Document version 0

November 23, 2010

RooStats

User's Guide

Kyle Cranmer, Grégory Schott, Lorenzo Moneta, Wouter Verkerke

With contributions from:

Sven Kreiss, 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 also uses a frequentist notion of probability (ie. does not require a prior for the hypothesis), but also is not guaranteed to poses the properties (ie. coverage) 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.

[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.

Types of Statistical Questions

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 **ConfidenceInterval**s.

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

2 Fundamental Interfaces in RooFit and RooStats

One of the fundamental design philosophies behind RooFit and RooStats is a conceptually clear mapping between mathematics and software. We have tried to identify fundamental concepts in statistics and assign corresponding C++ classes or interfaces [3].

2.1 RooRealVar, RooArgSet, RooAbsReal, & RooAbsPdf

In RooFit even simple real-valued parameters and observables are treated as a C++ objects, instead of just a simple floating point number. This allows the variable to carry around a name, units, upper- and lower- values for fitting and integration, etc. For real-valued variables, the class is **RooRealVar**, but there are equivalent classes for discrete categories etc.

In addition to real-valued variables, one can define real-valued functions, both of which inherit from **RooAbsReal** (for abstract real-valued object). The most important methods in the interface are listed below. They all inherit from an interface called **RooAbsArg** (for abstract argument).

```
class RooAbsReal : public RooAbsArg {
  // Return value and unit accessors
  virtual Double_t getVal(const RooArgSet* set=0) const ;

  // Analytical integration support
  virtual Double_t analyticalIntegral(Int_t code,const char* rangeName=0) const;

  // create a function of fewer variables, eliminating others via profiling
  virtual RooAbsReal* createProfile(const RooArgSet& paramsOfInterest) ;

  // create a new RooAbsReal that is the integral of this function
  RooAbsReal* createIntegral(const RooArgSet& iset,
                 const RooCmdArg arg1, ...) const;

  // User entry point for plotting
  virtual RooPlot* plotOn(RooPlot* frame, ...) const ;

  // Create 1,2, and 3D histograms from and fill it
  TH1 *createHistogram(const char *name, const RooAbsRealLValue& xvar,
                       const RooCmdArg& arg1=RooCmdArg::none(), ...) ;

  // ...
};
```

One can create sets and lists of these abstract arguments called **RooArgSet** and **RooArgList**, respectively. They are often used to represent a list of parameters or observables. Sets have unique elements, while lists can have repitition. These classes support basic list operations and use the ROOT TIterator syntax for iteration.

[3] Interfaces is a term used in object-oriented programing for an abstract specification of a C++ class. It defines what the class can/should/must do, but it does not provide an implementation. This is done because often there are many possible implementations that can satisfy that interface. This is especially true in statistics, where there seem to be several ways to do almost everything.

Probability density functions (pdfs) are clearly one of the most important concepts for RooFit and RooStats. A pdf is a real-valued function of some observables, it should be non-negative, and its integral over the domain of the observables should be 1. They are represented by the interface **RooAbsPdf**. Often pdfs are parametrized, for example a Gaussian is parametrized by its mean and standard deviation. In RooFit there is no intrensic notion of which variables are observables and which are parameters, that must be specified or implied by a given context. This also means that the normalization of the pdf is context dependent and managed by RooFit internally. The RooFit documentation provides more details and examples of this basic design decision and various options for analytic and numerical integration. While this may seem like an inconvenience, it provides a great deal of power. For instance, it allows one to treat a single instance of a Gaussian $G(x|\mu,\sigma)$ either as a pdf over x or as a poster of μ,σ given some measurement x.

```
class RooAbsPdf : public RooAbsReal {
  // raw, unnormalized evaluation of the pdf.
  // the user implements this when creating new pdfs
  virtual Double_t evaluate() const = 0 ;

  // Toy MC generation
  RooDataSet *generate(const RooArgSet &whatVars, Int_t nEvents, const RooCmdArg↩
      & arg1, ...);

  RooDataHist *generateBinned(const RooArgSet &whatVars, Double_t nEvents, const↩
       RooCmdArg& arg1, ...);

  // -log(L) fits to binned and unbinned data
  virtual RooFitResult* fitTo(RooAbsData& data, RooCmdArg arg1,...);

  // return object representing -log(L), same signature as fitTo
  virtual RooAbsReal* createNLL(RooAbsData& data, ...);

  // Project p.d.f into lower dimensional p.d.f
  virtual RooAbsPdf* createProjection(const RooArgSet& iset) ;

  // Create cumulative density function from p.d.f
  RooAbsReal* createCdf(const RooArgSet& iset, const RooArgSet& nset=RooArgSet()↩
      ) ;

  // Support for extended maximum likelihood, switched off by default
  enum ExtendMode { CanNotBeExtended, CanBeExtended, MustBeExtended } ;
  virtual ExtendMode extendMode() const ;

  // if extended, how many events are expected
  virtual Double_t expectedEvents(const RooArgSet* nset) const ;
};
```

Let's consider a basic example of a Gaussian distribution of x with mean μ and standard deviation σ. The mathematical representation is

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

In RooFit / RooStats code, one an construct this pdf via:

```
  // 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 using the workspace factory syntax 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)

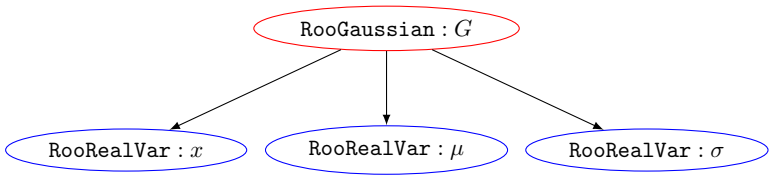

Figure 1: test

2.2 RooWorkspace & Model Config

RooWorkspace is a new core RooFit class that was developed in the context of RooStats. RooWorkspace brings the ability to save a model with all of its dependecies (variables, data, custom functions, etc.) into a ROOT file. Thus, RooWorkspace is they primary tool for RooStats digital publishing capabilities. In addition, the workspace provides a low level factory so that one can script the creation of complicated models.

