

Embedding Bayesian Tools in Mathematical Software

JOHN GEWEKE and WILLIAM MCCAUSLAND

University of Iowa, USA and University of Minnesota, USA

SUMMARY

The BACC software provides its users with tools for Bayesian Analysis, Computation and Communications. These tools are embedded in mathematical software applications such as Matlab and Gauss. From the user's perspective, there is a seamless integration of special-purpose BACC commands with the powerful built-in commands of the application. Several models are currently available, and BACC is designed to be extendible. We give a brief demonstration of the use of BACC for Matlab, and discuss the implementation of new models for BACC.

1. INTRODUCTION

The BACC software provides its users with tools for Bayesian Analysis, Computation and Communications. Most other Bayesian software is stand-alone. An important feature of the BACC software is that it implements its tools as extensions to popular mathematical applications such as Matlab and Gauss. From the user's perspective, there is a seamless integration of special-purpose BACC commands with built-in general-purpose commands for computation, graphics, and program flow control.

The user has available a number of *models*, which describe the proper joint distribution of unknown and known quantities. The user creates *model instances* by selecting one of the models and supplying values for its known quantities. BACC provides commands for model instance creation, prior and posterior simulation, marginal likelihood estimation, and much more.

Student users have found BACC very useful for learning about Bayesian analysis, computation and communication. A limited number of model specifications have been implemented for BACC, and so many users may want to become *model developers*, and implement their own models for BACC. BACC is designed to be extendible in this way.

Section 2 introduces an example of a model. The example appears throughout the paper to illustrate various points.

Section 3 gives a brief introduction to BACC from a user's perspective. It also describes how to get more information on using the BACC software.

Section 4 gives an overview of the BACC software. It describes the structure of the software, and discusses how the guiding design principles support the extendability of the software.

Section 5 describes how model developers can extend the functionality of BACC by implementing new models. Model developers specify posterior simulation algorithms by

implementing transition kernels that preserve the posterior distribution. This is in contrast to the BUGS software, described in Gilks et al., (1994) in which models are defined by specifying prior distributions and data distributions. BACC demands more effort to specify a model, but offers more flexibility.

Section 6 describes some resources for model developers.

Section 7 concludes the paper.

2. AN EXAMPLE

This section describes a particular model. We will revisit this example throughout the paper. The model is a variant of the Seemingly Unrelated Regressions (SUR) model. It allows cross-equation covariate coefficient equality restrictions. Special cases include the familiar single-equation regression model, and the usual SUR model (block diagonal covariate matrix, with T row blocks).

The number of observations for each variable is T . There are m T -vectors of endogenous observed variables: y_1, \dots, y_m , and m corresponding $T \times k$ matrices of exogenous observed variables: Z_1, \dots, Z_m . The unobserved quantities are a k -variate coefficient parameter β , and an $m \times m$ precision (inverse of variance) parameter H . The data distribution is described by

$$y = Z\beta + \epsilon$$

$$\epsilon|Z \sim N(0, H^{-1} \otimes I_T),$$

where

$$y \equiv \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad Z \equiv \begin{bmatrix} Z_1 \\ \vdots \\ Z_m \end{bmatrix}, \quad \epsilon \equiv \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_m \end{bmatrix}.$$

In the prior distribution of β and H , the two parameters are independent. The marginal prior distribution of β is normal, with mean $\underline{\beta}$ and precision \underline{H}_β . The marginal prior distribution of H is Wishart, with mean $\underline{\nu}\underline{S}^{-1}$ and degrees of freedom $\underline{\nu}$.

$$\beta \sim N(\underline{\beta}, \underline{H}_\beta^{-1}), \quad H \sim W(\underline{S}^{-1}, \underline{\nu}).$$

3. BACC FROM THE USER'S PERSPECTIVE

In this section, we discuss BACC from the perspective of a user. For definiteness, we assume that she uses BACC through Matlab. Users of other mathematical applications will have a similar perspective. To the Matlab user, BACC is an extension of Matlab, rather than a separate application: its tools for Bayesian analysis are implemented as Matlab commands. Extending Matlab to recognize BACC commands is simple. The user downloads the BACC software from the BACC website and modifies the Matlab path so that it knows where to look for the special commands. From then on, using BACC commands is just as

easy as using Matlab's built-in commands for calculations, graphics, I/O, and program flow control. The user can even use the Matlab help system to get help on BACC commands.

