BAT manual

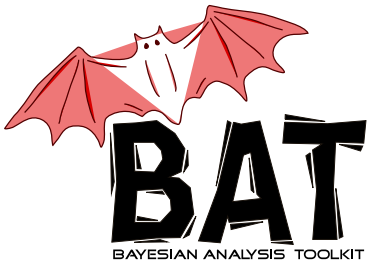
1.0.0

Generated by Doxygen 1.8.11

Contents

Chapter 1

Main Page



For the impatient

For a quick tutorial covering the basic features, jump directly to the basic tutorial.

The purpose of this document is to introduce the Bayesian Analysis Toolkit (BAT), a C++ package providing tools to

- compare model predictions with data,
- draw conclusions on the validity of a model as a representation of the data,
- and to extract the values of the free parameters of a model.

BAT was originally developed at the `Max Planck institute for physics` in Munich, Germany, in the context of particle physics. BAT uses ROOT for data handling and graphical output.

How to read this document

If you are new to BAT, we recommend to start with the [basic tutorial](Getting started) to quickly get you going. If you then want to learn more, check the brief introduction to the statistical basics in Bayesian Statistics. The core algorithm of BAT is Markov chain Monte Carlo, described in Markov chain Monte Carlo. Further chapters describe other tools of BAT, such as Optimization, specific statistical models, and advanced features.

Further help

Please visit our home page for an overview of papers, research, and other activities related to BAT. The code development takes place over at github.

This document is meant to is quickly enable to you to find your way around BAT. We describe the common workflow and explain how to solve some not-so-common tasks. To keep the size of this document manageable, we refer to the html reference guide in which all classes, methods etc. are documented. The BAT source code comes with many examples that illustrate how to use the various features. You can browse them online.

If you run into a problem with BAT, the preferred method is to create a new issue on github. This allows other users to easily find your issue online and benefit from the solution. If you provide some code for us to reproduce the problem, we can help you much faster. Have a look at issues with the label troubleshooting or question.

For general questions or comments that do not belong into the public in the form of issues, you can contact the developers at bat@mpp.mpg.de.

Chapter 2

Installation instructions

This document provides a short description of how to compile and use BAT on your computer.

Platforms

BAT has been developed on Linux. The installation, unit tests, and examples are run and known to work on Linux and Mac OS X. On Linux, we test with gcc and on Mac OS X we use clang but both compilers should work on either platform.

Windows is not supported.

Dependencies

It is understood that all commands shown below are to be entered into a terminal.

Required: Basic tools

BAT itself uses only C++03 features. Compilation and tests work fine with gcc >= 4.3 and clang >= 3.3. But recent versions of ROOT (see below) may require a C++11 compliant compiler.

Under Debian or Ubuntu, you can install the essential requirements with

```
sudo apt-get install build-essential curl
```

In order to use the development version of BAT instead of an official release, some more packages are needed

```
sudo apt-get install autoconf automake git-core libtool
```

Building and installing works with autoconf >= 2.63 and automake >= 1.10. To run the tests, a more recent automake version is needed, v1.15 is known to be sufficient.

Required: ROOT

ROOT is an object-oriented data-analysis framework. At http://root.cern.ch/, you can obtain the source code as well as binary distributions for a number of Linux distributions and Mac OS X versions. We advise to download the latest production release of ROOT. BAT is compatible with ROOT >=5.34.19 and ROOT 6. We regularly run unit tests with ROOT 5 and ROOT 6 to ensure backward compatibility.

On Linux, an alternative is to check your package manager for the availability of ROOT packages. Usually these packages are rather old but often they are good enough to build BAT. For example on Ubuntu systems up to 16.04, you can conveniently install the requirements with

```
sudo apt-get install libroot-graf2d-postscript-dev libroot-graf3d-g3d-dev\
                     libroot-math-foam-dev libroot-math-minuit-dev\
                     libroot-math-physics-dev libroot-math-mathmore-dev\
                     libroot-roofit-dev root-system-bin
```

Note

For the interface to RooFit/RooStats, ROOT must be compiled with support for RooFit and MathMore, the latter relies on the GNU scientific library (GSL).

Optional: Cuba

Cuba is a library containing general-purpose multidimensional integration algorithms. It can be obtained from http://www.feynarts.de/cuba/. BAT is compatible with Cuba versions 3.3 through at least 4.2.

Cuba is not necessary to run BAT. We recommend it for model comparison where expensive integrals are needed. Cuba provides integration routines tuned for performance, which are useful for integration in problems with not too many dimensions (~10). By default, Cuba will evaluate in parallel and take all idle cores; the number of cores can be set through an environment variable. For a single core, set

```
CUBACORES=1
```

The recommended way to get Cuba is to configure BAT with the option

```
--with-cuba=download
```

This will download a compatible version of Cuba to the local subdirectory `external/cuba-VERSION`, compile it, and configure BAT to use it.

If you want to compile Cuba manually, make sure it is built with position-independent code:

```
./configure CFLAGS='-fPIC -O3 -fomit-frame-pointer -ffast-math -Wall'
make
make install
```

Building

Obtaining BAT

You can download the latest release of BAT from http://mpp.mpg.de/bat/. Open a terminal, unpack the tarball usually named like BAT-x.x.tar.gz (here x.x is the version number) and switch to the directory

```
tar -xzf BAT-x.x.tar.gz
cd BAT-x.x
```

Alternatively, you can clone the git repository https://github.com/bat/bat (we recommend using the master branch):

```
git clone https://github.com/bat/bat
cd bat
./autogen.sh
```

Now start the configuration with

```
./configure
```

This will check your system for all components needed to compile BAT and set up the paths for installation. You can add the option `--prefix=/path/to/install/bat` to `./configure`. The BAT library files will then be installed to `$prefix/lib` and the include files to `$prefix/include`. The default installation prefix is `/usr/local`, which requires super-user privileges.

You can list all available options using

```
./configure --help
```

In the following, we describe the most useful options in detail.

ROOT

The configure script checks for ROOT availability in the system and fails if ROOT is not installed. You can specify the ROOTSYS directory using `--with-rootsys=/path/to/rootsys`

BAT support for RooFit/RooStats is turned off by default. The feature can be turned on explicitly with `--enable-roostats`. The configure script will check whether the version of ROOT is sufficient and whether ROOT was compiled with RooFit/RooStats support.

openMP

Support for openMP threading to run multiple Markov chains in parallel is available through the configure option `--enable-parallel`; it is disabled by default. This requires a version of gcc accepting the `-fopenmp` flag, anything >= 4.2 should suffice. Note that if threads are enabled, the default number of threads actually used is implementation dependent and may also depend on the current load of the CPU. Manual control over the number of threads is achieved entirely by openMP means such as setting the environment variable OMP_↩ NUM_THREADS before running an executable.

The default version of clang does not implement openMP.

Cuba

If you configured BAT with the option `--with-cuba=download`, BAT will download, compile, and use Cuba automatically. For manual configuration, use the configure option `--with-cuba[=DIR]` to enable Cuba. If you installed Cuba including the `partview` executable, the Cuba installation path will be derived from its location. Otherwise, the configure script will search for `libcuba.a` and `cuba.h` in the system paths. If you manually specify the Cuba install path as `DIR`, configure will look in `DIR/lib/` and `DIR/include/` instead. For more fine-grained control, use `--with-cuba-include-dir=/path/to/cuba/header` and `--with-cuba-lib-dir=/path/to/cuba/lib`.

Advanced options

If you want to be able to step through BAT line by line with a debugger, use `--enable-debug`. This slows down execution as it turns off code optimization but it improves the compilation time. Another way to speed up the build is to create only shared libraries if you don't need static libraries: `--disable-static`. Finally, you can reduce the output to the terminal with `--enable-silent-rules`.

Compile

After a successful configuration, run

```
make
make install
```

to compile and install BAT. Note that depending on the setting of the installation prefix you might need super-user privileges to be able to install BAT and run `sudo make install` instead of plain `make install`. In the former case, you might need to run `sudo ldconfig` just once to help the loader pick up the new libraries immediately.

System setup

After installation, BAT offers two mechanisms to make BAT available:

1. The script `bat-config` returns details of the BAT installation directories and compilation settings; see `bat-config`.
2. The file `bat.pc` contains the same information as above and can be used by the more powerful `pkg-config`; e.g.,

   ```
   pkg-config --modversion bat
   pkg-config --libs bat
   ```

If you do not install BAT to the system directories, you need to manually add the path to `bat-config`, `bat.pc`, the libraries, and the include files to the search paths. Depending on your shell, the set of commands on linux for bash-compatible shells is

```
BATPREFIX="/bat/install/prefix"
export PATH="$BATPREFIX/bin:$PATH"
export LD_LIBRARY_PATH="$BATPREFIX/lib:$LD_LIBRARY_PATH"
export CPATH="$BATPREFIX/include:$CPATH"
export PKG_CONFIG_PATH="$BATPREFIX/lib/pkgconfig:$PKG_CONFIG_PATH"
```

and for csh-compatible shells is

```
set BATPREFIX = /bat/install/prefix
setenv PATH              "${BATPREFIX}/bin:${PATH}"
setenv LD_LIBRARY_PATH   "${BATPREFIX}/lib:${LD_LIBRARY_PATH}"
setenv CPATH             "${BATPREFIX}/include:${CPATH}"
setenv PKG_CONFIG_PATH   "${BATPREFIX}/lib/pkgconfig:${PKG_CONFIG_PATH}"
```

If you want to make BAT permanently available, add the above commands to your login script, for example to `.profile` or to `.bashrc`.

On Mac OS X you do not have to set up LD_LIBRARY_PATH because we use the `rpath` option to make BAT compatible with the SIP feature enabled by default on Mac OS X starting with El Capitan.

Updating $CPATH is required if you work with interactive ROOT macros that use BAT (both for ROOT 5 and 6).

The minimal setup does not require setting PKG_CONFIG_PATH to run BAT unless you want to integrate BAT into another probject using `pkg-config`. BAT itself does not use `pkg-config`.

Including BAT in your project

The most basic way to compile and link a file `example.cxx` with BAT is

```
gcc ‘bat-config --cflags‘ ‘bat-config --libs‘ example.cxx -o
```

In a makefile, simply query `bat-config` to set appropriate variables. However, there will be an error at runtime, for example in interactive ROOT macros, if

```
libBAT.so, libBATmodels.so, libBATmtf.so,
libBAT.rootmap, libBATmodels.rootmap, libBATmtf.rootmap
```

are not in the directories searched by the library loader; see above how to setup the LD_LIBRARY_PATH and the CPATH.

Interactive ROOT macros

Due to problems in ROOT 6.02.00, it is important to create an instance of a BAT class before calling any free function defined in the BAT libraries. Else `cling` will emit confusing `error messages`. For example, the right order would be

```
int main() {
    BCLog::OpenLog("log.txt");
    BCAux::SetStyle();
    ...
}
```

instead of the other way around around because `OpenLog` creates a singleton object.

Contact

Please consult the BAT web page `http://mpp.mpg.de/bat/` for further information. In case of questions or problems, please don't hesitate to create an issue at `https://github.com/bat/bat/issues/` or contact the authors directly via email through `bat@mpp.mpg.de`.

Chapter 3

Getting started

To demonstrate the basic usage of BAT, we will build an example analysis step by step. Step one, naturally, is to install BAT—please refer to the installation chapter in this manual or online. To check if you have BAT installed and accessible, run the following command

```
bat-config
```

in your terminal. It should output a usage statement. This program outputs information about your BAT installation:

bat-config Flag	**Returned Information**
`--prefix`	Path to BAT installation
`--version`	BAT version identifier
`--libs`	Linker flags for compiling code using BAT
`--cflags`	C++ compiler flags for compiling code using BAT
`--bindir`	Path to BAT binaries directory
`--incdir`	Path to BAT include directory
`--libdir`	Path to BAT libraries directory

BAT has a second executable, which creates for you the files necessary to start a basic analysis. We will use this executable to initialize our tutorial project:

```
bat-project MyTut MyMod
```

This will create a directory called `MyTut` that contains

`Makefile`	a makefile to compile our tutorial project
`runMyTut.cxx`	the C++ source to an executable to run our tutorial project
`MyMod.h`	the C++ header for our tutorial model `MyMod`
`MyMod.cxx`	the C++ source for our tutorial model `MyMod`

If BAT is installed correctly, you can compile and run this project already:

```
cd MyTut
make
./runMyTut
```

BAT will issue a series of errors telling you your model has no parameters. Because of course we haven't actually put anything into our model yet. Let's do that.