```
class RooWorkspace : public TNamed {
  // ... a relevant subset of RooWorkspace public methods
  // Write this workspace to a ROOT file
  Bool_t writeToFile(const char* fileName, Bool_t recreate=kTRUE) ;

  // 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) const ;
  RooAbsData*       data(const char* name) const ;
  RooRealVar*       var(const char* name) const ;
  const RooArgSet*  set(const char* name) ;
  RooAbsReal*       function(const char* name) const ;
  RooCategory*      cat(const char* name) const ;
  RooAbsCategory*   catfunc(const char* name) const ;
  RooAbsArg*        arg(const char* name) const ;
  const RooArgSet&  components() const ;
  TIterator*        componentIterator() const ;
  TObject*          obj(const char* name) const ;

  // Print contents of the workspace
  void Print(Option_t* opts=0) const ;

  // Import functions for dataset, functions, generic objects
  Bool_t import(const RooAbsArg& arg, const RooCmdArg& arg1=RooCmdArg(),...) ;
  Bool_t import(const RooArgSet& args, const RooCmdArg& arg1=RooCmdArg(),...);
  Bool_t import(RooAbsData& data, const RooCmdArg& arg1=RooCmdArg(),...);
  Bool_t import(const char *fileSpec, const RooCmdArg& arg1=RooCmdArg(),...);
  Bool_t import(TObject& object, Bool_t replaceExisting=kFALSE) ;

  // import class code for custom classes (eg. custom PDFs and fuctions)
  static void autoImportClassCode(Bool_t flag) ;

  // define a RooArgSet with a name
  Bool_t defineSet(const char* name, const RooArgSet& aset, ...);

  // Import, load and save parameter value snapshots
  Bool_t saveSnapshot(const char* name, const RooArgSet& params,
              Bool_t importValues=kFALSE) ;
  Bool_t loadSnapshot(const char* name) ;

  // Group accessors
  RooArgSet allVars() const;
  RooArgSet allCats() const ;
  RooArgSet allFunctions() const ;
  RooArgSet allPdfs() const ;
  std::list<RooAbsData*> allData() const ;
  std::list<TObject*> allGenericObjects() const ;
} ;
```

ModelConfig is a simple class that holds configuration information specifying how a model should be used in the context of various RooStats tools. Because a RooAbsPdf has no intrensic notion of observable or parameter, it can be used in different ways. ModelConfig can be thought of as introducing the notion of a Hypothesis, and this class should carry all that is needed to specify how it should be used. In particular, it can identify what are the observables, which parameters are parameters of interest, and which are nuisance parameters. It can also identify which pdfs are part of the likelihood function (frequentist) and which are prior pdfs (Bayesian).

```
namespace RooStats {
 class ModelConfig : public TNamed {
  /* getter methods */
   /// get model PDF (return NULL if pdf  does not exist)
   RooAbsPdf * GetPdf() const ;
```

```
   /// get parameters prior pdf  (return NULL if not existing)
   RooAbsPdf * GetPriorPdf() const ;
   /// get Proto data set (return NULL if not existing)
   RooAbsData * GetProtoData()  const ;
   /// get RooArgSet containing the parameter of interest (return NULL does not ↩
       exist)
   const RooArgSet * GetParametersOfInterest() const ;
   /// get RooArgSet containing the nuisance parameters (return NULL if not ↩
       existing)
   const RooArgSet * GetNuisanceParameters() const ;
   /// get RooArgSet containing the constraint parameters (return NULL if not ↩
       existing)
   const RooArgSet * GetConstraintParameters() const ;
   /// get RooArgSet for observales  (return NULL if not existing)
   const RooArgSet * GetObservables() const ;
   /// get RooArgSet for conditional observales  (return NULL if not existing)
   const RooArgSet * GetConditionalObservables() const ;
   /// get RooArgSet for parameters for a particular hypothesis  (return NULL if↩
        not existing)
   const RooArgSet * GetSnapshot() const ;
   /// get the associated workspace
   const RooWorkspace * GetWS() const ;

   /* setter methods */
   // set a workspace that owns all the necessary components for the analysis
   virtual void SetWorkspace(RooWorkspace & ws);
   // Set the Pdf, add to the the workspace if not already there
   virtual void SetPdf(RooAbsPdf& pdf) ;
   // Set the Prior Pdf, add to the the workspace if not already there
   virtual void SetPriorPdf(RooAbsPdf& pdf) ;
   // Set the proto DataSet, add to the the workspace if not already there
   virtual void SetProtoData(RooAbsData & data) ;
   // ... the rest of the SetXxx methods ...

  };
}
```

2.3 ConfInterval & IntervalCalculator

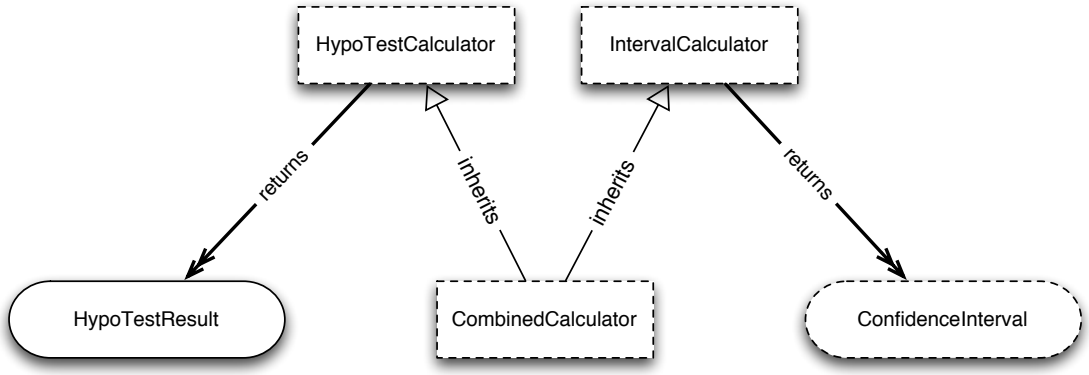

Figure 2: An class diagram of the interfaces for hypothesis testing and confidence interval calculations. The diagram shows the classes used to return the results of these statistical tests as well.

ConfInterval is an interface class for a generic interval in the RooStats framework. Any tool inheriting from IntervalCalculator can return a ConfInterval. There are many types of intervals, they may be a simple range [a,b] in 1 dimension, or they may be disconnected regions in multiple dimensions. So the common interface is simply to ask the interval if a given point "IsInInterval". The Interval also knows what confidence level it was constructed at and the space of parameters for which it was constructed. Note, one could use the same class for a Bayesian "credible interval".

