(M|b)$ as part of the model and $L(b)$ as a prior
- This Hybrid approach to nuisance parameters has been used in a NeymanConstruction before, but it is NOT the Feldman–Cousins technique
- See papers by [G. Hill](#), [Conrad et. al](#), [Tegenfeldt & Conrad](#) (they call it FHC²)



Goal of Bayesian-frequentist hybrid solutions is to provide a frequentist treatment of the main measurement, while eliminating nuisance parameters (deal with systematics) with an intuitive Bayesian technique.

$$P(n_{\text{on}}|s) = \int db \text{Pois}(n_{\text{on}}|s + b) \pi(b), \quad p = \sum_{n \in n_{\text{obs}}}^{\infty} P(n|s).$$

Principled version (eg. Z_{Γ}):

- ▶ clearly state prior $\eta(b)$; identify control samples (sidebands) and use:

$$\pi(b) = P(b|n_{\text{off}}) = \frac{P(n_{\text{off}}|b)\eta(b)}{\int db P(n_{\text{off}}|b)\eta(b)}.$$

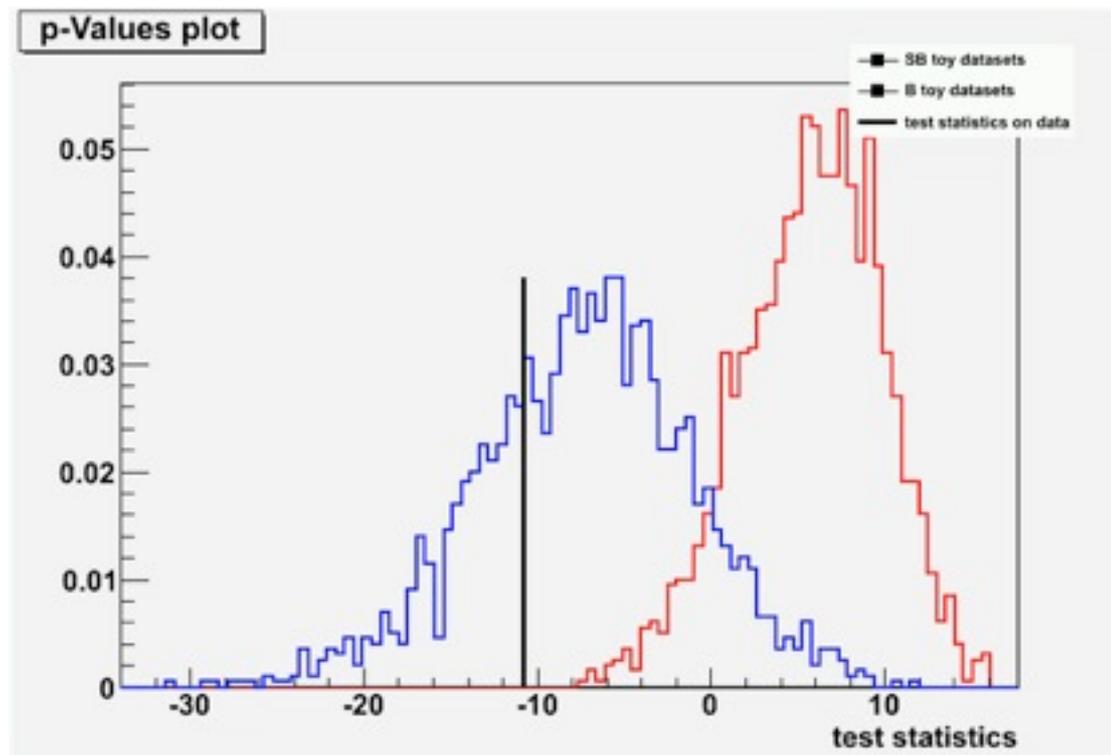
Ad-hoc version (eg. Z_{N}):

- ▶ unable or unwilling to justify $\pi(b)$, so go straight to some distribution
 - eg. a Gaussian, truncated Gaussian, log normal, Gamma, etc...
 - often the case for real systematic uncertainty (eg. MC generators, different background estimation techniques, etc.)

Recommendation: Avoid ad hoc priors if possible.

The HybridCalculator is undergoing a major redesign to use the same TestStatistic/TestStatSampler/SamplingDistribution design as the NeymanConstruction tools.

- ▶ target: end of summer 2010





A Little about Combinations

Start with a **model** for the data, eg. a probability density function for x written $P(x|\mu, \nu)$ that is parametrized by

- ▶ **parameters of interest:** $\mu : m_H, \sigma, \dots$
- ▶ **nuisance parameters:** $\nu : b, JES, \epsilon_b, \dots$

The **likelihood function** is given by

$$L(\mu, \nu) = \prod_{i \in \text{events}} P(x_i | \mu, \nu)$$

I will often refer to the **profile likelihood ratio:**

$$\lambda(\mu) = \frac{L(\mu, \hat{\nu})}{L(\hat{\mu}, \hat{\nu})}$$

Where $\hat{\nu}$ is the maximum likelihood estimator with μ fixed

$$L(\mu, \nu)$$

Basic formalism: one dataset

Start with a **model** for the data, eg. a probability density function for x written $P(x|\mu, \nu)$ that is parametrized by

- ▶ **parameters of interest:** $\mu : m_H, \sigma, \dots$
- ▶ **nuisance parameters:** $\nu : b, JES, \epsilon_b, \dots$

The **likelihood function** is given by

$$L(\mu, \nu) = \prod_{i \in \text{events}} P(x_i | \mu, \nu)$$

I will often refer to the **profile likelihood ratio**:

Where $\hat{\nu}$ is the maximum likelihood estimator with μ fixed

Remember, $L(\mu, \nu)$ is not a probability.

With two datasets, the observables may be different and one needs a **model** for each dataset.

- ▶ models may have different nuisance parameters, but should share parameters of interest, eg.

$$P_1(x|\mu, \nu) \quad P_2(y|\mu, \alpha)$$

The **likelihood function** is given by

$$L(\mu, \nu, \alpha) = \prod_{i \in \text{data1}} P_1(x_i|\mu, \nu) \cdot \prod_{i \in \text{data2}} P_2(y_i|\mu, \alpha)$$

In RooFit/RooStats, this type of situation is represented by:

- ▶ a “combined dataset” for data1, data2 with a “category label”

- ▶ a “simultaneous PDF”

- keeps track of P_1, P_2
- and associated category

x	y	category
2.7	-	1
1.3	-	1
-	5.1	2
-	7.2	2

The diagram shows two vertical boxes representing individual datasets. The first box has 'x' at the top, followed by values 2.7 and 1.3. The second box has 'y' at the top, followed by values 5.1 and 7.2. These are added together to form a larger table with three columns: 'x', 'y', and 'category'. The 'category' column has values 1 for the first two rows and 2 for the last two rows.

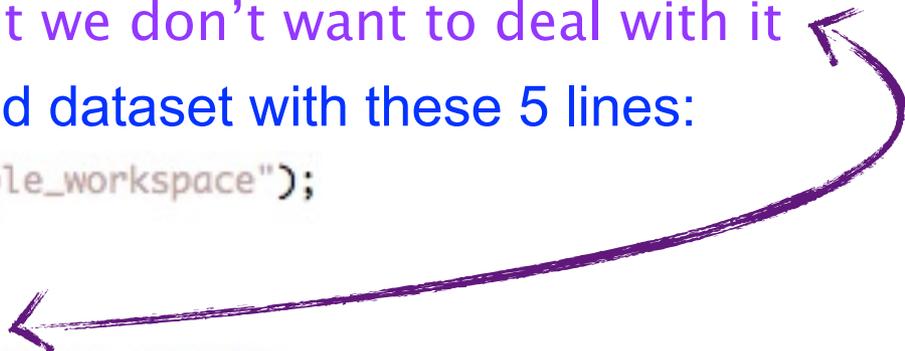
Providing ingredients via the workspace

To perform the combination, we need these basic ingredients:

- ▶ we need the full model (eg. the simultaneous PDF) for each channel
- ▶ we need the combined dataset for each channel
- ▶ we need to know what the names of the parameters of interest are and what they correspond to for each channel
- ▶ we do NOT need to know and we don't want to worry about:
 - structure and complexity of PDF
 - structure of the combined dataset
 - anything about the nuisance parameters (unless we want to introduce correlations between them in the combination)
- ▶ technically we need any custom code, but we don't want to deal with it

This can be achieved for an arbitrary model and dataset with these 5 lines:

```
RootWorkspace workspace("Example_workspace");  
workspace.import(*data);  
workspace.import(*pdf);  
workspace.importClassCode();  
workspace.writeToFile("myWorkspace.root");
```

