BAYESIAN ANALYSIS TOOLKIT

An introduction to BAT

BAT version 0.9.4

November 20, 2014

Contents

1 Introduction **4**
1.1 License . . . 4
1.2 Installation instructions . . . 4

2 Running BAT **4**
2.1 The analysis chain . . . 4
2.2 Getting started . . . 4
2.3 Creating a model . . . 5
2.3.1 Implementation of a statistical model as a C++ class . . . 5
2.3.2 Definition of parameters of a model . . . 6
2.3.3 A skeleton created by the bat-project script . . . 7
2.3.4 Roostats interface . . . 12
2.4 Data . . . 12
2.4.1 Data format and handling . . . 12
2.4.2 Constraining the values of data points . . . 13
2.5 Managing more than one model: the model manager . . . 13
2.6 Normalization and numerical integration . . . 14
2.6.1 Cuba interface . . . 15
2.7 Parameter estimation and marginalization . . . 15
2.7.1 Maximization of the full posterior probability density . . . 15
2.7.2 Marginalization . . . 16
2.8 Model comparison and hypothesis testing . . . 17
2.8.1 Model comparison . . . 17
2.8.2 Goodness-of-fit test . . . 17
2.9 Propagation of uncertainties . . . 19
2.10 Thread parallelization . . . 20
2.11 One- and two-dimensional histograms . . . 21

3 Tools and models **24**
3.1 Tools . . . 24
3.1.1 The summary tool . . . 24
3.2 Models for function fitting . . . 24
3.2.1 The Gaussian case . . . 25
3.2.2 The Poissonian case . . . 25
3.2.3 The Binomial case . . . 25
3.3 Multi-template fitter . . . 26
3.3.1 Mathematical formulation . . . 26
3.3.2 Creating the fitter . . . 27
3.3.3 Adding a channel . . . 27
3.3.4 Adding a data set . . . 27
3.3.5 Adding a process . . . 27
3.3.6 Adding systematic uncertainties . . . 28
3.3.7 Running the fit . . . 29
3.3.8 Output . . . 29
3.3.9 Settings . . . 30
3.3.10 Analysis facility . . . 30
3.3.11 Performing ensemble tests . . . 30
3.3.12 Performing automated analyses . . . 32

4 Output **34**
4.1 Log file . . . 34

4.2 Summary information 34
4.3 Histograms 34
4.4 The output class 35

5 Settings, options and special functions **36**
5.1 Markov Chain settings and options 36
5.2 Settings and options for Simulated Annealing 38

Bibliography **38**

1 Introduction

The Bayesian Analysis Toolkit, BAT, is a software package designed to help solve statistical problems encountered in Bayesian inference. Allowing to formulate models and their parameters, the main purpose of the toolkit is to provide methods to solve the numerical optimization and integration. It features the possibility to estimate parameters and to compare models. A procedure to estimate the goodness-of-fit is included and based on ensemble tests. A detailed introduction to BAT can be found in [1].

1.1 License

BAT can be downloaded from `http://mpp.mpg.de/bat`.

BAT is a free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License (LGPL) as published by the Free Software Foundation, either version 3 of the License, or any later version.

1.2 Installation instructions

The set of detailed instructions to install your version of BAT are in the `INSTALL` file that comes with the BAT source distribution. The latest version is available online at `https://github.com/bat/bat/blob/master/INSTALL.md`.

2 Running BAT

2.1 The analysis chain

The typical analysis chain in BAT is the following: one or several models are defined together with their parameters and corresponding ranges. Data is read in from a file and interfaced with each model. For each model parameters are estimated either from the posterior probability density or from the marginalized probability densities of the individual parameters. Models can be compared using direct probabilities or Bayes factors. A goodness–of–fit test can be performed by evaluating the likelihood for an ensemble of possible data sets given the best-fit parameters. The data sets are generated under the assumption of the model at hand and the best-fit parameters.

2.2 Getting started

BAT comes in form of a library. It can be linked against in any existing C++ code, or it can be used in an interactive ROOT session. The latter case is discussed later on in this manual. Several files need to be provided by the user in order to start a new project:

- A makefile in which the BAT library is linked.
- Include and source files of the classes defining the models used in the analysis (see next section).

- A main file in which the actual analysis is performed.

The executable script `BAT/tools/bat-project` can be used to create an empty analysis skeleton including the above listed files. This is a good starting point.

The script can take up to two parameters. The first parameter is the name of the project, the second one is the name of new model class. If only the name of the project is specified, only a Makefile and a main C++ file are provided. This can be useful if predefined classes are used in an analysis (see, e.g., the fast fitter classes described later in this manual). If also a model class name is given, a C++ include and source file are created. These can be modified to the need of user.

2.3 Creating a model

2.3.1 Implementation of a statistical model as a C++ class

The statistical models used in BAT are implemented in terms of C++ classes. All model classes inherit from the pure-virtual base class `BCModel`, which has one member function that must be overloaded:

```
virtual double BCModel::LogLikelihood(const std::vector<double> & params) = 0
```

This function calculates the conditional probability of the data given a set of parameter values, $p(x|\vec{\lambda})$. It returns the logarithm of the conditional probability for reasons of numerical stability.

The member function

```
virtual double BCModel::LogAPrioriProbability(const std::vector<double> & params)
```

may also be overloaded. It calculates the *a priori* probability for a set of parameter values. It returns the logarithm of the conditional probability for reasons of numerical stability. If the *a priori* probablity is separable in terms of the individual parameters, then this function need not be overloaded. In this case, the prior can be set for individual parameters through ROOT `TF1` objects (functions) with[1]

```
int BCModel::SetPrior(int index, TF1 * f).
```

Three frequently used priors are predefined:

- `int SetPriorGauss(int index, double mean, double sigma)`
  A Gaussian prior.
- `int SetPriorGauss(int index, double mean, double sigmadown, double sigmaup)`
  A Gaussian prior with upper and lower widths.
- `int SetPriorDelta(int index, double value)`
  A delta-function prior; the parameter range is also set to the specified value.

Additionally, priors may be set from ROOT `TH1` objects (1D histograms) with

```
int SetPrior(int index, TH1 * h, bool flag=false).
```

[1] `BCModel` also contains versions of all prior-setting functions that allow for reference to parameters by name rather than index.

The prior may also be chosen constant with respect to individual parameters or all of them with

```
int SetPriorConstant(int index)
int SetPriorConstantAll();
```

If the prior for a parameter is undefined, BAT assumes a constant prior and issues a warning.

Warning: calls to `SetPrior*` *have no effect if the concrete model* `MyModel : public BCModel` *overloads* `LogAPrioriProbability`*.*

2.3.2 Definition of parameters of a model

The parameters of a model are implemented as a C++ class `BCParameter`. They can be added to a model in two ways: Parameters can be defined explicitly and then added to a model via

```
BCParameter * parameter = new BCParameter(const char * name, double lowerlimit,
                                          double upperlimit,
                                          const char* latexname = "");
int BCModel::AddParameter(BCParameter * parameter),
```

where the arguments in the `BCParameter` constructor are the name, lower limit, and upper limit of the parameter. The optional `latexname` is used to label axes in output plots. When omitted, `name` is used instead. The user can take explicit control and have multiple models share one parameter:

```
model1.AddParameter(parameter);
model2.AddParameter(parameter);
```

Second, parameters can also be defined implicitly via

```
int BCModel::AddParameter(const char* name, double lowerlimit,
                          double upperlimit, const char* latexname = "").
```

Each parameter must have a unique name and valid limits. Once added to a model, each parameter is given a unique index starting at zero. A parameter can be referenced by its index (fast) or by its name (slow). It can be returned from a model using the methods

```
BCParameter * BCModel::GetParameter(int index),
BCParameter * BCModel::GetParameter(char * name).
```

It is possible to change the lower and upper limits of a parameter after it has been added to the model:

```
model.GetParameter(0)->SetLimits(min, max);
```

The parameter's current ranges are used in all algorithms. Parameters can also be fixed (unfixed) to particular values using

```
void BCParameter::Fix(double value).
void BCParameter::Unfix().
```

If `BCParameter::Fixed()` is true, marginalization algorithms will not store this parameter's 1D marginal distribution as well as any 2D combination with other potentially variable parameters. The same effect can be achieved with `BCParameter::FillHistograms(false)`.

For those parameters for which the marginal distribution is stored, the number of bins of the histogram is set via

```
BCParameter::SetNbins(unsigned nbins)
```

2.3.3 A skeleton created by the bat-project script

The `bat-project` script is a good starting point to implement a model and set up an analysis. After installation, it is in the `bin` subdirectory. Its usage is

```
bat-project <project> [<model>]
```

The script creates a new directory named `<project>` and a new model class named `<model>`, i.e., the header and source files for the class, together with a Makefile for the project.