3.1 Defining a model

To define a valid BAT model we must make three additions to the empty model that `bat-project` has created:

1. we must add parameters to our model;
2. we must implement a log-likelihood function;
3. and we must implement or state our priors.

We will start with a simple model that fits a normal distribution to data, and so has three parameters: the mode (μ) and standard deviation (σ) of our distribution, and a scaling factor ("height"). We will start with flat priors for all.

How you store and access your data is entirely up to you. For this example, we are going to fit to a binned data set that we store as a ROOT histogram in a private member of our model class. In the header, add to the class

```
#include <TH1D.h>

...

class MyMod : public BCModel
{

    ...

private:
    TH1D fDataHistogram

};
```

And in the source file, we initialize our `fDataHistogram` in the constructor; let's also fill it with some random data, which ROOT can do for us using a TF1:

```
#include "MyMod.h"
#include <TF1.h>

...

// ---------------------------------------------------------
MyMod::MyMod(const std::string& name)
    : BCModel(name),
      fDataHistogram("data", ";mass [GeV];count", 100, 5.0, 5.6)
{
    // create function to fill data according to
    TF1 data_func("data_func", "exp(-0.5*((x - 5.27926) / 0.04)^2)", 5.0, 5.6);

    // fill data histogram randomly from data_func 1,000 times
    fDataHistogram.FillRandom("data_func", 1e3);
}
```

3.1.1 Adding parameters

To add a parameter to a model, call its member function **BCModel::AddParameter**:

```
bool AddParameter(const std::string& name, double min, double max, const std::string& latexname = "", const
       std::string& unitstring = "")
```

You must indicate a parameter's name and its allowed range, via `min` and `max`. Each parameter must be added with a unique name. BAT will also create a "safe version" of the name which removes all non alpha-numeric characters except for the underscore, which is needed for naming of internal storage objects. Each parameter name should also convert to a unique safe name; BAT will complain if this is not the case (and `AddParameter` will return `false`).

Optionally, you may add a display name for the parameter and a unit string, both of which are used when BAT creates plots.

The most logical place to add parameters to a model is in its constructor. Let us now edit `MyMod.cxx` to add our mode, standard deviation, and height parameters inside the constructor:

```
MyMod::MyMod(const std::string& name)
    : BCModel(name),
      fDataHistogram("data", ";mass [GeV];count", 100, 5.0, 5.6)
{
    ...

    AddParameter("mu",    5.27, 5.29, "#mu", "[GeV]");
    AddParameter("sigma", 25e-3, 45e-3, "#sigma", "[GeV]");
    AddParameter("height", 0, 10, "", "[events]");
}
```

I have chosen the ranges because I have prior information about the data: the mode of my distribution will be between 5.27 and 5.29; the standard deviation will be between 0.025 and 0.045; and the height will be between 0 and 10. When we provide a prior for a parameter, $p_0(\lambda)$, the prior BAT uses is

$$P_0(\lambda) = \begin{cases} p_0(\lambda), & \text{if } \lambda \in [\lambda_{\text{min}}, \lambda_{\text{max}}], \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

So keep mind that the ranges your provide for parameters become part of the prior: BAT will not explore parameter space outside of the range limits you provide.

If you have written the code correctly for adding parameters to your model, your code should compile. BAT will again, though, issue an error if you try to run `runMyTut`, since we are still missing priors.

3.1.2 Setting prior distributions

There are two ways we may set the prior, $P_0(\vec{\lambda})$ for a parameter point: We can override the function that returns the log *a priori* probability for a model (**BCModel::LogAPrioriProbability**) and code anything we can dream of in C++:

```
double MyMod::LogAPrioriProbability(const std::vector<double>& pars)
{
   ...
}
```

In this case, you have to make sure the prior is properly normalized if you wish to compare different models as BAT will not do this for you.

Or we can set individual priors for each parameter, and BAT will multiply them together for us including the proper normalization. If each parameter has a prior that factorizes from all the others, then this is the much better option. In this case, we **must not** override `LogAPrioriProbability`. Instead we tell BAT what the factorized prior is for each parameter, by adding a **BCPrior** object to each parameter after we create it:

```
#include <BAT/BCGaussianPrior.h>

...

MyMod::MyMod(const std::string& name)
    : BCModel(name),
      fDataHistogram("data", ";mass [GeV];count", 100, 5.0, 5.6)
{
    ...

    // add parameters for Gaussian distribution
    AddParameter("mu",    5.27, 5.29, "#mu", "[GeV]");
    GetParameters().Back().SetPrior(new BCGaussianPrior(5.28, 2e-3));

    AddParameter("sigma", 25e-3, 45e-3, "#sigma", "[GeV]");
    GetParameters().Back().SetPrior(new BCGaussianPrior(35e-3, 3e-3));

    AddParameter("height", 0, 10, "", "[events]");
    GetParameters().Back().SetPriorConstant();
}
```

We could create a **BCConstantPrior** object for `height` just as we created a **BCGaussianPrior** object for `mu` and `sigma`, but BAT has a convenient function that creates one for us.

We can now compile our code and run it. The results will be meaningless, though, since we have yet to define our likelihood.

3.1.3 Defining a likelihood

The heart of our model is the likelihood function—more specifically, since BAT works with the natural logarithm of functions, our log-likelihood function.

Given a histogrammed data set containing numbers of events in each bin, our statistical model is the product of Poisson probabilities for each bin—the probability of the events observed in the bin given the expectation from our model function:

$$\mathcal{L}(\vec{\lambda}) \equiv \prod_i \text{Poisson}(n_i|\nu_i), \tag{3.2}$$

where n_i is the number of events in bin i and ν_i is the expected number of events given our model, which we will take as the value of our model function at the center of the bin, m_i,

$$\nu_i \equiv f(m_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{\left(-\frac{(m_i-\mu)^2}{2\sigma^2}\right)\right\}. \tag{3.3}$$

Working with the log-likelihood, this transforms into a sum:

$$\log \mathcal{L}(\vec{\lambda}) \equiv \sum_i \log \text{Poisson}(n_i|\nu_i). \tag{3.4}$$

BAT conveniently has a function to calculate the logarithm of the Poisson distribution (with observed x and expected λ) for you:

```
double BCMath::LogPoisson(double x, double lambda)
```

Let us code this into our log-likelihood function:

```
#include <BAT/BCMath.h>

#include <TMath.h>

...

// ---------------------------------------------------------
double MyMod::LogLikelihood(const std::vector<double>& pars)
{
    // store our log-likelihood as we loop through bins
    double LL = 0.;

    // loop over bins of our data
    for (int i = 1; i <= fDataHistogram.GetNbinsX(); ++i) {

        // retrieve observed number of events
        double x = fDataHistogram.GetBinContent(i);

        // retrieve bin center
        double m = fDataHistogram.GetBinCenter(i);

        // calculate expected number of events, using ROOT Gaus function
        double nu = TMath::Gaus(m, pars[0], pars[1], true);

        // add to log-likelihood sum
        LL += BCMath::LogPoisson(x, nu);

    }

    // return log-likelihood
    return LL;
}
```

3.2 Looking at the output

Our model class is now ready to go. Let's just make one edit to the `runMyTut.cxx` file to change our model name from the default "name_me" to something sensible:

```
// create new MyMod object
MyMod m("gaus_mod");
```

We can now compile and run our project:

```
1 make
2 ./runMyTut
```

This will sample from the posterior probability distribution and marginalize the results, saving plots to `gaus_mod_plots.pdf`. In that file you should see the 1D and 2D marginalizations of our three parameters:

\

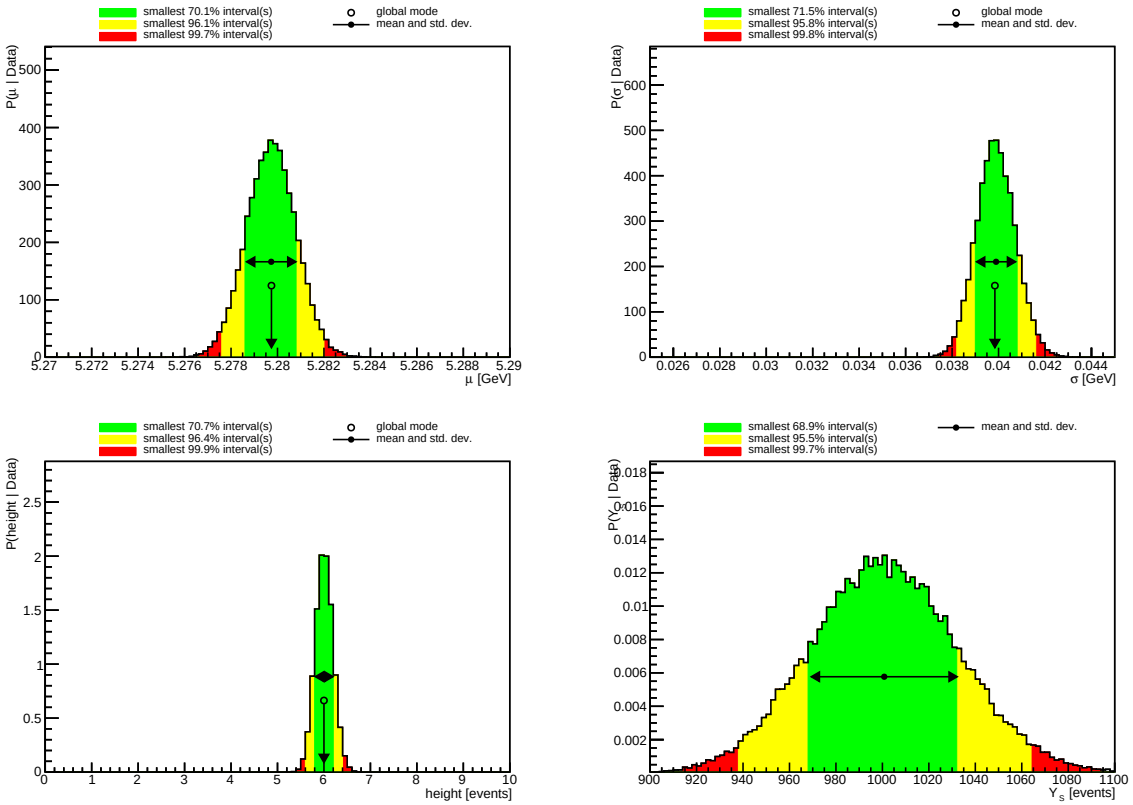


Figure 3.1 1D Posteriors of Gaussian distribution model.

\

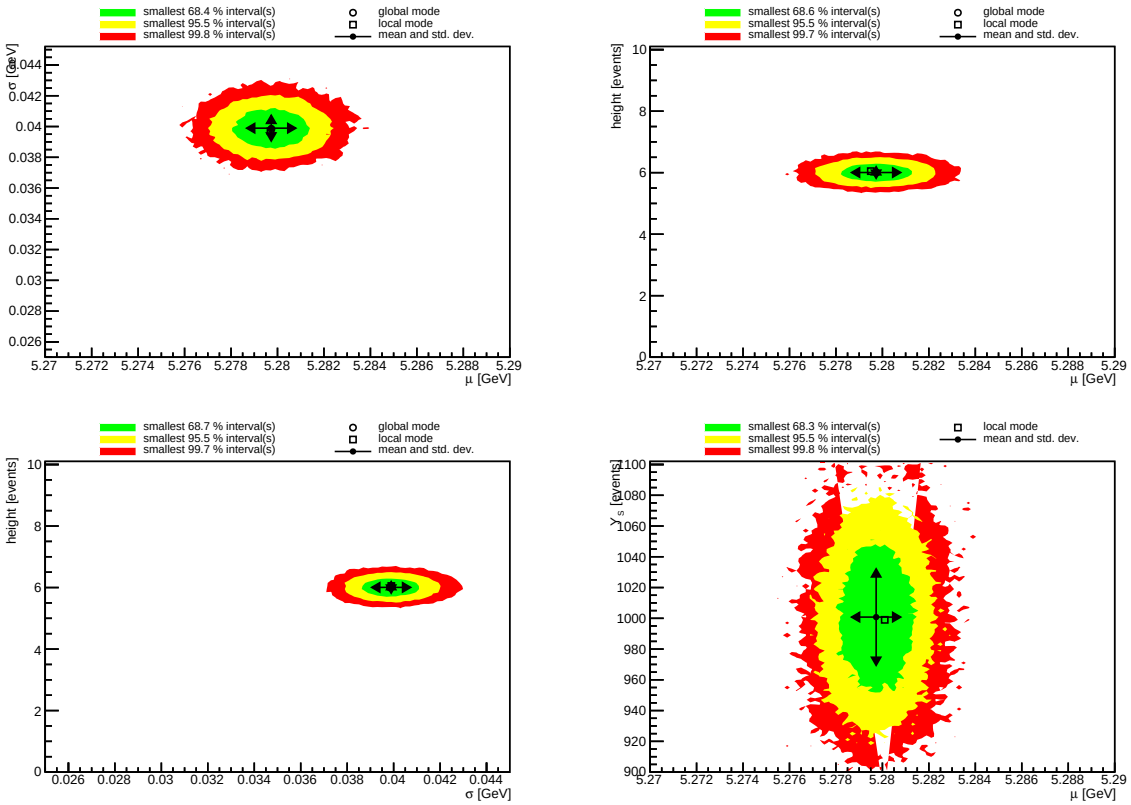


Figure 3.2 2D Posteriors of Gaussian distribution model.

