Document version 0

April 12, 2010

RooStats

User's Guide

Kyle Cranmer, Grégory Schott, Lorenzo Moneta, Wouter Verkerke

With contributions from:

Kevin Belasco, Danilo Piparo, Giacinto Piacquadio, Maurizio Pierini, George H. Lewis, Alfio Lazzaro, Mario Pelliccioni, Matthias Wolf

Contents

1 Introduction

The RooStats project aims to provide a comprehensive, flexible, and validated suite of statistical tools within ROOT. Early on in the project it was decided to leverage the data modeling approach developed in RooFit, which is already well-established within high-energy physics and beyond. Thus, RooStats can be seen as providing high-level statistical tools, while RooFit provides the core data modeling language as well as many low-level aspects of the functionality. In the ongoing process of developing RooStats, RooFit is also undergoing rapid development.

One of the major goals of RooStats is to provide a unified framework for different statistical methods. Early on in the project it was demonstrated that RooFit is well suited for the three major types of statistical inference:

Classical / Frequentist This "school" of statistics restricts itself to making statements of the form "probability of the data given the hypothesis". The definition of probability in this context is based on a limit of frequencies of various outcomes. In that sense it is objective.

Bayesian This "school" of statistics allows one to make statements of the form "probability of the hypothesis given the data", which requires a prior probability of the hypothesis. Often the definition of probability in this context is a "degree of belief".

Likelihood-based This intermediate approach does not require a prior for the hypothesis, but also is not guaranteed to poses the properties that frequentists methods aim to achieve (or achieve by construction). These approaches do "obey the likelihood principle" (as do Bayesian methods), while frequentist methods do not.

The developers of RooStats appreciate that there are many different approaches to answering the same types of problems. There are pros and cons of various techniques, and we aim to provide a framework which can accommodate any of them.

One Model, Many Methods

A common scenario that we hope to address with RooStats is the comparison of different statistical approaches for the same statistical problem. Without a unified framework, these comparisons are complicated by the fact that each method must: a) re-create the model (eg. probability density function(s) for the data) and b) represent the data itself. Having redundant modeling and data representation is error prone and often complicates the comparison (or makes it practically impossible).[1] In that sense, RooStats aims to be like TMVA[2], providing utilities to easily try and compare multiple statistical techniques. By relieving the technical overhead associated to these types of comparisons, the hard work can go into improved modeling of the problem at hand and asking better questions.

Types of Statistical Questions

[1] Of course, these types of cross-checks can also be very useful!

[2] The Toolkit for MultiVariate Analysis is also distributed in ROOT http://tmva.sourceforge.net.

One of the first steps in any statistical analysis is to carefully pose the question that one wishes to answer. Most of these questions can be classified as follows:

Parameter Estimation Find the most likely ('best fit') values of the parameters of a model given data.

Hypothesis Testing Based on the data accept or reject a given hypothesis. Often one tests a null hypothesis against an alternative. When the hypothesis has no free parameters it is called 'simple' and when it has free parameters it is called 'composite'.

Confidence intervals Find a region in the parameter space that is consistent with the data. In the frequentist setting, one desires for this interval or region to 'cover' the true parameter point with a specified probability (or confidence). In the Bayesian setting, one wishes for the interval or region to contain some fixed amount of the posterior probability.

Goodness of Fit Quantify how well a model fits the data, without reference to an alternative.

RooStats provides tools for each of these broad class of questions in addition to some miscellaneous utilities. The design of the software is explicitly organized around these broad classes of questions: for instance the interface `IntervalCalculator` is common to all tools that produce `ConfidenceIntervals`.

Combining Results & Digital Publishing

Combining results from multiple experiments in order to enhance sensitivity of a measurement or improve the power of a hypothesis test is common. The challenge of combining results is primarily logistical, since a proper combination requires low-level information from each experiment be brought together to form one large statistical test. Again, this is hindered by the fact that the ingredients to the combination are heterogeneous (eg. different formats, technologies, and conventions).

The major advancement that was made by the RooStats project is the concept of the *workspace*. The power of the workspace is that it allows one to save data and an arbitrarily complicated model to disk in a ROOT file. These files can then be shared or archived, and they provide all the low-level ingredients necessary for a proper combination in a unified framework. The `RooWorkspace` provides this low-level functionality, thus it is technically part of RooFit (along with its documentation and several tutorial macros).