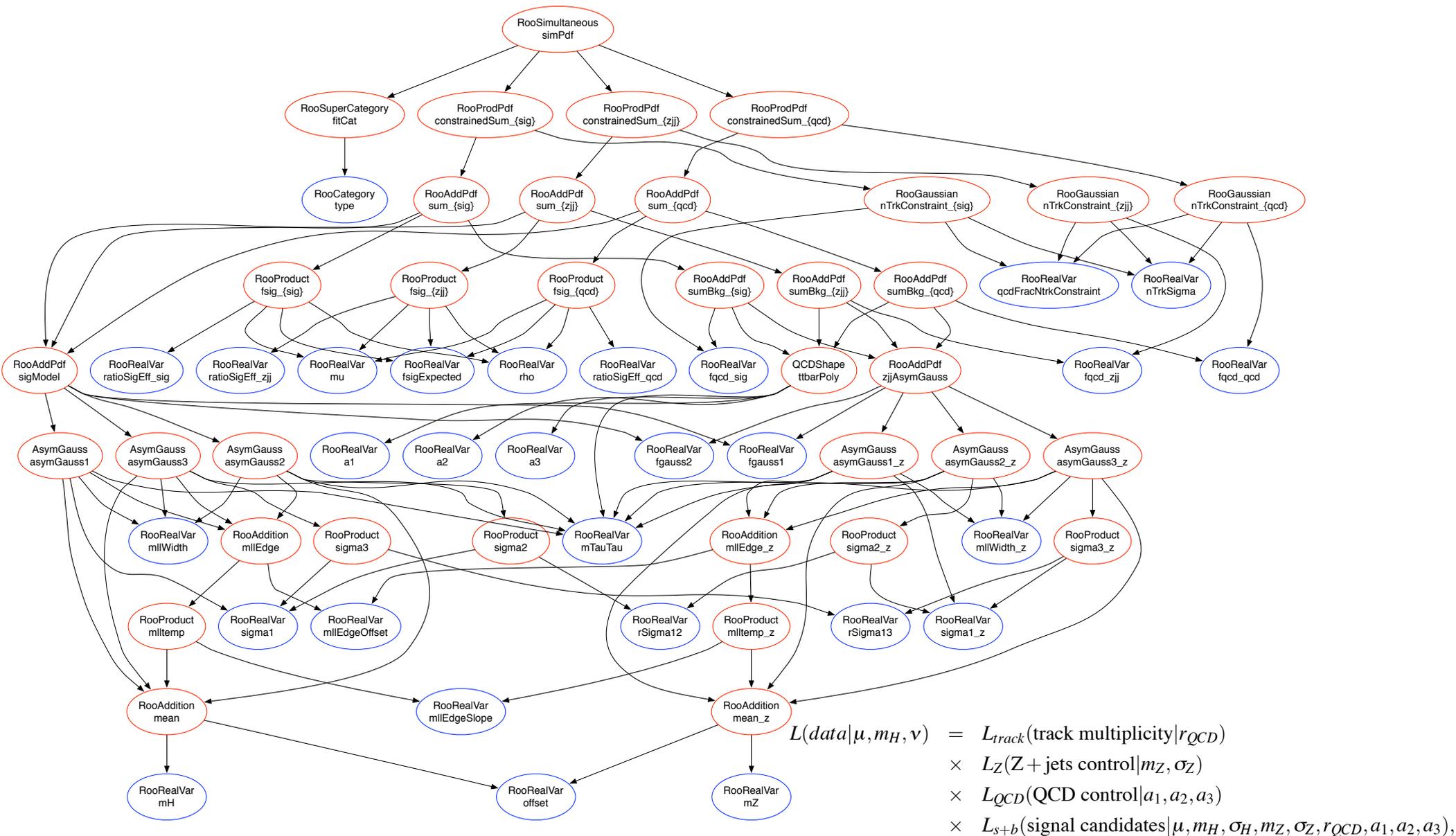


This is **exactly** what is needed from each channel to perform combination

One channel with control samples



An example search channel that already is a simultaneous pdf over a signal region, a Z+jets control sample, and a ttbar & W+jets control region



The actual combination code

These are the ~50 lines it took to perform combination of two complicated channels.

- ▶ This is being abstracted into a RooStats combination utility

```
void combine(string mass,double& mllSig, double& mlhSig, double& mcombSig){
```

```
TFile llf(("workspace_simul_ll_m"+mass+".root").c_str());  
RooWorkspace* llWS = (RooWorkspace*) llf.Get("Example_workspace");  
TFile lhf(("workspace_simul_lh_m"+mass+".root").c_str());  
RooWorkspace* lhWS = (RooWorkspace*) lhf.Get("Example_workspace");
```

```
// get original models
```

```
RooAbsPdf* llModel = llWS->pdf("simPdf");
```

```
RooAbsPdf* lhModel = lhWS->pdf("simPdf");
```

```
cout << "models = " << llModel << " " << lhModel << endl;
```

```
// should be able to use abs data, but Import(string, data) needs specific on
```

```
RooDataSet* lldata = (RooDataSet*) llWS->data("data");
```

```
RooDataSet* lhdata = (RooDataSet*) lhWS->data("data");
```

```
// add models to new WS with suffix
```

```
RooWorkspace* combWS = new RooWorkspace("combWS");
```

```
combWS->import(*llModel, RenameAllNodes("ll"), RenameAllVariables("ll"));
```

```
combWS->import(*lhModel, RenameAllNodes("lh"), RenameAllVariables("lh"));
```

```
// need to rename type_ll and type_lh back to common type
```

```
combWS->import(*llWS->cat("type"));
```

```
RooCategory* type = combWS->cat("type");
```

```
combWS->factory("EDIT::ll_tmp(simPdf_ll, type_ll=type)");
```

```
combWS->factory("EDIT::lh_tmp(simPdf_lh, type_lh=type)");
```

```
combWS->Print();
```

```
// get new WS back out
```

```
llModel = combWS->pdf("ll_tmp");
```

```
lhModel = combWS->pdf("lh_tmp");
```

```
RooCategory* channel = new RooCategory("channel","channel");
```

```
channel->defineType("ll",0);
```

```
channel->defineType("lh",1);
```

```
RooArgSet vars=((RooArgSet*)lldata->get()->clone("vars"));
```

```
vars.Print();
```

```
RooRealVar w("w","w",0,10000);
```

```
vars.add(w);
```

```
RooDataSet* combData = new RooDataSet("combData", "combined dataset",  
vars,  
Index(*channel),  
Import("ll",*lldata),  
Import("lh",*lhdata),  
WeightVar(w));
```

```
map<string, RooAbsPdf*> pdfMap;
```

```
string llStr="llStr";
```

```
string lhStr="lhStr";
```

```
pdfMap["ll"]=llModel;
```

```
pdfMap["lh"]=lhModel;
```

```
RooSimultaneous* combModel_split
```

```
= new RooSimultaneous("combModel_split", "", pdfMap, *channel);
```

```
combWS->import(*combModel_split);
```

```
channel = combWS->cat("channel"); // get pointer to one in WS
```

```
// here we establish correspondence of common variables
```

```
combWS->import(*llWS->var("mH"));
```

```
combWS->import(*llWS->var("mu"));
```

```
combWS->import(*llWS->var("mTauTau"));
```

```
combWS->factory("EDIT::combModel(combModel_split,mH_ll=mH,mH_lh=mH,\  
ll=mu,mu_lh=mu,mTauTau_lh=mTauTau,mTauTau_ll=mTauTau)");
```

```
RooAbsPdf* combModel = combWS->pdf("combModel");
```

```
// do the combined fit
```

```
combModel->fitTo(*combData, Constrained(), Hesse(kFALSE), Minos(kFALSE));
```

```
// do profile LR
```

```
RooRealVar* mu = combWS->var("mu");
```

```
RooAbsReal* nll = combModel->createNLL(*combData, Constrained());
```

```
RooAbsReal* profile = nll->createProfile(*mu);
```

```
mu->setVal(0);
```

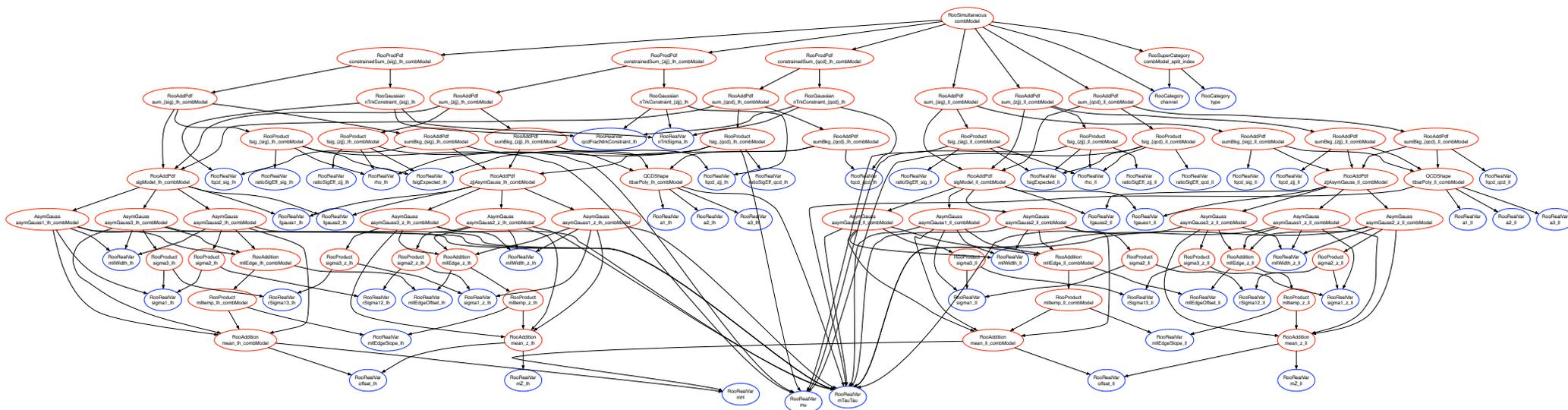
```
cout << "sqrt(-2loglambda(0)) = " << sqrt(2*profile->getVal()) << endl;
```

Resulting combined model

The resulting combined model combines 2 signal regions, 4 control regions, and 1 external measurement (track multiplicity of hadronic tau) that constrains QCD fraction in the lepton-hadron channel

- Since nuisance parameters are extracted from independent measurements, no additional terms were added to correlate the nuisance parameters between channels (14 nuisance in total)

- only shared parameters are m_H and $\mu = \frac{\sigma \times BR}{\sigma_{SM} \times BR_{SM}}$
- all the complexity of the model is hidden to combination code

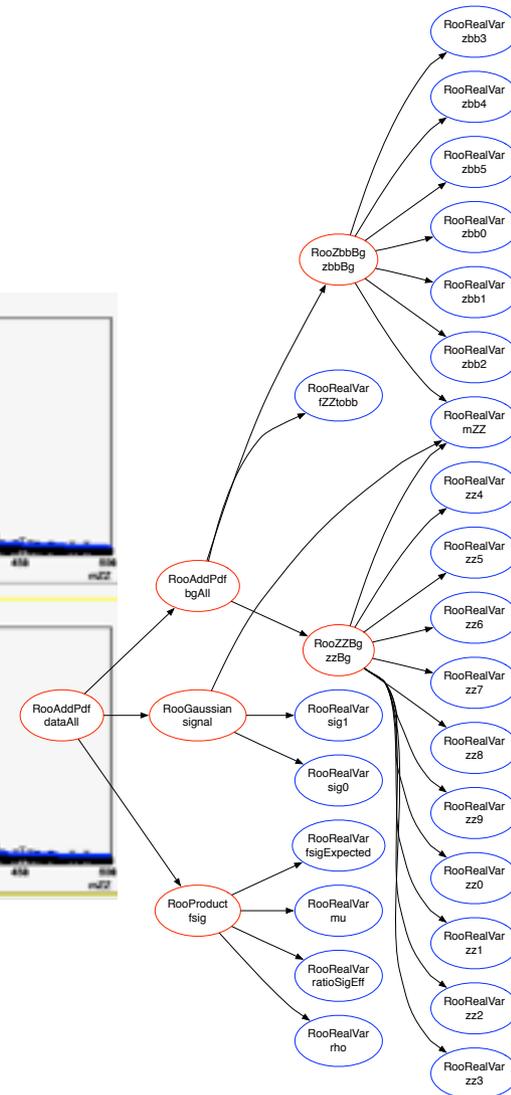
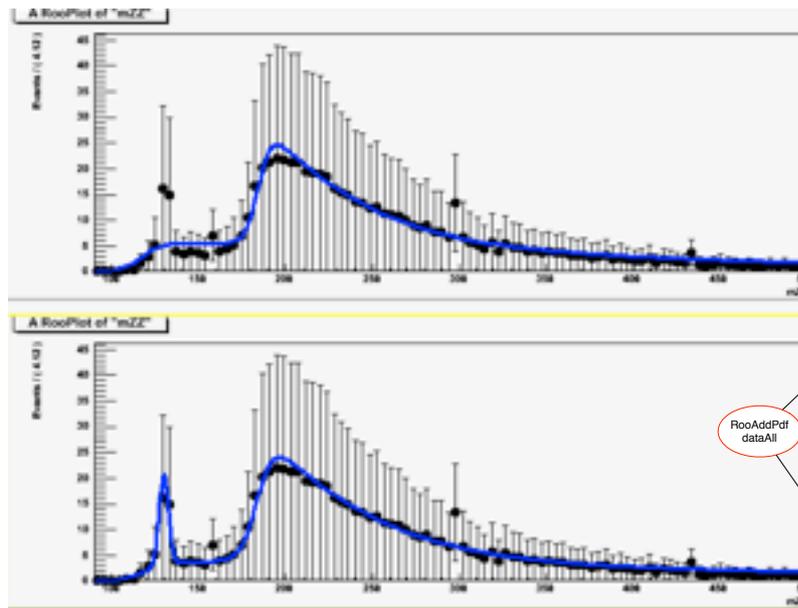
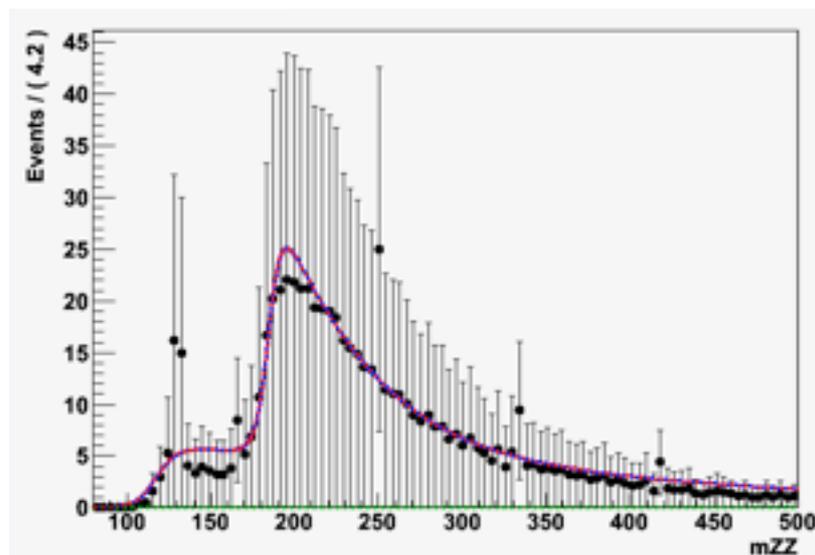