If `<model>` is omitted, the name of the model class will be set to `<project>`. To only create the project files, use `-` for `<model>`. To only create the model class, use `-` for `<project>`.

```
bat-project MyProject MyModel
---------------------------------------------------------------------------
The new BAT project was created in the directory 'MyProject'.
To test the configuration try to compile the project by running 'make'
inside the directory. In case there are some compilation errors you need
to adjust the parameters inside the 'Makefile'.

Once the program is compiled successfully, you can run it and it should
print some basic information on the screen.

Implement your model in files:   MyModel.h  MyModel.cxx
Implement your analysis in file: runMyProject.cxx

For details consult BAT webpage: http://mpp.mpg.de/bat
---------------------------------------------------------------------------
```

The include file will look like this

```
// ***************************************************************
// This file was created using the bat-project script.
// bat-project is part of Bayesian Analysis Toolkit (BAT).
// BAT can be downloaded from http://mpp.mpg.de/bat
// ***************************************************************

#ifndef __BAT__MYMODEL__H
#define __BAT__MYMODEL__H

#include <BAT/BCModel.h>

// This is a MyModel header file.
// Model source code is located in file MyProject/MyModel.cxx

// ---------------------------------------------------------
```

```
class MyModel : public BCModel
{
   public:

      // Constructors and destructor
      MyModel();
      MyModel(const char * name);
      ~MyModel();

      // Methods to overload, see file MyModel.cxx
      void DefineParameters();
      double LogAPrioriProbability(const std::vector<double> &parameters);
      double LogLikelihood(const std::vector<double> &parameters);
};
// ---------------------------------------------------------

#endif
```

The constructors and the destructor are defined as well as the key methods that define the likelihood and prior. The source files look like this

```
// ***************************************************************
// This file was created using the bat-project script.
// bat-project is part of Bayesian Analysis Toolkit (BAT).
// BAT can be downloaded from http://mpp.mpg.de/bat
// ***************************************************************

#include "MyModel.h"

#include <BAT/BCMath.h>

// ---------------------------------------------------------
MyModel::MyModel() : BCModel()
{
   // default constructor
   DefineParameters();
}

// ---------------------------------------------------------
MyModel::MyModel(const char * name) : BCModel(name)
{
   // constructor
   DefineParameters();
}

// ---------------------------------------------------------
MyModel::~MyModel()
   // default destructor
{
```

```
}

// ---------------------------------------------------------
void MyModel::DefineParameters()
{
   // Add parameters to your model here.
   // You can then use them in the methods below by calling the
   // parameters.at(i) or parameters[i], where i is the index
   // of the parameter. The indices increase from 0 according to the
   // order of adding the parameters.

//   AddParameter("x",  0.0, 10.0); // index 0
//   AddParameter("y", -5.0,  5.0); // index 1
}

// ---------------------------------------------------------
double MyModel::LogLikelihood(const std::vector<double> &parameters)
{
   // This methods returns the logarithm of the conditional probability
   // p(data|parameters). This is where you have to define your model.

   double logprob = 0.;

//   double x = parameters.at(0);
//   double y = parameters.at(1);

   // Gaussian distribution for x
//   logprob += BCMath::LogGaus(x, 6.0, 2.0);

   // Breit-Wigner distribution for y
//   logprob += BCMath::LogBreitWignerNonRel(y, 0.0, 1.0);

   return logprob;
}

// ---------------------------------------------------------
double MyModel::LogAPrioriProbability(const std::vector<double> &parameters)
{
   // This method returns the logarithm of the prior probability for the
   // parameters p(parameters).

   double logprob = 0.;

//   double x = parameters.at(0);
//   double y = parameters.at(1);

   // exponentially decreasing prior for x
//   logprob += -0.5*x;
```

```
   // Gaussian prior for y
//   logprob += BCMath::LogGaus(y, 2., 1.0);

   return logprob;
}
// ---------------------------------------------------------
```

and

```
// ***************************************************************
// This file was created using the bat-project script
// for project MyProject.
// bat-project is part of Bayesian Analysis Toolkit (BAT).
// BAT can be downloaded from http://mpp.mpg.de/bat
// ***************************************************************

#include <BAT/BCH1D.h>
#include <BAT/BCH2D.h>
#include <BAT/BCLog.h>
#include <BAT/BCAux.h>
#include <BAT/BCSummaryTool.h>

#include "MyModel.h"

int main()
{

   // set nicer style for drawing than the ROOT default
   BCAux::SetStyle();

   // open log file
   BCLog::OpenLog("log.txt");
   BCLog::SetLogLevel(BCLog::detail);

   // create new MyModel object
   MyModel* m = new MyModel();

   // set precision
   m->MCMCSetPrecision(BCEngineMCMC::kMedium);

   BCLog::OutSummary("Test model created");

   // create a new summary tool object
   BCSummaryTool * summary = new BCSummaryTool(m);

   // perform your analysis here

   // normalize the posterior, i.e. integrate posterior
```

```
   // over the full parameter space
//   m->SetIntegrationMethod(BCIntegrate::kDefault);
//   m->SetIntegrationMethod(BCIntegrate::kIntMonteCarlo);
//   m->SetIntegrationMethod(BCIntegrate::kIntCuba);
//   m->SetIntegrationMethod(BCIntegrate::kIntGrid);
//   m->Normalize();

   // run MCMC and marginalize posterior wrt. all parameters
   // and all combinations of two parameters
//   m->SetMarginalizationMethod(BCIntegrate::kMargDefault);
//   m->SetMarginalizationMethod(BCIntegrate::kMargMetropolis);
//   m->SetMarginalizationMethod(BCIntegrate::kMargMonteCarlo);
//   m->SetMarginalizationMethod(BCIntegrate::kMargGrid);
//   m->MarginalizeAll();

   // run mode finding; by default using Minuit
//   m->SetOptimizationMethod(BCIntegrate::kOptDefault);
//   m->SetOptimizationMethod(BCIntegrate::kOptMinuit);
//   m->SetOptimizationMethod(BCIntegrate::kOptSimAnn);
//   m->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
//   m->FindMode();

   // if MCMC was run before (MarginalizeAll()) it is
   // possible to use the mode found by MCMC as
   // starting point of Minuit minimization
//   m->FindMode( m->GetBestFitParameters() );

   // draw all marginalized distributions into a PostScript file
//   m->PrintAllMarginalized("MyModel_plots.pdf");

   // print individual histograms
//   m->GetMarginalized("x")->Print("x.pdf");
//   m->GetMarginalized("y")->Print("y.pdf");
//   m->GetMarginalized("x", "y")->Print("xy.pdf");

   // print all summary plots
//   summary->PrintParameterPlot("MyModel_parameters.pdf");
//   summary->PrintCorrelationPlot("MyModel_correlation.pdf");
//   summary->PrintKnowledgeUpdatePlots("MyModel_update.pdf");

   // calculate p-value
//   m->CalculatePValue( m->GetBestFitParameters() );

   // print results of the analysis into a text file
//   m->PrintResults("MyModel_results.txt");

   delete m;
   delete summary;
```

```
   BCLog::OutSummary("Test program ran successfully");
   BCLog::OutSummary("Exiting");

   // close log file
   BCLog::CloseLog();

   return 0;

}
```

2.3.4 Roostats interface

An alternative method to create a model is to create a Roostats workspace and to use the interface class `BCRooInterface` inheriting from `BCModel`. The model requires the measured `RooAbsData data` and the model to describe these data `RooAbsPdf model` to specify the likelihood. In addition, it needs the `RooAbsPdf prior`, and the specification of the parameters as a `RooArgSet` and more specifically the subset of parameters constituting the parameters of interest (POI) as a `RooArgSet`. The model can be initialized directly using

```
void BCRooInterface::Initialize(RooAbsData& data, RooAbsPdf& model,
                                RooAbsPdf& prior, const RooArgSet* params,
                                const RooArgSet& listPOI );
```

Assuming the workspace is stored in a ROOT file `rootFile`, set up the model with

```
BCRooInterface m;
m.Initialize(rootFile, wsName, "data", "model", "priorPOI",
             "priorNuisance", "parameters", "POI");
```

where the string arguments point to the corresponding elements in the workspace `wsName`. Note that priors are defined separately for parameters of interest and the nuisance parameters. An example on the usage of `BCRooInterface`, including a README, is shipped with BAT in the subdirectory `examples/advanced/roointerface`.

Warning: The posterior computed with `BCRooInterface` is not thread safe at present! If BAT is compiled with OpenMP support, set the number of threads to one when using `BCRooInterface`.

2.4 Data

2.4.1 Data format and handling

Data are managed in the form of data points that are combined to data sets. Data points and sets are implemented as classes `BCDataPoint` and `BCDataSet`.