```
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 an interface class for a tools which produce RooStats ConfIntervals. The interface currently assumes that any interval calculator can be configured by specifying:

- a model,

- a data set,
- a set of parameters of interest,
- a set of nuisance parameters (eg. parameters on which the model depends, but are not of interest), and
- a confidence level or size of the test (eg. rate of Type I error).

The interface allows one to pass the model, data, and parameters via a workspace and then specify them with names. The interface will be extended so that one does not need to use a workspace.

After configuring the calculator, one only needs to ask GetInterval, which will return a ConfInterval pointer.

The concrete implementations of this interface should deal with the details of how the nuisance parameters are dealt with (eg. integration vs. profiling) and which test-statistic is used (perhaps this should be added to the interface). The motivation for this interface is that we hope to be able to specify the problem in a common way for several concrete calculators.

```
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;

   };
}
```

2.4 HypoTestResult & HypoTestCalculator

HypoTestResult is the class used to hold the results of a hypothesis test.

```
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;
   };
}


#endif
```

HypoTestCalculator is an interface class for a tools which produce RooStats HypoTestResults. The interface currently assumes that any hypothesis test calculator can be configured by specifying:

- a model,
- a data set,
- a set of parameters of which specify the null (including values and const/non-const status), and
- a set of parameters of which specify the alternate (including values and const/non-const status).

The interface allows one to pass the model, data, and parameters via a workspace and then specify them with names. The interface will be extended so that one does not need to use a workspace.

After configuring the calculator, one only needs to ask GetHypoTest, which will return a HypoTestResult pointer.

The concrete implementations of this interface should deal with the details of how the nuisance parameters are dealt with (eg. integration vs. profiling) and which test-statistic is used (perhaps this should be added to the interface).

The motivation for this interface is that we hope to be able to specify the problem in a common way for several concrete calculators.

```
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) ;
   }
}
```

2.5 TestStatistic, TestStatSampler, and SamplingDistribution

TestStatistic is an interface class for test statistics. A test statistic is a real-valued number that summarizes the data, and usually p-values are based on integrals of the distribution of the test statistic. Thus, the TestStatistic interface defines the method `Evaluate(data, parameterPoint)`, which returns a double. These classes are used in conjunction with a **TestStatSampler** to generate a **SamplingDistribution**.

There are several choices for the test statistic (profile likelihood ratio, ratio of marginalized likelihoods, ...) and there are several ways for generating their sampling distribution (toy Monte Carlo, importance sampling, analytical techniques based on Fourier transforms, ...). These are outlined in Section 5

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

    // what is this called (for plotting)
    virtual const TString GetVarName() const = 0;
  }
}
```

```
namespace RooStats {
   class TestStatSampler {

      // Main interface to get a SamplingDistribution, pure virtual
      virtual SamplingDistribution* GetSamplingDistribution(RooArgSet& ↩
          paramsOfInterest);

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

```
      // Get the TestStatistic
      virtual const TestStatistic* GetTestStatistic()  const = 0;

      // Set the Pdf, etc. so we know how to sample and evaluate test stat
      virtual void SetModel(ModelConfig&) = 0;

      // Set the TestStatistic
      virtual void SetTestStatistic(TestStatistic& )  const = 0;

   };
}
```

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

    // Constructor for SamplingDistribution
    SamplingDistribution(const char *name,const char *title,
             std::vector<Double_t>& samplingDist,
             const TString varName = 0);

    // Constructor with weighted samples
    SamplingDistribution(const char *name,const char *title,
             std::vector<Double_t>& samplingDist,
             std::vector<Double_t>& sampleWeights,
             const TString varName = 0);

    // get the inverse of the Cumulative distribution function
    Double_t InverseCDF(Double_t pvalue);

    // get the inverse of the Cumulative distribution function
    Double_t InverseCDFInterpolate(Double_t pvalue);

    // merge two sampling distributions
    void Add(SamplingDistribution* other);

    // get values of test statistic
    const std::vector<Double_t> & GetSamplingDistribution() const;

    // get the sampling weights
    const std::vector<Double_t> & GetSampleWeights() const;

    // for plotting
    const TString GetVarName() const {return fVarName;}

  };
}

#endif
```

3 Quick Start

3.1 List of Tools

3.1.1 HypoTestCalculators

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

3.1.2 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

3.1.3 Plotting Classes

- LikelihoodIntervalPlot
- SamplingDistPlot
- HybridPlot (deprecated)
- HypoTestPlot
- MCMCIntervalPlot
- HypoTestInverterPlot

3.1.4 Test Statistics

- ProfileLikleihoodTestStat
- SimpleLikelihoodRatioTestStat
- RatioOfProfiledLikelihoodsTestStat
- NumEventsTestStat
- DebuggingTestStat

3.1.5 TestStatSamplers

- ToyMCSampler
- DebuggingSampler

3.2 Example Confidence Interval

```
using namespace RooStats ;
using namespace RooFit ;

void SimplestIntervalExample() {

  // set RooFit random seed for reproducible results
  RooRandom::randomGenerator()->SetSeed(3001);

