R News

The Newsletter of the R Project

Volume 6/1, March 2006

Editorial

by Martyn Plummer and Paul Murrell

Welcome to the first issue of R News for 2006. This is my first issue as Editor-in-Chief and, in the spirit of "start as you mean to go on", I have had someone else do all the work. Martyn Plummer is guest editor for this special issue, which has a distinctly Bayesian flavour. Thanks to Martyn and his merry band of contributors and reviewers for getting this special issue together.

Normal service and regular features will resume with a regular issue following the release of R 2.3.0. For now, sit back, relax, and prepare to gorge yourself on a feast of Bayes.

Paul Murrell
The University of Auckland, New Zealand
paul.murrell@R-project.org

This special issue of R News is dedicated to Bayesian inference and Markov Chain Monte Carlo (MCMC) simulation. The choice of articles for this issue is subjective. We aim to give you a snapshot of some current work on Bayesian statistical computing in R without any claim to comprehensiveness. A broader view is provided by the CRAN task view on Bayesian inference maintained by Jong Hee Park. This currently lists 33 packages, some of which have been discussed in previous R News articles (Yan, 2004; Raftery et al., 2005).

MCMC has become the numerical method of choice for many Bayesians. Although computationally expensive, it enables highly complex probability models to be analyzed. This capability, combined with cheap, abundant computing power, has contributed to the increasing popularity of Bayesian methods in applied statistics. The field has now matured to the point where most users should not expect to write custom software, but may use one of several existing "sampling engines". R provides a natural front end for these engines. This is nicely illustrated by the package **MCMCpack**, subject of the first article by Andrew Martin and Kevin Quinn. The computational back-end of **MCMCpack** is provided by the Scythe C++ statistical library. The front end is a collection of R functions for fitting models commonly used in the social and behavioural sciences. This is followed by an article on **coda**, which provides the infrastructure for analyzing MCMC output, and is used by **MCMCpack** among other packages.

Samantha Cook and Andrew Gelman then discuss validation of MCMC software using the **BayesValidate** package. Since the results of MCMC simulation are never exact, and the models used are typically very complex, validation of the sampling software can be quite difficult. However, the lack of validation of Bayesian software may be holding back its acceptance in regulated environments such as clinical trials approved by the US Food and Drug Administration, so this is an important problem.

We continue with a pair of articles on OpenBUGS. The BUGS (Bayesian inference Using Gibbs

Contents of this issue:

Sampling) project is a long-running project to provide a user-friendly language and environment for Bayesian inference. The first article, by Andrew Thomas and colleagues, describes the **BRugs** package which provides an R interface to the OpenBUGS engine. The second article by Andrew Thomas describes the BUGS language itself and the design philosophy behind it. Somewhat unusually for an article in R News, this article does not describe any R software, but it is included to highlight some of the differences in the way statistical models are represented in R and OpenBUGS.

The issue ends with an article by Jouni Kerman and Andrew Gelman, who give a personal perspective on what the next generation of Bayesian software may look like, and preview some of their own work in this area, notably the **rv** package for representing simulation-based random variables, and the forthcoming "Universal Markov Chain Sampler" package, **Umacs**.

Bibliography

A. E. Raftery, I. S. Painter, and C. T. Volinsky. BMA: An R package for bayesian model averaging. *R News*, 5(2):2–8, November 2005. URL `http://CRAN.R-project.org/doc/Rnews/`. 1

J. Yan. Fusing R and BUGS through Wine. *R News*, 4(2):19–21, September 2004. URL `http://CRAN.R-project.org/doc/Rnews/`. 1

Martyn Plummer
International Agency for Research on Cancer
Lyon, France
`plummer@iarc.fr`

Applied Bayesian Inference in R using MCMCpack

by Andrew D. Martin and Kevin M. Quinn

Introduction

Over the past 15 years or so, data analysts have become increasingly aware of the possibilities afforded by Markov chain Monte Carlo (MCMC) methods. This is particularly the case for researchers interested in performing Bayesian inference. Here, MCMC methods provide a fairly straightforward way for one to take a random sample approximately from a posterior distribution. Such samples can be used to summarize any aspect of the posterior distribution of a statistical model. While MCMC methods are extremely powerful and have a wide range of applicability, they are not as widely used as one might guess. At least part of the reason for this is the gap between the type of software that many applied users would like to have for fitting models via MCMC and the software that is currently available. **MCMCpack** (Martin and Quinn, 2005) is an R package designed to help bridge this gap.

Until the release of **MCMCpack**, the two main options for researchers who wished to fit a model via MCMC were to: a) write their own code in R, C, FORTRAN, etc., or b) write their own code (possibly relying heavily on the available example programs) using the BUGS language[1] in one of its various implementations (Spiegelhalter et al., 2004; Thomas, 2004; Plummer, 2005). While both of these options offer a great deal of flexibility, they also require non-trivial programming skills in the case of a) or the willingness to learn a new language and to develop some modest programming skills in the case of b). These costs are greater than many applied data analysts are willing to bear. **MCMCpack** is geared primarily towards these users.

The design philosophy of **MCMCpack** is quite different from that of the BUGS language. The most important design goal has been the implementation of MCMC algorithms that are model-specific. This comports with the manner in which people oftentimes think about finding software to fit a particular class of models rather than thinking about writing code from the ground up. The major advantage of such an approach is that the sampling algorithms, being hand-crafted to particular classes of models, can be made dramatically more efficient than black box approaches such as those found in the BUGS language, while remaining robust to poorly conditioned or unusual data. All the **MCMCpack** estimation routines are coded in C++ using the Scythe Statistical Library (Martin et al., 2005). We also think it is easier to call a single R function to fit a model than to code a model in the BUGS language. It should also be noted that **MCMCpack** is aimed primarily at so-

[1]The BUGS language is a general purpose language for simulation from posterior distributions of statistical models. BUGS exploits conditional independence relations implied by a particular graphical model in order to automatically determine an MCMC algorithm to do the required simulation. In order to fit a model, the user must specify a graphical model using either the BUGS language or (in the case of WinBUGS) a graphical user interface.

cial scientists. While some models (linear regression, logistic regression, Poisson regression) will be of interest to nearly all researchers, others (various item response models and factor analysis models) are especially useful for social scientists.