The class `BCDataPoint` contains a set of double precision values. Data points can be generated explicitly by the user with or without initial values with the constructors

```
BCDataPoint::BCDataPoint(int nvariables),
BCDataPoint::BCDataPoint(vector<double> x).
```

Values of a data point can be set either one-by-one or all at once with

```
void BCDataPoint::SetValue(int index, double value),
void BCDataPoint::SetValues(std::vector <double> values).
```

The value of the ith entry can be recalled with

```
double BCDataPoint::GetValue(int index).
```

A data point can be added to a data set with

```
void BCDataSet::AddDataPoint(BCDataPoint * datapoint).
```

Alternatively, data can be read in from a file

- `int BCDataSet::ReadDataFromFile(char* filename, char* treename, const char* branchnames)`
  Data points are read from a ROOT tree according to the comma-separated branch names in `branchnames`.
- `int BCDataSet::ReadDataFromFile(char* filename, int nvariables)`
  Data points are read from an ASCII file containing one data point per line; each data point contains `nvariables` values.

Once a data set is defined it can be assigned to a model with

```
void BCModel::SetDataSet(BCDataSet * dataset).
```

Similarly, a data set can be returned from a model with

```
BCDataSet * BCModel::GetDataSet().
```

Though one can define several data sets, a model can only have one data set at a time. This data set can be accessed in the overloaded method `BCModel::LogLikelihood`.

2.4.2 Constraining the values of data points

For two applications discussed later in this section—the goodness-of-fit test and the calculation of error bands—it is necessary to define the limits of the data points. This is done for each variable separately. The limits are defined by the model, not by the data set:

```
void BCModel::SetDataBoundaries(int index, double lowerboundary,
                                double upperboundary, bool fixed=false).
```

2.5 Managing more than one model: the model manager

In case more than one model is defined and all models use the same data set a model manager can be defined. It is implemented as a class named `BCModelManager`. Models and their prior probabilities are added to the model manager via

```
void BCModelManager::AddModel(BCModel * model, double probability=0.),
```

where `probability` is the prior probability for the model[2]. A common data set can be defined and will be used by all models added to the manager. This can be done either explicitly via

```
void BCModelManager::SetDataSet(BCDataSet * dataset),
```

or by reading data from a file via

```
int BCModelManager::ReadDataFromFile(char * filename, char * treename,
                                     const char * branchnames),
int BCModelManager::ReadDataFromFile(char * filename, int nvariables).
```

2.6 Normalization and numerical integration

The posterior probability density function (pdf) is normalized to unity in Bayes' theorem. The normalization is an integral of the conditional probability times the prior probability over the whole parameter range. Since the analytical form of the integral is not known in general this integral is solved numerically. `BCModel` inherits from `BCIntegrate`, which contains several methods for numerical integration. An integration method can be chosen by

```
void BCIntegrate::SetIntegrationMethod(BCIntegrate::BCIntegrationMethod method),
```

where `BCIntegrationMethod` can be one of the following

- `BCModel::kIntMonteCarlo`. Sample-mean integration.
- `BCModel::kIntCuba`. An interface to the CUBA library [3, 4]; cf. Section 2.6.1.
- `BCModel::kIntGrid`. Evaluation on a grid (histogram). Only available for 1- and 2-dimensional problems.
- `BCIntegrate::kIntDefault`. Default method. Use `BCIntegrate::kIntGrid` for 1- and 2-dimensional problems and `BCIntegrate::kIntMonteCarlo` for more than 2 dimensions. If CUBA is available, `BCModel::kIntCuba` is used for more than 2 dimensions.

The normalization can be performed for each model separately or for all models belonging to a model manager:

```
double BCModel::Normalize(),
void BCModelManager::Normalize().
```

The normalization is stored for each model. The value can be obtained by

```
double BCModel::GetNormalization() .
```

Once the integral is calculated the posterior pdf for a set of parameter values can be evaluated

```
double BCModel::Probability(const std::vector <double> & parameter),
double BCModel::LogProbability(const std::vector <double> & parameter),
```

where the latter returns the logarithm of the posterior pdf.

[2]Here the prior probability is for the model itself and is not to be confused with the priors on parameters *within* a given model.

2.6.1 Cuba interface

The Cuba library contains four different integration algorithms. In BAT one can choose the method using

```
void BCIntegrate::SetCubaIntegrationMethod(BCIntegrate::BCCubaMethod method)
```

where `BCCubaMethod` is one of `kCubaVegas` (default), `kCubaSuave`, `kCubaDivonne`, or `kCubaCuhre`. Note that the evaluation of the posterior is run in parallel by default. If the posterior is not thread safe it is recommended to set the environment variable `CUBACORES` to 1.

One can tune the individual methods' parameters using the structs `BCCubaOptions::Method` following methods as follows:

```
BCCubaOptions::Vegas o = model.GetCubaVegasOptions();
o.flags = 1;
model.SetCubaOptions(o);
model.IntegrateCuba(BCIntegrate::kCubaVegas);
```

Note that the default options for each algorithm—taken from the example shipping with Cuba—may require some tuning to yield accurate results; see the Cuba manual for details.

2.7 Parameter estimation and marginalization

The posterior pdf can be used to estimate the set of parameter values most suited to describe the data. This is done by searching for the most probable value, or mode, of the posterior pdf. Two approaches are followed: either the mode in the whole parameter space is searched for, or the pdf is marginalized with respect to the particular parameter under study. In the latter case, several quantities can used to describe the marginalized distributions.

2.7.1 Maximization of the full posterior probability density

The maximization of the full posterior pdf can be performed using the method

```
void BCModel::FindMode(std::vector<double> start = std::vector<double>(0)).
```

If no starting point is passed by the user, there are two cases. If any algorithm (e.g., MCMC or grid marginalization) has been previously run on the model that yields an estimate of the mode, that estimate is used as the starting point. Else the starting point is at the center along each parameter dimension.

The vector of parameter values which maximizes the posterior pdf can be obtained using

```
std::vector <double> BCModel::GetBestFitParameters(),
double BCModel::GetBestFitParameter(unsigned int index).
```

The implemented algorithms can be chosen with the command

```
BCIntegrate::SetOptimizationMethod(BCIntegrate::BCOptimizationMethod method).
```

Three methods are available in the current version (Version 0.9.4):

- `BCIntegrate::kOptMetropolis`. A sampling algorithm using the Metropolis algorithm.

- `BCIntegrate::kOptMinuit`. An interface to the ROOT version of Minuit.
- `BCIntegrate::kOptSimAnn`. A Simulated Annealing algorithm.
- `BCIntegrate::kOptDefault`. Default method. Set to `BCIntegrate::kOptMinuit`.

If the interface to Minuit is used, the estimated uncertainties on the parameters can be obtained using

```
std::vector <double> BCModel::GetBestFitParameterErrors(),
double BCModel::GetBestFitParameterError(unsigned int index).
```

Settings and options of the Simulated Annealing algorithm are summarized in section 5.2.

If several (optimization) algorithms are run one after the other, the best-fit parameters are updated only if an increase in the target function is detected. A flag controls whether to ignore the results from a previous optimization:

```
void BCIntegrate::SetFlagIgnorePrevOptimization(bool flag).
```

2.7.2 Marginalization

The single-parameter estimation is done via marginalization. If more than one parameter is studied it is most efficient to marginalize with respect to all parameters simultaneously. This can be done using

```
int BCModel::MarginalizeAll() .
```

One- and two-dimensional histograms are filled during the marginalization. They can be accessed by[3]

```
BCH1D * BCModel::GetMarginalized(BCParameter * parameter),
BCH2D * BCModel::GetMarginalized(BCParameter * parameter1,
                                  BCParameter * parameter2),
```

Different methods of marginalization are implemented and can be chosen via

```
void BCIntegrate::SetMarginalizationMethod(BCIntegrate::BCMarginalizationMethod method)
```

where `BCMarginalizationMethod` can be one of the following

- `BCIntegrate::kMargMonteCarlo`. Uncorrelated Monte Carlo sampling; i.e., samples are independently drawn from a uniform distribution on the parameter space.
- `BCIntegrate::kMargMetropolis`. Correlated sampling using Markov Chain Monte Carlo (Metropolis algorithm).
- `BCIntegrate::kMargGrid`. Evaluation on a grid (histogram). Only available for 1- and 2-dimensional problems.
- `BCIntegrate::kMargDefault`. Default method. Use `BCIntegrate::kMargGrid` for 1- and 2-dimensional problems and `BCIntegrate::kMargMetropolis` for more than 2 dimensions.

Settings and options of the Markov Chain Monte Carlo algorithm are summarized in section 5.1.

[3] In the following, functions with `BCParameter` arguments exist also as versions taking a parameter name or a parameter index as an argument instead.