In each plot, we see the global mode and the marginalized mean and standard deviation of the posterior distribution; and three credibility intervals.

BAT has also printed a summary of the results to the log file and command line. This includes the global mode

```
Summary :  Global mode:
Summary :  0)  Parameter "mu"          : 5.279741 +- 0.001072076
Summary :  1)  Parameter "sigma"       : 0.040267 +- 0.00089553
Summary :  2)  Parameter "height"      : 6 +- 0.1897
```

and marginalized posteriors:

```
Summary :   (0) Parameter "mu" :
Summary :       Mean +- sqrt(Variance):          5.279748 +- 0.001065279
Summary :       Median +- central 68% interval: 5.279749 +  0.00105881 - -0.00106418
Summary :       (Marginalized) mode:            5.2797
Summary :        5% quantile:                   5.278005
Summary :       10% quantile:                   5.278381
Summary :       16% quantile:                   5.278685
Summary :       84% quantile:                   5.280808
Summary :       90% quantile:                   5.281112
Summary :       95% quantile:                   5.281498
Summary :       Smallest interval containing 69.8% and local mode:
Summary :       (5.2786, 5.2808) (local mode at 5.2797 with rel. height 1; rel. area 1)
```

3.3 Adding an observable

We can store the posterior distribution of any function of the parameters of our model using BAT's **BC↩ Observable**'s. Let us suppose we want to know the posterior distribution for the total number of events our model predicts—the signal yield; and we want to know the standard deviation's relation to the mean: σ/μ. In our constructor, we add the observables in the same way we added our parameters:

```
// ---------------------------------------------------------
MyMod::MyMod(const std::string& name)
    : BCModel(name),
      fDataHistogram("data", "mass [GeV];count", 100, 5.0, 5.6)
{
    ...

    AddObservable("SignalYield", 900, 1100, "Y_{S}", "[events]");
    AddObservable("Resolution",
                  100. * GetParameter("sigma").GetLowerLimit() / GetParameter("mu").GetUpperLimit(),
                  100. * GetParameter("sigma").GetUpperLimit() / GetParameter("mu").GetLowerLimit(),
                  "#sigma / #mu", "[%]");
}
```

Note that an observable does not need a prior since it is not a parameter; but it does need a range for setting the marginalized histogram's limits. Since we know already that the answer will be 1000 events with an uncertainty of $\sqrt{1000}$, we set the range for the yield to $[900, 1100]$.

And we need to calculate this observable in our `CalculateObservables(...)` member function. Uncomment it in the header file `MyMod.h`:

```
void CalculateObservables(const std::vector<double> & pars);
```

and implement it in the source:

```
// ---------------------------------------------------------
void MyMod::CalculateObservables(const std::vector<double>& pars)
{
    // store total of number events expected
    double nu = 0;

    // loop over bins of our data
    for (int i = 1; i <= fDataHistogram.GetNbinsX(); ++i)
        // calculate expected number of events in that bin
        // and add to total expectation
        nu += pars[2] * TMath::Gaus(fDataHistogram.GetBinCenter(i), pars[0], pars[1], true);

    // store in the observable
    GetObservable(0) = nu;

    // Store sigma as percentage of mu:
    GetObservable(1) = 100. * pars[1] / pars[0];
}
```

Compile and run `runMyTut.cxx` and you will see new marginalized distributions and text output for the observables:

```
Summary :   (3) Observable "SignalYield" :
Summary :       Mean +- sqrt(Variance):         1000.9 +- 31.53
Summary :       Median +- central 68% interval: 1000.6 +  31.937 - -31.194
Summary :       (Marginalized) mode:            1003
Summary :        5% quantile:                   949.52
Summary :       10% quantile:                   960.52
Summary :       16% quantile:                   969.44
Summary :       84% quantile:                   1032.6
Summary :       90% quantile:                   1041.9
Summary :       95% quantile:                   1053.5
Summary :       Smallest intervals containing 68.7% and local modes:
Summary :       (970, 1032) (local mode at 1003 with rel. height 1; rel. area 0.97769)
Summary :     (1032, 1034) (local mode at 1033 with rel. height 0.57407; rel. area 0.022311)
```

As we expected, the mean of the total yield posterior is just the number of events in our data set and the standard deviation is its square root.

3.4 Further Output

BAT has a few more output options than the two mentioned above. The code for turning them on is already included in the `runMyTut.cxx` generated by `bat-project`. You can write the Markov-chain samples to a ROOT `TTree` for further postprocessing by uncommenting the following line:

```
m.WriteMarkovChain(m.GetSafeName() + "_mcmc.root", "RECREATE");
```

You can also modify the plotting output to include more plots per page. Without specifying, we used the default of one plot per page. Let's instead plot 4 plots (in a 2-by-2 grid):

```
m.PrintAllMarginalized(m.GetSafeName() + "_plots.pdf", 2, 2);
```

There are four more graphical outputs from BAT. Turn them on by uncommenting the following lines

```
m.PrintParameterPlot(m.GetSafeName() + "_parameters.pdf");
m.PrintCorrelationPlot(m.GetSafeName() + "_correlation.pdf");
m.PrintCorrelationMatrix(m.GetSafeName() + "_correlationMatrix.pdf");
m.PrintKnowledgeUpdatePlots(m.GetSafeName() + "_update.pdf", 3, 2);
```

(We have edited the last line to specify 3-by-2 printing.)

The parameter plot graphically summarizes the output for all parameters (and in a separate page, all observables) in a single image:

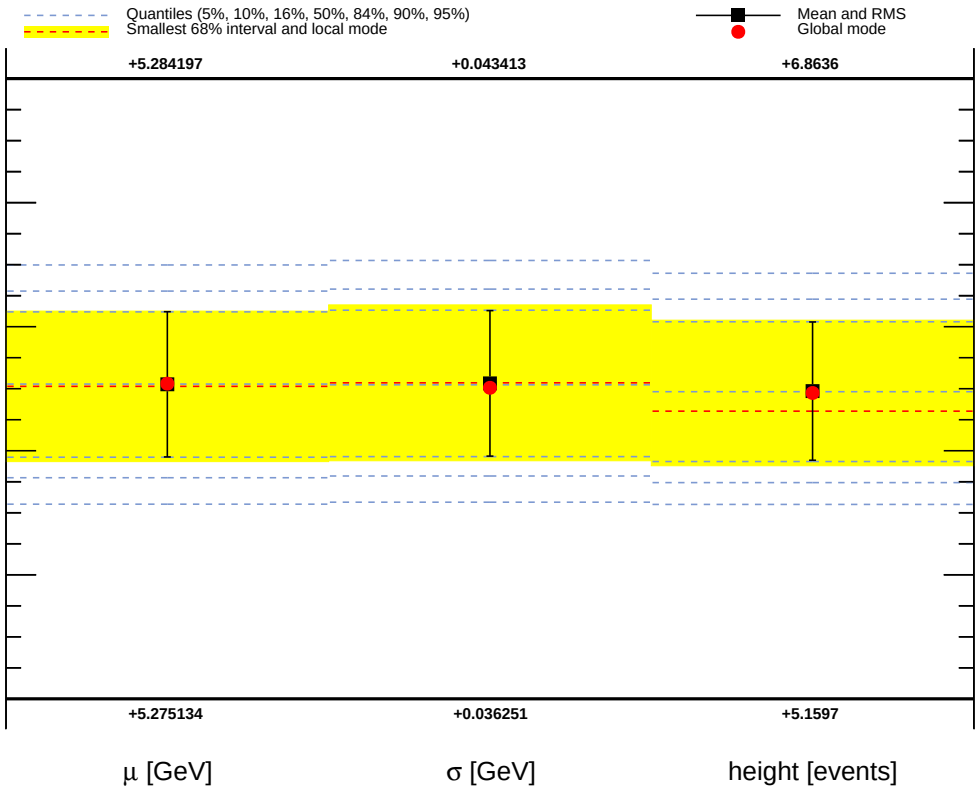


Figure 3.3 Summary of parameter marginalizations.

The correlation plot and the correlation matrix summarize graphically the correlations among parameters and observables:

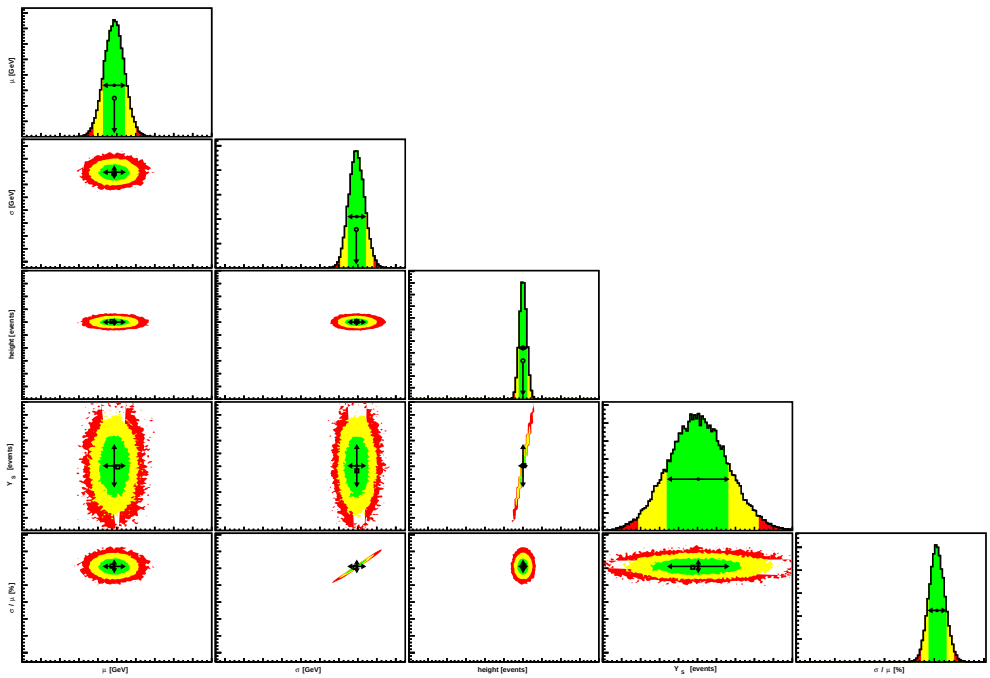


Figure 3.4 Parameter and observable correlation plots.