While we think **MCMCpack** has definite advantages over BUGS for many users, we emphasize that we view BUGS and **MCMCpack** as complimentary tools for the applied researcher. In particular, the greater flexibility of the BUGS language is perfect for users who need to build and fit custom probability models.

Currently **MCMCpack** contains code to fit the following models: linear regression (with Gaussian errors), a hierarchical longitudinal model with Gaussian errors, a probit model, a logistic regression model, a one-dimensional item response theory model, a K-dimensional item response theory model, a normal theory factor analysis model, a mixed response factor analysis model, an ordinal factor analysis model, a Poisson regression, a tobit regression, a multinomial logit model, a dynamic ecological inference model, a hierarchial ecological inference model, and an ordered probit model. The package also contains densities and random number generators for commonly used distributions that are not part of the standard R distribution, a general purpose Metropolis sampling algorithm, and some utilities for visualization and data manipulation. The package provides modular random number generators, including the L'Ecuyer generator which produces independent substreams, thus making (embarrassingly) parallel simulation using **MCMCpack** possible.

In the remainder of this article, we illustrate the user interface and functionality of **MCMCpack** with three examples.

User interface

The model fitting functions in **MCMCpack** have been written to be as similar as possible to the corresponding R functions for estimation of the models in question. This largely eliminates the need to learn a specialized model syntax for anyone who is even a novice user of R. For example, to fit a linear regression model with an improper uniform prior on the coefficient vector, an inverse gamma prior with shape and scale both equal to 0.0005 for the error variance, and the default settings for the parameters governing the MCMC algorithm, one would issue a function call nearly identical to the `lm()` command.

As an example, consider the `swiss` data that contains fertility and socioeconomic indicators from the 47 French-speaking provinces of Switzerland in 1888. To fit a Bayesian linear regression of fertility on a number of predictors in the dataset we use the Gibbs sampling algorithm to obtain a sample approximately from the appropriate posterior distribution. To do this with **MCMCpack** and to then summarize the results we issue the following command:

```
> data(swiss)
> posterior1 <- MCMCregress(Fertility ~
+                   Agriculture + Examination +
+                   Education + Catholic +
+                   Infant.Mortality,
+                   data=swiss)
> summary(posterior1)
```

The `MCMCregress()` function has a syntax similar to the `lm()` command. All model functions in **MCMCpack** return `mcmc` objects as defined by the **coda** package (Plummer et al., 2005). **MCMCpack** relies on coda to perform posterior summarization and convergence diagnostics on the simulated values. The summary method for `mcmc` objects prints various quantities of interest to the screen, including the posterior mean, standard deviation, and quantiles. The **coda** package provides a number of other facilities, including a plot method that produces marginal posterior kernel density plots and traceplots, and a suite of convergence diagnostics. See Figure 1 for the posterior summary for the Swiss fertility regression.

We note that diagnosing convergence is *critical* for any application employing MCMC methods. While we have ignored such diagnostics here for reasons of space, please note that simulation run lengths used in all of the examples have been chosen so that inferences are accurate.

Latent variable models in MCMCpack

One very active area of research in the field of political methodology involves modeling voting in committees using the spatial voting model. This explanation of voting asserts that actors have a preferred policy position (usually called an ideal point) in a K-dimensional issue space. For example, many European parliamentary systems are characterized by a two-dimensional model, with one dimension representing traditional left-right economic considerations, and the other dimension representing the issue of European integration. Voters cast votes on binary choices—one representing the status quo, the other an alternative policy. The goal of these models is to recover the ideal points of the actors, and a set of item-specific parameters that are functions of the alternative and status quo positions.

Under certain sets of assumptions, these empirical spatial voting models are the same as item response theory (IRT) models used in educational testing and psychometrics. The Bayesian approach to fitting these latent variable models provides many advantages over the frequentist approach. Model estimation is reasonably easy using data augmentation, identification of the model is straightforward,

```
> summary(posterior1)

Iterations = 1001:11000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 10000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

                    Mean       SD  Naive SE Time-series SE
(Intercept)      67.0208 11.08133 0.1108133      0.1103841
Agriculture      -0.1724  0.07306 0.0007306      0.0007149
Examination      -0.2586  0.26057 0.0026057      0.0024095
Education        -0.8721  0.18921 0.0018921      0.0017349
Catholic          0.1040  0.03602 0.0003602      0.0002965
Infant.Mortality  1.0737  0.39580 0.0039580      0.0042042
sigma2           54.0498 12.68601 0.1268601      0.1566833

2. Quantiles for each variable:

                     2.5%      25%     50%      75%    97.5%
(Intercept)      45.53200 59.56526 67.0600 74.31604 88.87071
Agriculture      -0.31792 -0.22116 -0.1715 -0.12363 -0.02705
Examination      -0.76589 -0.43056 -0.2579 -0.08616  0.24923
Education        -1.24277 -0.99828 -0.8709 -0.74544 -0.49851
Catholic          0.03154  0.08008  0.1037  0.12763  0.17482
Infant.Mortality  0.28590  0.81671  1.0725  1.33767  1.85495
sigma2           34.57714 45.06332 52.3507 61.03743 83.85127
```

Figure 1: Posterior summary from coda for the Swiss fertility regression fit using `MCMCregress()`

auxiliary information can be included in the analysis through the use of priors, and one can discuss quantities of interest on the scale of probability (Clinton et al., 2004; Martin and Quinn, 2002). **MCMCpack** contains a number of latent variable models, including one-dimensional and K-dimensional IRT models and factor analysis models for continuous, ordinal, and mixed data.

To illustrate the one-dimensional IRT model in **MCMCpack**, we will use some data from the U.S. Supreme Court. **MCMCpack** contains a dataset (`SupremeCourt`) of the votes cast by the nine sitting justices on the 43 non-unanimous cases decided during the October 2000 term. The data are just a (43×9) matrix of zeros, ones, and missing values. To identify the polarity of the model we constrain the ideal points of two justices in our one-dimensional latent space. We constrain Justice Stevens (a well-known liberal) to have a negative ideal point, and Justice Scalia (perhaps the most conservative members of the Court) to have a positive ideal point. (The one-dimensional IRT model in **MCMCpack** is identified through constraints on the ideal points / subject abilities while the K-dimensional model is identified through constraints in the item parameters). We use the default priors on the item and subject parameters. To fit the model, we issue the following command:

```
> data(SupremeCourt)
> posterior2 <- MCMCirt1d(t(SupremeCourt),
+   theta.constraints=list(Stevens="-",
+   Scalia="+"), burnin=5000, mcmc=100000,
+   thin=10, verbose=500)
```

By default, `MCMCirt1d` only retains the ideal points in the posterior sample. One can optionally store the item parameters. We illustrate our results from this sample analysis in Figure 2. This figure shows the standard normal prior density on the ideal points, and the marginal posterior densities of the ideal points. Justice Stevens is the leftmost Justice, followed by Breyer, Ginsburg, and Souter, who are essentially indistinguishable. Justice O'Connor is the pivotal median justice, closely followed by Justice Kennedy. Chief Justice Rehnquist is next, following by Justices Thomas and Scalia. The posterior sample can be used to answer any number of important substantive questions (see, for example, Clinton et al., 2004).