This form of digital publishing has great potential, consider a few examples: It allows for one to publish likelihood functions in n-dimensions instead of resorting to 2-dimensional contours. It allows for one to interface the likelihood function to even higher-level software packages (eg. extraction of fundamental lagrangian parameters from experimental measurements). It allows for one to generate toy data for the observables given any parameter point, which is necessary for a truly frequentist calculation. It allows for combinations with other experiments as already mentioned.

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.

1.2 Other Resources

The RooStats Web Page:
https://twiki.cern.ch/twiki/bin/view/RooStats/

The Root User's Guide:
http://root.cern.ch/drupal/content/users-guide

The RooFit User's Guide:
ftp://root.cern.ch/root/doc/RooFit_Users_Manual_2.91-33.pdf

RooFit Tutorials:
http://root.cern.ch/root/html/tutorials/roofit/

RooStats Tutorials:
http://root.cern.ch/root/html/tutorials/roostats/

1.3 Terminology used in this guide

model a probability density function that describes some observables. We use the term model for both parametric models (eg. a Gaussian is parametrized by a mean and standard deviation) and non-parametric models (eg. histograms or KEYS pdfs).

observable(s) quantities that are directly measured by an experiment and present in a data set. The distribution of the observables are predicted by the model. Models are normalized such that the integral of the model over the observables is 1.

auxiliary observable observables that are come from an auxiliary experiment (eg. a control sample or a preceding experiment).

parameter of interest quantities used to parametrize a model that are 'interesting' in the sense that one wishes to estimate their values, place limits on them, etc (eg. masses, cross-sections, and the like).

nuisance parameter quantities used to parametrize a model that are uncertain but not 'interesting' in the above sense (eg. background normalization, shape parameters associated to systematic uncertainties, etc.)

control sample a data set independent of the main measurement (defining auxiliary observables) often used to constrain nuisance parameters by simultaneous considering it together with the main measurement.0

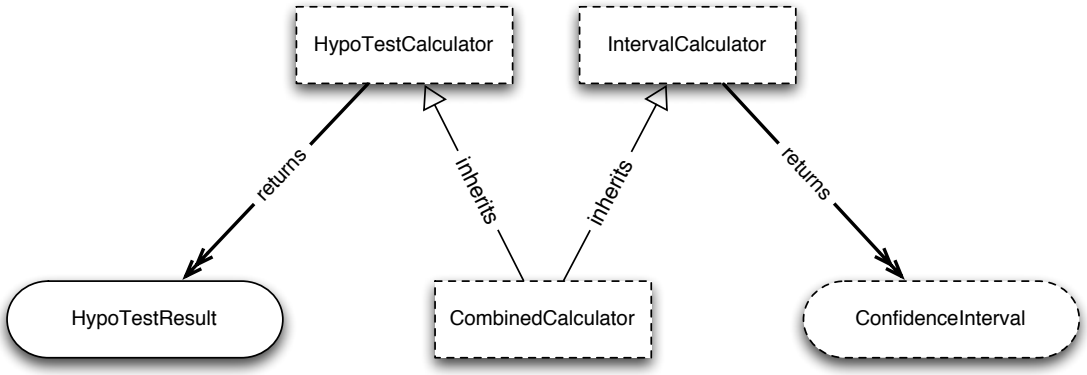

Figure 1: **default**

1.4 Types of Statistical Tools in RooStats

1.5 Design Philosophy: Mapping Mathematics to Software

Mathematical representation:

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

Representation in RooFit / RooStats code:

```
{
  // 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);
}
```

Graphical Representation

Figure 2: test

2 Parameter Estimation

Parameter estimation, or 'point estimation' is the general class of statistical tests in which one wishes to estimate the value of some parameter θ given some data x and a model $P(x|\theta)$. In high energy physics, this is usually referred to as "fitting", and the estimate that is given is usually the "maximum likelihood estimate" (MLE). The most common tool for providing maximum likelihood estimates in high energy physics is MINUIT.

In general an *estimator* for θ can be denoted $\hat{\theta}(x)$ and it has an expected value $E[\hat{\theta}]$. If the expectation value of the estimator is equal to the true value of the parameter, then the estimator is called *unbiased*. The *variance* of the estimator is $var[\hat{\theta}] = E[(\hat{\theta} - E[\hat{\theta}])^2]$.