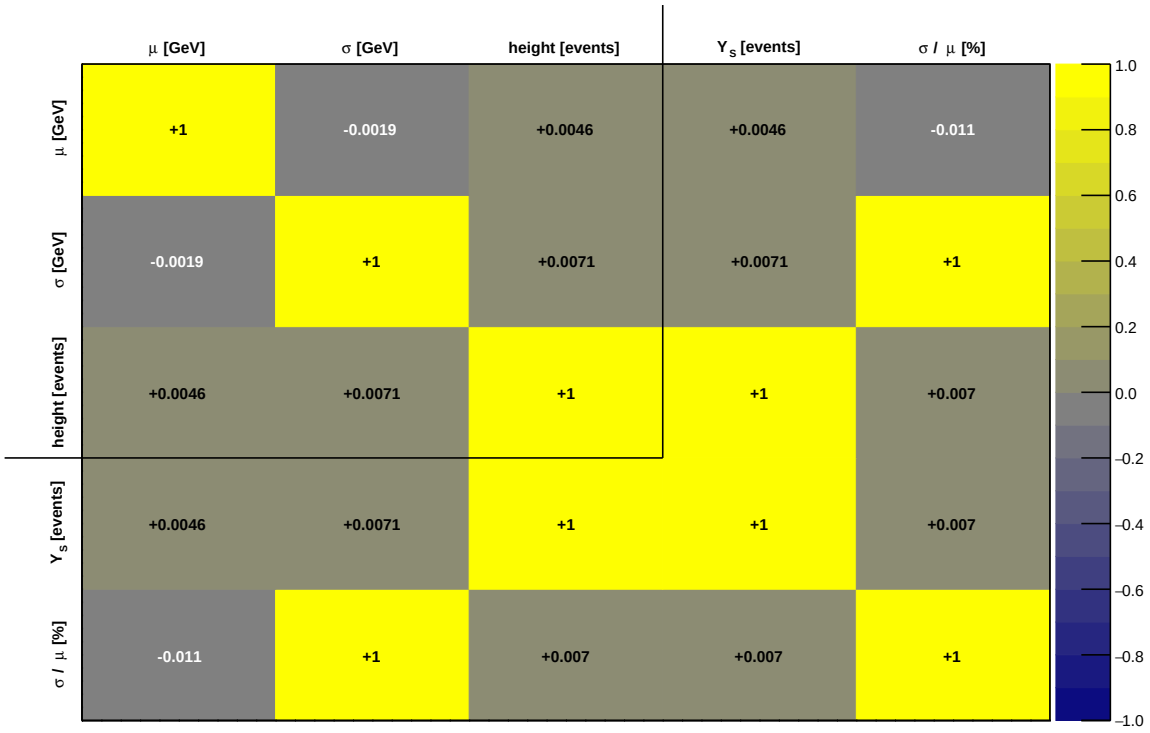


Figure 3.5 Parameter and observable correlation matrix.

The knowledge update plots show the marginalized priors and marginalized posteriors together in one plot for each variable (and also for 2D marginalizations):

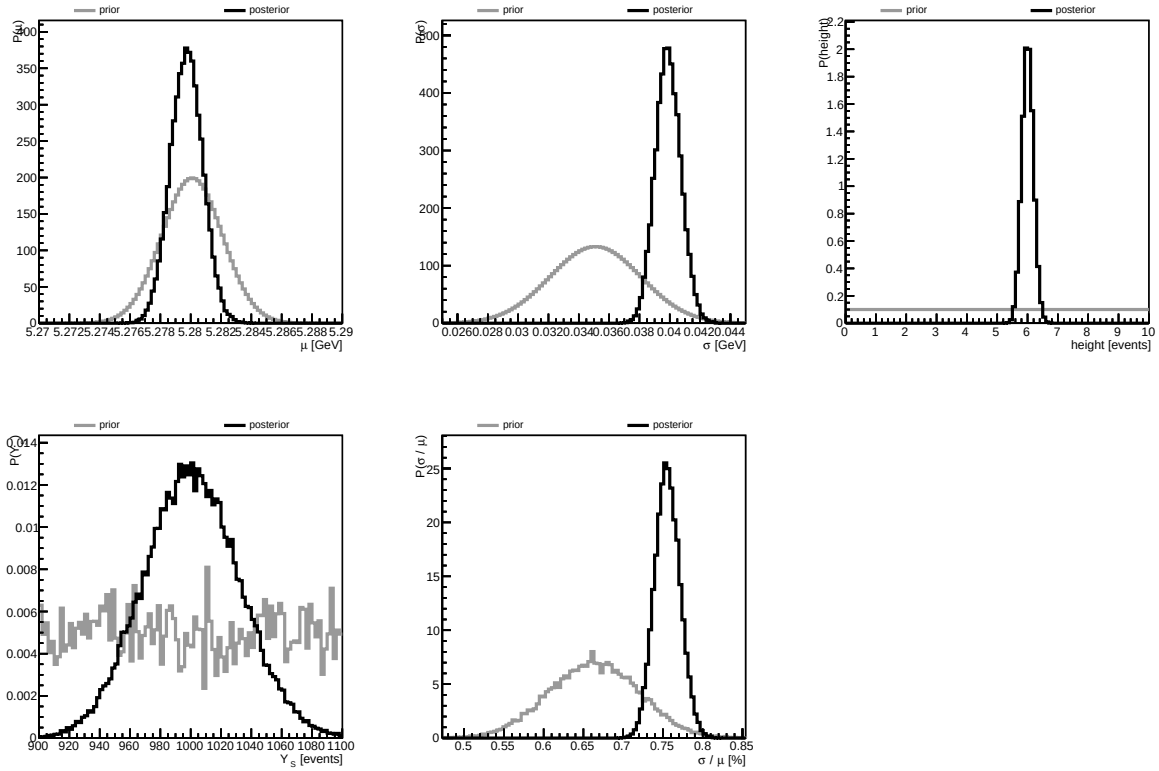


Figure 3.6 Knowledge update plots.

Note that this really shows the *marginalized* prior: Here, having used factorized priors, we see exactly the factorized priors. But had we used a multivariate prior, then we'd see what this looks like for each parameter. We also see what the prior of our observables look like given the priors of the variables they are functions of. These are very useful things to know.

Chapter 4

Bayesian Statistics

In this chapter, we give a concise and necessarily incomplete overview of Bayesian statistics. Our intention is to cover the bare minimum and introduce those quantities that appear in BAT. As a toolkit, BAT allows the user complete freedom in modeling. There is vast literature on the theory, applications, and modeling; for example, consider [8] [7] [14] [10] [3] [9] [5] for a comprehensive overview.

4.1 Basic terminology

Bayesian statistics is a framework for quantitative inductive or plausible reasoning; i.e., the optimal processing of incomplete information. The basic idea is to associate a degree of belief to logical propositions; e.g., the mass of the Higgs boson is 125 GeV. From fundamental axioms about reasoning, one can then show that the calculus of degree of belief is simply the ordinary calculus of probability theory; see [8] for a thorough discussion. Deductive reasoning is included as the limiting case in which the degree of belief is either 0 or 1.

From now on, we will use the symbol $P(A)$ to denote both the *degree of belief* in proposition A and the *probability* of A. In our applications below, A is often one value out of a continuum, so we use $P(A)$ also to denote the *probability density* of A. The *conditional probability* of A given B is $P(A \mid B)$.

The two central tasks of the natural sciences are to learn about nature from data and to make predictions for (future) experiments. A *model* M is a proxy for all discrete pieces of information relevant to calculating the degree of belief. The model can contain *parameters* $\boldsymbol{\theta}$ that take on values in a continuum, perhaps subject to constraints as for example $\theta_1 \geq 0$. Bayesian reasoning provides an update rule to adjust the degree of belief based on new information available in the form of *observed data* D. This update rule is the celebrated *Bayes' theorem*

$$\boxed{P(\boldsymbol{\theta} \mid D, M) \propto P(D|\boldsymbol{\theta}, M) P(\boldsymbol{\theta} \mid M)} \tag{4.1}$$

$P(\boldsymbol{\theta} \mid M)$ is the *prior density*, $P(D|\boldsymbol{\theta}, M)$ is called the *probability of the data* when treated as a function of D, and known as the *likelihood* when considering the dependence on $\boldsymbol{\theta}$, for fixed D. The model-dependent normalization constant is known as the *evidence* or *marginal likelihood*:

$$Z = \int \mathrm{d}\boldsymbol{\theta}\, P(D|\boldsymbol{\theta}, M) P(\boldsymbol{\theta} \mid M). \tag{4.2}$$

Finally, the left-hand side $P(\boldsymbol{\theta}|D, M)$ is the *posterior density*. Prior and posterior (*density* is usually omitted) represent the state of knowledge about the parameter $\boldsymbol{\theta}$ before and after seeing the data. Note that $\boldsymbol{\theta}$ appears on opposite sides of | in $P(D|\boldsymbol{\theta}, M)$ and $P(\boldsymbol{\theta}|D, M)$. That's why Bayes' theorem is also known as the theorem of *inverse probability*.

4.1.1 Marginalization

Suppose there are two parameters, $\boldsymbol{\theta} = (\theta_1, \theta_2)$, and θ_1 is the *parameter of interest* whereas θ_2 is a *nuisance parameter*. In Bayes' theorem, there is no fundamental distinction between parameters of interest and nuisance parameter, they are all just parameters. But often the goal of the analysis is to extract the posterior of θ_1 while θ_2 is only needed at an intermediate stage; for example in order to correctly model the measurement process of D. From the joint posterior $P(\theta_1, \theta_2|D)$, we compute the *marginalized* posterior and can remove the dependence on θ_2 by integration

$$P(\theta_1|D) = \int \mathrm{d}\theta_2 \, P(\theta_1, \theta_2|D). \tag{4.3}$$

4.1.2 Model comparison

If there is only a single model under consideration, and no potential for cconfusion, the model label M is implied and usually omitted from the equations. But suppose that there are two competing models, M_1, M_2, with parameters $\theta_{1,2}$, that quantitatively predict the outcome D of an experiment. The task is to find the model with the higher degree of belief. Using Bayes' theorem, the *posterior odds* of the models are easily found as

$$\frac{P(M_1|D)}{P(M_2|D)} = B_{12} \cdot \frac{P(M_1)}{P(M_2)}, \tag{4.4}$$

where the *Bayes factor* of M_1 versus M_2, B_{12}, is just the ratio of the evidences

$$B_{12} = \frac{P(D|M_1)}{P(D|M_2)} = \frac{Z_1}{Z_2} = \frac{\int \mathrm{d}\boldsymbol{\theta}_1 \, P(D|\boldsymbol{\theta}_1, M_1) P(\boldsymbol{\theta}_1, M_1)}{\int \mathrm{d}\boldsymbol{\theta}_2 \, P(D|\boldsymbol{\theta}_2, M_2) P(\boldsymbol{\theta}_2, M_2)} \tag{4.5}$$

The *prior odds* $P(M_1)/P(M_2)$ represent the relative degree of belief in the models, independent of the data. % The data are accounted for in the Bayes factor. The Bayes factor quantifies the relative shift of degree of belief induced by the data. In general, $\dim \boldsymbol{\theta}_1 \neq \dim \boldsymbol{\theta}_2$, and without loss of generality let $\dim \boldsymbol{\theta}_1 < \dim \boldsymbol{\theta}_2$. The Bayes factor automatically penalizes M_2 for its larger complexity, as the prior mass is spread out over a higher-dimensional volume. However, this can be compensated if the likelihood $P(D|\boldsymbol{\theta}_2, M_2)$ is significantly higher in regions of reasonably high prior density; i.e. the Bayes factor implements Occam's razor the simplest model that describes the observations is preferred.

4.2 Goodness of fit

In the Bayesian approach, there is, however, no straightforward answer to the following question: if there is only one model at hand, how to decide if that model is sufficient to explain the data, or if the search for a better model needs to continue? The standard procedure to tackle this problem of evaluating the *goodness of fit* is to define a test statistic $T = T(D)$ and to evaluate the following tail-area probability, that is the p value

$$p \equiv \int_{T>T_{obs}} \mathrm{d}T \, P(T|M). \tag{4.6}$$

Care has to be taken in the usage, computation, and interpretation of p values. An introduction a Bayesian interpretation of p values with applications in BAT is available at [1] ; see also the references therein.