  // make a simple model via the workspace factory
  RooWorkspace* wspace = new RooWorkspace();
  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("Example 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
  RooDataSet* data = wspace->pdf("normal")->generate(*wspace->set("obs"),100);
  data->Print();

  // for convenience later on
  RooRealVar* x = wspace->var("x");
  RooRealVar* mu = wspace->var("mu");

  // set confidence level
  double confidenceLevel = 0.95;

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

  // some plots
  TCanvas* canvas = new TCanvas("canvas", "canvas", 600,300);
  canvas->Divide(2);

  // plot the data
  canvas->cd(1);
  RooPlot* frame = x->frame();
  data->plotOn(frame);
  data->statOn(frame);
  frame->Draw();

  // plot the profile likeihood
  canvas->cd(2);
  LikelihoodIntervalPlot plot(plInt);
  plot.Draw();

  // for this example we know the expected intervals analytically
  double expectedLL = data->mean(*x)
    + ROOT::Math::normal_quantile(  (1-confidenceLevel)/2,1)
    / sqrt(data->numEntries());
  double expectedUL = data->mean(*x)
    + ROOT::Math::normal_quantile_c((1-confidenceLevel)/2,1)
    / sqrt(data->numEntries()) ;
```

```
  // Print the analytic and the RooStats intervals
  std::cout << endl << "expected interval is [ " <<
    expectedLL << ", " <<
    expectedUL << "]" << endl;

  cout << "plc interval is [" <<
    plInt->LowerLimit(*mu) << ", " <<
    plInt->UpperLimit(*mu) << "]" << endl;

}
```

```
///////////////////////////////////////////////////////////////////////
// An example construction of confidence intervals
// with four techniques.
///////////////////////////////////////////////////////////////////////

//#include "RooGlobalFunc.h"
#include "RooStats/ConfInterval.h"
#include "RooStats/PointSetInterval.h"
#include "RooStats/ConfidenceBelt.h"
#include "RooStats/FeldmanCousins.h"
#include "RooStats/ProfileLikelihoodCalculator.h"
#include "RooStats/MCMCCalculator.h"
#include "RooStats/BayesianCalculator.h"
#include "RooStats/MCMCIntervalPlot.h"
#include "RooStats/LikelihoodIntervalPlot.h"

#include "RooStats/ProofConfig.h"
#include "RooStats/ToyMCSampler.h"

#include "RooRandom.h"
#include "RooDataSet.h"
#include "RooRealVar.h"
#include "RooConstVar.h"
#include "RooAddition.h"
#include "RooDataHist.h"
#include "RooPoisson.h"
#include "RooPlot.h"

#include "TCanvas.h"
#include "TTree.h"
#include "TStyle.h"
#include "TMath.h"
#include"Math/DistFunc.h"
#include "TH1F.h"
#include "TMarker.h"
#include "TStopwatch.h"

#include <iostream>

// use this order for safety on library loading
using namespace RooFit ;
using namespace RooStats ;


void IntervalExamples()
{

  // Time this macro
  TStopwatch t;
```

```
t.Start();


// set RooFit random seed for reproducible results
RooRandom::randomGenerator()->SetSeed(3001);

// make a simple model via the workspace factory
RooWorkspace* wspace = new RooWorkspace();
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("Example 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
RooDataSet* data = wspace->pdf("normal")->generate(*wspace->set("obs"),100);
data->Print();

// for convenience later on
RooRealVar* x = wspace->var("x");
RooRealVar* mu = wspace->var("mu");

// set confidence level
double confidenceLevel = 0.95;

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

// example use of Feldman-Cousins
FeldmanCousins fc(*data, *modelConfig);
fc.SetConfidenceLevel( confidenceLevel);
fc.SetNBins(100); // number of points to test per parameter
fc.UseAdaptiveSampling(true); // make it go faster
fc.FluctuateNumDataEntries(false); // rows in dataset are not for individual ↩
    events, the summarize counts in experiment
// Proof

ProofConfig pc(*wspace, 4, "workers=4");    // proof-lite
//ProofConfig pc(w, 8, "localhost");    // proof cluster at "localhost"
ToyMCSampler* toymcsampler = (ToyMCSampler*) fc.GetTestStatSampler();
toymcsampler->SetProofConfig(&pc);     // enable proof

PointSetInterval* interval = (PointSetInterval*) fc.GetInterval();


// 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();
```

```
// example use of MCMCInterval
MCMCCalculator mc(*data, *modelConfig);
mc.SetConfidenceLevel( confidenceLevel);
// special options
mc.SetNumBins(200);      // bins used internally for representing posterior
mc.SetNumBurnInSteps(500); // first N steps to be ignored as burn-in
mc.SetNumIters(100000);    // how long to run chain
mc.SetLeftSideTailFraction(0.5); // for central interval
MCMCInterval* mcInt = mc.GetInterval();

// for this example we know the expected intervals
double expectedLL = data->mean(*x)
  + ROOT::Math::normal_quantile(  (1-confidenceLevel)/2,1)
  / sqrt(data->numEntries());
double expectedUL = data->mean(*x)
  + ROOT::Math::normal_quantile_c((1-confidenceLevel)/2,1)
  / sqrt(data->numEntries()) ;

// Use the intervals
std::cout << "expected interval is [ " <<
  expectedLL << ", " <<
  expectedUL << "]" << endl;

cout << "plc interval is [" <<
  plInt->LowerLimit(*mu) << ", " <<
  plInt->UpperLimit(*mu) << "]" << endl;

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

cout << "bc interval is [" <<
  bcInt->LowerLimit() << ", " <<
  bcInt->UpperLimit() << "]" << endl;

cout << "mc interval is [" <<
  mcInt->LowerLimit(*mu) << ", " <<
  mcInt->UpperLimit(*mu) << "]" << endl;

mu->setVal(0);
cout << "is mu=0 in the interval? " <<
  plInt->IsInInterval(RooArgSet(*mu)) << endl;