2.8 Model comparison and hypothesis testing

2.8.1 Model comparison

Models M_i can be compared by their posterior probability. Technically, the models are added to a model manager and given a prior probability (see Section 2.5). The posterior probability for the ith model, given the data, is simply

$$p(\mathrm{M_i}|\mathrm{data}) = \frac{N_\mathrm{i} \cdot p_0(\mathrm{M_i})}{\sum_{\mathrm{j}=1}^{N} N_\mathrm{j} \cdot p_0(\mathrm{M_j})} \,, \tag{1}$$

where N_i is the normalization of the ith model posterior pdf and $p_0(M_i)$ is the prior probability for the ith model. The posterior probability for a model can be evaluated once the model manager is initialized and all numerical integrations are performed via

```
void BCModelManager::Normalize().
```

The posterior probability can be returned from the model using the following method:

```
double BCModel::GetModelAPosterioriProbability().
```

Alternatively, Bayes factors can be calculated for two models using

```
double BCModelManager::BayesFactor(const unsigned int imodel1,
                                   const unsigned int imodel2),
```

where the arguments are the indices of the models in the model manager.

2.8.2 Goodness-of-fit test

Once the most suitable set of parameters, $\vec{\lambda}^*$, for a given model and data set, D, is estimated it is necessary to verify that the one model under consideration gives a reasonable representation of the data (regardless of any alternative models). To this end, one can define a test statistic and calculate a p-value. Many more details on the interpretation of p-values and the various choices of test statistics for common fitting problems can be found in [6].

For the most general model, this is accomplished as follows. Data sets, $\{\tilde{D}\}$, are generated under the assumption of the model and the best-fit-parameters. A frequency distribution f of the obtained conditional probability $k = p(\tilde{D}|\vec{\lambda}^*)$ is calculated and interpreted as probability density. k is used as the test statistic. The p-value is defined as the probability to find a conditional probability $p(\tilde{D}|\vec{\lambda}^*)$ equal or less than that found for the original data set, $k_0 = p(D|\vec{\lambda}^*)$, i.e.

$$p \equiv \int_0^{k_0 = p(D|\vec{\lambda}^*)} f(k)\, \mathrm{d}k \,. \tag{2}$$

In the most general case, the p-value is calculated using Markov Chain Monte Carlo. The dimension of the sampled space is the number of datapoints. The calculation can be started by

```
BCH1D * BCModel::CalculatePValue(std::vector<double> par,
                                 bool flag_histogram=false),
```

where `par` is a vector of the best-fit-parameters. The method returns a pointer to a `BCH1D` if the flag is set to true. The p-value is calculated from this distribution and can be obtained by `double BCModel::GetPValue().`

For a number of models the distribution of test certain statistics is approximately known. Hence the CPU intensive generation of data sets can be avoided, and the p-value is computed much faster.

Gauss For *Gaussian* problems (handled by the model `BCGraphFitter`, sec. 3.2.1) the standard χ^2 statistic can be used, including the correction for the number of fitted parameters, from

```
double BCModel::GetPvalueFromChi2NDoF(std::vector<double> par, int sigma_index)
```

where `sigma_index` is the index of the standard deviation in each data point.

Poisson In the *Poisson* case (described by the model `BCHistogramFitter`) there are the following three specific choices to obtain a p-value (for details see [6], Sec. III.D and IV.B):

1. ```
   int BCHistogramFitter::CalculatePValueFast(const std::vector<double> & par,
                                              double & pvalue,
                                              double & pvalueCorrected,
                                              int nIterations=100000)
   ```

   which uses a discrete Markov chain to vary the bin counts. The conditional probability k is then recomputed, and again the proportion of datasets with lower k is reported as the p-value. In addition, the corrected p-value is calculated in an attempt to produce a more uniformly distributed p-value. The p-value if first transformed to a χ^2 using the number of bins as degrees of freedom. Then this χ^2 is transformed back into a p-value with the degrees of freedom reduced by the number of fit parameters. A more thorough explanation of the algorithm is given in the appendix of [6].

2. ```
   int BCHistogramFitter::CalculatePValueLikelihood(const std::vector<double> & par,
                                                    double & pvalue)
   ```

   ues the fact that the rescaled likelihood ratio defined in Eq. (32.12) of [5] has an approximate χ^2-distribution if all the bin counts aren't too small; i.e. $n_i \geq 5$.

3. ```
   int BCHistogramFitter::CalculatePValueLeastSquares(const std::vector<double> & par,
                                                      double & pvalue,
                                                      bool weightExpect=true)
   ```

   calculates the sum of squared differences between observed counts and expected counts,

   $$\chi^2 = \sum_{i=1}^{N} \frac{(n_i - \nu_i(\lambda^*))^2}{\sigma_i^2},$$

   where the weights σ_i can be chosen as the expectation values from the Poisson distribution $\sigma_i^2 = \nu_i$ (`weightExpect=true`) or as the observed counts, $\sigma_i^2 = n_i$ (`weightExpect=false`). The latter choice is especially problematic for no observed count, $n_i = 0$. In that case the weight is arbitrarily set to unity. χ^2 has an approximate χ^2-distribution with $N - dim(\lambda)$ degrees of freedom for $n_i > 5$.

In all three methods the return value is an integer error code, and the resulting p-value is stored in the reference `double &pvalue`.
   ```

Binomial For binomial uncertainties handled by `BCEfficiencyFitter`, the same fast p-value methods are available as for the Poisson case (see above).

2.9 Propagation of uncertainties

During the marginalization, each point in parameter space is sampled with a frequency proportional to the posterior pdf at this point. It is possible in BAT to calculate any (user-defined) function of the parameters during the marginalization and thus obtain a frequency distribution for the function value(s). This in turn can be interpreted as the probability density for the function value(s). The uncertainties on the parameters are thus propagated to the function under study. An example for the propagation of uncertainties is the calculation of the uncertainty band for the case of function fitting (see Section 3.2). Uncertainty propagation can be done by overloading[4]

`BCEngineMCMC::MCMCUserIterationInterface()`

which is called at every iteration during the main run of the MCMC. The user has to loop over all chains and parameters using the protected variable `fMCMCx`, which is a vector of double values with a length of the number of chains times the number of parameters. An example code is given here which calculates a radius in 3D for each iteration (and chain) and fills it into a histogram:

```
void MyModel::MCMCIterationInterface()
{
  // get number of chains
  int nchains = MCMCGetNChains();

  // get number of parameters
  int npar = GetNParameters();

  // loop over all chains and fill histogram
  for (int i = 0; i < nchains; ++i) {
    // get the current values of the parameters x, y, z. These are
    // stored in fMCMCx.
    double x = fMCMCx.at(i * npar + 0); // parameter with index 0 in chain i
    double y = fMCMCx.at(i * npar + 1); // parameter with index 1 in chain i
    double z = fMCMCx.at(i * npar + 2); // parameter with index 2 in chain i

    // calculate the radius
    double r = sqrt(x*x + y*y + z*z);

    // fill the histogram
    myHistogramR->Fill(r);
  }
}
```

An example for the propagation of uncertainties can be found in `examples/basic/errorpropagation`.

[4] `BCIntegrate` inherits from `BCEngineMCMC`; `BCModel` in turn inherits from `BCIntegrate`.

2.10 Thread parallelization

At present, threads are used only to speed up the Markov chain sampling by running each chain in parallel. Assuming BAT is configured with parallelization enabled, the number of threads can be selected at runtime without recompilation with `./program OMP_NUM_THREADS=N`. Due to the overhead from thread creation, the sampling is faster only if the likelihood is sufficiently slow to compute. As a rule of thumb, if the unparallelized sampling takes minutes or even hours, the parallelized version should get close to the maximum speedup given by the number of cores. For very simple likelihoods, the overhead may actually slow down the entire process.

If BAT is compiled with support for threads and the number of threads is not set explicitly, it is implementation dependent whether only one thread is created or as many as there are available cores. To enforce serial execution, simply set `OMP_NUM_THREADS=1`.

For guidance, tests showed that it is beneficial to have as many threads as your computer provides. Working on a quad core machine with hyper-threading, we observed a speedup factor of 3.5 – 3.9 with 8 and 16 chains on 8 cores; the calculation took around 3 hours.

Warning: The results of the sampling become nonsense if multiple threads are used but the likelihood itself is not thread safe.

A simple example of a non-thread-safe likelihood is

```
double MyModel::LogLikelihood(const std::vector <double> & parameters)
{
  // assign member variable
  this->member = parameters[0];

  // value of member used in MyModel::Method
  return this->Method();
}
```

If the method `MyModel::Method` is called simultaneously from different threads with different `parameters`, its return value depends on the arbitrary call order of individual threads. A simple solution is to move `Method` into `OtherClass`, and to keep an independent copy of `OtherClass` for each thread.

```
#include <omp.h>

class MyModel
{
public:
  ...
  double LogLikelihood(const std::vector <double> & parameters)
  {
    return other.at(omp_get_thread_num()).Method(parameters);
  }
private:
  std::vector<OtherClass> other;
};
```

2.11 One- and two-dimensional histograms

The classes `BCH1D` and `BCH2D` are one- and two-dimensional histogram classes wrapping ROOT classes `TH1D` and `TH2D`. They are filled, e.g., during marginalization. To change the number of bins used for a particular parameter use

```
void BCParameter::SetNbins(unsigned nbins)
```

Pointers to the ROOT histograms can be returned using

```
TH1D * BCH1D::GetHistogram(),
TH2D * BCH2D::GetHistogram().
```

For one-dimensional histograms, once the histograms are filled, summary information can be obtained by

```
double BCH1D::GetMean(),
double BCH1D::GetMode(),
double BCH1D::GetMedian(),
```

and the quantiles of the distribution can be returned using

```
double BCH1D::GetQuantile(double probability),
```

where `probability` is a number between 0 and 1. This information can be used to estimate uncertainties (e.g., the central 64% probability region) or limits on parameters (e.g., the quantile for 0.95). Alternatively, the smallest set of intervals containing a certain probability can be obtained by

```
double BCH1D::GetSmallestInterval(double & min, double & max, double content=0.68).
```

Both types of histograms can be drawn to a ROOT `TCanvas` using the methods

```
BCH1D::Draw(std::string options, std::vector<double> intervals),
BCH2D::Draw(std::string options, std::vector<double> intervals),
```

where the options are summarized in Table 1 and Table 2.