Generic metropolis sampling

MCMCpack model functions allow users to choose prior distributions by picking the parameters of a particular parametric family that is specific to each

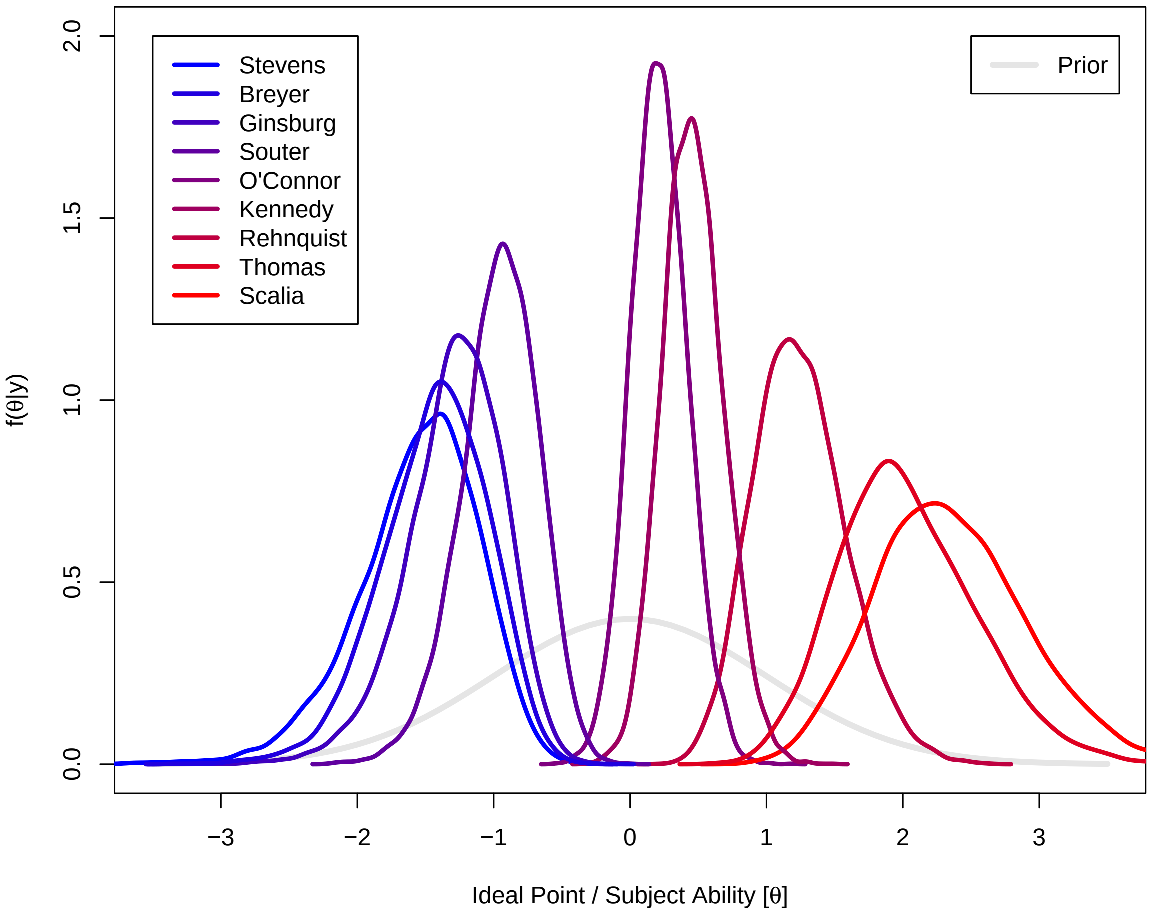

Figure 2: Posterior densities of ideal points for the U.S. Supreme Court justices, 2000 term, as estimated by `MCMCirt1d()`

model. This approach is reasonable for many applications, but at times users would like to use more flexible prior specifications. Similarly, there are numerous models that are not currently implemented *directly* in **MCMCpack**, but whose posterior densities are easy to write down up to a constant of proportionality. One area of **MCMCpack** that is currently under development is a set of functions to perform generic sampling from a user-supplied (log)-posterior distribution. While the user-supplied density is written as an R function, the simulation itself is performed in compiled C++ code, so it is much more efficient than doing the simulation in R itself.

While **MCMCpack** is not designed to be a general purpose sampling engine in the manner of BUGS, the ability to fit a models with relatively small numbers of parameters by specifying a (log)-posterior is very attractive to many social scientists who are accustomed to calculating maximum likelihood estimates using numerical optimization routines. For many of these researchers, writing an R function that evaluates a (log)-posterior is much more intuitive than specifying the equivalent graphical model in BUGS.

As an example, suppose one would like to fit a logistic regression model to the `birthwt` data from the **MASS** package (Venables and Ripley, 2002) with a complicated prior distribution. We assume our dichotomous dependent variable (the low birthweight indicator) $y_i \sim Bernouli(\pi_i)$ for observations $i = 1, \ldots, n$ with inverse-link function:

$$\pi_i = \frac{1}{1 + \exp(-x_i'\beta)}$$

The parameter vector β is of dimensionality $(p \times 1)$, and x_i is a column vector of observed covariates.