// make a reasonable style
gStyle->SetCanvasColor(0);
gStyle->SetCanvasBorderMode(0);
gStyle->SetPadBorderMode(0);
gStyle->SetPadColor(0);
gStyle->SetCanvasColor(0);
gStyle->SetTitleFillColor(0);
gStyle->SetFillColor(0);
gStyle->SetFrameFillColor(0);
gStyle->SetStatColor(0);


// some plots
TCanvas* canvas = new TCanvas("canvas");
canvas->Divide(2,2);

// plot the data
canvas->cd(1);
```

```
  RooPlot* frame = x->frame();
  data->plotOn(frame);
  data->statOn(frame);
  frame->Draw();

  // plot the profile likeihood
  canvas->cd(2);
  LikelihoodIntervalPlot plot(plInt);
  plot.Draw();

  // plot the MCMC interval
  canvas->cd(3);
  MCMCIntervalPlot* mcPlot = new MCMCIntervalPlot(*mcInt);
  mcPlot->SetLineColor(kGreen);
  mcPlot->SetLineWidth(2);
  mcPlot->Draw();

  canvas->Update();

  t.Stop();
  t.Print();

}
```

3.3 Example Hypothesis Test

3.4 Coding Hints

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 shows 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. For example:

```
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 your C++ file.

4 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.

5 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.

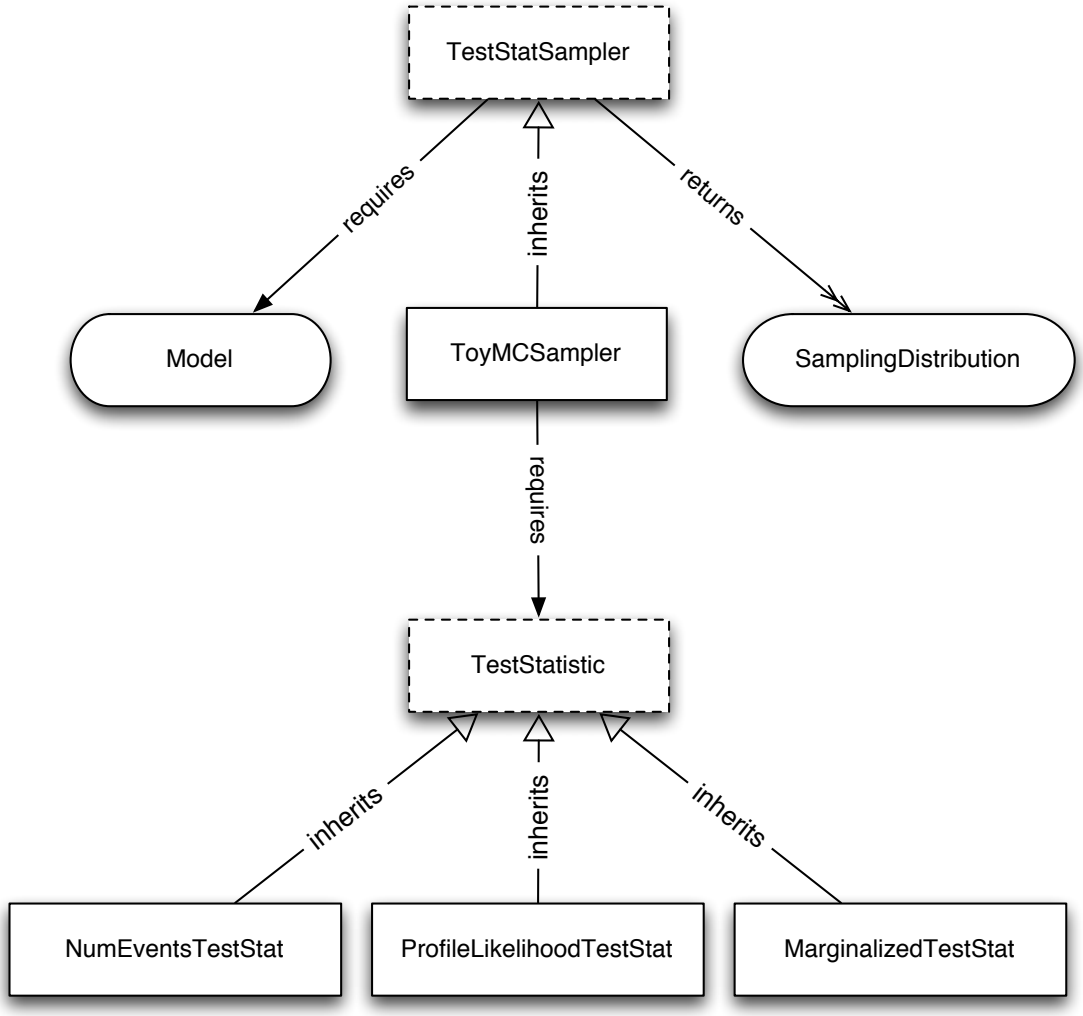

Figure 3: Class diagram showing the relationship between the interface classes TestStat-Samler and TestStatistic. The diagram shows a few concrete implementations of the TestStatistic interface and the TestStatSampler interface.