The following Matlab session illustrates the use of BACC within Matlab. Suppose the user is interested in Bayesian inference using an instance of the normal linear model described in Section 2. We will assume that the user has already created or loaded the Matlab matrices β , H , ν , S , X , and y , and they take on the desired values of β , H , ν , S , X , and y . The Matlab session might then proceed as follows.

```
MATLAB>> myMI = minst('nlm','beta','H',beta_,H_,nu_,S_,X,y);
MATLAB>> postsim(myMI,10000,1);
```

```
Posterior simulation successful.
Number of samples generated and appended: 10000
Total number of samples: 10000
```

```
MATLAB>> postfilter(myMI,101:10000);
```

```
Posterior filtering successful.
Model specification: nlm
Previous number of posterior samples: 10000
Number of samples passing through filter: 9900
```

```
MATLAB>> mlike(myMI,[0.1 0.5 0.9]);
```

Truncation parameter	Log marginal likelihood estimate
1.0000000e-001	-7.2624694e+001
5.0000000e-001	-7.2659648e+001
9.0000000e-001	-7.2675457e+001

The `minst` (Model INSTance) command sets up a model instance. The user provides the name of the model specification (here, 'nlm', which stands for normal linear model), names she wants to use for the unknown quantities β and H (here, 'beta' and 'H'), and the known quantities. The command returns a value (here, myMI) which serves as an identifier for the model instance. The `minst` command checks the known matrices for correct dimensions and structure, and gives an error message if appropriate.

The `postsim` (POSTerior SIMulation) command simulates a Markov chain whose invariant distribution is the posterior distribution of (β, H) . The choice of Markov chain is up to the developer who implements the model specification. The user has provided the model instance identifier myMI, the number of draws to record in the simulation matrix (here, 10000), and the number of draws to generate for each one recorded (here, 1).

The `postfilter` command filters samples from the posterior simulation matrix. It is useful for removing burn-in draws and subsampling. The user has provided the model instance identifier, and a vector of indices specifying the draws to keep. Here, the user has removed the first 100 samples.

The `mlike` (Marginal LIKelihood) command calculates estimates of marginal likelihoods, using the algorithm of Gelfand and Dey (1994). The user has specified three differ-

ent values of a truncation parameter and the displayed results are the three corresponding estimates of the marginal likelihood. The `mlike` command can also be used to obtain standard errors of the estimates of the marginal likelihood.

The user has full access to the simulated posterior values of the unobserved quantities, and using the mathematical software's built-in commands, can produce publication-quality graphics. Importantly, the mathematical software provides a great deal of flexibility. The user can create time series plots of parameter values to help assess convergence, histograms and scatterplots of prior and posterior functions of unknown quantities, and an infinite variety of graphs not envisioned by the BACC developers.

For details on the commands featured above and others, and tutorial introductions, see the manuals for users at the BACC website. There is also other introductory material available there, including McCausland (2000), an introduction to BACC for users. It features a much longer example, with illustrations of other commands, and several examples of plots.

4. AN OVERVIEW OF THE BACC SOFTWARE

This section sketches the organization of the BACC software. The organization is invisible to the user, and even the developer is not directly concerned with some of what follows. The purpose of the section is to demonstrate that the structure of the BACC software makes it extendible.

The BACC software consists of four distinct components: core code, application specific code, model implementation code, and the matrix library.