Adding Higgs to ZZ to 4 leptons to Combination

Extend previous example with H->4leptons

- ▶ working gorup quickly provided workspaces
- ▶ workspace included custom code for ZZ and Zbb background shapes

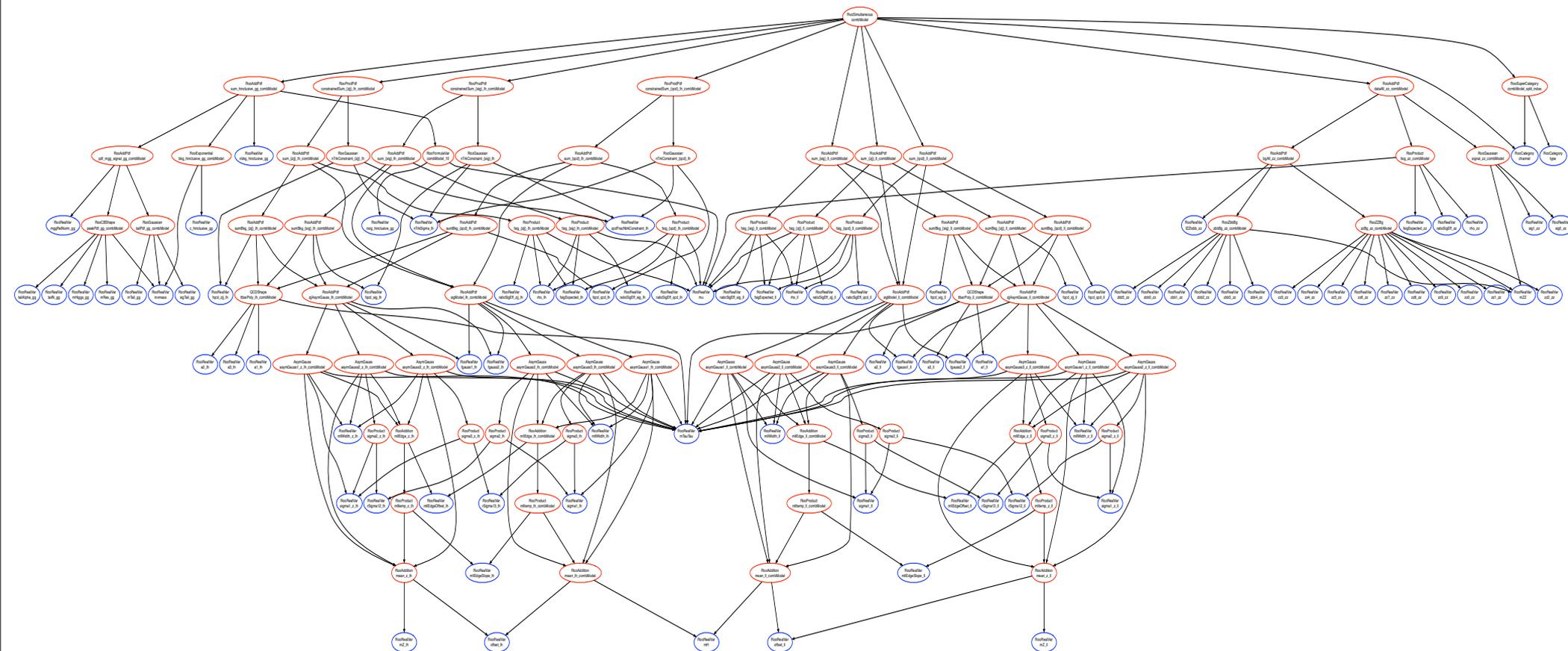
$$\frac{p0}{(1+e^{p7})(1+e^{\frac{mZZ-p8}{p9}})} + \frac{p1}{(1+e^{\frac{p2-mZZ}{p3}})(1+e^{\frac{p4-mZZ}{p5}})}$$



Combining the inputs

Using the same code as previous slide, with a few extra lines for the new channels, we arrive at the combined dataset & model

- ▶ here the only common parameter is μ , the master signal strength
 - could easily make Higgs mass be the same for all three channels
- ▶ the combined model has 27 nuisance parameters



Combining the inputs

Here with one workspace we can see the combination of 4 Higgs channels, together with their control samples, and plot of likelihood ratio

Color code:

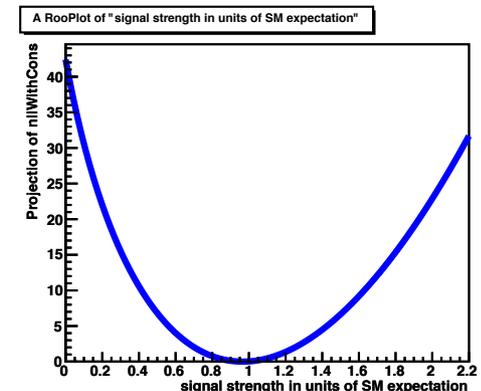
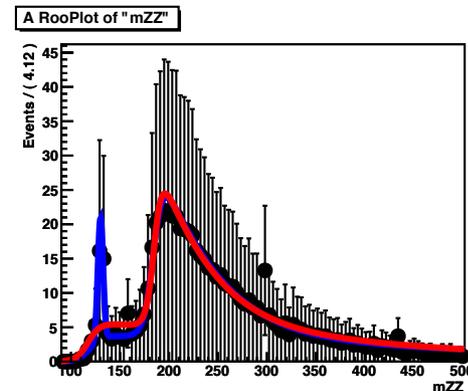
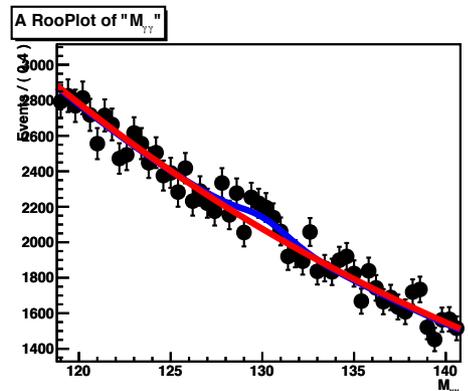
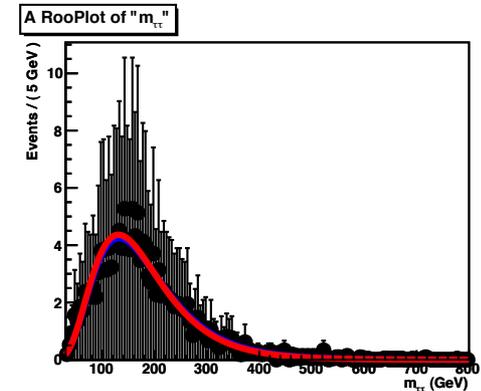
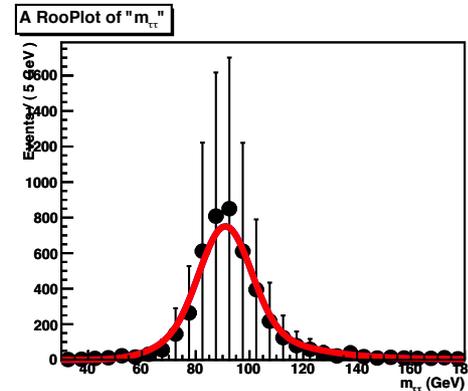
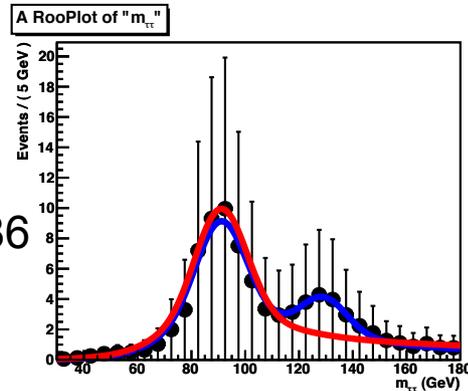
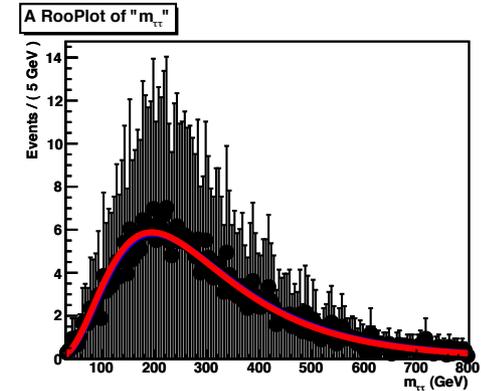
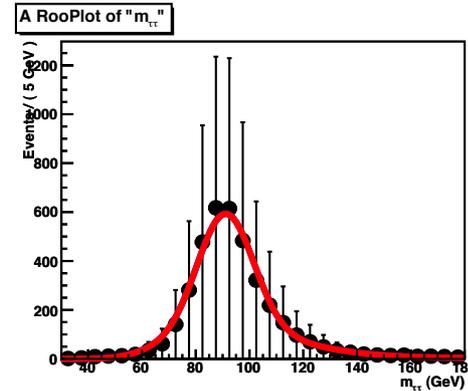
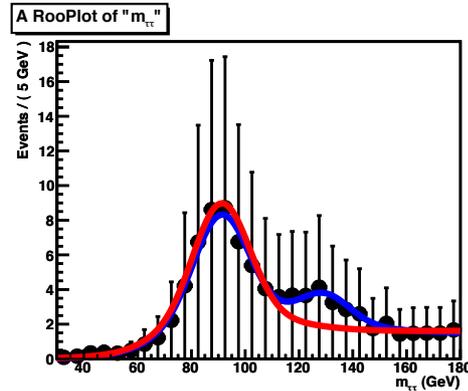
$$\mu = 0$$

$$\hat{\mu}$$

Results:

$$-2 \log \lambda(0) = 71.86$$

$$\sigma = 8.47703$$





HistFactory

Many analyses are based on template histograms (ROOT TH1)

