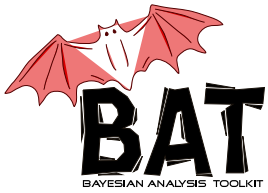

Quick-start Guide to BAT

BAT 0.9.4

This guide is intended to provide all the necessary information to quickly set up a BAT-based analysis code and is aimed at the programmer experienced with object-oriented C++. It explains how to create the barest classes necessary for a successful compilation and analysis of a single model. For more detailed use of the BAT software, including model comparison, and installation instructions, consult the full BAT introduction.

The simplest nontrivial analysis code one can build with BAT contains a model, a data set, and a main program to instatiate the first two. The following instructions will create the pieces necessary to map out the *a posteriori* (or posterior) likelihood $P(\vec{\lambda}|D)$ for a given parameter set $\vec{\lambda}$ given our data set D. This is done according to Bayes theorem

$$P(\vec{\lambda}|D) \propto P(D|\vec{\lambda})P_0(\vec{\lambda}),$$

where $P(D|\vec{\lambda})$ is the conditional probability of the data given the parameter set, and $P_0(\vec{\lambda})$ is the *a priori* (or prior) probability of $\vec{\lambda}$.

1 The Model

Our model class `MyModel` must inherit from the `BCModel` class. This class must setup the parameters of our model and report back the log of the likelihood and the *a priori* probability for our parameter set.

1.1 Model Parameters

We can add parameters to a model through

```
int BCEngineMCMC::AddParameter(const char * name,
                               double min, double max,
                               const char * latexname = "")
```

```
int BCEngineMCMC::AddParameter(BCParameter * parameter).
```

We must add parameters (with unique names) to our model before running our analysis. Preferably this is done in the constructor to `MyModel` or through an additional method, say,

```
void MyModel::DefineParameters()
{
     this -> AddParameter("par1", 0, 100);
     this -> AddParameter("par2", -1, 1);
     ...
}.
```

1.2 Priors

Our model class reports the *a priori* probability, $P_0(\vec{\lambda})$, for a set of parameter values through the method

```
double BCModel::LogAPrioriProbability(const std:vector<double> & parameters),
```

which need not be overloaded if our parameters are independent in our *a priori* probability.

1.2.1 Independent Priors

In this case, we may set the *a priori* probability for each parameter to a value returned by an arbitary function with[1]

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

or the corresponding value from an arbitrary histogram with

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

We may also quickly set the *a priori* probability of a parameter to one of four often-used distributions:

```
int BCModel::SetPriorDelta(int index, double value)
int BCModel::SetPriorGauss(int index, double mean, double sigma)
int BCModel::SetPriorGauss(int index, double mean,
                           double sigmadown, double sigmaup)
int BCModel::SetPriorConstant(int index),
```

where the second `SetPriorGauss` creates a normalized Gaussian distribution with distinct upper and lower widths.

1.2.2 Single Prior Function

We may also overload `BCModel::LogAPrioriProbability`, especially when our model parameters are not independent of each other, to return a double through any arbitrary C++ function.

[1] All functions in this section have corresponding partners that allow access to the parameter by name instead of index.

The BCMath namespace contains methods for the logarithms of several often-used distributions that can be used in this method. It contains, among many others, LogGaus and LogPoisson.

1.3 Likelihood

It is essential that MyModel overload

BCModel::LogLikelihood(const std::vector<double> & parameters),

which reports the log of the conditional probability, $P(D|\vec{\lambda})$, of the data set given particular values of our parameters. Again, the namespace BCMath contains methods to return the logarithms of many useful often-used distributions.

2 Data Set

The other essential ingredient to an analysis in BAT is a data set to analyze. We need not necessarily write our own class that inherits from BCDataSet, since it contains all the essential minimal functionality for a simple analysis—in this case, we may read the data in directly in main().

We may read our data directly from a file through the methods

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

or add data points directly to the object through

void BCDataSet::AddDataPoint(BCDataPoint * datapoint),

and the BCDataPoint class' methods for directly setting the data values.

We must be careful that all data points have the same dimensionality, which is set in the constructor of the BCDataPoint class

BCDataPoint::BCDataPoint(int nvariables).

3 Putting It All Together

Our main() program need now only instatiate an object of MyModel, read in a data set, and add it to our model, before running the analysis. This is accomplished (depending on how we've written MyModel) as follows:

```
#include <BAT/BCDataSet.h>
#include "MyModel.h"

int main()
{
     BCDataSet * data = new BCDataSet();
```

```
    ... [read data from source] ...

    MyModel model("MyModel");
    // MyModel::DefineParameters() is called in constructor

    model.SetDataSet(data);

    ...
}
```

The final step of the program is to map out the posterior likelihood in our parameter space. This is achieved by

```
void BCModel::MarginalizeAll(),
```

which produces marginalized likelihoods for each parameter (in 1D) and each pair of parameters (in 2D). These are accessed through[2]

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

where `BCH1D` and `BCH2D` are wrapper classes for ROOT's `TH1D` and `TH2D`. The marginalization method is set through

```
BCModel::SetMarginalizationMethod(BCMarginalizationMethod method)
```

where `BCMarginizationMethod` is an `enum` defined in `BCIntegrate`. We can also set the level of precision and the number of iterations for the marginalization through

```
BCEngineMCMC::MCMCSetPrecision(BCEngineMCMC::Precision precision),
BCEngineMCMC::MCMCSetNIterationsRun(unsigned int n),
```

where `Precision` is an `enum` defined in `BCEngineMCMC`. Of particular use for initial testing is the prevision level `BCEngineMCMC::kLow`, which sets the number of Markov chains to one.

The best-fit parameter set is accessed through

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

and the associated errors through

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

The best-fit value for each parameter according to its 1D marginalized distribution is accessed through,

```
std::vector<double> BCEngineMCMC::GetBestFitParametersMarginalized()
double BCEngineMCMC::GetBestFitParameterMarginalized(unsigned int index).
```

Marginalized distributions can also be printed directly to file using

[2]These methods also exist with the `BCParameter` arguments replaced by `const char` pointers to access the parameters by name.

```
int BCModel::PrintAllMarginalized1D(const char * filebase)
int BCModel::PrintAllMarginalized2D(const char * filebase)
int BCModel::PrintAllMarginalized(const char * filebase,
                                  unsigned int hdiv, unsigned int ndiv)
```

3.1 Generating Log Output

Detailed information on the running of BAT can be output to the screen or file through the static class `BCLog`. At the start of our program, we can designate a log file and the level or output to the file and (independently) the screen through

```
BCLog::OpenLog(const char * filename, BCLog::LogLevel loglevelfile,
               BCLog::LogLevel loglevelscreen),
```

where `BCLog::LogLevel` is an `enum` with the following values

`debug` lowest level of output;

`detail` details of functions and status of marginalization;

`summary` results, including best-fit values and normalization;

`warning` warning messages;

`nothing` no output.

3.2 Further Accessing & Outputting Marginalized Distributions

The results of a BAT analysis can be saved to file through the `BCModelOutput` class. Through ROOT classes, it handles the storage of the marginalized distributions and the results of the Markov Chain used to generate them.

The classes `BCH1D` and `BCH2D` have several methods for obtaining useful information from the marginalized distributions, including access to the mean, median, mode, and limits at given credibility levels. As well, these classes contain methods for drawing their distributions with the information such as credibility intervals highlighted.