4.3 Representation in BAT

In BAT, a model M is represented as a C++ subclass of **BCModel**. The crucial parts are to define the likelihood $P(D \mid \theta, M)$ **BCModel::LogLikelihood**, the prior $P(\theta \mid M)$ **BCModel::LogAPrioriProbability**, and the parameters θ. From these quantities, BAT can compute the unnormalized posterior $P(\theta \mid D, M)$ **BCModel::↩ LogProbabilityNN**. To avoid numerical overflow, BAT operates on the log scale whenever possible. The key methods of **BCModel** that a user has to implement are

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

The parameter values are passed in simply as numbers to likelihood and prior, all parameters are assumed to be real and continuous. Discrete parameters are not supported. The support of θ is a hyperrectangle whose bounds are given by the bounds of the individual parameters when added to the model with **BCModel::AddParameter**

```
bool BCModel::AddParameter(const std::string& name, double min, double max,
                           const std::string& latexname = "",
                           const std::string& unitstring = "")
```

The optional `latexname` and `unitstring` are used only for labeling plot axes; it's intended usage is to pretty up plots. For example, a parameter `theta` representing a time measured in seconds is defined as

```
AddParameter("theta", 0, 1, "#theta", "[s]");
```

and whenever `theta` appears on the axis of a plot, it will appear as θ [s]. Note that the plots are created with ROOT, so the `latexname` has to be in ROOT syntax which is basically *LaTeX* syntax but the backslash \ is replaced by the hash #.

Chapter 5

Integration

5.1 Motivation

The reason that BAT exists is that nearly any Bayesian analysis these days is too complicated to be handled analytically. To address typical questions like

- What is known about a single parameter taking into account the uncertainty on all other parameters?
- How are parameters correlated?

one needs to be able to compute and visualize 1D and 2D *marginal distributions*; cf. Marginalization. These are defined as integrals over the posterior; for example the 2D marginal distribution is

$$P(\theta_1, \theta_2 \mid D) = \int \prod_{i \neq 1,2} \mathrm{d}\theta_i \, P(\boldsymbol{\theta} \mid D). \tag{5.1}$$

To do model comparison, one has to compute the evidence, that is the integral over all parameters

$$Z = \int \mathrm{d}\boldsymbol{\theta} \, P(D|\boldsymbol{\theta}, M) P(\boldsymbol{\theta} \mid M). \tag{5.2}$$

Therefore Bayesian inference in practice requires good integration techniques. These methods are bundled in the class **BCIntegrate** with the exception of the Markov chain code in **BCEngineMCMC**; see also the Structure of the Code.

For low-dimensional problems, deterministic integration methods are usually the fastest and most robust but for higher dimensions, Monte Carlo techniques are the most efficient tools known. Depending on the setting, BAT defaults to deterministic methods for $d \leq 2, 3$ dimensions and uses Monte Carlo for $d > 3$.

5.2 Marginalization

The main use case for BAT is to estimate and visualize the marginal distributions of the posterior. Given a model `m`, all marginal distributions are estimated as histograms by

```
m.MarginalizeAll();
```

The individual distributions can be accessed using `unsigned` or `std::string` as keys

```
// one-dimensional
BCH1 d0 = m.GetMarginalized(0);
BCH1D d1 = m.GetMarginalized("name");

// two-dimensional
BCH2D d2 = m.GetMarginalized(3, 4);
BCH2D d3 = m.GetMarginalized("name_of_first_parameter", "name_of_second_parameter");
```

Directly access the underlying histogram with

```
// one-dimensional
BCH1 h0 = m.GetMarginalizedHistogram(0);

// two-dimensional
BCH2D h1 = m.GetMarginalizedHistogram(3, 4);
```

In case of many (nuisance) parameters, it may be useful to constrain which histograms are stored because the number of 2D marginals grows like $n \cdot (n-1)/2$ with the number of parameter n. This can be achieved either for all 1D or 2D marginal distributions or for individual combinations. For example:

```
m.AddParameter("x", 0, 1);
m.AddParameter("y", 0, 1);
m.SetFlagFillHistograms(false); // no 1D marginals for ‘x‘, ‘y‘ stored
m.AddParameter("z", 0, 1);
m.SetFillHistogramParPar(0, 2, false); // no ‘x‘ vs ‘z‘ 2D marginal
```

See also

BCEngineMCMC::SetFillHistogramParPar, **BCEngineMCMC::SetFillHistogramParObs**, **BCEngineMCMC::SetFillHistogramObsObs**, **BCEngineMCMC::SetFlagFillHistograms**

The method for marginalizing can be selected as follows

```
m.SetMarginalizationMethod(method);
```

where `method` is an `enum` in **BCIntegrate::BCMarginalizationMethod**

method	**Details**
`kMargMetropolis`	Metropolis algorithm; see Markov chain Monte Carlo.
`kMargMonteCarlo`	Sample mean integration in each histogram bin. Least efficient method.
`kMargGrid`	Evaluate target at each bin center. Most efficient for 1D and 2D.
`kMargDefault`	Use `kMargGrid` for $d \leq 2$, else `kMargMetropolis`.

The availability of marginalization methods can be queried at runtime using **BCIntegrate::CheckMarginalizationAvailability**.

In case any pre- or postprocessing needs to happen to set up data structures, we provide the hooks `virtual void` **BCIntegrate::MarginalizePreprocess** and `virtual void` **BCIntegrate::MarginalizePostprocess**. They are empty by default and can be overloaded in a user model.

5.3 Evidence

In BAT terminology, the evidence or normalization constant of a model `m` with the default method is computed as

```
double evidence = m.Normalize();
```

Once the normalization has been determined, **BCModel::LogProbability** returns the normalized value of the posterior as opposed to the not normalized result from **BCModel::LogProbabilityNN**.

Note

Internally only **BCModel::LogProbabilityNN** is called when integrating or optimizing because it only requires the user to override **BCModel::LogLikelihood** and to set a prior.

BCModel::Normalize returns the evidence on the linear scale. Choose the method of integration explicitly with

```
double evidence = m.Integrate(method);
```

where `method` is an `enum` in **BCIntegrate::BCIntegrationMethod**.

BCIntegrationMethod	**Details**
`kIntMonteCarlo`	Sample mean. Usually least efficient.
`kIntCuba`	Use a method from cuba.
`kIntGrid`	Approximate Riemann sum over a dense grid for $d \leq 3$.
`kIntLaplace`	Laplace approxation. Only approximately valid for peaked unimodal integrands. Incorrect if distribution is heavy tailed or if the mode is near a boundary. Very fast. No uncertainty estimate. Only method implemented on the log scale.
`kIntDefault`	Use Cuba if available. Else use `kIntGrid` for $d \leq 3$ and `kIntMonteCarlo` for $d > 3$.

General termination criteria for all integration methods except `kIntLaplace` are the desired absolute precision ϵ_a and relative precision ϵ_r and the minimum and maximum number of iterations:

```
m.SetAbsolutePrecision(1e-6);
m.SetRelativePrecision(1e-8);
m.SetNIterationsMin(2000);
m.SetNIterationsMax(50000);
```

The integration terminates if

$$|Z - \hat{Z}| \leq \max(\epsilon_a, \epsilon_r Z) \tag{5.3}$$

where $\hat{Z}$ is the current estimate of the evidence.

5.3.1 Rescaling

In case the log likelihood is very small or very large, going to the linear scale may take it to exactly zero or infinity in finite precision. This often happens in practice if the log likelihood is a sum of N terms. For example, assume each factor is 0.5. Then $\exp(0.5N) = \infty$ on the computer for $N \geq 1420$ using double precision. To avoid this problem, you can rescale the log likelihood by manually subtracting the value at the mode inside `LogLikelihood`.

Another option, if requirements are satisfied, is to use the Laplace method. It is naturally implemented on the log scale. While the standard interface to all integration methods via **BCIntegrate::Normalize()** always transforms to the linear scale, calling **BCIntegrate::IntegrateLaplace()** directly returns the evidence on the log scale.

5.3.2 Cuba

The Cuba package itself has four different integration methods. Cuba is an external dependency; see the installation instructions on how to build BAT with Cuba support. If Cuba is available, select the Cuba method with

```
m.SetCubaIntegrationMethod(method);
m.Integrate();
// alternative
m.IntegrateCuba(method);
```

where `method` is an `enum` in **BCIntegrate::BCCubaMethod**

BCCubaMethod	Details
kCubaVegas	VEGAS algorithm by Lepage
kCubaSuave	Suave algorithm.
kCubaDivonne	Divonne algorithm.
kCubaCuhre	Cuhre algorithm. Essentially quadrature in higher dimensions. Suffers from curse of dimensionality but most efficient and robust in low dimensions.
kCubaDefault	For $d = 1$, use VEGAS, for $d = 2, 3$, use Cuhre, and for $d > 3$, use Divonne.

Note

Cuba evaluates the posterior in parallel by default. If the posterior is not thread-safe (see Multithreading and Thread Safety), it is recommended to set the environment variable CUBACORES to 1.

Each Cuba method comes with various parameter values to set. We have taken over default values from the example that comes with Cuba but they are by no means optimal for every problem. Please experiment and consult the Cuba manual that comes with the Cuba source code from http://www.feynarts.de/cuba/. All options are accessible in the namespace **BCCubaOptions**. An example with bogus values

```
BCCubaOptions::Suave o = m.GetCubaSuaveOptions();
o.flatness = 5;
o.nnew = 5000;
o.nmin = 15;
m.SetNIterationsMax(1e7);
m.SetCubaOptions(o);
m.IntegrateCuba(BCIntegrate::kCubaSuave);
```

5.4 Slices

To get a quick visualization of a complicated posterior, a slice, or projection, may be preferable to a full marginalization. That is, all parameters except one [two] are held fixed (instead of integrated over as in marginalization) and the remaining parameter[s] are evaluated on a regular grid. In other words, the conditional distribution

$$P(\theta_1 \mid \boldsymbol{\theta}_{\setminus 1}, D) = \frac{P(\boldsymbol{\theta} \mid D)}{P(\boldsymbol{\theta}_{\setminus 1} \mid D)}. \tag{5.4}$$

The slice is returned as one [two] dimensional histogram. This is achieved with the various variants of **BC↩ Integrate::GetSlice**.

In a posterior where all parameters are independent, the marginal and conditional distributions coincide because

$$P(\theta_1 \mid \boldsymbol{\theta}_{\setminus 1}, D) = \frac{P(\boldsymbol{\theta}_{\setminus 1} \mid D) P(\theta_1 \mid D)}{P(\boldsymbol{\theta}_{\setminus 1} \mid D)} = P(\theta_1 \mid D). \tag{5.5}$$

Note

Independence rarely holds in practice, so marginalization is still useful.

For example, in a Gaussian posterior with independent parameters, we first find the mode and use these parameter values to find the 1D conditional distribution of the first parameter. In this example, the result is just a Gaussian. Here is how to find the result in general, for non-Gaussian or non-independent posteriors:

```
#include <TH1.h>
#include <TCanvas.h>
...
int main()
{
...
    m.FindMode(m.GetBestFitParameters());

    TCanvas c;
    unsigned nIterations;
    double log_max;
    int nbins = 100;
    bool normalize = true;
    TH1* slice = m.GetSlice(0, nIterations, log_max, m.GetBestFitParameters(), nbins, normalize);
    slice->Draw("HISTSAME");
    c.Print("slice.pdf");
    delete slice;
...
}
```

Chapter 6

Markov chain Monte Carlo

6.1 Motivation

Among the integration methods introduced in Integration, the Monte Carlo method is the most powerful one in high dimensions. The term Monte Carlo is used as a synonym for the use of pseudo-random numbers. Markov chains are a particular class of Monte Carlo algorithms designed to generate correlated samples from an arbitrary distribution. The central workhorse in BAT is an adaptive Markov chain Monte Carlo (MCMC) implementation based on the Metropolis algorithm. It allows users to marginalize a posterior without requiring manual tuning of algorithm parameters. In complicated cases, tweaking the parameters can substantially increase the efficiency, so BAT gives users full access to all tuning parameters.

6.2 Foundations

6.2.1 Monte Carlo integration

We begin with the *fundamental Monte Carlo* principle. Suppose we have a posterior probability density $P(\boldsymbol{\theta}|D)$, often called the *target* density, and an arbitrary function $f(\boldsymbol{\theta})$ with finite expectation value under P

$$E_P[f] = \int \mathrm{d}\boldsymbol{\theta}\, P(\boldsymbol{\theta}|D) f(\boldsymbol{\theta}) < \infty. \tag{6.1}$$

Then a set of draws $\{\boldsymbol{\theta}_i : i = 1 \ldots N\}$ from the density P, that is $\boldsymbol{\theta}_i \sim P$, is enough to estimate the expectation value. Specifically, the integral can be replaced by the estimator (distinguished by the symbol $\widehat{\ }$)

$$\boxed{\widehat{E_P[f]} \approx \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{\theta}_i),\ \boldsymbol{\theta} \sim P} \tag{6.2}$$

As $N \to \infty$, the estimate converges almost surely at a rate $\propto 1/\sqrt{N}$ by the strong law of large numbers if $\int \mathrm{d}\boldsymbol{\theta}\, P(\boldsymbol{\theta}|D) f^2(\boldsymbol{\theta}) < \infty$ [11] . This is true for independent samples from the target but also for correlated

samples. The only thing that changes is the increased variance of the estimator due to correlation.