The core code implements commands for Bayesian analysis, computation, and communication, in an application- and operating system-independent way. Some commands manipulate model instances. For example, the `minst` command creates a model instance, and the `postsim` command appends to the posterior simulation matrix of a model instance. Some commands do not. The `expect1` command takes a weighted random sample, and calculates estimates of the mean, its numerical standard error, and the relative numerical efficiency. The sample may, but need not, be taken from a simulation matrix. New commands may be added to the core. One could add, for example, a command to compute estimates of the marginal likelihood using a different algorithm, or a command to compute convergence diagnostics. The core code uses application-independent data structures. All core code is in ANSI C, and is thus portable across platforms and operating systems.

Application specific code serves as an interface between the user's mathematical application and the core code. Applications each have their own data structures, and their own way of loading dynamically linked libraries and passing information to them. The application specific code consists of code in the application's interpreted language as well as C code. Application specific code may also be operating system specific. At the moment, BACC has the application specific code necessary to implement versions of BACC for Windows Matlab, Unix Matlab, Windows Gauss, Unix Gauss, Windows console, and Unix console. The console versions are stand-alone, and do not run with mathematical applications. Users issue BACC commands directly at the operating system prompt. These versions are useful for debugging new model implementations, and as versions of last resort for those who do not have any of the supported mathematical applications. At the time of writing, versions for Windows Splus and Unix Splus are in progress.

Model implementation code defines a model and includes routines for prior and posterior MCMC simulation. A model developer implements a new model for BACC by writing a new model implementation file, which is a C language source file with a structured collection of model-specific computational routines. Section 5 describes the implementation of models in more detail. The BACC Developer’s Manual, at the BACC website, does so in even more detail.

The matrix library contains useful matrix routines for model developers. Routines implement low-level matrix operations such as multiplication, inversion, and Cholesky decomposition, as well as higher level procedures for evaluating multivariate densities and drawing from multivariate distributions. Developers who do not find a matrix routine for a procedure they need can write the required procedure and add it to the matrix library. There is a particular matrix data structure used for all matrix manipulation, described in documentation in the BACC developer’s distribution. Matrix routines must be able to manipulate these structures to conform to BACC requirements.

The structure of BACC observes some important design principles. First, as little code as possible is application-specific. As far as possible, code implementing commands is core code. This facilitates both the extension of BACC to new mathematical applications and the implementation of new commands.

Second, as little code as possible is model-specific. As much as possible, the implementation of a model includes only code that is unique to that model. Tasks that fall to the core code include processing user input; doing I/O; memory allocation; scheduling proposal draws, density evaluations, and other calculations; and compiling simulation matrices. This makes the model developer’s task easier and less error-prone.

Third, implementing BACC as a dynamically linked library to be linked with a mathematical application gives the user a great deal of flexibility.

5. MODEL DEVELOPMENT IN BACC

This section describes how a model developer implements a model for BACC.

The model developer defines, in a highly structured way, the joint distribution of the known and unknown quantities, and two MCMC algorithms. The invariant distributions of these Markov chains are the proper prior and posterior distributions of the unknown quantities. Defining the MCMC algorithm for the prior is not obligatory, but prior simulation may well be useful. For example, it allows the user to compare prior and posterior distributions of various function of interest.

Suppose the model developer is implementing the normal linear model described in section 2. For the prior distribution, the model developer might choose to draw β and H from their respective priors.

For the posterior distribution, the model developer might choose a two-block Gibbs sampler based on the following conditional posterior distributions:

$$\beta|H, y, Z \sim N(\bar{\beta}, \bar{H}_\beta^{-1}), \quad H|\beta, y, Z \sim W(\bar{S}^{-1}, \bar{v}),$$

where

$$\bar{H}_\beta \equiv \underline{H}_\beta + Z'(H \otimes I_T)Z,$$

$$\bar{\beta} \equiv \bar{H}_{\beta}^{-1} [\underline{H}_{\beta} \beta + Z'(H \otimes I_T)y],$$

$$\bar{S} \equiv \underline{S} + [s_{ij}] \quad s_{ij} \equiv u_i' u_i \equiv (y_i - Z_i \beta)' (y_i - Z_i \beta),$$

$$\bar{\nu} \equiv \underline{\nu} + T.$$

The developer has available routines to do matrix manipulation and random number generation. This is described in more detail in section 6.1.