Two naturally desirable properties of estimators are for them to be unbiased and have minimal mean squared error (MSE). These cannot in general both be satisfied simultaneously: a biased estimator may have lower mean squared error (MSE) than any unbiased estimator: despite having bias, the estimator variance may be sufficiently smaller than that of any unbiased estimator, and it may be preferable to use, despite the bias; see estimator bias [Wikipedia].

Among unbiased estimators, there often exists one with the lowest variance, called the minimum variance unbiased estimator (MVUE). In some cases an unbiased efficient estimator exists, which, in addition to having the lowest variance among unbiased estimators, satisfies the Cramér-Rao bound, which is an absolute lower bound on variance for statistics of a variable [Wikipedia].

RooFit provides a general interface to determine the MLE for all probability density functions via: `RooAbsPdf::fitTo(RooAbsData& data,...)`. It is possible to use other minimization algorithms other than MINUIT, which is outlined in the RooFit documentation.

Currently, RooStats does not provide functionality for point estimation beyond what is found directly in RooFit. However, RooStats does offer significantly more options when one wants to consider frequentist confidence intervals or Bayesian credible intervals on the parameters.

3 Test Statistics and Sampling Distributions

A *test statistic* $T(x)$ is a general term for a numerical summary of the data. Typically a test statistic is a single number, though it can be have multiple components. Usually a test statistic is best thought of as a mapping of the data to a real number, and that mapping often depends on the values of the parameters of a model (implicit in the notation $T(x)$.

For instance, for simple hypothesis testing the Neyman-Pearson lemma states that the likelihood ratio is the most powerful test statistic for testing a null hypothesis against an alternative. In a simple number counting experiment, the number of events usually serves as the test statistic. In more complicated cases one might use the profile likelihood ratio as the test statistic to summarize the preference of the data toward one hypothesis or the other. As will be described below, RooStats provides a general interface and several implementations for the notion of a test statistic.

Note, a *sufficient statistic* is a statistic which has the property of sufficiency with respect

to a statistical model and its associated unknown parameter, meaning that "no other statistic which can be calculated from the same sample provides any additional information as to the value of the parameter". In most situations relevant for high energy physics, the only distributions in the exponential family have sufficient statistics.

A *sampling distribution* is very familiar to high energy physicists, it is the distribution for some quantity that one obtains from sampling some parent distribution n times. It is most commonly the result of Monte Carlo sampling, and often obtained from "toy Monte Carlo" or "pseudo-experiments". In statistical tests, it is common to build the sampling distribution for some test statistic T by first using Monte Carlo techniques to sample x from $P(x|\theta)$ and then evaluating the test statistic $T(x)$. There are other techniques for building these sampling distributions.

3.1 TestStatistic interface and implementations

We added a new interface class called TestStatistic. It defines the method `Evaluate(data, parameterPoint)`, which returns a double. This class can be used in conjunction with the ToyMCSampler class to generate sampling distributions for a user-defined test statistic.

The following concrete implementations of the TestStatistic interface are currently available

ProfileLikelihoodTestStatReturns the log of profile likelihood ratio. Generally a powerful test statistic. NumEventsTestStatReturns the number of events in the dataset. Useful for number counting experiments. DebuggingTestStat Simply returns a uniform random number between 0,1. Useful for debugging. SamplingDistribution and the TestStatSampler interface and implementations

We introduced a "result" or data model class called SamplingDistribution, which holds the sampling distribution of an arbitrary real valued test statistic. The class also can return the inverse of the cumulative distribution function (with or without interpolation).

We introduced an interface for any tool that can produce a SamplingDistribution, called TestStatSampler. The interface is essentially GetSamplingDistribution(parameterPoint) which returns a SamplingDistribution based on a given probability density function. We foresee a few versions of this tool based on toy Monte Carlo, importance sampling, Fourier transforms, etc. The following concrete implementation of the TestStatSampler interface are currently available

ToyMCSampler uses a Toy Monte Carlo approach to build the sampling distribution. The pdf's generate method to generate is used to generate toy data, and then the test statistic is evaluated at the requested parameter point. DebuggingSampler Simply returns a uniform distribution between 0,1. Useful for debugging. NeymanConstruction and FeldmanCousins