The data encodes risk factors associated with low birth weight. We prepare the data for analysis as follows:

```
> attach(birthwt)
> race <- factor(race, labels = c("white",
+  "black", "other"))
> ptd <- factor(ptl > 0)
> ftv <- factor(ftv)
> levels(ftv)[-(1:2)] <- "2+"
> bwt <- data.frame(low = factor(low), age,
+   lwt, race, smoke = (smoke > 0), ptd,
+   ht = (ht > 0), ui = (ui > 0), ftv)
> detach(birthwt)
```

We could obtain the maximum likelihood estimates with the command:

```
glm.out <- glm(low ~ ., binomial, bwt)
```

We will store these estimates and use them as starting values. We could also fit the model with `MCMClogit()` which assumes a multivariate normal prior on the β vector.

Suppose, however, that we would like to fit the model where our prior on β is:

$$p(\beta) \propto \mathbb{I}(\beta_6 > 0)\mathbb{I}(\beta_8 > \beta_7) \prod_{i=1}^{k} \frac{1}{2(1 + (\beta_i/2)^2)}$$

This is an independent Cauchy prior with location parameter 0 and scale parameter 2 truncated to a sub-region of the parameter space. See Geweke (1986) for some situations where such constraints may be of interest.

To fit this model with **MCMCpack**, one has to code the log-posterior density in R. This function can be written as:

```
> logit.log.post <- function(beta){
+   ## constrain smoking coefficient to be
+   ## greater than zero
+   if (beta[6] <=0) return(-Inf)
+
+   ## constrain coefficient on ht to be
+   ## greater than coefficient on ptd
+   if (beta[8] <= beta[7]) return(-Inf)
+
+   ## form posterior
+   eta <- X %*% beta
+   p <- 1.0/(1.0+exp(-eta))
+   log.like <- sum(Y * log(p) +
+      (1-Y)*log(1-p))
+   log.prior <- sum(dcauchy(beta,
+      0, 2, log=TRUE))
+   return(log.like + log.prior)
+ }
```

Note that the argument to the function only contains the parameter vector. The data must be in the environment from which the function is called, which is done automatically by the model-fitting function. See the documentation for details. To prepare the data for analysis, we will build a vector that holds y_i and a matrix that holds x_i:

```
> Y.vec <- as.numeric(bwt$low) - 1
> X.mat <- model.matrix(glm.out)
```

The function we will use to simulate from the posterior is `MCMCmetrop1R()`. This function samples in a single block from a user-defined (log)-density using a random walk Metropolis algorithm with a multivariate normal proposal distribution. To simulate from the posterior distribution of this model, we issue the command:

```
> posterior3 <- MCMCmetrop1R(logit.log.post,
+   theta.init=coef(glm.out), burnin=1000,
+   mcmc=200000, thin=20, tune=.7,
+   Y=Y.vec, X=X.mat, verbose=500)
```

Here we use the results from the `glm()` function as our starting values. We choose a tuning parameter to produce an acceptance rate of about 25%. The data are passed with the `Y=Y.vec, X=X.mat` options. `MCMCmetrop1R()` puts these into the appropriate environment such that the data are available to the function when performing simulation.

As with all **MCMCpack** model functions, this code returns an `mcmc` object. Here, to summarize the results, we first label our variables, and then summarize the posterior:

```
> varnames(posterior3) <- colnames(X.mat)
> summary(posterior3)
```

We report just the posterior medians and central 95% credible intervals from the coda summary in Figure 3.

```
                 2.5%      50%     97.5%
(Intercept) -1.24909  0.70661  2.918533
age         -0.10475 -0.03132  0.040915
lwt         -0.02965 -0.01595 -0.003622
raceblack    0.10524  1.10791  2.176933
raceother   -0.09613  0.73250  1.606134
smokeTRUE    0.11367  0.78682  1.587147
ptdTRUE      0.35179  1.19072  2.076352
htTRUE       1.01740  2.01728  3.313555
uiTRUE      -0.21642  0.67817  1.573814
ftv1        -1.35557 -0.41235  0.459527
ftv2+       -0.75103  0.14322  1.009499
```

Figure 3: Posterior medians and 2.5th and 97.5th percentiles from the constrained logistic regression model fit using `MCMCmetrop1R()`

Future developments

MCMCpack is a work in progress and under current active development. Going forward, we intend to implement more standard models—especially those used commonly in the social sciences. To illustrate the use of **MCMCpack** for applied problems, we will provide detailed vignettes and more datasets. We also intend to continue improving the flexibility of **MCMCpack**. One approach to this is to give users the ability to provide non-standard priors to any of the standard model fitting functions. We also plan to expand the number of general-purpose sampling functions. The website for the **MCMCpack** project (`http://mcmcpack.wustl.edu`) contains a more detailed list of things to come. We welcome comments and suggestions from the R community about **MCMCpack** and how we can make it a better tool for applied Bayesian inference.

Acknowledgements

We gratefully acknowledge support from the United States National Science Foundation (Grants SES-0350646 and SES-0350613), the Department of Political Science and the Weidenbaum Center at Washington University, and the Department of Government and the Institute for Quantitative Social Science at Harvard University. Neither the Foundation, Washington University, nor Harvard University bear any responsibility for this software.

Bibliography

J. Clinton, S. Jackman, and D. Rivers. The statistical analysis of roll call data. *American Political Science Review*, 98:355–370, 2004. 4

J. Geweke. Exact inference in the inequality constrained normal linear regression model. *Journal of Applied Econometrics*, 1(2):127–141, 1986. 6

A. D. Martin and K. M. Quinn. Dynamic ideal point estimation via Markov chain Monte Carlo for the U.S. Supreme Court, 1953-1999. *Political Analysis*, 10:134–153, 2002. 4

A. D. Martin and K. M. Quinn. *MCMCpack: Markov chain Monte Carlo (MCMC) Package*, 2005. URL http://mcmcpack.wustl.edu. R package version 0.6-3. 2

A. D. Martin, K. M. Quinn, and D. B. Pemstein. *Scythe Statistical Library*, 2005. URL http://scythe.wustl.edu. Version 1.0. 2

M. Plummer. *JAGS: Just Another Gibbs Sampler*, 2005. URL http://www-fis.iarc.fr/~martyn/software/jags/. Version 0.8. 2

M. Plummer, N. Best, K. Cowles, and K. Vines. *coda: Output Analysis and Diagnostics for MCMC*, 2005. URL http://www-fis.iarc.fr/coda/. R package version 0.9-2. 3

D. Spiegelhalter, A. Thomas, N. Best, and D. Lunn. *WinBUGS*, 2004. URL http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/. Version 1.4.1. 2