5.1. The Elements of a Model Specification Implementation

The elements of a model specification are *dimension parameters*, *known quantities*, *unknown quantities*, *auxiliary quantities*, and *blocks*.

Dimension parameters specify the dimensions of matrix-valued quantities. Examples include the number of observations in a data set, the number of individuals in a cross section, the number of equations in a system of equations, and the number of states in a Markov chain. In the example, T , k , and m are dimension parameters.

Known quantities include fixed prior parameters and observed data. The core BACC code makes no distinction. The user supplies values for the known quantities, and they remain constant during simulation. In the example, the known quantities are $\underline{\beta}$, \underline{H} , $\underline{\nu}$, \underline{S} , X , and y . Recall how the user supplied these values in the Matlab code of Section 3.

Unknown quantities may be model parameters, latent variables, or missing data. Again, there is no distinction. These quantities change value as the state of the Markov chain changes. The unknown quantities in the example are β and H .

Auxiliary quantities are any other quantities used in calculations. They may be sufficient statistics or useful intermediate quantities such as a vector of residuals. In the example, S , u , and \bar{H} may be defined as auxiliary quantities. In the current implementation of this model specification, there are 13 auxiliary quantities, including these three.

Known, unknown and auxiliary quantities are all implemented as BACC matrices, even if they are scalars or vectors.

The block is the unit of organization of the simulation routines. A block consists of a routine to draw from a proposal distribution, and possibly a function used to evaluate a Hastings ratio. Here, there are two prior blocks and two posterior blocks. In both cases, there is one block for β and another for H .

5.2. Computational Routines

The model developer implements a model specification by providing a structured collection of model-specific computational routines. Some of these routines may already be available in the BACC matrix library. Others may not, and so the model developer must write new routines.

Armed with the requisite routines, the model developer's task is more like filling out a form than programming. The core expects particular routines for particular tasks, and the model developer simply identifies which routines do which tasks.

The routines fall under four categories: density evaluation routines, auxiliary quantity calculation routines, unknown quantity simulation routines, and miscellaneous routines.

The BACC core uses density evaluation routines to evaluate the log prior density and log data density at given values of the unknown quantities. For each unknown quantity, the model developer provides two functions that sum to its log prior density. One may depend on the unknown quantity (for example an unnormalized density), but the other may not (for example, a normalizing factor). The latter is calculated only once, and this provides an efficient way to avoid the recalculation of integration constants. The sum of the log prior densities over all unknown quantities must be the joint log prior density, but the decomposition need not be into marginals. Similarly, the model developer also provides two functions that sum to the log data density.

The BACC core uses auxiliary quantity calculation routines to update the values of auxiliary quantities. For each auxiliary quantity, the model developer provides a routine that computes the value of that quantity. It is not the model developer's responsibility to update the auxiliary quantities when the Markov chain changes state. The BACC core updates auxiliary quantities automatically, and like the Unix "make" utility, does so when and only when necessary. This arrangement has many advantages. Choosing by hand when to update each auxiliary quantity is time consuming, error-prone, and inflexible: for example, if the model developer wanted to change the order of the blocks, then she would have to rethink the timing of the updates.

The BACC core uses unknown quantity simulation routines for simulating the two (prior and posterior) Markov chains. Each Markov chain consists of an initial distribution and a transition distribution. Draws from the initial distribution may be broken into one or more blocks, with each block representing a subset of the unknown quantities. For each block, the model developer provides a routine to draw a particular subset of the unknown quantities. The initial distribution may be degenerate: it may, for example, put probability one on the mode of the appropriate distribution. Draws from the transition distribution may also be broken into one or more blocks. For each block, the model developer provides a routine to draw from a proposal density, and possibly a function used to evaluate the Hastings ratio. If no such function is provided, it is assumed to be identically equal to unity (as, for example, for a pure Gibbs block).

Miscellaneous routines are used for such things as checking user input for correctness and performing parameter transformations.