A flexible framework for the Neyman Construction was added in this release. The NeymanConstruction is a concrete implementation of the IntervalCalculator interface, but it needs several additional components to be specified before use. The design factorizes the choice of the parameter points to be tested, the choice of the test statistic, and the generation of sampling distribution into separate parts (described above). Finally, the NeymanConstruction class is simply in charge of using these parts (strategies) and constructing the confidence belt and confidence intervals. The ConfidenceBelt class is still under development, but the

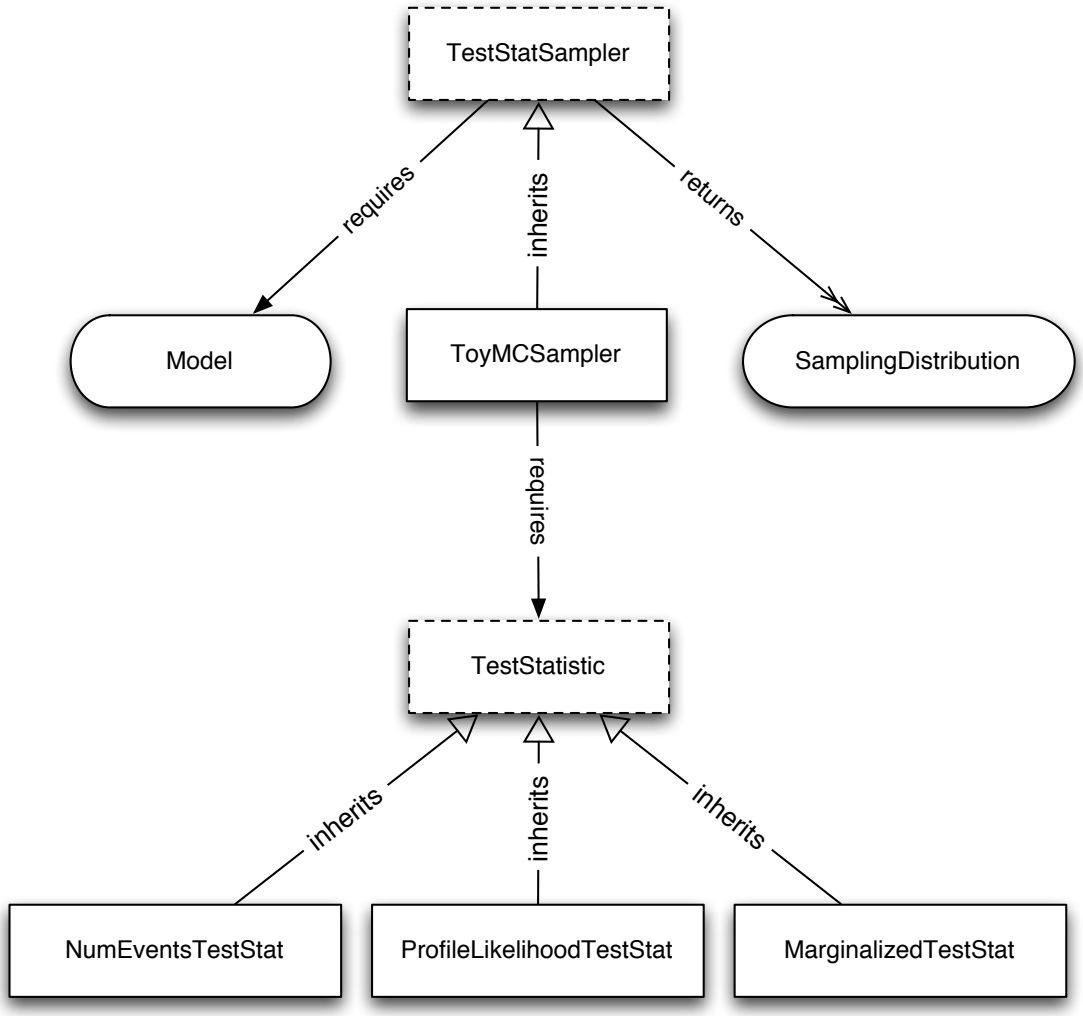

Figure 3: Class diagram showing the relationship between the interface classes TestStat-Samler and TestStatistic. The diagram shows a few concrete implementations of the TestStatistic interface and the TestStatSampler interface.

current version works fine for producing ConfidenceIntervals. We are also working to make this class work with parallelization approaches, which is not yet complete.