- ▶ provide a tool that allows one to use RooStats statistical tools without knowing RooFit's data modeling language

In this approach, user provides other templates corresponding to variations of individual systematics

- ▶ this is done for each source of systematic and for each signal and background individually
- ▶ It is straightforward to provide a combined model for several channels and to identify the same systematic in each channel

The user specifies all of these systematics via an XML file and a compiled command line executable parses the XML file to produce the combined model

- ▶ by default, it also runs a profile likelihood analysis on the parameters of interest

For each sig & bkg estimate, the expected number of events is modeled as

$$N_{exp} = L f \epsilon(\alpha) \sigma(x; \alpha)$$

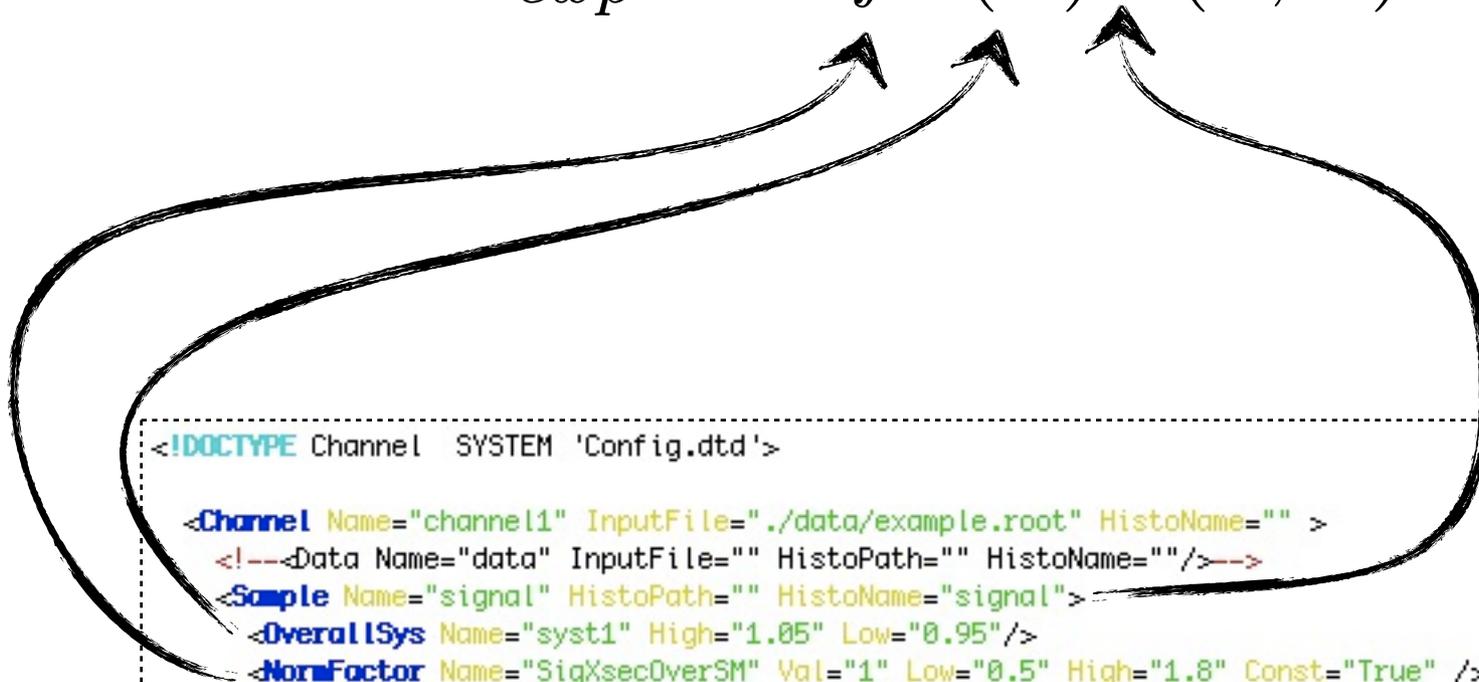
- For data-driven estimates, $L=L_0$, the nominal luminosity
- For theory-driven estimates L is an nuisance parameter (constrained)
- f is an overall scaling factor that is left unconstrained
 - these are typically things we measure, like $\mu=\sigma/\sigma_{SM}$
 - can also be a ratio of cross-sections $r=\sigma_{tt}/\sigma_Z$ or $r=\sigma_{\mu\mu}/\sigma_{e\mu}$
- $\epsilon(\alpha)$ is an efficiency or acceptance term assembled from the individual systematics, and there is an α for each source of systematic
- $\sigma(x;\alpha)$ is a histogram for the variable x (in units of cross-section) that interpolates between different variational histograms

By using the same name for the systematic source or scale factor, one can assemble complex combined models that are very general

A 1-channel example, where signal histogram normalization multiplied by “SigXsecOverSM”, which is considered the parameter of interest.

- Nuisance parameters α_j : “Lumi”, “syst1” (sig only), “syst2” (bkg1 only), “syst3” (bkg2 only)

$$N_{exp} = L f \epsilon(\alpha) \sigma(x; \alpha)$$



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

<Channel Name="channel1" InputFile="./data/example.root" HistoName="" >
  <!--<Data Name="data" InputFile="" HistoPath="" HistoName="" />-->
  <Sample Name="signal" HistoPath="" HistoName="signal">
    <OverallSys Name="syst1" High="1.05" Low="0.95"/>
    <NormFactor Name="SigXsecOverSM" Val="1" Low="0.5" High="1.8" Const="True" />
  </Sample>
  <Sample Name="background1" HistoPath="" NormalizeByTheory="True" HistoName="background1">
    <OverallSys Name="syst2" Low="0.95" High="1.05"/>
  </Sample>
  <Sample Name="background2" HistoPath="" NormalizeByTheory="True" HistoName="background2">
    <OverallSys Name="syst3" Low="0.95" High="1.05"/>
    <!-- <HistoSys Name="syst4" HistoPathHigh="" HistoPathLow="histForSyst4"/>-->
  </Sample>
</Channel>
```

ROOT now has a new executable in \$ROOTSYS/bin called **hist2workspace**

- command line: **hist2workspace myAnalysis.xml**
- Can drive parameter settings, constraints, etc. via XML

```
<!DOCTYPE Combination SYSTEM 'Config.dtd'>

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

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

  <Measurement Name="Example" Lumi="10" LumiRelErr="0.05" BinLow="0" BinHigh="2" Mode="comb">
    <POI>SigXsecOverSM</POI>
    <ParamSetting Const="True">Lumi alpha_syst1</ParamSetting>
  </Measurement>

</Combination>
```

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

<Channel Name="channel1" InputFile="./data/example.root" HistoName="" >
  <!--<Data Name="data" InputFile="" HistoPath="" HistoName="" />-->
  <Sample Name="signal" HistoPath="" HistoName="signal">
    <OverallSys Name="syst1" High="1.05" Low="0.95"/>
    <NormFactor Name="SigXsecOverSM" Val="1" Low="0.5" High="1.8" Const="True" />
  </Sample>
  <Sample Name="background1" HistoPath="" NormalizeByTheory="True" HistoName="background1">
    <OverallSys Name="syst2" Low="0.95" High="1.05"/>
  </Sample>
  <Sample Name="background2" HistoPath="" NormalizeByTheory="True" HistoName="background2">
    <OverallSys Name="syst3" Low="0.95" High="1.05"/>
    <!-- <HistoSys Name="syst4" HistoPathHigh="" HistoPathLow="histForSyst4"/>-->
  </Sample>
</Channel>
```

Parametrized templates (eg. systematics)

Given nominal histograms and +/- variations for each source of systematic α_j produce a family of predictions parametrized by α_j with linear interpolation:

$$\varepsilon_{jk}(\alpha_j) = \begin{cases} \tilde{\varepsilon}_{jk} + \alpha_j(\varepsilon_{jk}(\alpha_j^+) - \tilde{\varepsilon}_{jk}) & \text{if } \alpha_j > 0 \\ \tilde{\varepsilon}_{jk} & \text{if } \alpha_j = 0 \\ \tilde{\varepsilon}_{jk} - \alpha_j(\varepsilon_{jk}(\alpha_j^-) - \tilde{\varepsilon}_{jk}) & \text{if } \alpha_j < 0. \end{cases}$$

Now expectation is:

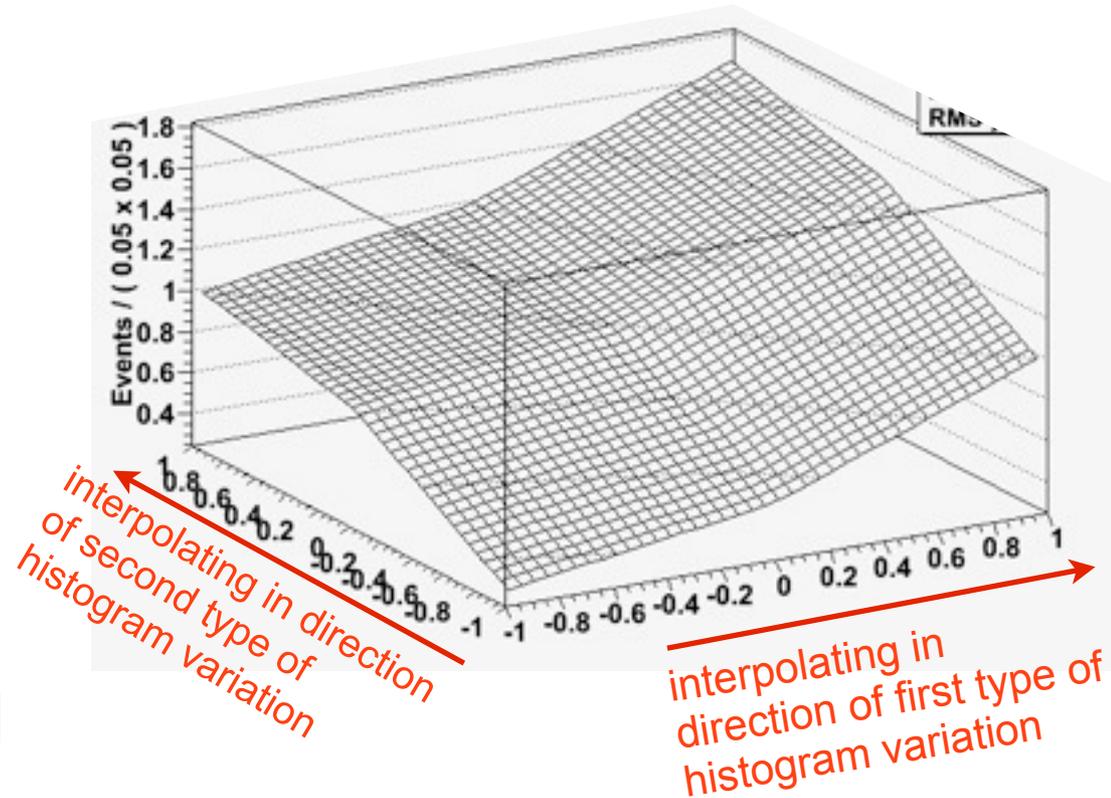
$$N_k^{exp} = \mathcal{L} \sigma_k \prod_j \tilde{\varepsilon}_{jk} \frac{\varepsilon_{jk}(\alpha_j)}{\tilde{\varepsilon}_{jk}} = \tilde{N}_k^{exp} \prod_j \frac{\varepsilon_{jk}(\alpha_j)}{\tilde{\varepsilon}_{jk}}$$

Then, constrain α_j with Normal

$$L(\sigma_{sig}, \mathcal{L}, \alpha_j) = Pois(N^{obs} | N_{tot}^{exp}) \times Gaus(\tilde{\mathcal{L}} | \mathcal{L}, \sigma_{\mathcal{L}}) \times \prod_j Gaus(\tilde{\alpha}_j = 0 | \alpha_j, \Delta_{\alpha_j} = 1).$$

Generalization for multiple bins and multiple channels:

$$L(\sigma_{sig}, \mathcal{L}, \alpha_j) = \prod_{l \in \{ee, \mu\mu, e\mu\}} \left\{ \prod_{i \in bins} \left[Pois(N_i^{obs} | N_{i,tot}^{exp}) Gaus(\tilde{\mathcal{L}} | \mathcal{L}, \sigma_{\mathcal{L}}) \prod_{j \in syst} Gaus(0 | \alpha_j, 1) \right] \right\}$$



Support for different constraints

For large uncertainties, truncated Gaussian are a bad choice

- ▶ often lead to optimistic p-values, short tail, bad behavior at 0

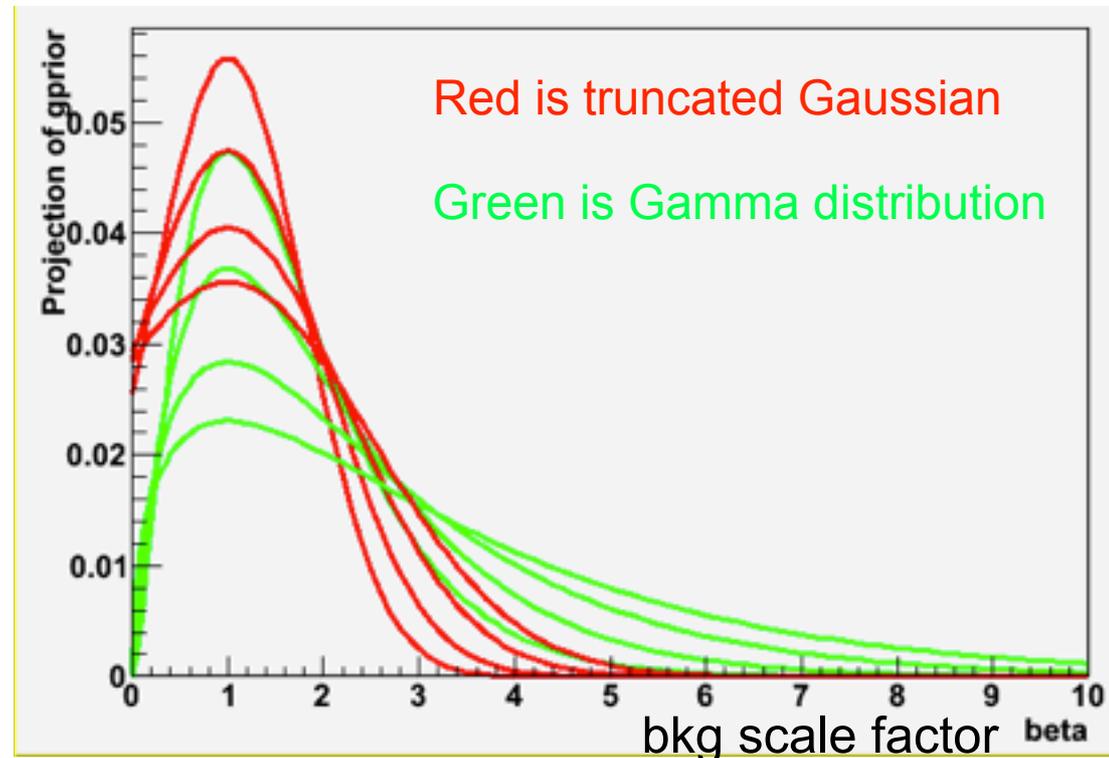
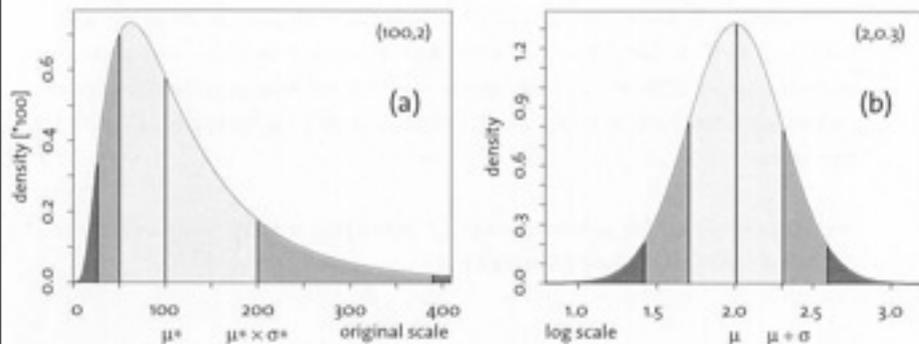
Gamma (reasonable for systematics constrained from measurements)

- ▶ longer tail, good behavior near 0, natural choice if auxiliary is based on counting

Log-normal and Uniform are other popular choices: **Now implemented**

- ▶ Thanks to Dominique Tardif for helping with XML parsing!
- ▶ standard conventions relate parameters to physicists notion of “relative uncertainty”
- ▶ For small relative error, tool produces consistent results for Gaus, Gamma, LogNormal

PDF	Prior	Posterior
Gaussian	uniform	Gaussian
Poisson	uniform	Gamma
Log-normal	reference	Log-Normal





SPlots in RooStats

The *sPlot* technique is developed in the above context of a maximum Likelihood method making use of discriminating variables.

One considers a data sample in which are merged several species of events. These species represent various signal components and background components which all together account for the data sample. The different terms of the log-Likelihood are:

N the total number of events in the data sample,

N_s the number of species of events populating the data sample,

N_i the number of events expected on the average for the i^{th} species,

$f_i(y_e)$ the value of the PDFs of the discriminating variables y for the i^{th} species and for event e ,

x the set of control variables which, by definition, do not appear in the expression of the Likelihood function L

The extended log-Likelihood reads:

$$\mathcal{L} = \sum_{e=1}^N \ln \left\{ \sum_{i=1}^{N_s} N_i f_i(y_e) \right\} - \sum_{i=1}^{N_s} N_i . \quad (8)$$

From this expression, after maximization of L with respect to the N_i parameters, a weight can be computed for every event and each species, in order to obtain later the true distribution $\mathbf{M}_i(x)$ of variable x . If n is one of the N_s species present in the data sample, the weight for this species is defined by:

$${}_s\mathcal{P}_n(y_e) = \frac{\sum_{j=1}^{N_s} \mathbf{V}_{nj} f_j(y_e)}{\sum_{k=1}^{N_s} N_k f_k(y_e)} , \quad (9)$$

where \mathbf{V}_{nj} is the covariance matrix resulting from the Likelihood maximization. This matrix can be used directly from the fit, but this is numerically less accurate than the direct computation:

$$\mathbf{V}_{nj}^{-1} = \frac{\partial^2(-\mathcal{L})}{\partial N_n \partial N_j} = \sum_{e=1}^N \frac{f_n(y_e) f_j(y_e)}{(\sum_{k=1}^{N_s} N_k f_k(y_e))^2} . \quad (10)$$

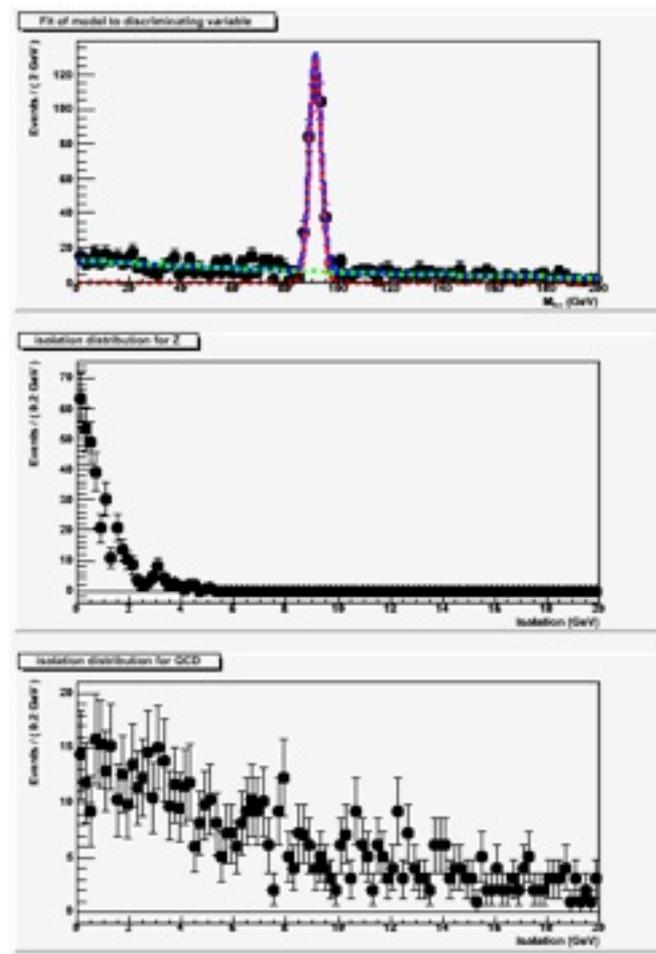
The distribution of the control variable x obtained by histogramming the weighted events reproduces, on average, the true distribution $\mathbf{M}_n(x)$.