A. Thomas. *OpenBUGS*, 2004. URL http://mathstat.helsinki.fi/openbugs/. 2

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. 5

Andrew D. Martin
Department of Political Science
Washington University in St. Louis, USA
admartin@wustl.edu

Kevin M. Quinn
Department of Government
Harvard University, USA
kevin_quinn@harvard.edu

CODA: Convergence Diagnosis and Output Analysis for MCMC

by Martyn Plummer, Nicky Best, Kate Cowles and Karen Vines

At first sight, Bayesian inference with Markov Chain Monte Carlo (MCMC) appears to be straightforward. The user defines a full probability model, perhaps using one of the programs discussed in this issue; an underlying sampling engine takes the model definition and returns a sequence of dependent samples from the posterior distribution of the model parameters, given the supplied data. The user can derive any summary of the posterior distribution from this sample. For example, to calculate a 95% credible interval for a parameter α, it suffices to take 1000 MCMC iterations of α and sort them so that $\alpha_1 < \alpha_2 < \ldots < \alpha_{1000}$. The credible interval estimate is then $(\alpha_{25}, \alpha_{975})$.

However, there is a price to be paid for this simplicity. Unlike most numerical methods used in statistical inference, MCMC does not give a clear indication of whether it has converged. The underlying Markov chain theory only guarantees that the distribution of the output will converge to the posterior in the limit as the number of iterations increases to infinity. The user is generally ignorant about how quickly convergence occurs, and therefore has to fall back on *post hoc* testing of the sampled output. By convention, the sample is divided into two parts: a "burn in" period during which all samples are discarded, and the remainder of the run in which the chain is considered to have converged sufficiently close to the limiting distribution to be used. Two questions then arise:

1. How long should the burn in period be?

2. How many samples are required to accurately

estimate posterior quantities of interest?

The **coda** package for R contains a set of functions designed to help the user answer these questions. Some of these convergence diagnostics are simple graphical ways of summarizing the data. Others are formal statistical tests.

History of CODA

The **coda** package has a long history. The original version of **coda** (Cowles, 1994) was written for S-PLUS as part of a review of convergence diagnostics (Cowles and Carlin, 1996). It was taken up and further developed by the BUGS development team to accompany the prototype of WinBUGS now known as "classic BUGS" (Spiegelhalter et al., 1995). Classic BUGS had limited facilities for output analysis, but dumped the sampled output to disk, in a form now known as "CODA format", so that it could be read into **coda** for further analysis.

Later BUGS versions, known as WinBUGS (Spiegelhalter et al., 2004), had a sophisticated graphical user interface which incorporated all of the features of **coda**. However, as the name suggests, WinBUGS only ran on Microsoft Windows (until the recent release of its successor OpenBUGS which also runs on Linux on the x86 platform). BUGS users on UNIX and Linux were either limited to using classic BUGS or they developed their own MCMC software, and a residual user base for **coda** remained.

The **coda** package for R arose out of an attempt to port the **coda** suite of S-PLUS functions to R. Differences between S-PLUS and R made this difficult, and the porting process ended with a more substantial rewrite. Likewise, changes in S-PLUS 5.0 meant that **coda** ceased to run on S-PLUS [1], and an initial patch by Brian Smith, led to a complete rewrite known as **boa** (Bayesian Output Analysis), which has subsequently been ported to R (Smith, 2005).

MCMC objects

S-PLUS **coda** had a menu-driven interface aimed at the casual S-PLUS user. The menu interface was retained in the R package as the `codamenu()` function, but one of the design goals was to build this interface on top of an object-based infrastructure so that the diagnostics could also be used on the command line. A new class called `mcmc` was created to hold MCMC output. The `mcmc` class was designed from the starting point that MCMC output can be viewed as a time series. More precisely, MCMC output and time series share some characteristics, but there are important differences in the way they are used.

- An MCMC time series evolves in discrete time (measured in iterations) and time is always positive.
- The time series is not assumed to be stationary. In fact the primary goal of convergence diagnosis is to identify and remove any non-stationary parts from the beginning of the series. *A priori* an MCMC time series is more likely to be stationary at iteration 10000 than at iteration 1.
- An MCMC time series is artificially generated. This means it can be extended, if necessary. It can also be replicated. A replicated time series arises from a so-called "parallel" chain, derived from the same model, but using different starting values for the parameters and a different seed for the random number generator.
- The autocorrelation structure of the time series is a nuisance. A maximally informative series of a given length has no autocorrelation: each iteration is an independent sample from the posterior distribution. In order to obtain such a series we may choose to lengthen the MCMC run by a factor of n and take every nth iteration, a process known as "thinning".

To reflect this close relation with time series, `mcmc` objects have methods for the generic time series functions `time`, `start`, `end`, `frequency`, and `window`. The `thin()` function is used to extract the "thinning interval", *i.e.* the number of iterations between consecutive samples in a chain that has been thinned. The `window()` function is used to get a subset of iterations from an `mcmc` object, usually by removing the inital part of the chain, or increasing the thinning interval.

```
x <- window(x, start=100, thin=5)
```

Numeric vectors or matrices in R can be converted to `mcmc` objects using the `mcmc()` function, and `mcmc` objects representing parallel chains can be combined with `mcmc.list()`. As the name suggests, the `mcmc.list()` function returns a list of `mcmc` objects, but it also checks that each component of the list contains data on the same variables over the same set of iterations. It is not sufficient to combine parallel chains using the `list()` function, since functions in the **coda** package require the presence of the `mcmc.list` class attribute as proof of consistency between the list components.

[1]It was no longer possible to use a replacement function on an object unless that object already existed, a language feature also shared by R.