The FeldmanCousins class is a separate concrete implementation of the IntervalCalculator interface. It uses the NeymanConstruction internally, and enforces specific choices of the test statistic and ordering principle to realize the Unified intervals described by Feldman and Cousins in their paper Phys.Rev.D57:3873-3889,1998.

4 Hypothesis Testing

5 Confidence Intervals

5.1 Profile Likelihood Ratio (the method of MINOS)

5.2 Neyman Construction

5.3 Feldman-Cousins

5.4 Neyman Construction with nuisance parameters

5.5 The "Profile Construction"

5.6 Markov Chain Monte Carlo

Another way to determine a confidence interval for parameters is to use Markov Chain Monte-Carlo with the MCMCCalculator. Like the other IntervalCalculator derived classes, once you initialize and configure your MCMCCalculator, just call GetInterval(), which returns an MCMCInterval (ConfInterval derived class).

5.6.1 MCMCCalculator

MCMCCalculator runs the Metropolis-Hastings algorithm (section 5.6.3) with the parameters of your model, a $-log(\text{Likelihood})$ function constructed from the model and data set, and a ProposalFunction (section 5.6.2). This generates a Markov chain posterior sampling, which is used to create an MCMCInterval (section 5.6.4).

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 5.6.3 and 5.6.4 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();
```

5.6.2 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 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 cacheing mechanism.

5.6.3 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 5.6.2) 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

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

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();
```

5.6.4 MCMCInterval

MCMCInterval is a ConfInterval that determines the confidence interval on your parameters from a MarkovChain (section 5.6.6) 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 ∼4GB 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 5.6.1) 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.

5.6.5 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();
```

5.6.6 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
}
```

6 Goodness of Fit

7 Coverage Studies

8 Utilities

8.1 The Number Counting PDF Factory

8.2 RooStatsUtils

8.2.1 Standalone number counting functions

8.2.2 Converting between p-values and significance

8.3 SPlot

This initial description of `RooStats::SPlot` is taken directly from the documentation of http://root.cern.ch/root/html/TSPlot.html. It mainly describes the method, which is common to both the RooStats implementation and `TSPlot`. The main difference between

the implementations is that the RooStats implementation allows one to use arbitrary models created with RooFit's data modeling language.

A common method used in High Energy Physics to perform measurements is the maximum Likelihood method, exploiting discriminating variables to disentangle signal from background. The crucial point for such an analysis to be reliable is to use an exhaustive list of sources of events combined with an accurate description of all the Probability Density Functions (PDF).

To assess the validity of the fit, a convincing quality check is to explore further the data sample by examining the distributions of control variables. A control variable can be obtained for instance by removing one of the discriminating variables before performing again the maximum Likelihood fit: this removed variable is a control variable. The expected distribution of this control variable, for signal, is to be compared to the one extracted, for signal, from the data sample. In order to be able to do so, one must be able to unfold from the distribution of the whole data sample.

The TSPlot method allows to reconstruct the distributions for the control variable, independently for each of the various sources of events, without making use of any a priori knowledge on this variable. The aim is thus to use the knowledge available for the discriminating variables to infer the behaviour of the individual sources of events with respect to the control variable.

TSPlot is optimal if the control variable is uncorrelated with the discriminating variables.

A detail description of the formalism itself, called ${}_s\mathcal{P}lot$

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

8.3.1 The method

The ${}_s\mathcal{P}lot$ technique is developped 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,
- $\mathrm{N_s}$ the number of species of events populating the data sample,
- $\mathrm{N_i}$ the number of events expected on the average for the i^{th} species,
- $\mathrm{f}_i(y_e)$ 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:

$$\mathcal{L} = \sum_{e=1}^{N} \ln \Big\{ \sum_{i=1}^{\mathrm{N_s}} N_i \mathrm{f}_i(y_e) \Big\} - \sum_{i=1}^{\mathrm{N_s}} N_i \ . \qquad (1)$$