The default 1D options are `"BTsiB3CS1D0Lmeanmode"` to show a histogram with the three smallest regions containing 68%, 95%, and 99.7% of the probability displayed as filled bands in green, yellow, and red. In addition, the marginal mean and standard deviation, as well as the projection of the global mode onto the one-dimensional subspace are drawn. The symbols are explained in a legend drawn above the histogram. Additional contour levels can be specified via `intervals`; e.g., `intervals = std::vector<double>(0.5)` requests a 50% probability contour. Similarly in 2D, the default options `"BTfB3CS1meangmodelmode"` further include a marker for the local marginal mode.

Table 1: Drawing options for one-dimensional histograms. Set lower part shows additional options when using `BCH1D::Print()`.

Option	Style	Comment
`B1`	Draw one band between values specified in `intervals`	Default
`B2`	Draw two bands between values specified in `intervals`	
`B3`	Draw three bands between values specified in `intervals`	
`BTci`	Band type is central interval	
`BTsi`	Band type is/are smallest interval(s)	Default
`BTul`	Band type is upper limit	
`BTll`	Band type is lower limit	
`D0`	Draw distribution as a histogram	Default
`D1`	Draw distribution as a smooth curve	
`CS0`	Color scheme 0 (b&w)	
`CS1`	Color scheme 1 (green/yellow/red)	Default
`CS2`	Color scheme 2 (blueish colors)	
`CS3`	Color scheme 3 (redish colors)	
`smooth1`	Use ROOT smoothing algorithm once	
`smooth3`	Use ROOT smoothing algorithm three times	
`smooth5`	Use ROOT smoothing algorithm five times	
`smooth10`	Use ROOT smoothing algorithm ten times	
`mode`	Indicate global mode	Default
`mean`	Indicate mean value and standard deviation	Default
`median`	Indicate median and central interval	
`quartiles`	Indicate quartiles	
`deciles`	Indicate deciles	
`percentiles`	Indicate percentiles	
`L`	Add a legend	Default
`same`	Add histogram on top of another histogram	
`R`	Rescale canvas so that histogram is square	
`logx`	Draw x-axis in log-scale	
`logy`	Draw y-axis in log-scale	

Table 2: Drawing options for two-dimensional histograms. Set lower part shows additional options when using `BCH2D::Print`.

Option	Style	Comment
`B1`	Draw one band between values specified in `intervals`	Default
`B2`	Draw two bands between values specified in `intervals`	
`B3`	Draw three bands between values specified in `intervals`	
`BTf`	Band type is a filled area	Default
`BTc`	Band type contour	
`CS0`	Color scheme 0 (b&w)	
`CS1`	Color scheme 1 (green/yellow/red)	Default
`CS2`	Color scheme 2 (blueish colors)	
`CS3`	Color scheme 3 (redish colors)	
`smooth1`	Use ROOT smoothing algorithm once	
`smooth3`	Use ROOT smoothing algorithm three times	
`smooth5`	Use ROOT smoothing algorithm five times	
`smooth10`	Use ROOT smoothing algorithm ten times	
`gmode`	Indicate global mode	Default
`lmode`	Indicate local mode	
`mean`	Indicate mean value and standard deviation	Default
`profilex`	Draw the profile line vs. x using the mode	
`profiley`	Draw the profile line vs. y using the mode	
`nL`	Remove the legend	
`R`	Rescale canvas so that histogram is square	
`logz`	Draw z-axis in log-scale	

3 Tools and models

3.1 Tools

3.1.1 The summary tool

A summary tool, `BCSummaryTool`, is provided with BAT, which summarizes the results of the marginalization. It creates a set of plots and tables. An instance of the class can be created by

```
BCSummaryTool() ,
BCSummaryTool(BCModel * model);
```

If the model is not set during construction it can be set using the method

```
void BCSummaryTool::SetModel(BCModel * model) .
```

After the marginalization of the posterior has been performed, a set of plots can be produced with the following methods:

- `int BCSummaryTool::PrintParameterPlot(const char* filename="parameters.pdf")`
  Creates an overview plots of the marginalized mode, standard deviation, the most important quantiles and the global mode for all parameters.
- `int BCSummaryTool::PrintCorrelationPlot(const char* filename="correlation.pdf")`
  Prints a two-dimensional correlation matrix of the parameters.
- `int BCSummaryTool::PrintKnowlegdeUpdatePlots(const char* filename="update.pdf")`
  Prints the marginalized distributions for the prior and posterior probablity. This illustrates the update in knowledge due to the data. Calling this function will re-run the analysis without the use of the `LogLikelihood` information.

In addition, a latex table of the parameters and the results can be produced with the method:

```
int BCSummaryTool::PrintParameterLatex(const char * filename) .
```

3.2 Models for function fitting

A common application in data analysis is fitting a function, $y(x)$, to a (one-dimensional) distribution or a set of data points. BAT offers three dedicated tools for this purpose, depending on the uncertainties on each data point. These classes and the assumed uncertainties are summarized in the following.

In all three cases, the uncertainties for each data point (or bin content) are assumed to be independent of each other. I.e., in the case of histograms bin-by-bin migration is not included. The overall conditional probability is a product of the individual probabilities for the expectation value given the y-value (or bin content).

An uncertainty band is calculated for each fit. During the marginalization, each point in parameter space is sampled with a frequency proportional to the posterior pdf at this point (if the Markov chain has converged). The uncertainty band is obtained by evaluating the fit function, $y(x)$, for each x at each point in parameter space. The values are histogrammed in x–$y(x)$–space. Each slice of x is normalized to unity and interpreted as probability density for y given

x. The 0.16 and 0.84 quantiles are then interpreted as the uncertainty on y at that particular x. The uncertainty band can be returned from a model using

```
TGraph * BCModel::GetErrorBandGraph(double level1, double level2) ,
```

where the levels correspond to the quantiles of the distribution $p(y(x))$ (default: 0.16 and 0.84).

In all three cases, the fast methods for evaluating the p-value are implemented. The p-value is evaluated automatically and returned together with a summary.

3.2.1 The Gaussian case

The class `BCGraphFitter` allows to fit a ROOT function (`TF1`) to a ROOT graph (`TGraphErrors`). The uncertainties on y at a given x, defined by the uncertainties of the `TGraphErrors`, are assumed to be Gaussian, i.e., the uncertainty on y corresponds to the width, σ, of the Gaussian. The uncertainties on x are not taken into account. An example for this fitter can be found in `examples/basic/graphFitter`.

3.2.2 The Poissonian case

The class `BCHistogramFitter` allows to fit a ROOT function (`TF1`) to a ROOT histogram (`TH1D`). The uncertainty on the expectation value in each bin is assumed to be Poissonian, and thus non-symmetric around the number of entries in this bin. An example for this fitter can be found in `examples/basic/histogramFitter`.

3.2.3 The Binomial case

The class `BCEfficiencyFitter` allows to fit a ROOT function (`TF1`) to the ratio of two ROOT histograms (`TH1D`). The uncertainty on the expectation value in each bin is assumed to be Binomial, and thus non-symmetric around the ratio of entries in this bin. The ratio is assumed to be between 0 and 1, i.e., one histogram contains a subset of the other. An example for this fitter can be found in `examples/basic/efficiencyFitter`.

3.3 Multi-template fitter

The Multi Template Fitter (MTF) is a tool which allows to fit several template histograms to a data histogram. The content of the bins in the templates are assumed to fluctuate independently according to Poisson distributions. Several channels can be fitted simultaneously.

3.3.1 Mathematical formulation

The multi-template fitter is formulated in terms of Bayesian reasoning. The posterior probability is proportional to the product of the Likelihood and the prior probability. The latter can be freely chosen by the user whereas the Likelihood is predefined. It is a binned Likelihood which assumes that the fluctuations in each bin are of Poisson nature and independent of each other. All channels, processes and sources of systematic uncertainties are assumed to be uncorrelated.