Documentation taken from M. Pivik's TSPlot tool

M. Pivik and F. R. Le Diberder, Nucl. Inst. Meth. A (in press), physics/0402083.

SPlot: Working SPlot implementation that works for arbitrary models

- rewritten from original code from BaBar
- more general than TSPLOT class
- http://root.cern.ch/root/html/tutorials/roostats/rs301_splot.C.html





Bernstein Correction Utility

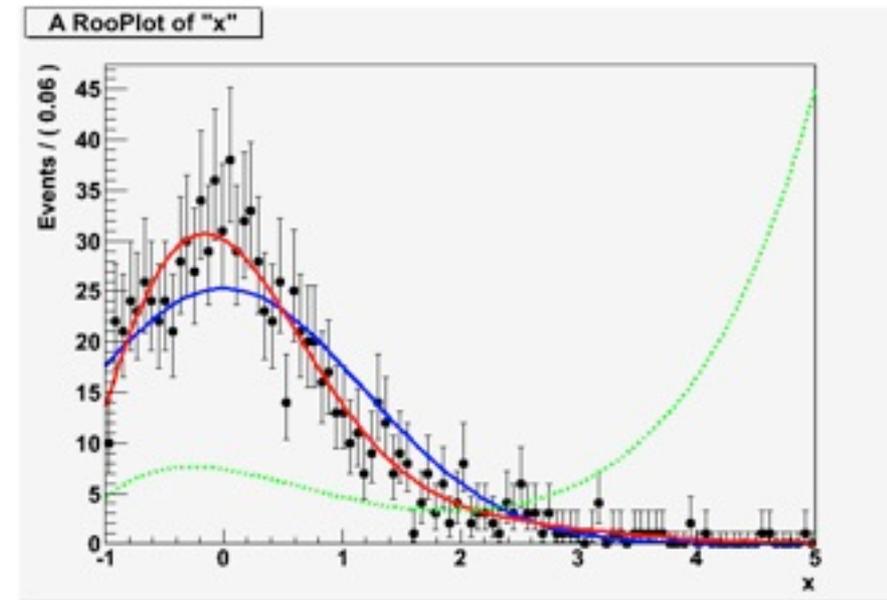
BernsteinCorrection: prompted by work in our statistics forum, automate procedure of correcting nominal model to data by including a correction factor based on a polynomial whose degree is chosen in a principled way

- http://root.cern.ch/root/html/tutorials/roostats/rs_bernsteinCorrection.C.html

9.5 Bernstein Correction

BernsteinCorrection is a utility in RooStats to augment a nominal PDF with a polynomial correction term. This is useful for incorporating systematic variations to the nominal PDF. The Bernstein basis polynomials are particularly appropriate because they are positive definite.

This tool was inspired by the work of Glen Cowan together with Stephan Horner, Sascha Caron, Eilam Gross, and others. The initial implementation is independent work. The major step forward in the approach was to provide a well defined algorithm that specifies the order of polynomial to be included in the correction. This is an empirical algorithm, so in addition to the nominal model it needs either a real data set or a simulated one. In the early work, the nominal model was taken to be a histogram from Monte Carlo simulations, but in this implementation it is generalized to an arbitrary PDF (which includes a RooHistPdf). The algorithm basically consists of a hypothesis test of an n -th order correction (null) against a $n+1$ -th order correction (alternate). The quantity $q = -2 \log LR$ is used to determine whether the $n+1$ -th order correction is a major improvement to the n -th order correction. The distribution of q is expected to be roughly χ^2 with one degree of freedom if the n -th order correction is a good model for the data. Thus, one only moves to the $n+1$ -th order correction if q is relatively large. The chance that one moves from the n -th to the $n+1$ -th order correction when the n -th order correction (eg. a type 1 error) is sufficient is given by the $\text{Prob}(\chi_1^2 > \text{threshold})$. The constructor of this class allows you to directly set this tolerance (in terms of probability that the $n+1$ -th term is added unnecessarily).



data

nominal model

corrected model

correction factor



Jeffreys's Prior

Jeffreys's Prior is an "objective" prior based on formal rules
(it is related to the Fisher Information and the Cramér-Rao bound)

$$\pi(\vec{\theta}) \propto \sqrt{\det \mathcal{I}(\vec{\theta})}. \quad (\mathcal{I}(\theta))_{i,j} = -E \left[\frac{\partial^2}{\partial \theta_i \partial \theta_j} \ln f(X; \theta) \middle| \theta \right].$$

Eilam, Glen, Ofer, and I showed in [arXiv:1007.1727](https://arxiv.org/abs/1007.1727) that the Asimov data provides a fast, convenient way to calculate the Fisher Information

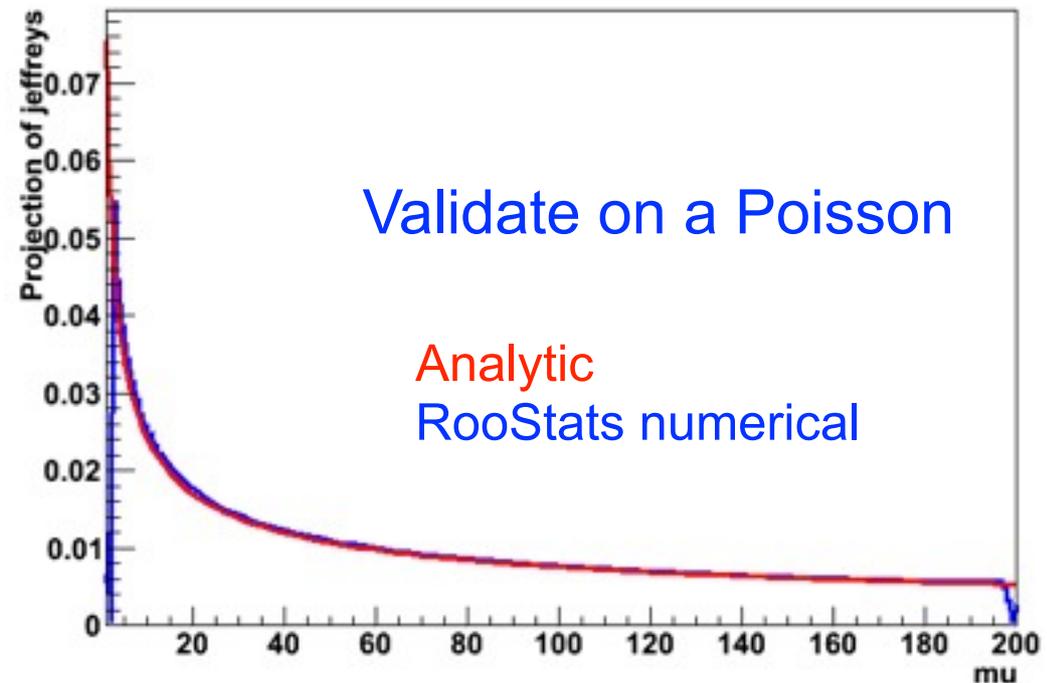
$$V_{jk}^{-1} = -E \left[\frac{\partial^2 \ln L}{\partial \theta_j \partial \theta_k} \right] = -\frac{\partial^2 \ln L_A}{\partial \theta_j \partial \theta_k} = \sum_{i=1}^N \frac{\partial \nu_i}{\partial \theta_j} \frac{\partial \nu_i}{\partial \theta_k} \frac{1}{\nu_i} + \sum_{i=1}^M \frac{\partial u_i}{\partial \theta_j} \frac{\partial u_i}{\partial \theta_k} \frac{1}{u_i}$$

Use this as basis to calculate
Jeffreys's prior for an arbitrary PDF!

```
RooWorkspace w("w");
w.factory("Uniform::u(x[0,1])");
w.factory("mu[100,1,200]");
w.factory("ExtendPdf::p(u,mu)");

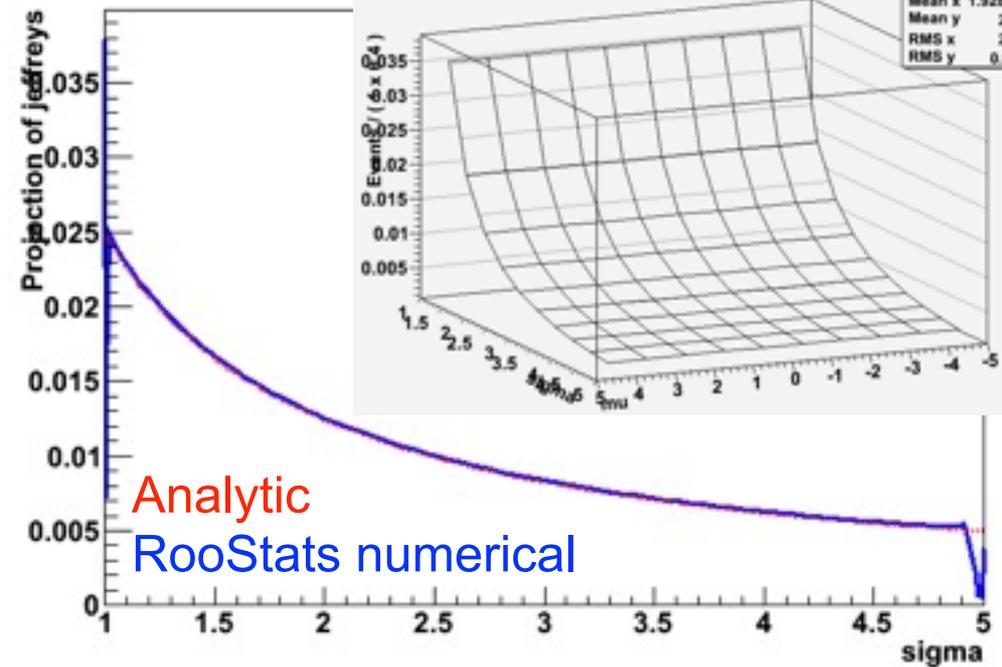
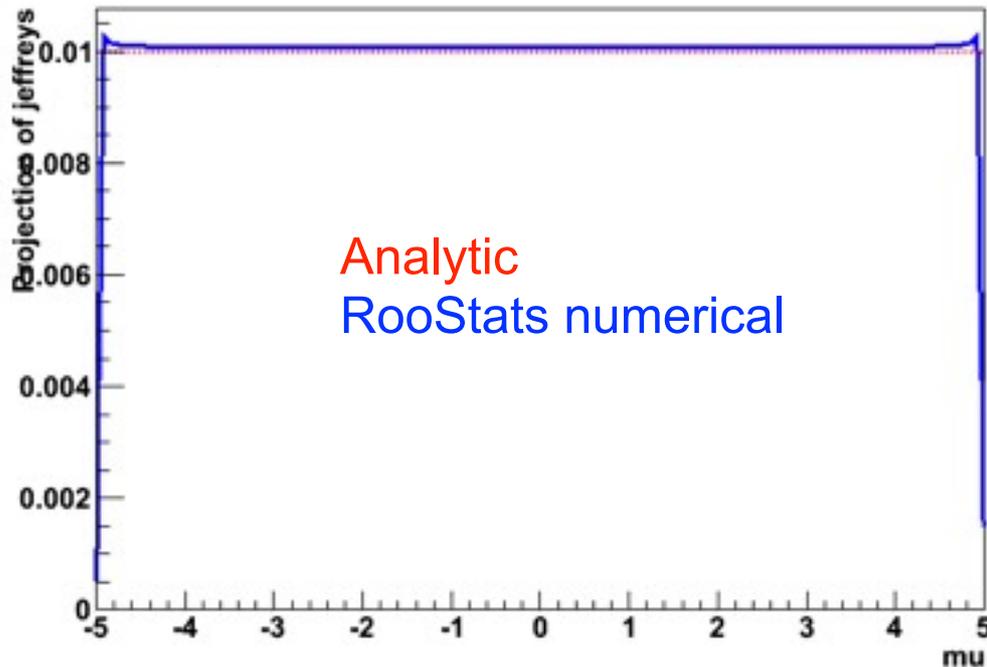
w.defineSet("poi","mu");
w.defineSet("obs","x");
// w.defineSet("obs2","n");
```

```
RooJeffreysPrior pi("jeffreys","jeffreys",*w.pdf("p"),*w.set("poi"),*w.set("obs"));
```