6. RESOURCES FOR DEVELOPERS

This section discusses resources available for the model developer to facilitate model implementation. The Matrx (sic) Library consists of routines to perform useful matrix operations. Workspace support makes it easier for model developers to set up a programming environment in which to implement new model specifications. The Metropolis check offers a tool for developers to check the results of their simulations.

6.1. *The Matrix Library*

The Matrix Library is a library of routines for performing matrix operations. All matrix routines manipulate matrices conforming to the special BACC data structure. This data structure includes the dimensions of the matrix, its type (double precision floating-point or integer), and its value. Use of this structure allows the developer to treat a matrix as a single unit, and renders impossible many potential programming errors.

Some basic matrix operations in the library include multiplication, stacking, inversion and Cholesky decomposition.

There is a collection of routines to evaluate the normalization constant and density kernels of various multivariate distributions, including the multivariate normal, Wishart, and Dirichlet distributions. These routines are designed to be used “off the shelf” as the density evaluation routines described in 5.2.

There are many routines for random number generation, including routines to draw from the multivariate normal, Wishart and Dirichlet distributions. These routines are also designed for easy inclusion in model specification files.

The model developer will not find all the required matrix operations implemented in the Matrix library, and will need to write new routines. These routines can be added to the Matrix library, extending it for the benefit of others.

6.2. *Workspace Support*

The current developers of the BACC software use the Microsoft Visual C++ software development environment. For the convenience of developers, the BACC developer’s distribution for Windows includes not only the BACC source code, but also the supporting files (workspace and project files) that govern how BACC code is built. We have taken care to make the Windows distribution portable. A developer should be able to install the BACC developer’s distribution on any directory of any drive, and be able to build any Windows version of BACC.

The developer’s distribution for UNIX includes makefiles, which serve the same purpose as the workspace and project files, but are somewhat less convenient. We have taken care to use ANSI C and standard makefile features, and have tested BACC on different UNIX platforms.

6.3. *Metropolis Check*

The Metropolis Check is a facility that allows the model developer to check the posterior simulation results of a new model implementation. It is a simple random-walk Metropolis chain that has the same invariant distribution as the posterior. The model developer can check the results of a new posterior simulator against the results of this Metropolis chain by comparing estimates of posterior moments and marginal likelihoods. If the estimates of posterior moments and the marginal likelihood for the Metropolis chain are close to their counterparts for the model developer’s chain, relative to their standard errors, this is evidence in support of the claim that the model developer’s chain is working correctly. If they are not, this suggests that there is a problem with the model developer’s chain. The differences in the outputs of the two chains may be helpful in diagnosing the problem.

The Metropolis chain is run after the posterior simulation that it is intended to check. It starts at the posterior mean, and each candidate step has a multivariate normal distribution

with mean zero and covariance proportional to the posterior covariance. The user chooses the constant of proportionality. For unknown quantities such as stochastic matrices and precision matrices, whose natural dimensions are smaller than their number of elements, transformation routines are used.

Caveats apply to the use of the Metropolis Check. In particular, the Metropolis chain may not converge well. But it may still be useful in many situations.

7. CONCLUSION

We have given a brief introduction to the use of the BACC software and described the task of creating new models for it. Much more information is available at the BACC website, www.econ.umn.edu/~bacc. BACC allows users to use Bayesian tools in a very flexible, feature-rich environment. BACC allows developers to focus on writing model-specific code. It also shortens the time lag between creating new models and/or methods for Bayesian inference, and their widespread availability to applied scientists who are not programmers.

REFERENCES

- Gelfand, A. E., and Dey, D. K. (1994). Bayesian Model Choice: Asymptotics and Exact Calculations. *J. Roy. Statist. Soc. B* **56**, 501–514
- Geweke, J. (1999). Using Simulation Methods for Bayesian Econometric Models: Inference, Development, and Communication. *Econometric Reviews* **18**, 1–126
- Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994). A language and program for complex Bayesian modelling. *The Statistician* **43**, 169–78
- McCausland, W. J. (2000). Using the BACC Software for Bayesian Inference. *Mimeo*