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 $\mathrm{N_s}$ species present in the data sample, the weight for this species is defined by:

$$\boxed{{}_s\mathcal{P}_\mathrm{n}(y_e) = \frac{\sum_{j=1}^{\mathrm{N_s}} \mathbf{V}_{\mathrm{n}j}\mathrm{f}_j(y_e)}{\sum_{k=1}^{\mathrm{N_s}} N_k \mathrm{f}_k(y_e)}} \ , \tag{2}$$

where $\mathbf{V}_{\mathrm{n}j}$ 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:

$$\mathbf{V}^{-1}_{\mathrm{n}j} \ = \ \frac{\partial^2(-\mathcal{L})}{\partial N_\mathrm{n} \partial N_j} \ = \ \sum_{e=1}^{N} \frac{\mathrm{f}_\mathrm{n}(y_e)\mathrm{f}_j(y_e)}{(\sum_{k=1}^{\mathrm{N_s}} N_k \mathrm{f}_k(y_e))^2} \ . \tag{3}$$

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

The class TSPlot allows to reconstruct the true distribution $\mathbf{M}_\mathrm{n}(x)$ of a control variable x for each of the $\mathrm{N_s}$ species from the sole knowledge of the PDFs of the discriminating variables $\mathrm{f}_i(y)$ The plots obtained thanks to the TSPlot class are called ${}_s\mathcal{P}lots$

8.3.2 Some properties and checks

Beside reproducing the true distribution, ${}_s\mathcal{P}lots$ bear remarkable properties:

Each x distribution is properly normalized:

$$\sum_{e=1}^{N} {}_s\mathcal{P}_\mathrm{n}(y_e) \ = \ N_\mathrm{n} \ . \tag{4}$$

For any event:

$$\sum_{l=1}^{\mathrm{N_s}} {}_s\mathcal{P}_l(y_e) \ = \ 1 \ . \tag{5}$$

That is to say that, summing up the $\mathrm{N_s}$ ${}_s\mathcal{P}lots$ one recovers the data sample distribution in x and summing up the number of events entering in a ${}_s\mathcal{P}lot$ for a given species, one recovers the yield of the species, as provided by the fit.

$$\sigma[N_\mathrm{n} \ {}_s\tilde{\mathrm{M}}_\mathrm{n}(x)\delta x] \ = \ \sqrt{\sum_{e \subset \delta x} ({}_s\mathcal{P}_\mathrm{n})^2} \ . \tag{6}$$

reproduces the statistical uncertainty on the yield N_n, as provided by the fit: $\sigma[N_\mathrm{n}] \equiv \sqrt{\mathbf{V}_{\mathrm{nn}}}$ Because of that and since the determination of the yields is optimal when obtained using a Likelihood fit, one can conclude that the ${}_s\mathcal{P}lot$ technique is itself an optimal method to reconstruct distributions of control variables.

- A maximum Likelihood fit is performed to obtain the yields N_i of the various species. The fit relies on discriminating variables y uncorrelated with a control variable x the later is therefore totally absent from the fit.
- The weights ${}_s\mathcal{P}$ are calculated using Eq. **??** where the covariance matrix is taken from Minuit.
- Histograms of x are filled by weighting the events with ${}_s\mathcal{P}$
- Error bars per bin are given by Eq. **??**.

The ${}_s\mathcal{P}lots$ reproduce the true distributions of the species in the control variable x within the above defined statistical uncertainties.

8.4 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 polynomails 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 emperical 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 nth-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 of 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 Prob($\chi^2_1 >$ 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).

9 Tutorials

Several new tutorials were added for RooStats. They are located in your `$ROOTSYS` area under `tutorials/roostats`. They can be run from the command line by typing

```
root $ROOTSYS/tutorials/roostats/<tutorial>
```

(note you can also add a + to the end of the command to have the macro compiled).

For your convenience, you can browse them online:
RooFit Tutorials: http://root.cern.ch/root/html/tutorials/roofit/
RooStats Tutorials: http://root.cern.ch/root/html/tutorials/roostats/

rs301_splot.C Demonstrates use of RooStats sPlot implementation

rs401c_FeldmanCousins.C Demonstrates use of FeldmanCousins interval calculator with a Poisson problem, reproduces resulst from table IV and V of the original paper Phys.Rev.D57:3873-3889,1998.

rs401d_FeldmanCousins.C Demonstrates use of FeldmanCousins interval calculator with the neutrino oscillation toy example described in the original paper Phys.Rev.D57:3873-3889,1998. Reproduces figure 12.

rs_bernsteinCorrection.C Demonstrates use of BernsteinCorrection class, which corrects a nominal PDF with a polynomial to agree with observed or simulated data.