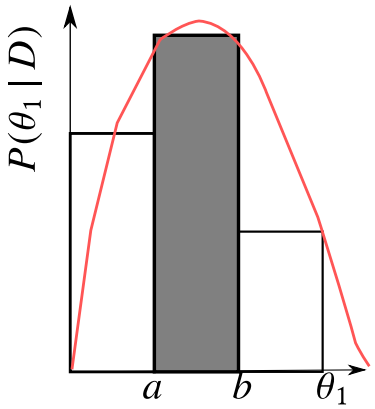


Figure 6.1 Histogram approximation to the 1D marginal.

How does this Eq. 6.2 relate to Bayesian inference? Upon applying Bayes' theorem to real-life problems, one usually has to marginalize over several parameters, and this can usually not be done analytically, hence one has to resort to numerical techniques. In low dimensions, say $d \leq 2$, quadrature and other grid-based methods are fast and accurate, but as d increases, these methods generically suffer from the *curse of dimensionality*. The number of function evaluations grows exponentially as $\mathcal{O}\left(m^d\right)$, where m is the number of grid points in one dimension. Though less accurate in few dimensions, Monte Carlo — i.e., random-number based — methods are the first choice in $d \gtrsim 3$ because the computational complexity is (at least in principle) independent of d. Which function f is of interest to us? For example when integrating over all but the first dimension of $\boldsymbol{\theta}$, the marginal posterior probability that θ_1 is in $[a, b)$ can be estimated as

$$P(a \leq \theta_1 \leq b|D) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{\theta_1 \in [a,b)}(\boldsymbol{\theta}_i)$$

with the *indicator function*

$$\mathbf{1}_{\theta_1 \in [a,b)}(\boldsymbol{\theta}) = \begin{cases} 1, \theta_1 \in [a, b) \\ 0, \text{else} \end{cases}$$

This follows immediately from the Monte Carlo principle with $f(\boldsymbol{\theta}) = \mathbf{1}_{\theta_1 \in [a,b)}(\boldsymbol{\theta})$. The major simplification arises as we perform the integral over $d - 1$ dimensions simply by ignoring these dimensions in the indicator function. If the parameter range of θ_1 is partitioned into bins, then the above holds in every bin, and defines the histogram approximation to $P(\theta_1|D)$. In exact analogy, the 2D histogram approximation is computed from the samples for 2D bins in the indicator function. For understanding and presenting the results of Bayesian parameter inference, the set of 1D and 2D marginal distributions is the primary goal. Given samples from the full posterior, we have immediate access to *all* marginal distributions at once; i.e., there is no need for separate integration to obtain for example $P(\theta_1|D)$ and $P(\theta_2|D)$. This is a major benefit of the Monte Carlo method in conducting Bayesian inference.

6.2.2 Metropolis algorithm

The key ingredient in BAT is an implementation of the Metropolis algorithm to create a Markov chain; i.e. a sequence of (correlated) samples from the posterior. We use the shorthand MCMC for Markov chain Monte Carlo.

Efficient MCMC algorithms are the topic of past and current research. This section is a concise overview of the general idea and the algorithms available in BAT. For a broader overview, we refer the reader to the abundant literature; e.g., [11] [2] .

In BAT, there are several variants of the random-walk Metropolis Hastings algorithm available. The basic idea is captured in the 2D example plot. Given an initial point $\boldsymbol{\theta}_0$, the Metropolis algorithm produces a sample in each iteration $t = 1 \ldots N$ as follows:

- Propose a new point $\tilde{\boldsymbol{\theta}}$
- Generate a number u from the uniform distribution on [0,1]
- Set $\boldsymbol{\theta}_t = \tilde{\boldsymbol{\theta}}$ if $u < \frac{P(\tilde{\boldsymbol{\theta}} \mid D)}{P(\boldsymbol{\theta}_{t-1} \mid D)}$
- Else stay, $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1}$

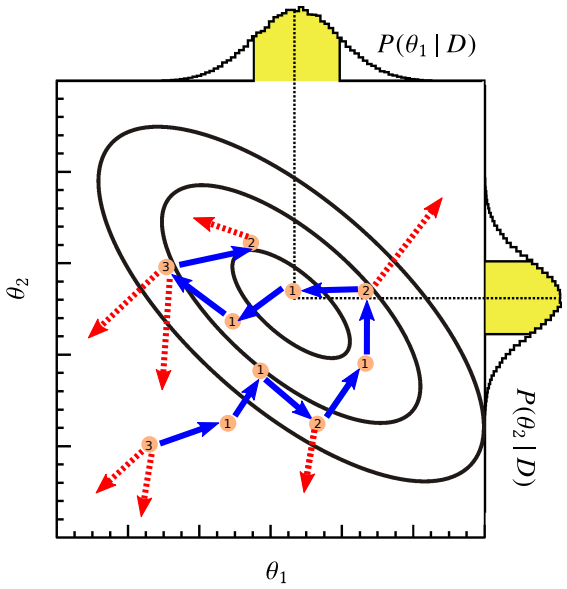


Figure 6.2 2D random walk with the Metropolis algorithm.

In the example plot, the chain begins in the lower left corner. Rejected moves are indicated by the dashed arrow, accepted moves are indicated by the solid arrow. The circled number is the number of iterations the chain stays at a given point $\boldsymbol{\theta} = (\theta_1, \theta_2)$.

In each iteration t, one updates the estimate of the 1D marginal distribution $P(\theta_1|D)$ by adding the first coordinate of $\boldsymbol{\theta}_t$ to a histogram. Repeat this for all other coordinates to update the other $(d-1)$ 1D marginals. And redo it for all pairs of coordinates to estimate the 2D marginals.

As a concrete example, suppose the chain has 5 iterations in 2D:

t	$\boldsymbol{\theta}_t$
1	(1.1, 2.3)
2	(1.1, 2.3)
3	(3.8, 1.8)
4	(2.4, 5.2)
5	(1.8, 4.2)

Let us choose a histogram to approximate the 1D marginal posterior for θ_1 with five bins from $[n, n+1)$ for $n = 0 \ldots 4$ such that the right edge of the bin is not included. Up to $t = 5$, the histogram is

n	weight
0	0
1	3
2	1
3	1
4	0

In the end, we usually normalize the histogram so it estimates a proper probability density that integrates to 1.

6.2.3 Convergence

Since samples are not independent, the initial point has some effect on Markov chain output. The asymptotic results guarantee that, under certain conditions (see [11] or [2]) a chain of infinite length is independent of the initial point. In practice, we can only generate a finite number of points so a decision has to be made when the chain has run long enough. One helpful criterion is to run multiple chains from different initial positions and to declare convergence if the chains mixed; i.e. explore the same region of parameter space. Then the chains have forgotten their initial point.

Non-convergence is a problem that can have many causes including simple bugs in implementing the posterior. But there are properly implemented posteriors for which a Markov chain has difficulties to explore the parameter space efficiently, for example because of strong correlation, degeneracies, or multiple well separated modes.

6.3 Implementation in BAT

Implementing the Metropolis algorithm, one has to decide on how to propose a new point based on the current point, that is one needs the *proposal function* $q(\tilde{\boldsymbol{\theta}} \mid \boldsymbol{\theta}_t, \xi)$ with adjustable parameters ξ. The main difference between MCMC algorithms is typically given by different choices of q. The Metropolis algorithm doesn't specify which q to choose, so we can and have to select a function q and tune ξ according to our needs.

In BAT, the proposal is *symmetric* around the current point

$$q(\tilde{\boldsymbol{\theta}} \mid \boldsymbol{\theta}_t, \xi) = q(\boldsymbol{\theta}_t \mid \tilde{\boldsymbol{\theta}}, \xi). \tag{6.3}$$

The Markov property implies that the proposal may only depend on the current point $\boldsymbol{\theta}_t$ and not on any previous point. If the value of ξ is set based on a past sequence of iterations of the chain, we need two stages of sampling in BAT, the *prerun* and the *main run*. In the prerun, the chain is run and periodically ξ is updated based on the past iterations. In contrast, ξ is kept fixed in the main run to have a proper Markov chain.

6.3.1 Proposal functions

BAT offers two kinds of proposal function termed *factorized* and *multivariate*. The general form is either a Gaussian or Student's t distribution. In the factorized case, the joint distribution is a product of 1D distributions. In the multivariate case, a dense covariance matrix is used that allows correlated proposals. In either case, the default is Student's t distribution with one degree of freedom (dof); i.e., a Cauchy distribution. Select the proposal like this:

```
m.SetProposeMultivariate(true);
m.SetProposalFunctionDof(5); // Student's t with 5 degrees of freedom
m.SetProposalFunctionDof(-1); // Gaussian
```

6.3.1.1 Multivariate proposal

Since

Introduced and set as the default in v1.0

Changing all d parameters at once within one iteration is an all-or-nothing approach. If the proposed move is accepted, all parameters have changed for the price of a single evaluation of the posterior. If the move is rejected, the new point is identical to the old point and the chain does not explore the parameter space.

We implement the adaptive algorithm by Haario et al. [6], [15]. In brief, the proposal is a multivariate Gaussian or Student's t distribution whose covariance is learned from the covariance of samples in the prerun. An overall scale factor is tuned to force the acceptance rate into a certain range.

the multivariate normal distribution

$$\mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}} |\boldsymbol{\Sigma}|^{-1/2} \exp\left(-\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu})\right) \tag{6.4}$$

or the multivariate Student's t distribution

$$\mathcal{T}(\boldsymbol{\theta}|\boldsymbol{\mu},\boldsymbol{\Sigma},\nu) = \frac{\Gamma((\nu+d)/2)}{\Gamma(\nu/2)(\pi\nu)^{d/2}} |\boldsymbol{\Sigma}|^{-1/2} (1+\frac{1}{\nu}(\boldsymbol{\theta}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}))^{-(\nu+d)/2} \tag{6.5}$$

can adapt in such a way as to efficiently generate samples from essentially any smooth, unimodal distribution. The parameter ν, the degree of freedom, controls the "fatness" of the tails of $\mathcal{T}$; the covariance of $\mathcal{T}$ is related to the scale matrix $\boldsymbol{\Sigma}$ as $\frac{\nu}{\nu-2} \times \boldsymbol{\Sigma}$ for $\nu > 2$, while $\boldsymbol{\Sigma}$ is the covariance of $\mathcal{N}$. Hence for finite ν, $\mathcal{T}$ has fatter tails than $\mathcal{N}$, and for $\nu \to \infty$, $\mathcal{T}(\boldsymbol{\theta}|\boldsymbol{\mu},\boldsymbol{\Sigma},\nu) \to \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu},\boldsymbol{\Sigma})$.

Before delving into the details, let us clarify at least qualitatively what we mean by an efficient proposal. Our requirements are

- that it allow to sample from the entire target support in finite time,
- that it resolve small and large scale features of the target,
- and that it lead to a Markov chain quickly reaching the asymptotic regime.

An important characteristic of Markov chains is the acceptance rate α, the ratio of accepted proposal points versus the total length of the chain. We argue that there exists an optimal α for a given target and proposal. If $\alpha = 0$, the chain is stuck and does not explore the state space at all. On the contrary, suppose $\alpha = 1$ and the target distribution is not globally uniform, then the chain explores only a tiny volume where the target distribution changes very little. So for some $\alpha \in (0, 1)$, the chains explore the state space well.