The parameters of the model are thus the expectation values of the different processes, λ_k, and the nuisance parameters, δ_l.

Excluding systematic uncertainies. In case no sources of systematic uncertainty are taken into account the Likelihood is defined as

$$L = \prod_{i=1}^{N_{\mathrm{ch}}} \prod_{j=1}^{N_{\mathrm{bin}}} \frac{\lambda_{ij}^{n_{ij}}}{n_{ij}!} e^{-\lambda_{ij}} \, , \tag{3}$$

where N_{ch} and N_{bin} are the number of channels and bins, respectively. n_{ij} and λ_{ij} are the observed and expected number of events in the jth bin of the ith channel. The expected number of events are calculated via

$$\lambda_{ij} = \sum_{k=1}^{N_{\mathrm{p}}} \lambda_{ijk} \tag{4}$$

$$= \sum_{k=1}^{N_{\mathrm{p}}} \lambda_k \cdot f_{ijk} \cdot \epsilon_{ik} \, , \tag{5}$$

where f_{ij} is the bin content of the jth bin in the normalized template of the kth process in the ith channel. ϵ_{ik} is the efficiency of the kth process in the ith channel specified when setting the template. λ_k is the contribution of the kth process and is a free parameter of the fit.

Including systematic uncertainties. In case sources of systematic uncertainties are taken into account, the efficiency ϵ_{ik} is modified according to a nuisance parameter:

$$\epsilon_{ik} \rightarrow \epsilon_{ik} \cdot (1 + \sum_{l=1}^{N_{\mathrm{syst}}} \delta_l \cdot \Delta\epsilon_{ijkl}) \, , \tag{6}$$

where δ_l is the nuisance parameter associated with the source of systematic uncertainty and $\Delta\epsilon_{ijkl}$ is the change in efficiency due to the lth source of systematic uncertainty in the ith channel and jth bin for the kth process.

3.3.2 Creating the fitter

The main MTF class `mtf` is derived from the `BCModel` class. A new instance can be created via

```
BCMTF::BCMTF()
BCMTF::BCMTF(const char * name) ,
```

where the name of the MTF can be specified via argument `name`.

3.3.3 Adding a channel

The MTF fits several channels simultaneously. These channels can be physics channels, e.g., $Z^0 \to e^+e^-$ and $Z^0 \to \mu^+\mu^-$, samples with disjunct jet multiplicity or entirely different classes altogether. A new channel can be added using the following method:

```
int BCMTF::AddChannel(const char * name) ,
```

where `name` is the name of the process, and the return value is an error code. Note that at least one channel has to be added.

3.3.4 Adding a data set

Each channel added to the MTF has a unique data set which comes in form of a (`TH1D`) histogram. It can be defined using the following method:

```
int BCMTF::SetData(const char * channelname,
                   TH1D hist) ,
```

where `channelname` is the name of the channel and `hist` is the histogram representing the data. The return value is an error code.

3.3.5 Adding a process

Each template that is fit to the data set corresponds to a process, where one process can occur in several channels. The fit then defines the contribution of the process and thus each process comes with one model parameter. A process can be added using the following method:

```
int BCMTF::AddProcess(const char * name,
                      double nmin = 0.,
                      double nmax = 1.) ,
```

where `name` is the name of the process and `nmin` and `nmax` are the lower and upper bound of the parameter associated with the contribution of the process. The parameter is denoted λ_k $(k = 1 \dots N_\mathrm{p})$ in section 3.3.1. Note that at least one process has to be added. A prior needs to be defined for each process, using the default `BCModel` methods.

It is likely that a single process will have different shapes in different channels. Thus, templates for a process need to be defined for each channel separately using the following method:

```
int BCMTF::SetTemplate(const char * channelname,
                       const char * processname,
                       TH1D hist,
                       double efficiency = 1.) ,
```

where channelname and processname are the names of the channel and the process, respectively. The parameter hist is the (TH1D) histogram (or template) which represents the process. The histogram will be normalized to unity and the entries in the normalized histogram are the probabilities to find an event of a process k and channel i in bin j. This probability is denoted f_{ijk} ($i = 1 \dots N_{\mathrm{ch}}$, $j = 1 \dots N_{\mathrm{b}}$, $k = 1 \dots N_{\mathrm{p}}$) in section 3.3.1. The last parameter, efficiency, is the efficiency of the process in that channel and is used to scale to template during the fit. This is needed if a process contributes with different amounts in two separate channels. The efficiency is denoted ϵ_{ik} ($i = 1 \dots N_{\mathrm{ch}}$, $k = 1 \dots N_{\mathrm{p}}$) in section 3.3.1. The return value is an error code. Note that templates do not have to be set if the process does not contribute to a particular channel.

3.3.6 Adding systematic uncertainties

Systematic uncertainties can alter the shape of a template. Sources of systematic uncertainty can be included in the fit using nuisance parameters. This nuisance parameter is assumed to alter the original template linearly, where values of -1, 0, and 1 correspond to the "downwards" shifted, nominal and "upwards shifted" template, respectively. The nuisance parameters are denoted δ_l ($l = 1 \dots N_{\mathrm{syst}}$) in section 3.3.1. Shifted refers to a change of one standard deviation. An example for a nuisance parameter could be the jet energy scale (JES). With a nominal JES of 1 and and uncertainty of 5%, the scaled templates correspond to a JES of 0.95 and 1.05, respectively. A prior needs to be defined for each nuisance parameter which is usually chosen to be a standard normal distribution. A source of systematic uncertainty can be added using the following method:

```
int BCMTF::AddSystematic(const char * name,
                         double min = -5.,
                         double max = 5.) ,
```

where name is the name of the source of systematic uncertainty and min and max are the lower and upper bound of the nuisance parameter, respectively. The return value is an error code.

Since the different sources of systematic uncertainty have an individual impact on each process and in each channel, these need to be specified. Two method can be used to define the impact:

```
int BCMTF::SetSystematicVariation(const char * channelname,
                                  const char * processname,
                                  const char * systematicname,
                                  TH1D hist_up,
                                  TH1D hist_down) ,
```

where channelname, processname and systematicname are the names of the channel, the process and the source of systematic uncertainty. The (TH1D) histograms hist_up and hist_down are the histograms corresponding to an "up"- and "down"-scaling of the systematic uncertainty of one standard deviation, i.e., for each bin entry y they are calculated as

$$\Delta_{\mathrm{up}} = (y_{\mathrm{up}} - y_{\mathrm{nominal}})/y_{\mathrm{nominal}} \,, \tag{7}$$

$$\Delta_{\text{down}} = (y_{\text{nominal}} - y_{\text{down}})/y_{\text{nominal}} \,. \tag{8}$$

Note the sign of the down-ward fluctuation. These histograms define the change of the bins in each template in the efficiency which is denoted $\Delta\epsilon_{ijkl}\,(i = 1\ldots N_{\text{ch}},\, j = 1\ldots N_{\text{b}},\, k = 1\ldots N_{\text{p}},\, l = 1\ldots N_{\text{syst}})$. For example, if the value for a particular bin of `hist_up` is 0.05, i.e., if the systematic uncertaintiy is 5% in that bin, then the efficiency of the process in that channel will be multiplied by $(1 + 0.05)$. The return value is an error code.

The second variant does not take the difference in efficiency, but calculates it internally from the absolute values:

```
int BCMTF::SetSystematicVariation(const char * channelname,
                                  const char * processname,
                                  const char * systematicname,
                                  TH1D hist,
                                  TH1D hist_up,
                                  TH1D hist_down) ,
```

where `channelname`, `processname` and `systematicname` are the names of the channel, the process and the source of systematic uncertainty. The (`TH1D`) histograms `hist`, `hist_up` and `hist_down` are the nominal histogram and the histograms corresponding to an "up"- and "down"-scaling of the systematic uncertainty of one standard deviation. In this case, the histograms are not the relative differences but the absolute values. The return value is an error code.

3.3.7 Running the fit

The fit can be started using one of the standard `BCModel` fitting methods, e.g.

```
BCMTF::MarginalizeAll() ,
BCMTF::FindMode() .
```

3.3.8 Output

The MTF produces several outputs:

- `PrintAllMarginalized(const char* name)` prints the marginalized distributions in 1D and 2D for all parameters, i.e., the processes and nuisance parameters into a PostScript file `name`.
- `PrintResults(const char* name)` writes a summary of the fit into a text file `name`.
-
```
PrintStack(int channelindex,
           const std::vector<double> & parameters,
           const char * filename = "stack.pdf",
           const char * options = "")
PrintStack(const char * channelname,
           const std::vector<double> & parameters,
           const char * filename = "stack.pdf",
           const char * options = "")
```

prints a stacked histogram of the templates and the data histogram in the file `name` using a set of parameters `parameters`. For example, these could be the best fit results. Several options can be specified:

- `logx`: uses a log-scale for the x-axis.
- `logy`: uses a log-scale for the y-axis.
- `logx`: plot the x-axis on a log scale
- `logy`: plot the y-axis on a log scale
- `bw`: plot in black and white
- `sum`: draw a line corresponding to the sum of all templates
- `stack`: draw the templates as a stack
- `e0`: do not draw error bars
- `e1`: draw error bars corresponding to sqrt(n)
- `b0`: draw an error band on the expectation corresponding to the central 68% probability
- `b1`: draw bands showing the probability to observe a certain number of events given the expectation. The green (yellow, red) bands correspond to the central 68% (95%, 99.8%) probability

3.3.9 Settings

Several settings can be changed which impact the fit.