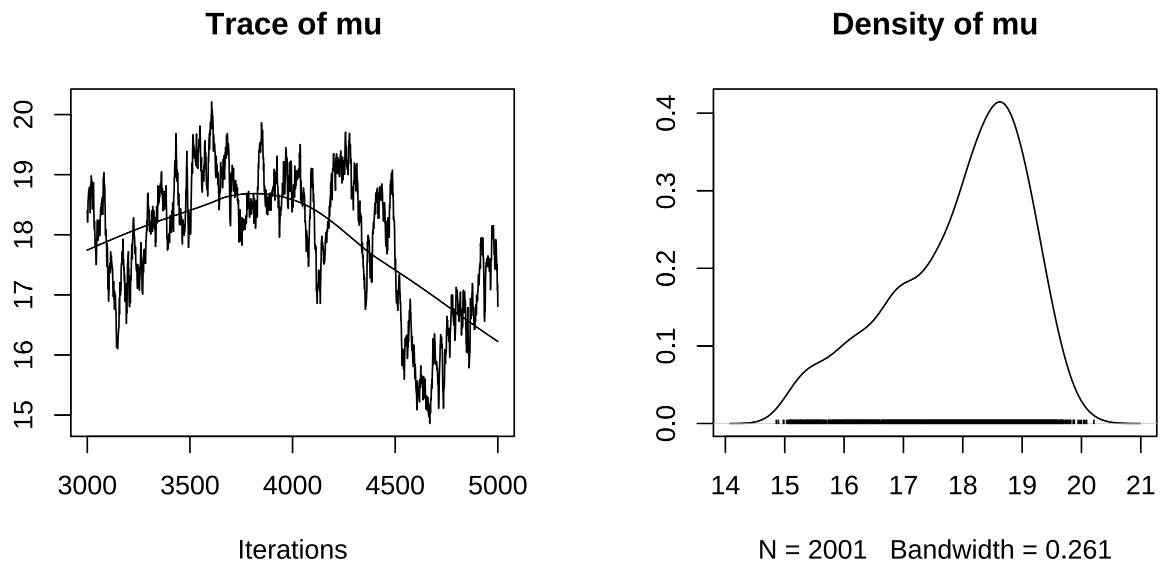

Figure 1: Example of a trace plot and density plot produced by the plot method for `mcmc` objects.

Reading MCMC data into R

Externally generated MCMC output can be read into R from files written in CODA format. In this format, each parallel chain has its own *output file* and there is a single *index file*. The `read.coda()` function reads output from an output/index file pair and returns an `mcmc` object.

Short-cut functions are provided for output from JAGS (Plummer, 2005) and OpenBUGS. In JAGS, the output file is, by default, called 'jags.out' and the index file 'jags.ind'. A call to `read.jags()`, without any arguments, will read the data in from these files. In OpenBUGS, the index file is, by default, 'CODAindex.txt', and the output files are 'CODAchain1.txt', 'CODAchain2.txt', *etc.*. The `read.openbugs()` function reads these files and returns an `mcmc.list` object containing output from all chains.

Graphics

The **coda** package contains several graphics functions for visualising MCMC output. The graphical output from plotting functions is quite extensive. A separate plot is produced for each scalar parameter, and for each element of a vector or array parameter. A single function call can thus create a large number of plots. In order to make the plotting functions more user-friendly, an appropriate multi-frame layout is automatically chosen and interactive plotting devices are paused in between pages.

The `plot` method for the `mcmc` class creates two plots for each parameter in the model, illustrated in Figure 1. The first is a trace plot, which shows the evolution of the MCMC output as a time series. The second is a density plot, which shows a kernel density estimate of the posterior distribution. Trace plots are useful for diagnosing very poor mixing, a phenomenon in which the MCMC sampler covers the support of the posterior distribution very slowly. Figure 1 shows an extreme example of this. Poor mixing invalidates the kernel density estimate, as it implies that the MCMC output is not a representative sample from the posterior distribution. The density plots produced by **coda** have some useful features: distributions that are bounded on $[0, 1]$ or $[0, \infty)$ are recognized automatically and the density plots are modified so that the smooth density curve does not spill over the boundaries. For integer-valued parameters, a bar plot is produced instead of a density plot.

Two additional plotting functions allow the correlation structure of the parameters to be explored. The function `autocorr.plot()` produces an `acf` object from the MCMC output and plots it. The resulting plot can be useful in identifying slow mixing, and may suggest a suitable thinning interval for the sample to attain a sequence of approximately independent samples from the posterior. The function `crosscorr.plot()` shows an image of the posterior correlation matrix. It identifies parameters that are highly correlated (a frequent cause of slow mixing when using Gibbs sampling) and may suggest reparameterization of the model, or the use of an sampling method that updates these parameters together in a block. Figure 2 shows `crosscorr.plot()` output from the same example as Figure 1. It is clear that there is a strong negative correlation between `mu` and `alph[1]`.

Further plotting functions are available in the **coda** package. In particular, Lattice plots have recently been added by Deepayan Sarkar.

Summary statistics

The `summary` method for the `mcmc` class prints a fairly verbose summary of each parameter, giving the mean, standard deviation, standard error of the mean and a selection of quantiles.

Calculation of the standard error of the mean requires estimating the spectral density of the mcmc series at zero. This is done by the low-level function `spectrum0()`, which is also used by several other functions in **coda**. It uses a variation of the estimator proposed by Heidelberger and Welch (1981) and fits a generalized linear model to the lower part of the periodogram. Unfortunately MCMC outout can have extremely high autocorrelation, which may cause `spectrum0()` to crash. A more robust estimator, based the best-fitting autoregressive model, is provided by the function `spectrum0.ar()`.

One of the most important uses of `spectrum0.ar()` is in the function `effectiveSize()`. This answers the question "How many independent samples from the posterior distribution contain the same amount of information?". In the example illustrated in Figure 1 there are 3000 sampled iterations, but the "effective size" of the sample is only 6.9, clearly inadequate for any further inference.

Formal convergence tests

There are four formal convergence tests at the core of the **coda** package. A brief explanation of the underlying theory is given on the corresponding help pages along with appropriate references, so the details will not be repeated here. Briefly, `geweke.diag()` and `gelman.diag()` aim to diagnose lack of convergence using a single chain and multiple parallel chains, respectively. These functions also have graphical versions that show how convergence is improved by discarding extra burn-in iterations at the beginning of the series. The other two diagnostics are designed for run length control based on accurate estimation of the mean (`heidel.diag()`) or a quantile (`raftery.diag()`).

Outlook

Although the **coda** package continues to evolve incrementally, its core functionality has not substantially changed in the last 12 years. This is largely due to the lack of integration between between **coda** and the underlying MCMC engine, which means that **coda** must fall back on *post hoc* analysis of the output, assuming nothing about how it was generated. Closer integration of MCMC engines into R would enable R functions to interrogate the transition kernel of the Markov chain and get better estimates of convergence rates. Conversely, run length control could be done automatically from R. Both of these changes would improve the practice of Bayesian data analysis. Currently, the use of MCMC methods imposes an extra burden on the user to check for nonconvergence of the MCMC output before it can be used. Not only does this create extra work, it is also a distraction from the more important process of model criticism. Eventually this layer of complexity may be hidden from the user.