5.1 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

ProfileLikelihoodTestStat returns the log of profile likelihood ratio. Generally a powerful test statistic. To introduce notation, this is

$$\log \Lambda(\mu, \hat{\hat{\theta}}) = \log \frac{L(\mu, \hat{\hat{\theta}})}{L(\hat{\mu}, \hat{\theta})}$$

where two hats denote a conditional maximum likelihood estimate. **SimpleLikelihoodRatioTestStat** returns the log of the ratio of two simple hypotheses, namely the background hypothesis b and the signal+background hypothesis sb.

$$\log \frac{L(\mu_{sb})}{L(\mu_b)}.$$

RatioOfProfiledLikelihoodsTestStat returns the log of the ratio of the two profiled likelihoods under the two hypotheses. The default is

$$\log \frac{\Lambda(\mu_{sb}, \hat{\hat{\theta}}_{sb})}{\Lambda(\mu_b, \hat{\hat{\theta}}_b)}.$$

Alternatively, using SetSubtractMLE(false), it is

$$\log \frac{L(\mu_{sb}, \hat{\hat{\theta}}_{sb})}{L(\mu_b, \hat{\hat{\theta}}_b)}.$$

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 is used to generate toy data at the requested parameter point, and then the test statistic is evaluated. If the pdf is extended, the number of events to generate is determined automatically. If the pdf is not extended or one wants to set the number of events to a specific value, one can use SetNEventsPerToy(const Int_t). As an option, ToyMCSampler also supports the generation of toys from an "importance density" instead of the pdf. It then re-weights the values of the test statistic to recover a distribution according to the pdf.[4] The importance density has to be similar to the pdf to avoid anomalously large values after reweighting. This method can improve the sampling of small tails of a distribution significantly. Another feature of the ToyMCSampler is its capability to leverage Proof. To activate this feature, one has to call SetProofConfig() with a ProofConfig instance. ProofConfig supports runs with Proof-Lite and runs on Proof clusters. A blank Proof-hostname implies Proof-Lite. Generic options for Proof(-Lite) are also supported, e.g. "workers=2" to force the use of exactly two workers irrespective of how many cores are available. For example:

```
ProofConfig pc(workspace, 10, "workers=2");     // proof-lite, force 2 workers
//ProofConfig pc(workspace, 8, "localhost");    // proof cluster at "localhost"
toymcsampler.SetProofConfig(&pc);               // enable proof
```

The integer number in the ProofConfig constructor defines the number of "Proof events".

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 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.

6 Hypothesis Test Calculators

Hypothesis testing is one of the most common procedures in science. In *simple hypothesis testing* one considers a null hypothesis H_0 and an alternate hypothesis H_1, both of which

[4] This is based on the talk by Michael Woodroofe at Banff, where he was able to populate the tail of a 5σ test with 3600 toys.

are completely specified and provide models for the data.

Define: Type I & Type II errors, size, and power.

The Neyman-Pearson lemma states that the likelihood ratio $P(x|H_1)/P(x|H_0)$ provides the most powerful test statistic for testing the alternate against the null. This is the basis for the ubiquity of the use of the likelihood ratio; however, it is only the most powerful test in this very limited situation.

More commonly one does not have a fully specified simple hypothesis, but a *composite hypothesis* $P(x|\mu,\nu)$ that is parametrized. It may be parametrized according to some parameter of interest μ (e.g., the mass of a particle) or by some nuisance parameter ν (e.g., an unknown background rate). In these cases, the Neyman-Pearson lemma is not applicable. One would like to find a test statistic that is most powerful over the full range of μ – which is referred to as Uniformly Most Powerful (UMP). Unfortunately, UMP tests don't exist in general.

There is a precise dictionary that relates hypothesis tests to confidence intervals. In fact, the frequentist method for deriving confidence intervals is referred to as an "inverted hypothesis test".

In RooStats we wish to support several different techniques for hypothesis testing. They mainly differ in the way that they incorporate systematic effects. There are only very few cases in which hypothesis tests with systematics have a clear "solution" from first principles. In a more general setting the field has developed several techniques, and new ones are being advocated given recent interaction with the professional statistics community. Some of the techniques included in RooStats are mainly avaialbe for consistency with historical practice in the field.

As denoted in Fig. 2, the interface for tools that peform hypothesis tests in RooStats is called **HypoTestCalculator**. After configuring a HypoTestCalculator, one simply requests the result via `HypoTestCalculator::GetHypoTest()`.

Below we describe three concrete implementatons.

6.1 The Hybrid Calculator

This is the traditional frequentist ToyMC method; however, because of its Bayesian marginalization of nuisance parameters it is not purely frequentist. Hence the technique is often referred to as a "Bayesian-Frequentist Hybrid". We define H_b as the hypothesis that no signal is present over the background and H_{sb} the hypothesis that signal is also present. In order to quantify the degree in which each hypotheses are favoured or excluded by the experimental observation one chooses a test-statistics which ranks the possible experimental outcomes. A commonly used test statistics consist as the ratio of the likelihood function in both hypotheses: $Q = L_{sb}/L_b$ and the quantity $-2\ln Q$ may also be used instead of Q. The HybridCalculator class of RooStats provides alternative choices for the test statistics such as the number of events or the profiled likelihood ratio.

A comparison of $Q_{\rm obs}$ for the data being tested to the probability distributions dP/dQ

expected in both hypotheses allows to compute the confidence levels:

$$CL_{sb} = P_{sb}(Q < Q_{\rm obs}), \quad \text{where} \quad P_{sb}(Q < Q_{\rm obs}) = \int_{-\infty}^{Q_{\rm obs}} \frac{dP_{sb}}{dQ} dQ, \tag{1}$$

$$CL_{b} = P_{b}(Q < Q_{\rm obs}), \quad \text{where} \quad P_{b}(Q < Q_{\rm obs}) = \int_{-\infty}^{Q_{\rm obs}} \frac{dP_{b}}{dQ} dQ. \tag{2}$$

Small values of CL_{sb} (resp. CL_b) point out poor compatibility with the H_{sb} (resp. H_b) hypothesis and favour the H_b (resp. H_s) hypothesis. The functional form of the dP_{sb}/dQ and dP_b/dQ pdfs not being known a priori, a large amount of toy Monte-Carlo experiments are performed in order to determine it for two families of pseudo datasets: the ones in the signal+background and the ones in the background-only hypothesis (see figure 4).

Figure 4: The distributions of $-2\ln Q$ in the background-only (red, on the right) and signal+background (blue, on the left) hypotheses. The black line represents the value of $-2\ln Q$ on the tested data. The shaded areas represent $1 - CL_b$ (red) and CL_{sb} (blue).