- `SetFlagEfficiencyConstraint` sets a flag if the overall efficiency (calculated from the value given when setting a template and the corresponding systematic uncertainties) is constrained to be between 0 and 1 or not. The default value is `true`.

3.3.10 Analysis facility

The analysis facility allows to perform a variety of analyses and ensemble tests for a given MTF. It can be created using the constructor:

```
BCMTFAnalysisFacility::BCMTFAnalysisFacility(BCMTF * mtf) ,
```

where `mtf` is the corresponding MTF object.

3.3.11 Performing ensemble tests

Ensemble testing is done in two steps: first, ensembles are generated according to the processes defined in the MTF. The ensembles are stored in root files. In a second step, the ensembles are analyzed using the MTF specified.

Creating ensembles. Ensembles can be generated using several methods. A single ensemble can be generated using the following method:

```
std::vector<TH1D> BCMTFAnalysisFacility::
                  BuildEnsemble(const std::vector<double> & parameters) ,
```

where `parameters` is a set of parameters which corresponds to those in the template fitter, i.e., the process contributions and nuisance parameters. For most applications, the best fit parameters of the data set at hand is used. The return value is a set of histograms corresponding to a pseudo data set for the different channels.

A similar method is used to generate multiple ensembles:

```
std::vector<TH1D> BCMTFAnalysisFacility::
                  BuildEnsembles(const std::vector<double> & parameters,
                                 int nensembles) ,
```

where `nensembles` is the number of ensembles to be generated. The return value is a pointer to a `TTree` object in which the ensembles are stored. The entries in the tree are the parameters and the number of entries in each bin of the data histograms.

The third method is based on a tree where the tree contains a set of parameters for each ensemble. This option is preferred if, e.g., the ensembles should be varied accoring to the prior probabilities. The method used to generate ensembles is

```
std::vector<TH1D> BCMTFAnalysisFacility::
                  BuildEnsembles(TTree * tree, int nensembles) ,
```

where `tree` is the input tree. Note that the ensembles are randomized, i.e., the first event in the tree does not correspond to the first ensemble. This is done to avoid biases if the tree itself is the output of a Markov Chain.

Analyzing ensembles. Ensemble tests can be performed usign the ensembles defined earlier or using a set of parameters. In the former case, the method is:

```
TTree * BCMTFAnalysisFacility::
        PerformEnsembleTest(TTree * tree, int nensembles) ,
```

where `tree` is the tree of ensembles and `nensembles` is the number of ensembles to be analyzed. The return value is a tree containing the information about the analyzed ensemble. The list of variables is

- `parameter_i`: the ith parameter value used at the generation of the ensemble.
- `mode_global_i`: the ith global mode.
- `std_global_i`: the ith standard deviation evaluated with the global mode.
- `chi2_generated_i`: the χ^2 calculated using the parameters at generation of the ensemble for channel i.
- `chi2_mode_i`: the χ^2 calculated using the global mode parameters for channel i.
- `cash_generated_i`: the Cash statistic (Likelihood ratio) calculated using the parameters at generation of the ensemble for channel i.

- `cash_mode_i`: the Cash statistic (Likelihood ratio) calculated using the global mode parameters for channel i.
- `n_events_i`: the number of events in the ensemble in channel i.
- `chi2_generated_total`: the total χ^2 calculated using the parameters at generation of the ensemble.
- `chi2_mode_total`: the total χ^2 calculated using the global mode parameters.
- `cash_generated_total`: the total Cash statistic calculated using the parameters at generation of the ensemble.
- `cash_mode_total`: the total Cash statistic calculated using the global mode parameters.
- `n_events_total`: the total number of events in the ensemble.

Ensemble tests can also be performed using the following methd:

```
TTree * BCMTFAnalysisFacility::
        PerformEnsembleTest(const std::vector<double> & parameters,
                            int nensembles) ,
```

in which case the ensembles are generated internally using the parameters and are then analyzed.

By default the log messages for both the screen and the log-file are suppressed while performing the ensemble test. This can be changed using method:

```
void BCMTFAnalysisFacility::SetLogLevel(BCLog::LogLevel level) .
```

3.3.12 Performing automated analyses

The analysis facility also allows to perform an automated analysis over individual channels and or systematic uncertainties.

Performing single channel analyses. The current data set can be analyzed automatically for each channel separately using the analysis facility method

```
int BCMTFAnalysisFacility::
    PerformSingleChannelAnalyses(const char * dirname,
                                 const char * options = "")
```

where `dirname` is the name of a directory which will be created and into which all plots will be copied. If `mcmc` is specified in the `options` then the MCMC will be run for each channel. The method creates all marginalized distributions and results as well as an overview plot. If the option `nosyst` is specified, the systematic uncertainties are all switched off.

Performing single systematic analyses. Similarly, the method

```
int BCMTFAnalysisFacility::
    PerformSingleSystematicAnalyses(const char * dirname,
                                    const char * options = "")
```

can be used to perform a set of analyses for each systematic uncertainty separately.

Performing calibration analyses. Ensemble tests for different sets of parameters can be automized by using the method

```
int BCMTFAnalysisFacility::
    PerformCalibrationAnalysis(const char * dirname,
                               const std::vector<double> & default_parameters,
                               int index,
                               const std::vector<double> & parametervalues,
                               int nensembles = 1000)
```

which can be used to easily generate calibration curves. The ensembles are generated for a set of parameters, `default_parameters` where one of the parameters, `index`, can vary. The paramter values are defined by `parametervalues`. `nensembles` defines the number of pseudo data sets used for each ensemble.

4 Output

4.1 Log file

The class BCLog is used to write out information during the runtime of the program. The output is written to screen and to a log file. The level of detail can be set independently for both via

```
void BCLog::SetLogLevelFile(BCLog::LogLevel level),
void BCLog::SetLogLevelScreen(BCLog::LogLevel level),
void BCLog::SetLogLevel(BCLog::LogLevel levelscreen, BCLog::LogLevel levelfile),
```

where the level is one of the following

- `debug`: Lowest level of information.
- `detail`: Details of functions, such as the status of the Markov chains, etc.
- `summary`: Results, such as best-fit values, normalization, etc.
- `warning`: Warning messages
- `nothing`: Nothing is written out.

A log file has to be opened in the beginning of the main file using

```
void BCLog::OpenLog(const char * filename).
```

4.2 Summary information

A summary of the MCMC run can be written to file or printed to the screen using

```
void BCModel::PrintSummary(),
void BCModel::PrintSummary(const char * file).
```

The summary contains information about the convergence status, the models, their parameters and respective ranges as well as information about the marginalization (e.g., mean and rms, median and central 68% interval, and the smallest interval containing 68% probability). The results of the global maximization of the posterior probability is also summarized.

4.3 Histograms

Histograms of the marginalized distributions can be stored to disk with the functions

```
int BCIntegrate::PrintAllMarginalized1D(const char * filebase),
int BCIntegrate::PrintAllMarginalized2D(const char * filebase),
int BCIntegrate::PrintAllMarginalized(const char * file,
    std::string options1d="BTsiB3CS1D0pdf0Lmeanmode",
    std::string options2d="BTfB3CS1meangmode",
    unsigned int hdiv=1, unsigned int ndiv=1),
```

where `file` (or `filebase`) is the filename, the options for individual plots are explained in section 2.11, and `hdiv` and `ndiv` define the number of divisions in the plots. Alternatively, the histograms can be obtained from the model (see section 2.7.2) and then plotted, printed or stored in a ROOT file. Supported formats are `.ps` and `.pdf` (default).

4.4 The output class

The results of an analysis can be stored in a ROOT file using the output class `BCModelOutput`. This class is assigned a model class and a file. It contains a ROOT tree which stores the most important information of the analysis outcome, such as the global mode, the marginalized mode, means, limits, etc. The constructors are

```
BCModelOutput()
BCModelOutput(BCModel * model, const char * filename).
```

The model and filename can be set after construction using

```
BCModelOutput::SetModel(BCModel * model),
BCModelOutput::SetFile(const char * filename).
```

The marginalized distributions can also be stored in the output file using

```
BCModelOutput::WriteMarginalizedDistributions().
```

The single points of the Markov Chain(s) can also be stored in the output file together with the posterior probability at these points. This can be done by setting a flag before the Markov Chain is run:

```
BCModelOutput::WriteMarkovChain(bool flag = true).
```

Please note that the file size can be large in case you chose this options. Each chain is stored as a ROOT tree. This option allows for offline diagnostics of the chains. The variables stored are:

- Phase: describes the phase of the running (1: pre-run, 2: main run),
- Cycle: described the cycle of the chain in the pre-run,
- Iteration: the current iteration number,
- NParameters: the number of parameters,
- LogProbability: the log of the posterior probability,
- Parameter$i$: the parameter value of the ith parameter.