How should the proposal function be adapted? After a chunk of N_{update} iterations, we change two things. First, in order to propose points according to the correlation present in the target density, the proposal scale matrix $\boldsymbol{\Sigma}$ is updated based on the sample covariance of the last n iterations. Second, $\boldsymbol{\Sigma}$ is multiplied with a scale factor c that governs the range of the proposal. c is tuned to force the acceptance rate to lie in a region of $0.15 \leq \alpha \leq 0.35$. The α range is based on empirical evidence and the following fact: for a multivariate normal proposal function, the optimal α for a normal target density is 0.234, and the optimal scale factor is $c = 2.38^2/d$ as the dimensionality d approaches ∞ and the chain is in the stationary regime [12] . We fix the proposal after a certain number of adaptations, and then collect samples for the final inference step. However, if the Gaussian proposal function is adapted indefinitely, the Markov property is lost, but the chain and the empirical averages of the integrals represented by Eq. 6.1 still converge under mild conditions [6].

The efficiency can be enhanced significantly with good initial guesses for c and $\boldsymbol{\Sigma}$. We use a subscript t to denote the status after t updates. It is often possible to extract an estimate of the target covariance by running a mode finder like MINUIT that yields the covariance matrix at the mode as a by product of optimization. In the case

of a degenerate target density, MINUIT necessarily fails, as the gradient is not defined. In such cases, one can still provide an estimate as

$$\mathbf{\Sigma}^0 = \mathrm{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2) \tag{6.6}$$

where σ_i^2 is the prior variance of the i-th parameter. The updated value of $\mathbf{\Sigma}$ in step t is

$$\mathbf{\Sigma}^t = (1 - a^t)\mathbf{\Sigma}^{t-1} + a^t \boldsymbol{S}^t \tag{6.7}$$

where $\boldsymbol{S}^t$ is the sample covariance of the points in chunk t and its element in row m and column n is computed as

$$(\boldsymbol{S}^t)_{mn} = \frac{1}{N_{\text{update}} - 1} \sum_{i=(t-1)\cdot N_{\text{update}}}^{t\cdot N_{\text{update}}} ((\boldsymbol{\theta}^i)_m - \widehat{E_P[(\boldsymbol{\theta})_m]})((\boldsymbol{\theta}^i)_n - \widehat{E_P[(\boldsymbol{\theta})_n]}) \tag{6.8}$$

The weight $a^t = 1/t^\lambda$, $\lambda \in [0, 1]$ is chosen to make for a smooth transition from the initial guess to the eventual target covariance, the implied cooling is needed for the ergodicity of the chain if the proposal is not fixed at some point [6]. One uses a fixed value of λ, and the particular value has an effect on the efficiency, but the effect is generally not dramatic; in this work, we set $\lambda = 0.5$ [15].

We adjust the scale factor c as described in the pseudocode shown below. The introduction of a minimum and maximum scale factor is a safeguard against bugs in the implementation. The only example we can think of that would result in large scale factors is that of sampling from a uniform distribution over a very large volume. All proposed points would be in the volume, and accepted, so $\alpha \equiv 1$, irrespective of c. All other cases that we encountered where $c > c_{max}$ hinted at errors in the code that performs the update of the proposal.

```
// default values
αmin = 0.15; αmax = 0.35;
cmin = 1e-5; cmax = 100;
β = 1.5;

// single update of the covariance scale factor
if (α > αmax && c < cmax) {
  c *= β * c
} else if (α < αmin && c > cmin) {
  c /= β
}
```

See also

> **BCEngineMCMC::SetMultivariateCovarianceUpdateLambda**, **BCEngineMCMC::SetMultivariate↩ Epsilon**, **BCEngineMCMC::SetMultivariateScaleMultiplier**

6.3.1.2 Factorized proposal

Since

> Factorized was the default and only choice prior to v1.0 and continues to be available using BCEngine↩ MCMC::SetProposeMultivariate(false)

The factorized proposal in d dimensions is a product of 1D proposals.

We sequentially vary one parameter at a time and complete one iteration of the chain once a new point has been proposed in *every* direction. This means the chain attempts to perform a sequence of axis-aligned moves in one iteration.

Each 1D proposal is a Cauchy or Breit-Wigner function centered on the current point. The scale parameter is adapted in the prerun to achieve an acceptance rate in a given range that can be adjusted by the user. Note that there is a separate scale parameter in every dimension.

This means the posterior is called d times in every iteration. Since the acceptance rate is typically different from zero or one, the factorized proposal typically generates a new point in every iteration that differs from the previous point in some but not all dimensions.

6.3.1.3 Comparison

Comparing the factorized proposal to the multivariate proposal, we generally recommend the multivariate for most purposes.

Use the factorized proposal if you can speed up the computation of the posterior if you know that some parameters did not change. This can be useful if the computation is expensive if some but not all parameters change.

6.3.2 Prerun

During the prerun, the proposal is updated. BAT considers three criteria to decide when to end the prerun. The prerun takes some minimum number of iterations and is stopped no matter what if the maximum number of prerun iterations is reached. In between, the prerun terminates if the efficiency and the R value checks are ok. To perform the prerun manually, do

```
m.MetropolisPreRun();
```

In most cases this is not needed, because **BCModel::MarginalizeAll** calls **BCEngineMCMC::Metropolis**, which will take care of the prerun and the main run and all the data handling associated with it. To force a prerun to be run again, perhaps after it failed and some settings have been adjusted, do:

```
m.SetFlagPreRun(true);
m.MarginalizeAll();
```

6.3.2.1 Efficiency

The *efficiency*, or acceptance rate, is the ratio of the accepted over the total number proposal moves. A small efficiency means the chain rarely moves but may then make a large move. A large efficiency means the chain explores well locally but may take a long time to explore the entire region of high probability. Optimality results exists only for very special cases: Roberts and Rosenthal showed that for a Gaussian target with d independent components and a Gaussian proposal, the optimal target efficiency is 23.4 % for $d \geq 5$ is but should be larger for small d; e.g., 44 % is best in one dimension [13] . Based on our experience, we use a default range for the efficiency as $[0.15, 0.35]$.

See also

BCEngineMCMC::SetMinimumEfficiency, **BCEngineMCMC::SetMaximumEfficiency**

6.3.2.2 R value

The *R value* [4] by Gelman and Rubin quantifies the estimated scale reduction of the uncertainty of an expectation value estimated with the samples if the chain were run infinitely long. Informally, it compares the mean and variance of the expectation value for a single chain with the corresponding results of multiple chains. If the chains mix despite different initial values, then we assume that they are independent of the initial value, the burn-in is over, and the samples produce reliable estimates of quantities of interest. For a single chain, the R value cannot be computed.

In BAT, we monitor the expectation value of each parameter and declare convergence if all R values are below a threshold. Note that the R values are estimated from batches of samples, and they usually decrease with more iterations but they may also increase, which usually is a clear indication that the chains do not mix, perhaps due to multiple modes that trap the chains.

See also

BCEngineMCMC::SetRValueParametersCriterion Set the maximum allowed R value for all parameters. By defition, R cannot go below 1 except for numerical inaccuracy. **Default: 1.1**
BCEngineMCMC::SetCorrectRValueForSamplingVariability The strict definition of R corrects the sampling variability due finite batch size. **Default: false**
BCEngineMCMC::GetRValueParameters R values are computed during the prerun and they can be retrieved but not set.

6.3.2.3 Prerun length

Defining convergence automatically based on the efficiency or the R value is convenient may be too conservative if the user knows a good initial value, a good proposal, etc. For more control the minimum and maximum length of the prerun can be set, too.

See also

BCEngineMCMC::SetNIterationsPreRunMin, **BCEngineMCMC::SetNIterationsPreRunMax**
BCEngineMCMC::SetNIterationsPreRunCheck sets the number of iterations between checks

If desired, the statistics can be cleared to remove the effect of a bad initial point with **BCEngineMCMC::SetPreRunCheckClear** after some set of iterations

For the user's convenience, multiple settings related to precision of the Markov chain can be set at once using **BCEngineMCMC::SetPrecision**. The default setting is `m.SetPrecision(BCEngineMCMC::kMedium)`.

6.3.3 Main run

In the main run, the proposal is held fixed and each chain is run for **BCEngineMCMC::GetNIterationsRun()** iterations.

See also

BCEngineMCMC::SetNIterationsRun

To reduce the correlation between samples, a lag can be introduced to take only every 10th element with **BCEngineMCMC::SetNLag**.

Chapter 7

Optimization

While sampling from a posterior with Markov chains, BAT will store the maximum value of the posterior that it has seen (and the corresponing point in parameter space). But the sampling algorithm is designed to sample, not to find a maximum value. To find the maximum point, BAT includes optimization methods. These are called via **BCIntegrate::FindMode**, which takes optional arguments: You may give it a starting point; if you do not specify one, BAT will start from the current maximum point. You can also specify the optimization method when you call the function; or you can specify it in advance via `BCIntegrate::SetOptimzationMethod`.

By default, BAT replaces its currently held maximum point only if the newly found one has a greater posterior. You can tell BAT to replace the currently held one regardless of whether optimization improves upon the currently stored one via the function **BCIntegrate::SetFlagIgnorePrevOptimization**.

7.1 Minuit

BAT uses Minuit through ROOT's interface to it. To use it, set the optimization method to **BCIntegrate::kOptMinuit**.

Minuit is what is known as a gradient follower—it moves from a starting point in the direction that increases the posterior until it finds a maximum. Minuit is much better at finding the exact location of a maximum than sampling with Markov chains is, but it does not gaurantee that this maximum is the global maximum or only a local one. Sampling with Markov chains can better identify the region of the global maximum than Minuit can. So we recommend that you first marginalize your model and then call `FindMode`.

7.2 Simulated Annealing

BAT provides a simulated annealing algorithm for optimization. To use it, set the optimization method to **BCIntegrate::kOptSimAnn**.

This algorithm is similar to the Metropolis-Hastings sampling one in that involves proposal of new points to move to randomly in a neighborhood of the current point. But the neighborhood and the acceptance criteria are regulated in such a way as to encourage motion towards the global maximum.

Using BAT, simulated annealing will in general not find the maximum point as precisely or rapidly as Minuit, but it can more reliably find the global maximum instead of a local maximum. We therefore recommend calling `FindMode` with **BCIntegrate::kOptMinuit** after calling it with `kOptSimAnn`. This is similar to first sampling and then optimizing with Minuit; but simulated annealing is less calculationally expensive than sampling—that is, it will call your likelihood less often. However, simulated annealing is not a sampling technique and will not provide output samples for further offline use.

There are several parameters that govern the running of simulated annealing. Please consult the code documentation and C. Brachem's *Implementation and test of a simulated annealing algorithm in the Bayesian Analysis Tookit* (2009) for how to set them. The most important option you may change is the proposal function. This is set via **BCIntegrate::SetSASchedule**. The options are **BCIntegrate::kSACauchy**, **BCIntegrate::kSABoltzmann**, and **BCIntegrate::kSACustom**. (The latter requires you to set your own proposal function.)

Chapter 8

Predefined Models

Chapter 9

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.

9.1 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.

9.1.1 Excluding systematic uncertainies

in case no sources of systematic uncertainty are taken into account the likelihood is defined as

$$l = \prod_{i=1}^{n_{\text{ch}}} \prod_{j=1}^{n_{\text{bin}}} \frac{\lambda_{ij}^{n_{ij}}}{n_{ij}!} e^{-\lambda_{ij}} \,, \tag{9.1}$$

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

$$\lambda_{ij} = \sum_{k=1}^{N_{\text{p}}} \lambda_{ijk} \tag{9.2}$$

$$= \sum_{k=1}^{N_{\text{p}}} \lambda_k \cdot f_{ijk} \cdot \epsilon_{ik} \,, \tag{9.3}$$

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.

9.1.2 Including Systematic Uncertainies

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

$$\epsilon_{ik} \rightarrow \epsilon_{ik} \cdot (1 + \sum_{l=1}^{N_{\text{syst}}} \delta_l \cdot \Delta\epsilon_{ijkl}), \tag{9.4}$$

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.

9.2 Creating the Fitter

The main MTF class **BCMTF** 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`.

9.3 Adding a Channel

The MTF fits several channels simultaneously. These channels can be physics channels, e.g., $Z^0 \rightarrow e^+e^-$ and $Z^0 \rightarrow \mu^+\mu^-$, 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.

9.4 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.

9.5 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 the Mathematical formulation. 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 the Mathematical formulation. 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 the Mathematical formulation. 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.