Acknowledgements

Many people have provided useful feedback and extensions to the **coda** package. In particular we would like to thank Deepayan Sarkar, Russel Almond, Douglas Bates and Andrew Martin for their contributions to **coda**.

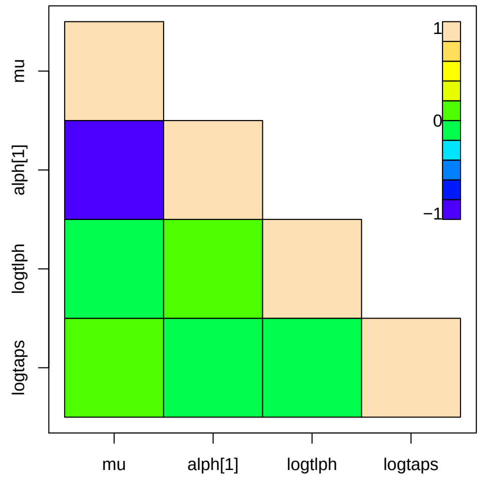

Figure 2: Example output from the function `crosscorr.plot`

Bibliography

M. K. Cowles. *Practical issues in Gibbs sampler implementation with application to Bayesian hierarchical modelling of clinical trial data*. PhD thesis, Division of Biostatistics, University of Minnesota, 1994. 8

M. K. Cowles and B. P. Carlin. Markov Chain Monte Carlo diagnostics: a comparative review. *J. Am. Statist. Ass.*, 91:883–904, 1996. 8

P. Heidelberger and P. D. Welch. A spectral method for confidence interval generation and run length control in simulations. *Communications of the ACM*, 24:233–245, 1981. 10

M. Plummer. *JAGS 0.90 User Manual*. IARC, Lyon, September 2005. URL http://www-ice.iarc.fr/~martyn/software/jags. 9

B. J. Smith. The boa package. 2005. URL http://www.public-health.uiowa.edu/boa/. 8

D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks. *BUGS 0.5: Bayesian inference Using Gibbs Sampling - Manual (version ii)*. Medical Research Council Biostatistics Unit, Cambridge, Cambridge, 1995. 8

D. Spiegelhalter, A. Thomas, N. Best, and D. Lunn. *WinBUGS user manual, version 2.0*. Medical Research Council Biostatistics Unit, Cambridge, Cambridge, June 2004. URL http://www.math.stat.helsinki.fi/openbugs. 8

Martyn Plummer
International Agency for Research on Cancer
Lyon, France
plummer@iarc.fr

Nicky Best
Department of Epidemiology and Public Health
Faculty of Medecine
Imperial College
London, UK

Kate Cowles
Department of Biostatistics, College of Public Health
The University of Iowa, USA

Karen Vines
Department of Statistics
The Open University
Milton Keynes, UK

Bayesian Software Validation

by Samantha Cook and Andrew Gelman

BayesValidate is a package for testing Bayesian model-fitting software. Generating a sample from the posterior distribution of a Bayesian model often involves complex computational algorithms that are programmed "from scratch." Errors in these programs can be difficult to detect, because the correct output is not known ahead of time; not all errors lead to crashes or results that are obviously incorrect. Software is often tested by applying it to data sets where the "right answer" is known or approximately known. Cook et al. (2006) extend this strategy to develop statistical assessments of the correctness of Bayesian model-fitting software; this method is implemented in **BayesValidate**. Generally, the validation method involves simulating "true" parameter values from the prior distribution, simulating fake data from the model, performing inference on the fake data, and comparing these inferences to the "true" values. Geweke (2004) presents an alternative simulation-based method for testing Bayesian software.

More specifically, let $\theta^{(0)}$ represent the "true" parameter value drawn from the prior distribution $p(\theta)$. Data y are drawn from $p(y|\theta^{(0)})$, and the posterior sample of size L to be used for inference, $\theta^{(1)}, \ldots, \theta^{(L)}$, is drawn using the to-be-tested software. With this sampling scheme, $\theta^{(0)}$ as well as $\theta^{(1)}, \ldots, \theta^{(L)}$ are, in theory, draws from $p(\theta|y)$. If the Bayesian software works correctly, then, $\theta^{(0)}$ should look like a random draw from the empirical distribution $\theta^{(1)}, \ldots, \theta^{(L)}$, and therefore the (empirical) posterior quantile of $\theta^{(0)}$ with respect to $\theta^{(1)}, \ldots, \theta^{(L)}$ should follow a Uniform(0, 1) distribution. Testing the software amounts to testing that the posterior quantiles for scalar parameters of interest are in fact uniformly distributed.

One "replication" of the validation simulation consists of: 1) Generating parameters and data; 2) generating a sample from the posterior distribution; and 3) calculating posterior quantiles. Performing many replications creates, for each scalar parameter whose posterior distribution is generated by the model-fitting software, a collection of quantiles whose distribution will be uniform if the software works correctly. If N_{rep} is the number of replications and $q_1, q_2, \ldots, q_{N_{rep}}$ are the quantiles for a scalar parameter, the quantity $\sum_{i=1}^{N_{rep}} \left(\Phi^{-1}(q_i)\right)^2$ will follow a $\chi^2_{N_{rep}}$ distribution if the software works correctly, where Φ^{-1} represents the inverse normal cumulative distribution function (CDF). For each scalar parameter, a p-value is then obtained by comparing the sum of the transformed quantiles with the $\chi^2_{N_{rep}}$ distribution. **BayesValidate** analyzes each scalar parameter separately, but also creates combined summaries for each vector parameter; these scalar results and summaries are in the graphical output as well.

BayesValidate performs a specified number of replications and calculates a p-value for each scalar parameter. The function returns a Bonferroni-adjusted p-value and a graphical display of the z_θ statistics, which are the inverse normal CDFs of the p-values. Figures 1 and 2 show the graphical output for two versions of a program written to fit a simple hierarchical normal model with parameters σ^2, τ^2, μ, and $\alpha_1, \alpha_2, \ldots, \alpha_6$; one version correctly samples from the posterior distribution and one has an error.