A significance estimation can be obtained using formula 7 on CL_{sb}. Moreover, the tested data can be said to be excluded at a given CL if $1 - CL_{sb}$ is smaller than this CL (or alternatively the CL_s prescription can be used (see below)). By varying the hypothesis being tested (for example varying the signal cross-section as on figures **??**, **??**, **??** and **??**) one may also scan for the type of model that can be excluded with the given data. It should be observed that these confidence intervals do not have the same meaning as the ones obtained with the profile likelihood method or the Bayesian credibility intervals.

Systematics uncertainties are taken into account using Bayesian Monte-Carlo sampling. For each toy Monte-Carlo experiment the effective value of the nuisance parameters is varied before generating the toy Monte-Carlo sample itself (that includes in addition the Poisson fluctuations). The whole phase space of the nuisance parameters is thus sampled through Monte-Carlo integration. The final net effect consist in a broadening of the $-\ln Q$ distribution and thus, as expected in presence of systematic uncertainties, a degraded separation of the hypotheses.

7 Confidence Interval Calculators

7.1 Profile Likelihood Calculator

Suppose we have, for each of N events in a collection, a set of measured quantities $\underline{x} = (x_a, x_b, x_c, \ldots)$ whose distributions are described by a joint probability density function, or pdf, $f(\underline{x}, \underline{\theta})$, where $\underline{\theta} = (\theta_1, \theta_2, \theta_3, \ldots)$ is a set of K parameters. The likelihood function is then defined by the equation:

$$L(\underline{x}, \underline{\theta}) = \prod_{i=1}^{N} f(\underline{x}_i, \underline{\theta}). \tag{3}$$

For simplicity, let's focus on one dimensional case where we are have a parameter of interest, θ_0 and K-1 nuisance parameters. The profile likelihood is a function of θ_0 and it is defined as

$$\lambda(\theta_0) = \frac{L(\theta_0, \hat{\hat{\underline{\theta}}}_{i\neq 0})}{L(\hat{\theta}_0, \hat{\underline{\theta}}_{i\neq 0})} \tag{4}$$

The numerator, $L(\hat{\theta}_0)$ is the maximum likelihood value, obtained from the best fit to the data, while the denominator is the maximum value obtained by fixing θ_0 and varying the remaining K-1 parameters. It can be shown (Wilks' theorem) that asymptotically $-2ln\lambda(\theta_0)$ is distributed as a $\chi^2(1)$ ($\chi^2(n)$ in the case of n parameter of interest). This is not surprising since in the asymptotic limit the Likelihood function becomes a Gaussian centered around the ML estimator θ^0, and the profile likelihood curve has a parabolica shape:

$$-2ln\lambda(\theta_0) = -2(\ln L(\theta_0) - \ln L(\hat{\theta}_0)) = n_\sigma^2, \text{ with } n_\sigma = \frac{\theta_0 - \hat{\theta}_0}{\sigma}. \tag{5}$$

where σ represent the Gaussian standard deviations of the parameter θ_0. From this construction, it is possible to easily obtained the one- or two-sided confidence interval we are interested in. Even in case of non parabolic log-likelihood functions, it can be shown, due to the invariance property of the likelihood ratios, that this approach is also valid. This is a method known in the physics community, it has been called the method of MINOS since it has been implemented by the MINOS algorithm of the Minuit program**??**.

The mapping between the desired confidence level (CL) and the value of Gaussian n_σ is given by the formulae:

$$n_\sigma = \sqrt{2} \cdot \mathrm{Erf}^{-1}(CL) \qquad (\text{two} - \text{sided}) \tag{6}$$

$$n_\sigma = \sqrt{2} \cdot \mathrm{Erf}^{-1}(2 \cdot CL - 1) \quad (\text{one} - \text{sided}) \tag{7}$$

where Erf^{-1} is the inverse error function. In the later case, an upper limit on the parameter θ_0 would then correspond then to the value $\theta_0 > \hat{\theta}_0$ obeying equations 5 and 7.

7.2 Neyman Construction

This is the general class for performing a NeymanConstruction. Some details in the Test-Statsitic section, more to come.

7.3 Feldman-Cousins

This is a light weight class that encodes a specific configuration of the more general purpose Neyman Construction class.

7.3.1 Neyman Construction with nuisance parameters

7.3.2 The "Profile Construction"

7.4 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).

7.4.1 MCMCCalculator

MCMCCalculator runs the Metropolis-Hastings algorithm (section 7.4.3) with the parameters of your model, a $-log(\text{Likelihood})$ function constructed from the model and data set, and a ProposalFunction (section 7.4.2). This generates a Markov chain posterior sampling, which is used to create an MCMCInterval (section 7.4.4).

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 7.4.3 and 7.4.4 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();
```

7.4.2 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.

7.4.3 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 7.4.2) 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 like-

lihood 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 ±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();
```

7.4.4 MCMCInterval

MCMCInterval is a ConfInterval that determines the confidence interval on your parameters from a MarkovChain (section 7.4.6) 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 7.4.1) 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.

7.4.5 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();
```

7.4.6 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
}
```

7.5 The BayesianCalculator

7.6 The HypoTestInverter

8 Plotting Classes

9 Goodness of Fit

10 Coverage Studies

11 Utilities

11.1 Converting between p-values and significance

```
namespace RooStats {

  // returns one-sided significance corresponding to a p-value
  inline Double_t PValueToSignificance(Double_t pvalue);

  // returns p-value corresponding to a 1-sided significance
  inline Double_t SignificanceToPValue(Double_t Z);

  // remove from a set all parameters that are set to constant (ie. "fixed" )
  inline void RemoveConstantParameters(RooArgSet* set);

  // Assuming all values in set are RooRealVars, randomize their values.
  inline void RandomizeCollection(RooAbsCollection& set,
                                  Bool_t randomizeConstants = kTRUE);
}
```

11.2 Standalone number counting hypothesis tests

These are RooStats standalone utilities that calculate the p-value or Z value (eg. significance in 1-sided Gaussian standard deviations) for a number counting experiment. This is a hypothesis test between background only and signal-plus-background. The background estimate has uncertainty derived from an auxiliary or sideband measurement.