9.6 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 the Mathematical formulation. 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

$$\Delta_{\text{up}} = (y_{\text{up}} - y_{\text{nominal}})/y_{\text{nominal}}\,, \tag{9.5}$$

$$\Delta_{\text{down}} = (y_{\text{nominal}} - y_{\text{down}})/y_{\text{nominal}}\,. \tag{9.6}$$

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.

9.7 Running the Fit

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

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

9.8 Output

The MTF produces several outputs:

1. `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`.
2. `PrintResults(const char* name)` writes a summary of the fit into a text file `name`.
3. To print a stacked histogram of the templates and the data histogram in the file `name` using a set of parameters `parameters`, do

```
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 = "")
```

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

9.9 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`.

9.10 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.

9.11 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.

9.12 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.

9.13 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

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

9.14 Performing Automated Analyses

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

9.14.1 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.

9.14.2 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.

9.14.3 Performing Calibration Analyses

Ensemble tests for different sets of parameters can be automized by using the method

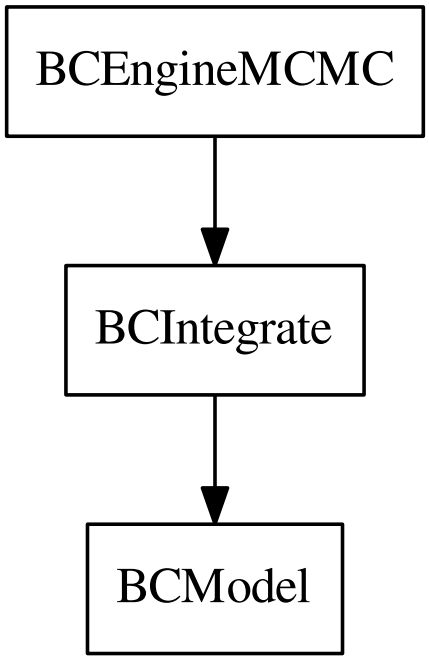
```
int BCMTFAnalysisFacility::PerformCalibrationAnalysis(const char * dirname,
                                const std::vector<double> & default_parameters,
                                int index,
                                const std::vector<double> & parametervalues,
                    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

Chapter 10

Structure of the Code

BAT is object-oriented and uses inheritance for code reuse. The single most important hierarchy is that of **BCModel**. In most applications, a user would implement a model `MyModel` by either inheriting from **BCModel** directly or creating an instance of one of the Predefined Models and thus would inherit from **BCModel** indirectly.



This structure has the benefit that `MyModel` has native access to all things related to Markov chains due to **BCEngineMCMC** and one can immediately optimize or integrate the posterior with the methods in **BCIntegrate**. However, one has to keep in mind that multiple Markov chains for the *same* instance of `MyModel` share the entire state of the object. This can lead to complications when `MyModel::LogLikelihood` is called concurrently; see Multithreading and Thread Safety for more details.

Chapter 11

Model Comparison

In case more than one model is defined to explain the same data set the **BCModelManager** can come in handy. Separate **BCModel**s `m1` and `m2` and their prior probabilities (70 % vs 30 %) are added to the model manager via

```
BCModelManager mgr;
mgr.AddModel(&m1, 0.7);
mgr.AddModel(&m2, 0.3);
```

For convenience, some of the most important methods for handling **BCModel**s are forwarded to each model in the manager. For example

```
mgr.SetPrecision(BCEngineMCMC::kQuick);
mgr.MarginalizeAll();
```

is equivalent to

```
for (unsigned i = 0; i < mgr.GetNModels(); ++i) {
  mgr.GetModel(i)->SetPrecision(BCEngineMCMC::kQuick);
  mgr.GetModel(i)->MarginalizeAll();
}
```

To do model comparison, the evidence is needed for each model, then the Bayes factors and the posterior odds are immediately available. Models are index by an `unsigned`; the first model has index 0 etc.:

```
mgr.Integrate();
double B = mgr.BayesFactor(0, 1);
mgr.PrintModelComparisonSummary();
```

See also

- `examples/advanced/polynomialFit`

Chapter 12

Output Options

Chapter 13

Defining a factorized prior

If a model does not overload **BCModel::LogAPrioriProbability**, then its prior is the product of the individual priors for each of its parameters. We call these the factorized priors. To set a parameter's prior call

```
BCParameter::SetPrior(BCPrior* const prior);
```

You can call factorized priors within your overloaded `LogAPrioriProbability` by querying the log of the prior of a particular parameter with

```
BCParameter::GetLogPrior(double x)
```

The prior you set for a parameter need only inherit from **BCPrior**. BAT has several built in prior classes, such as **BCConstantPrior** and **BCGaussianPrior**. You can implement new factorized priors by inheriting from the **BCPrior** class. To illustrate how to do this, we will work through the construction of the Gaussian prior:

```
class BCGaussianPrior : public BCPrior
```

BCPrior is a pure-virtual class with three methods that must be overloaded:

```
double GetLogPrior(double x);
BCPrior* clone() const;
bool IsValid() const;
```

The first one contains the meat of our new prior:

```
double GetLogPrior(double x)
{
  return -0.5 * (x - fMean) * (x - fMean) / fSigma / fSigma - log(fSigma) - 0.5 * log(2 * M_PI);
}
```

This is, naturally, the log of a Guassian prior. The second one should simply return a copy of the prior:

```
BCPrior* clone() const
{
  return new BCGaussianPrior(*this);
}
```

The last function is required for checking that everything that is needed by the prior is properly set. You'll notice that our example calls on two member variables in its `GetLogPrior`: `fMean` and `fSigma`. So we must check whether they have been properly set:

```
bool IsValid() const
{
  return std::isfinite(fMean) and std::isfinite(fSigma) and fSigma > 0;
}
```

This checks that the mean and standard deviation are both finite and that the standard deviation is positive semi-definite, as it need be.

Naturally to be of use, we also create a constructor that allows us to set the mean and standard deviation at creation; and getters and setters that allow us to access them. (See the source code of **BCGaussianPrior** for their implementation.)

This is all that is required to create a new prior. **BCPrior** has several methods for getting properties of the prior: the mode; the integral over a range; the n'th raw, central, and standardized moments; the mean; the variance; the standard deviation; the skewness; and the kurtosis. These functions make their own internal calculations and need not be overloaded. If you wish to speed them up, or provide exact results, you may overload them. But take care that you overload them to give back proper results! For example, the mode of the Gaussian distribution is not simply `fMean`, since this presumes the range over which we query contains `fMean`. The more proper implementation is

```
double GetMode(double xmin, double xmax)
{
  if (fMean < xmin)
    return xmin;
  if (fMean > xmax)
    return xmax;
  return fMean;
}
```

Chapter 14

Sharing / Loading samples

You can output the samples of BAT's Markov chains in the form of a ROOT `TTree` saved in a `TFile`. You can read these samples back into BAT using **BCEmptyModel**:

```
BCEmptyModel m("MyModel_mcmc.root");
m.Remarginalize();
```

where `MyModel\_mcmc.root` is a `.root` file produced by a BAT analysis—that is, it contains two `Tree`'s, `X\_mcmc.root` and `X\_pars.root`, where X is a prefix common to both trees. BAT will search the file for two such trees matching the structures expected from BAT output. Alternatively, the constructor can be called with the prefix specified explicitly:

```
BCEmptyModel::BCEmptyModel(const std::string& filename, const std::string& name, bool loadObservables).
```

The last argument tells switches on or off the loading of **BCObservable**'s stored in the `TTree`.

You can also use an instance of your own model instead of **BCEmptyModel** if you provide a constructor that calls the

```
BCModel::BCModel(const std::string\&
filename, const std::string\& name, bool loadObservables)
```

constructor.

Note that instead of calling `MarginalizeAll()`, a reloaded BAT analysis is marginalized by calling `Remarginalize()`.

All subsequent drawing or accessing of **BCH1D** and **BCH2D** objects is the same as if the analysis had been run directly rather than reloaded. For most summary reports of BAT, the actual model itself is not needed—only the samples. So you can share your results by sharing your ROOT output files without sharing your model implementation.

Likewise, when outputting to a ROOT file, BAT (by default) autosaves the Markov chains to the ouput file at regular intervals. You can use the steps outlined above to check the state of an ongoing analysis using **BCEmptyModel**.

Chapter 15

Multithreading and Thread Safety

In a demanding analysis, a single evaluation of the posterior may take seconds or more. In an MCMC analysis, this is then the bottle neck. The overall time to solution can then be reduced by running multiple chains on multiple threads.

Assuming BAT is configured with parallelization enabled (see the Installation instructions), 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 (say one millisecond per evaluation), the overhead from synchronizing threads usually slows down the entire process. We invite the user to experiment which number of threads gives the best performance.

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 for a likelihood that takes more than a second to evaluate.

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();
}

double MyModel::Method()
{
  return -member * member;
}
```

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. There is a race condition on `member` because it is read and written simultaneously by multiple threads so the result of `LogLikelihood` is not deterministic. There are several solutions.

Avoid state

In this contrived example, it is easiest to avoid using any member variable or method call and do everything inside `LogLikelihood`

```
double MyModel::LogLikelihood(const std::vector<double>& parameters)
{
  return -parameter[0] * parameter[0];
}
```

Independent copies of state

In practice, it may be impractical or even inevitable to maintain state and call into other functions that require state. A clean solution is to move `Method` into `OtherClass`, and to keep an independent copy of `OtherClass` for each thread or each chain. For this purpose, we provide the hook **BCEngineMCMC::MCMCUserInitialize** that is called before any chain attempts to evaluate the likelihood. For example:

```
class SomeState
{
public:
  double Method() { ... }
private:
  double t;
}

class MyModel
{
public:
  ...
  virtual void MCMCUserInitialize()
  {
    state.resize(GetNChains());
  }

  virtual double LogLikelihood(const std::vector<double>& parameters)
  {
    SomeState& s = state.at(GetCurrentChain());
    s.t = parameters[0];
    return s.Method();
  }

private:
  std::vector<SomeState> state;
};
```

Note

In BAT, only the Markov chain sampling uses multiple threads. During optimization or any algorithm, only one thread is active. In the above example, **BCEngineMCMC::GetCurrentChain** returns 0 in such a context.

Bibliography

[1] F. Beaujean, A. Caldwell, D. Kollár, and K. Kröninger. p-values for model evaluation. *Phys.Rev.D*, 83:012004, 2011. 22

[2] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov chain Monte Carlo*. CRC press, 2011. 32, 34

[3] Giulio D'Agostini. *Bayesian Reasoning in Data Analysis: A Critical Introduction*. World Scientific, 2003. 21

[4] A. Gelman and D.B. Rubin. Inference from iterative simulation using multiple sequences. *Stat.Sci.*, 7(4):457–472, 1992. 37

[5] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. *Bayesian data analysis*, volume 2. Chapman & Hall/CRC Boca Raton, FL, USA, 2014. 21

[6] Heikki Haario, Eero Saksman, and Johanna Tamminen. An adaptive metropolis algorithm. *Bernoulli*, 7(2):pp. 223–242, 2001. 35, 36

[7] John A. Hartigan. *Bayes theory*. Springer, 1983. 21

[8] Edwin T. Jaynes and G. Larry Bretthorst. *Probability theory*. Cambridge University Press, 2003. 21

[9] Maurice George Kendall, Alan Stuart, Anthony O'Hagan, Jonathan Forster, and J. K. Ord. *Kendall's Advanced Theory of Statistics: Bayesian Inference*. Arnold, 2004. 21

[10] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. 21

[11] C.P. Robert and G. Casella. *Monte Carlo statistical methods*. Springer, 2004. 31, 32, 34

[12] G. O. Roberts, A. Gelman, and W. R. Gilks. Weak convergence and optimal scaling of random walk metropolis algorithms. *Ann.Appl.Probab.*, 7(1):pp. 110–120, 1997. 35

[13] Jeffrey S Rosenthal et al. Optimal proposal distributions and adaptive mcmc. *Handbook of Markov Chain Monte Carlo*, pages 93–112, 2011. 37

[14] D. S. Sivia and John Skilling. *Data analysis: a Bayesian tutorial*. Oxford University Press, 2006. 21

[15] Darren Wraith, Martin Kilbinger, Karim Benabed, Olivier Cappe, Jean-Francois Cardoso, et al. Estimation of cosmological parameters using adaptive importance sampling. *Phys.Rev.D*, 80:023507, 2009. 35, 36