Validate Jeffreys's Prior on a Gaussian μ , σ , and (μ, σ)

```
RoofWorkspace w("w");  
w.factory("Gaussian::g(x[0,-20,20],mu[0,-5,5],sigma[1,0,10])");  
w.factory("n[10,.1,200]");  
w.factory("ExtendPdf::p(g,n)");  
w.var("n")->setConstant();  
  
w.var("sigma")->setConstant();  
w.defineSet("poi","mu");  
w.defineSet("obs","x");  
RootJeffreysPrior pi("jeffreys","jeffreys",*w.pdf("p"),*w.set("poi"),*w.set("obs"));
```



In [arXiv:1007.1727](https://arxiv.org/abs/1007.1727), Eilam, Glen, Ofer, and I also outlined the generalization of Wilks's Thm. called Wald's Thm., which states asymptotic distribution of $\lambda(\mu)$ for $\mu \neq \mu_{\text{true}}$ is a non-central χ^2 with parameter

$$\Lambda_r = \sum_{i,j=1}^r (\theta_i - \theta'_i) \tilde{V}_{ij}^{-1} (\theta_j - \theta'_j)$$

Three forms:

without MathMore, sum of χ^2

$$f_X(x; k, \lambda) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} f_{Y_{k+2i}}(x),$$

with MathMore: $k \geq 2$, use Incomplete Bessel

$$f_X(x; k, \lambda) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{k/4-1/2} I_{k/2-1}(\sqrt{\lambda x})$$

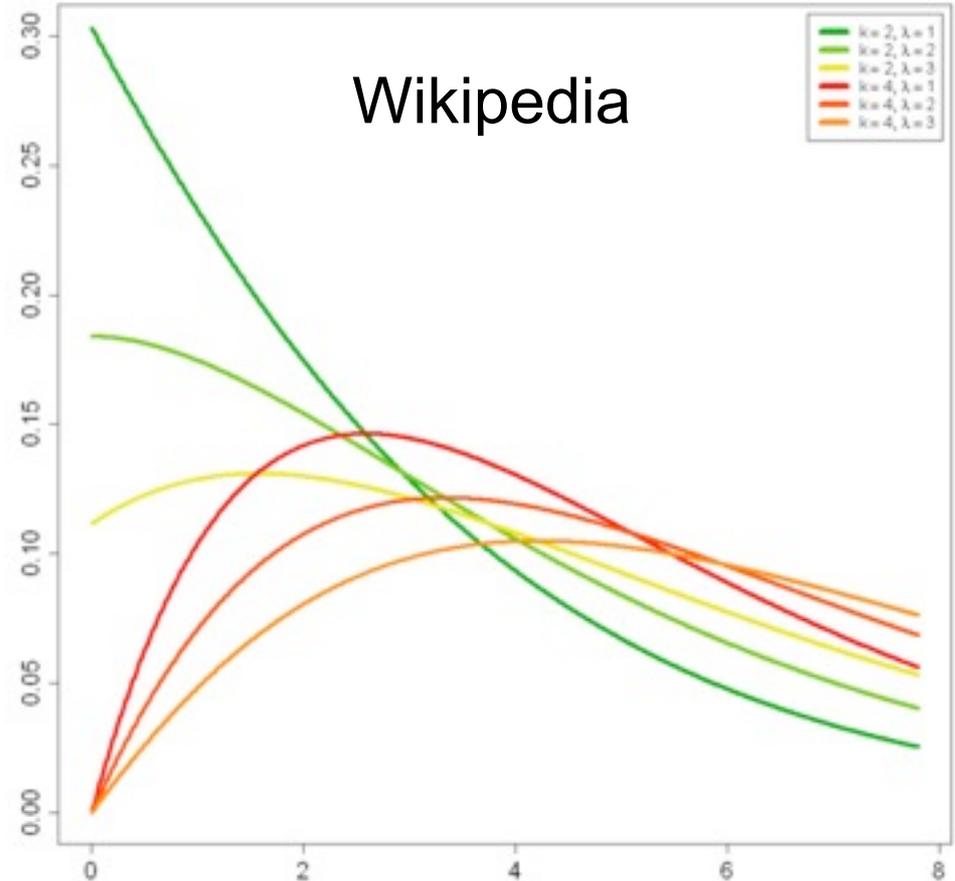
for $k < 2$ confluent hypergeometric functions

$$f_X(x; k, \lambda) = e^{-\lambda/2} {}_0F_1(; k/2; \lambda x/4) \frac{1}{2^{k/2} \Gamma(k/2)} e^{-x/2} x^{k/2-1}.$$

Test $x=5, k=3, \Lambda=1.5$:

RooStats 0.0972573

Matlab, R 0.097257



In [arXiv:1007.1727](https://arxiv.org/abs/1007.1727), Eilam, Glen, Ofer, and I also outlined the generalization of Wilks's Thm. called Wald's Thm., which states asymptotic distribution of $\lambda(\mu)$ for $\mu \neq \mu_{\text{true}}$ is a non-central χ^2 with parameter

$$\Lambda_r = \sum_{i,j=1}^r (\theta_i - \theta'_i) \tilde{V}_{ij}^{-1} (\theta_j - \theta'_j)$$

Three forms:

without MathMore, sum of χ^2

$$f_X(x; k, \lambda) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} f_{Y_{k+2i}}(x),$$

with MathMore: $k \geq 2$, use Incomplete Bessel

$$f_X(x; k, \lambda) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{k/4-1/2} I_{k/2-1}(\sqrt{\lambda x})$$

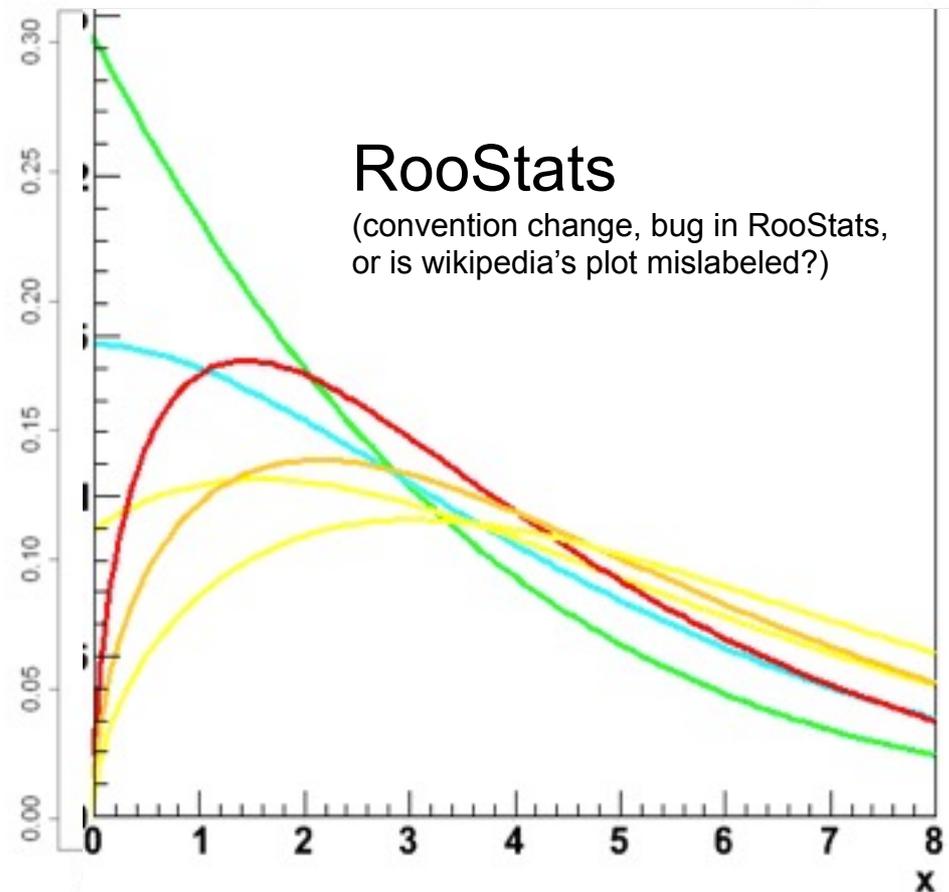
for $k < 2$ confluent hypergeometric functions

$$f_X(x; k, \lambda) = e^{-\lambda/2} {}_0F_1(; k/2; \lambda x/4) \frac{1}{2^{k/2} \Gamma(k/2)} e^{-x/2} x^{k/2-1}.$$

Test $x=5, k=3, \Lambda=1.5$:

RooStats 0.0972573

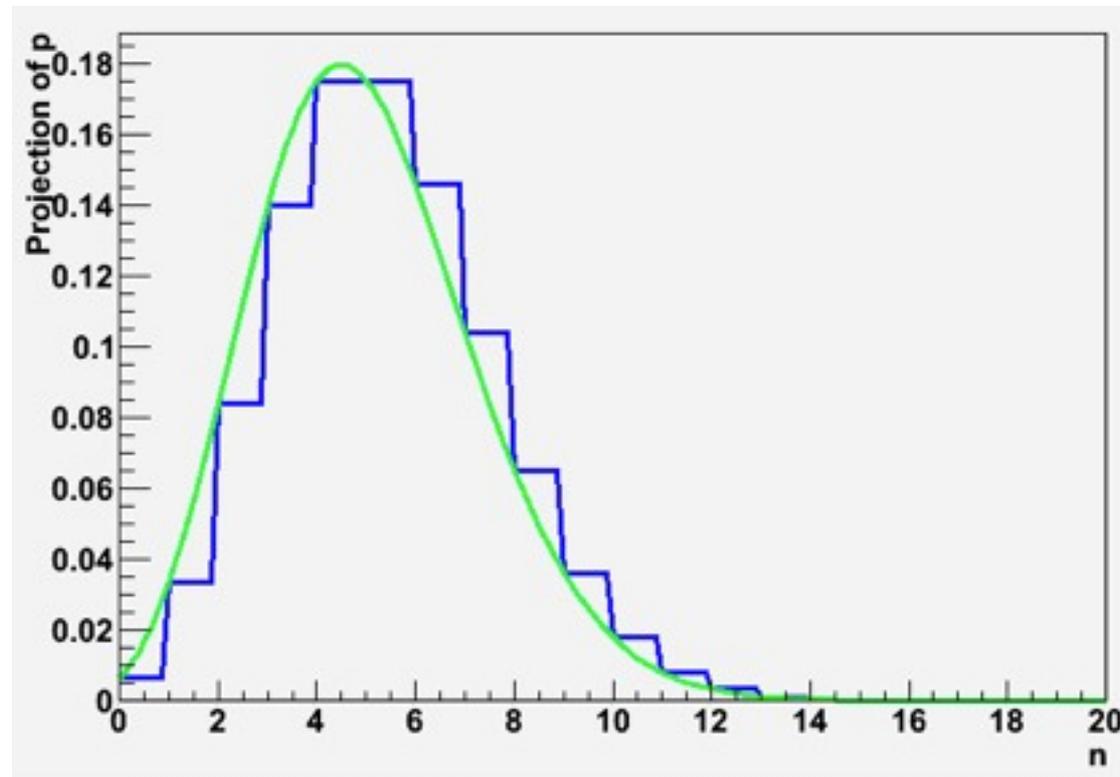
Matlab, R 0.097257



Added **Legender Polynomial** and **Spherical Harmonic** functions.

Request to have flag in Poisson PDF so that count is not rounded down to nearest integer.

- ▶ useful for expected data with non-integer counts
- ▶ **WARNING:** no consistent way to keep $\text{integral}=1$ and value at integer values invariant.



RooStats currently provides many useful tools for common statistical problems using well known statistical techniques

- ▶ Bayesian, Frequentist, and Likelihood-based methods available

The tools are still developing rapidly, so keep up to date with the current status via the RooStats web page and recent ROOT releases

<https://twiki.cern.ch/twiki/bin/view/RooStats/WebHome>

It is an open project for the field, and we would love to hear your ideas and incorporate any tools that you develop.

- ▶ please see the RooStats developers list



Some Statistical Recommendations (in the context of the prototype “on/off” problem)

This is a simplified problem that has been studied quite a bit to gain some insight into our more realistic and difficult problems

- ▶ **number counting with background uncertainty**
 - main measurement: observe n_{on} with $s+b$ expected
 - sideband measurement: observe n_{off} with τb expected

$$\underbrace{P(n_{\text{on}}, n_{\text{off}} | s, b)}_{\text{joint model}} = \underbrace{\text{Pois}(n_{\text{on}} | s + b)}_{\text{main measurement}} \underbrace{\text{Pois}(n_{\text{off}} | \tau b)}_{\text{sideband}}$$

- ▶ **Note: sideband is used to constrain background uncertainty**
 - In this approach “background uncertainty” is a statistical error
- ▶ **Contrast to:** $P(n_{\text{on}} | s) = \int db \text{Pois}(n_{\text{on}} | s + b) \pi(b)$,
 - where $\pi(b)$ is usually a Gaussian that is randomized in Toy MC
 - it is a Bayesian prior, resulting model is a Bayesian-averaged model
 - does not explicitly use knowledge of sideband $\text{Pois}(n_{\text{off}} | \tau b)$



Recommendation: where possible, one should express uncertainty on a parameter as a statistical (random) process

- ▶ explicitly include terms that represent auxiliary measurements in the likelihood

Recommendation: when using a Bayesian technique, one should explicitly express and separate the prior from the objective part of the probability density function

Example:

- ▶ **By writing** $P(n_{\text{on}}, n_{\text{off}} | s, b) = \text{Pois}(n_{\text{on}} | s + b) \text{Pois}(n_{\text{off}} | \tau b)$.
 - the objective statistical model is for the background uncertainty is clear
- ▶ One can then explicitly express a prior $\eta(b)$ and obtain:

$$\pi(b) = P(b | n_{\text{off}}) = \frac{P(n_{\text{off}} | b) \eta(b)}{\int db P(n_{\text{off}} | b) \eta(b)}.$$

Goal of Bayesian-frequentist hybrid solutions is to provide a frequentist treatment of the main measurement, while eliminating nuisance parameters (deal with systematics) with an intuitive Bayesian technique.

$$P(n_{\text{on}}|s) = \int db \text{Pois}(n_{\text{on}}|s + b) \pi(b), \quad p = \sum_{n \in n_{\text{obs}}}^{\infty} P(n|s).$$

Principled version (eg. Z_{Γ}):

- ▶ clearly state prior $\eta(b)$; identify control samples (sidebands) and use:

$$\pi(b) = P(b|n_{\text{off}}) = \frac{P(n_{\text{off}}|b)\eta(b)}{\int db P(n_{\text{off}}|b)\eta(b)}.$$

Ad-hoc version (eg. Z_{N}):

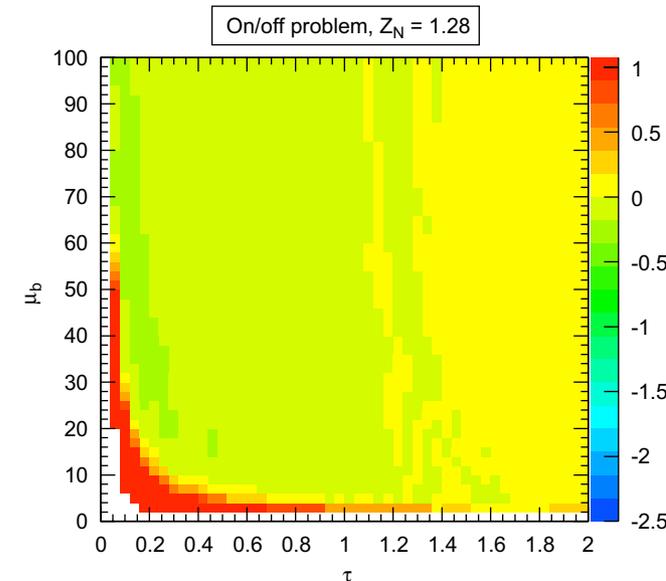
- ▶ unable or unwilling to justify $\pi(b)$, so go straight to some distribution
 - eg. a Gaussian, truncated Gaussian, log normal, Gamma, etc...
 - often the case for real systematic uncertainty (eg. MC generators, different background estimation techniques, etc.)

Recommendation: Avoid ad hoc priors if possible.

Hybrid solutions are common (used by LEP & Tevatron Higgs)

- ▶ this is what was done for TDR and several ATLAS & CMS results
- ▶ It is known that using an ad hoc prior can lead to optimistic results (eg. overstate significance, undercover).
 - it can be fairly significant, must be studied on a case-by-case basis

- [1] K. Cranmer, *Statistical challenges for searches for new physics at the LHC, Proceedings of PhyStat05: Statistical Problems in Particle Physics, Astrophysics and Cosmology, Oxford, England, United Kingdom (2005) SPIRES 6994415*
- [2] R. D. Cousins, J. T. Linnemann and J. Tucker, “Evaluation of three methods for calculating statistical significance when incorporating a systematic uncertainty into a test of the background-only hypothesis for a Poisson process,” *Nucl. Instrum. Meth. A* **595**, 480 (2008).
- [16] T. Junk, *Nucl. Instrum. Methods Phys. Res., Sec. A* **434**, 435 (1999)
- [19] G. C. Hill, “Comment on ‘Including systematic uncertainties in confidence interval construction for Poisson statistics’,” *Phys. Rev.* **D67**, 118101 (2003). [physics/0302057].



Recommendation: Avoid ad hoc priors if possible

Recommendation: When using an ad-hoc prior π , one should be honest about the assumptions that are involved and the limitations to the inference that they impose.

Warning: If one is not able to accurately characterize the prior beyond $Z\sigma$, then one cannot consistently discuss a deviation from the expected distributions beyond the $Z\sigma$ level.



The Z_{Bi} solution is a specific, analytical solution to the on/off problem

- ▶ not of general use

The general frequentist solution to this problem is the Neyman Construction, which provides coverage "by construction"

- ▶ However, the Neyman Construction is not uniquely determined; one must also specify:
 - the test statistic $T(n_{on}, n_{off}; s, b)$, which depends on data and parameters
 - a well-defined ensemble that defines the sampling distribution of T
 - the limits of integration for the sampling distribution of T
 - parameter points to scan (including the values of any nuisance parameters)
 - how the final confidence intervals in the parameter of interest are established

We need to specify all of these things to generalize Z_{Bi}

- ▶ See slides on Neyman-Construction and FeldmanCousins tool

No clear Bayesian solution among Z_{Bi} , Z_{Γ} , Z_{PL} , Z_N

- Bayesian solution would generically have a prior for the parameters of interest as well as nuisance parameters

See discussion in PDG

Recommendation: When performing a Bayesian analysis one should separate the objective likelihood function from the prior distributions to the extent possible.

Recommendation: When performing a Bayesian analysis one should investigate the sensitivity of the result to the choice of priors.

Warning: Flat priors in high dimensions can lead to unexpected and/or misleading results.

Recommendation: When performing a Bayesian analysis for a single parameter of interest, one should attempt to include Jeffreys's prior in the sensitivity analysis.



To support the points raised above, here are some quotes from professional statisticians (taken from selected PhyStat talks and selections from Bob Cousins lectures):

- “Perhaps the most important general lesson is that the facile use of what appear to be uninformative priors is a dangerous practice in high dimensions.” – Brad Efron
- “meaningful prior specification of beliefs in probabilistic form over very large possibility spaces is very difficult and may lead to a lot of arbitrariness in the specification.” – Michael Goldstein
- “Sensitivity Analysis is at the heart of scientific Bayesianism.” – Michael Goldstein
- “Non-subjective Bayesian analysis is just a part – an important part, I believe of a healthy sensitivity analysis to the prior choice...” J.M. Bernardo
- “Objective Bayesian analysis is the best frequentist tool around” – Jim Berger