This is based on code and comments from Bob Cousins and on the following papers:

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 Authors: Robert D. Cousins, James T. Linnemann, Jordan Tucker http://arxiv.org/abs/physics/0702156 NIM A 595 (2008) 480–501

Statistical Challenges for Searches for New Physics at the LHC Authors: Kyle Cranmer http://arxiv.org/abs/physics/0511028

Measures of Significance in HEP and Astrophysics Authors: J. T. Linnemann http://arxiv.org/abs/physics/03

The problem is treated in a fully frequentist fashion by interpreting the relative background uncertainty as being due to an auxiliary or sideband observation that is also Poisson distributed with only background. Finally, one considers the test as a ratio of Poisson means where an interval is well known based on the conditioning on the total number of events and the binomial distribution.

In short, this is an exact frequentist solution to the problem of a main measurement x distributed as a Poisson around s+b and a sideband or auxiliary measurement y distributed as a Poisson around τb. Eg.

$$L(x, y|s, b, \tau) = \mathrm{Pois}(x|s + b)\mathrm{Pois}(y|\tau b)$$

Naming conventions: Exp = Expected Obs = Observed P = p-value Z = Z-value or significance in sigma (one-sided convention)

```
namespace RooStats{
   namespace  NumberCountingUtils {

  // Expected P-value for s=0 in a ratio of Poisson means.
  // Here the background and its uncertainty are provided directly and
  // assumed to be from the double Poisson counting setup described in the
  // BinomialWithTau functions.
  // Normally one would know tau directly, but here it is determiend from
  // the background uncertainty.  This is not strictly correct, but a useful
  // approximation.
     Double_t BinomialExpZ(Double_t sExp, Double_t bExp, Double_t ↩
         fractionalBUncertainty);

  // See BinomialWithTauExpP
     Double_t BinomialWithTauExpZ(Double_t sExp, Double_t bExp, Double_t tau);
  // See BinomialObsP
     Double_t BinomialObsZ(Double_t nObs, Double_t bExp, Double_t ↩
         fractionalBUncertainty);
  // See BinomialWithTauObsP
     Double_t BinomialWithTauObsZ(Double_t nObs, Double_t bExp, Double_t tau);
  // See BinomialExpP
     Double_t BinomialExpP(Double_t sExp, Double_t bExp, Double_t ↩
         fractionalBUncertainty);

  // Expected P-value for s=0 in a ratio of Poisson means.
  // Based on two expectations, a main measurement that might have signal
  // and an auxiliarly measurement for the background that is signal free.
  // The expected background in the auxiliary measurement is a factor
  // tau larger than in the main measurement.
     Double_t BinomialWithTauExpP(Double_t sExp, Double_t bExp, Double_t tau);
```

```
  // P-value for s=0 in a ratio of Poisson means.
  // Here the background and its uncertainty are provided directly and
  // assumed to be from the double Poisson counting setup.
  // Normally one would know tau directly, but here it is determiend from
  // the background uncertainty.  This is not strictly correct, but a useful
  // approximation.
     Double_t BinomialObsP(Double_t nObs, Double_t, Double_t ↩
         fractionalBUncertainty);

  // P-value for s=0 in a ratio of Poisson means.
  // Based on two observations, a main measurement that might have signal
  // and an auxiliarly measurement for the background that is signal free.
  // The expected background in the auxiliary measurement is a factor
  // tau larger than in the main measurement.
     Double_t BinomialWithTauObsP(Double_t nObs, Double_t bExp, Double_t tau);
   }
}
```

11.3 The Number Counting PDF Factory

11.4 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

11.4.1 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:

$$\mathcal{L} = \sum_{e=1}^{N} \ln \Big\{ \sum_{i=1}^{\mathrm{N_s}} N_i \mathrm{f}_i(y_e) \Big\} - \sum_{i=1}^{\mathrm{N_s}} N_i \ . \tag{8}$$

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

$$\boxed{{}_s\mathcal{P}_{\mathrm{n}}(y_e) = \frac{\sum_{j=1}^{\mathrm{N_s}} \mathbf{V}_{\mathrm{n}j} \mathrm{f}_j(y_e)}{\sum_{k=1}^{\mathrm{N_s}} N_k \mathrm{f}_k(y_e)}} \ , \tag{9}$$

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:

$$\mathbf{V}^{-1}_{\mathrm{n}j} \ = \ \frac{\partial^2(-\mathcal{L})}{\partial N_{\mathrm{n}} \partial N_j} \ = \ \sum_{e=1}^{N} \frac{\mathrm{f}_{\mathrm{n}}(y_e)\mathrm{f}_j(y_e)}{(\sum_{k=1}^{\mathrm{N_s}} N_k \mathrm{f}_k(y_e))^2} \ . \tag{10}$$

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 ${}_s\mathcal{P}lots$

11.4.2 Some properties and checks

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

Each x distribution is properly normalized:

$$\sum_{e=1}^{N} {}_s\mathcal{P}_{\rm n}(y_e) \;=\; N_{\rm n} \;. \tag{11}$$

For any event:

$$\sum_{l=1}^{\rm N_s} {}_s\mathcal{P}_l(y_e) \;=\; 1 \;. \tag{12}$$

That is to say that, summing up the $\rm N_s$ ${}_s\mathcal{P}lots$ 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.

$$\sigma[N_{\rm n}\ {}_s\tilde{\rm M}_{\rm n}(x)\delta x] \;=\; \sqrt{\sum_{e\subset\delta x} ({}_s\mathcal{P}_{\rm n})^2} \;. \tag{13}$$

reproduces the statistical uncertainty on the yield $N_{\rm n}$, as provided by the fit: $\sigma[N_{\rm n}] \equiv \sqrt{\mathbf{V}_{\rm nn}}$ 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.

11.5 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 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).

12 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.