If the flag `BCIntegrate::SetFlagWriteSAToFile()` is set to true, the output file will also contain a tree with the points evaluated for the simulated annealing method. The variables stored are:

- Iteration: the iteration
- Temperature: the temperature (defines the step length)
- LogProbability: the log of the posterior probability,
- Parameter$i$: the parameter value of the ith parameter.

5 Settings, options and special functions

5.1 Markov Chain settings and options

The most important options for the Markov Chains are listed here. For further reference see the reference guide:

- `BCEngineMCMC::MCMCSetNChains(int n)` Sets the number of chains which are run in parallel (default: 5).
- `BCEngineMCMC::MCMCSetNLag(int n)` Sets the lag of the algorithm, i.e., only every n-th point is stored (default: 1).
- `BCEngineMCMC::MCMCSetNIterationsMax(int n)` Sets the maximum number of iterations of the pre-run (default: 1,000,000).
- `BCEngineMCMC::MCMCSetNIterationsPreRunMin(int n)` Sets the minimum number of iterations of the pre-run (default: 1,000,000).
- `BCEngineMCMC::MCMCSetNIterationsRun(int n)` Sets the number of iterations of the analysis run (default: 100,000).
- `BCEngineMCMC::MCMCSetNIterationsUpdate(int n)` Sets the number of iterations (default: 1,000) after which the chains are updated in the pre-run (e.g., for the calculation of the efficiency and convergence tests). Note that if there are k parameters, each changed one at a time, the actual number of posterior evaluations between two convergence checks is $\min(k \cdot n, 10000)$, where n is the number of iterations in which a new value of exactly one of the k parameters is proposed.
- `BCEngine::MCMCSetFlagOrderParameters(bool flag)`. Decides if all parameters should be varied one after each other (true) or all at the same time (default: true).
- `BCEngine::MCMCSetFlagFillHistograms(bool flag)`. Sets the flag for all parameters to either fill histograms or not.
- `BCEngine::MCMCSetFlagPreRun(bool flag)`. Sets the flag if a prerun should be performed or not.
- `BCEngineMCMC::MCMCSetFlagInitialPosition(int flag)` Decides how to chose the initial positions (0: center of the parameter boundaries, 1: random positions (default), 2: user defined positions).
- `BCEngineMCMC::MCMCSetFlagFillHistograms(int index, bool flag)` and `BCEngineMCMC::MCMCSetFlagFillHistograms(bool flag)`. Set flag to fill the marginalized distribution for a single or all parameters. Not filling the distributions might increase the speed of MCMC run.
- `BCEngineMCMC::MCMCSetWriteChainToFile(bool flag)` Sets flag to write MCMC to file.
- `BCEngineMCMC::MCMCSetWritePreRunToFile(bool flag)` Sets flag to write pre run to file.
- `BCEngineMCMC::MCMCSetInitialPositions(std::vector<double> x0s)` and `MCMCSetInitialPositions(std::vector< std::vector<double> > x0s)` Set the initial

positions of all parameters in all chains.

- `BCEngineMCMC::MCMCSetMinimumEfficiency(double efficiency)` and `MCMCSetMaximumEfficiency(double efficiency)`. Set the minimum (default: 15%) and maximum (default: 50%) efficiency of the Markov Chains. The efficiency found in the pre-run has to be within these limits otherwise the pre-run continues.

- `BCEngineMCMC::MCMCSetRValueCriterion(double r)` Set the R-value criterion for convergence of a set of chains (default: 0.1).

- `BCEngineMCMC::MCMCSetRValueParametersCriterion(double r)`. Sets the parameter R-value criterion for convergence of all chains.

- `BCEngineMCMC::MCMCSetRValueStrict(bool flag)` Calculate the R-value criterion for convergence with the strict definition (true) by [8] or use a relaxed version (false) which doesn't guarantee $R \geq 1$, but allows for similarly robust convergence checking in slightly less iterations. (default: relaxed definition)

- `BCEngineMCMC::MCMCSetTrialFunctionScaleFactor(std::vector <double> scale)` Set the the initial scale for all one dimensional proposal functions (default: 1). BAT updates the individual scales until the efficiency is in the desired range (see also `BCEngineMCMC::MCMCSetMinimumEfficiency`). If it is known that the. support of the posterior concentrates in a small region of parameter space,. setting the scales to a value smaller than 1 will help the chain to achieve. the desired efficiency faster. Alternatively, one can shrink the parameter range.

- `BCEngineMCMC::WriteMarkovChain(bool flag)`. Set a flag to write the Markov Chain into a ROOT file. See section 4.4 on how to handle output in BAT.

- `BCEngineMCMC::MCMCSetPrecision(BCEngineMCMC::Precision precision)`. Set pre-defined values for running the algorithm with different precision. Possible varguments are `BCEngineMCMC::kLow` (for quick runs), `BCEngineMCMC::kMedium` (for "normal" running), `BCEngineMCMC::kHigh` (for publications).

- `BCEngineMCMC::MCMCSetRandomSeed(unsigned seed)`. Set the random number generator used with the Markov chain. The default seed is 0, i.e., it is randomly initialized. To obtain reproducible results, use a non-zero seed. E.g. `m->MCMCSetRandomSeed(21340)`.

Proposal function The proposal function is set to a Breit-Wigner function per default. The width of the Breit-Wigner is adjusted during the pre-run to match the required efficiency of the sampling. The user can overload the method

```
void BCEngineMCMC::MCMCTrialFunction(int chain, std::vector <double> &x)
void BCEngineMCMC::MCMCTrialFunctionSingle(int ichain,
                                           int iparameter,
                                           std::vector <double> &x)
```

The first method is used in the case of unordered sampling, the second one for ordered sampling. The vector `x` is filled with a random number preferably in the range of 0 to 1. The numbers will be scaled to the valid parameter range. An example for changing the trial function can be found in `examples/expert/TrialFunction`.

5.2 Settings and options for Simulated Annealing

The most important options for the implemented Simulated Annealing algorithm are listed here. For further reference and the information about the implementation see the reference guide and Ref. [9]:

- `BCIntegrate::SetSASchedule(BCIntegrate::BCSASchedule)`. Set the annealing schedule and thus the shape of the proposal function. Possible values are
  - `BCIntegrate::kSACauchy`,
  - `BCIntegrate::kSABoltzmann`,
  - `BCIntegrate::kSACustom`.
- `BCIntegrate::SetSAT0(double T0)`. Set the starting temperature. The temperature defines the step size.
- `BCIntegrate::SetSATmin(double Tmin)`. Set the threshold temperature. The temperature defines the step size.
- `BCIntegrate::SetFlagWriteSAToFile(bool flag)`. Set a flag to write the individual steps of the simulated annealing into a ROOT file. See section 4.4 on how to handle output in BAT.

References

[1] A. Caldwell, D. Kollar and K. Kroeninger, *BAT - The Bayesian Analysis Toolkit*, *Comp. Phys. Comm.* **180** (2009) 2197-2209 [arXiv:0808.2552]. `http://www.mpp.mpg.de/bat/`

[2] `http://root.cern.ch/`

[3] T. Hahn, *CUBA: A library for multidimensional numerical integration*, *Comp. Phys. Comm.* **168** (2005) 78 [arXiv:hep-ph/0404043].

[4] `http://www.feynarts.de/cuba/`

[5] Particle Data Group, Mathematical Tools or Statistics, Monte Carlo, Group Theory, Physics Letters B, Volume 667, Issues 1-5, Review of Particle Physics, 11 September 2008, Pages 316-339, ISSN 0370-2693, DOI: 10.1016/j.physletb.2008.07.030. (http://www.sciencedirect.com/science/article/B6TVN-4T4VKPY-G/2/32a7641753a1a6d41124d1992263243a).

[6] F. Beaujean and A. Caldwell and D. Kollár and K. Kröninger, *p-values for model evaluation*, *Phys. Rev. D* **83** (2011) [arxiv:1011.1674].

[7] V.E. Johnson, *A Bayesian chi2 Test for Goodness-of-Fit*, The Annals of Statistics 32 (2004) 2361.

[8] A. Gelman and D. B. Rubin, *Inference from Iterative Simulation Using Multiple Sequences*, *Statistical Science* **7** (1992) 4, 457.

[9] C Brachem, *Implementation and test of a simulated annealing algorithm in the Bayesian Analysis Toolkit (BAT)*, Bachelor thesis, II.Physik-UniG-Bach2009/03 (July 2009).