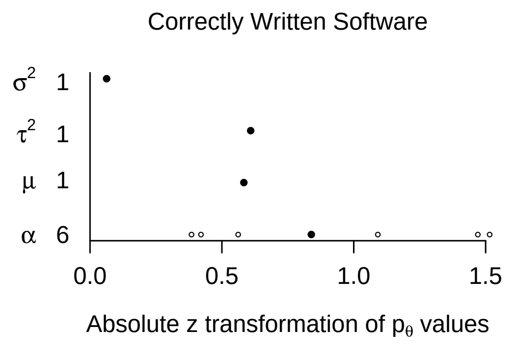

Figure 1: *z_θ statistics: Correctly written software. Each row represents a scalar parameter or batch of parameters; the circles in each row represent the z_θ statistics associated with that parameter or batch of parameters. Solid circles represent the z_θ statistics associated with the mean of that batch of parameters. The numbers on the y axis indicate the number of parameters in the batch. The z_θ statistics are all within the expected range for standard normal random variables.*

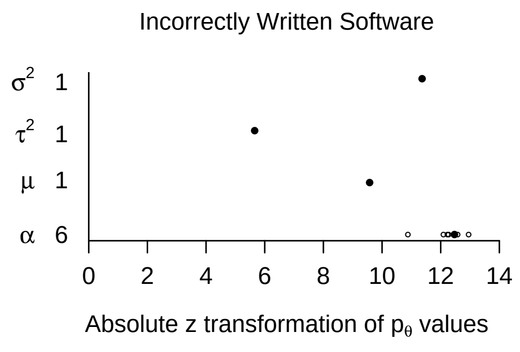

Figure 2: *z_θ statistics: Incorrectly written software (error sampling the parameter α). Each row represents a scalar parameter or batch of parameters; the circles in each row represent the z_θ statistics associated with that parameter or batch of parameters. Solid circles represent the z_θ statistics associated with the mean of that batch of parameters. The numbers on the y axis indicate the number of parameters in the batch. Values of z_θ larger than 2 indicate a potential problem with the software; this plot provides convincing evidence that the software has an error.*

Bibliography

S. Cook, A. Gelman, and D. B. Rubin. Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics*, 2006. To appear. 11

J. Geweke. Getting It Right: Joint Distribution Tests of Posterior Simulators. *Journal of the American Statistical Association*, 99:799–804, 2004. 11

Samantha Cook, Andrew Gelman
Department of Statistics
Columbia University, NY, USA

Making BUGS Open

by Andrew Thomas, Bob O'Hara, Uwe Ligges, and Sibylle Sturtz

BUGS[1] (Bayesian inference Using Gibbs Sampling, Spiegelhalter et al., 2005) is a long running software project aiming to make modern Bayesian analysis using Markov Chain Monte Carlo (MCMC) simulation techniques available to applied statisticians in an easy to use Windows package. With the growing realization of the advantages of Open Source software we decided to release the source code of the BUGS software plus full program level documentation on the World Wide Web[2]. We call this release OpenBUGS (Thomas, 2004). We hope the BUGS user community will be encouraged to correct, improve and extend this software.

We follow a brief outline of how the BUGS software works with a more detailed discussion of the software technology used during the development of BUGS. We then try and explain why BUGS was developed using non-standard tools. We hope to convince the reader that although unfamiliar, our tools are very powerful and simple to use.

[1]http://www.mrc-bsu.cam.ac.uk/bugs/
[2]http://mathstat.helsinki.fi/openbugs/

Much of the ease of use of the BUGS software comes from its graphical user interface and the idea of the compound document as a container for different types of information. However, much is to be gained by interfacing BUGS with other software. R has many useful built-in and contributed functions but as yet little in the way of Bayesian analysis tools. The **BRugs** interface to the BUGS software plus a small suite of R functions is an attempt to improve this situation.

All of these (**BRugs** interface, the whole OpenBUGS software, and the R functions) have been organized for the R users' convenience in an R package also called **BRugs**. This package is distributed over the CRAN network. Its current version (0.2-5) is only available for Windows.

In R, the user with Internet connection can simply type

```
R> install.packages("BRugs")
R> library("BRugs")
```

and then happily start sampling, benefiting from the strengths of both OpenBUGS and R.

The **R2WinBUGS** package by Sturtz et al. (2005) already provides an approach to connecting BUGS and R. This has the disadvantage that it is impossible to interact during processing/sampling by WinBUGS in any way. If you need Gibbs sampling in R on other operating systems than Windows, we recommend to take a look at JAGS (Just Another Gibbs Sampler) by Plummer (2005).

How BUGS works

The software creates lots of objects, wires the objects together and then gets the objects to talk to each other. More formally a dynamic data structure, a directed acyclic graph, of objects is build to represent the Bayesian model. This graph is able to exploit conditional independence assumptions to efficiently calculate conditional probabilities. A layer of updater objects is created to sample parameters of the model and copy them into the graph data structure. Finally a layer of monitor objects can be created to monitor (watch) the values of the sampled parameters and provide summary statistics for them.

How is the graph of objects built? The user writes a description of the Bayesian model in the BUGS language. This model description is also a description of the graph of objects that BUGS should build. A compiler turns the textual representation of the Bayesian model into the graph of objects. Objects of base class 'updater' have a method which is able to decide if objects of that particular class can (and should) act as updaters for a particular parameter in the model based on the functional form of its conditional distribution.

Compilation and inference

Compilation of the description of a Bayesian model in the BUGS language involves a number of stages. Firstly lexical analysis, scanning, is performed to break the stream of characters representing the model into tokens. Secondly syntactical analysis, parsing, is performed to build a tree representation of the model. Thirdly the graph of objects is constructed by a post order traversal of the parse tree with objects whose values have been observed marked as data. Finally conditional independence is used to produce lists of graph objects that when multiplied together calculate conditional distributions.

BUGS uses MCMC simulation algorithms to make inference. These algorithms are computationally expensive but robust to details of the problem they are applied to. This robustness is an important property in a system such as BUGS which automatically chooses the inference algorithm. BUGS is able to match a wide choice of MCMC algorithm, such as single site Gibbs, slice sampling and continuously adapting block Metropolis, to the model parameters that need updating.

Software development

BUGS is written in the language Component Pascal (CP) using the BlackBox Component Builder from *Oberon microsystems*[3]. CP is a very modern compiled language with both modular and object orientated features. The language is highly dynamic with runtime loading and linking of modules. Compiled modules contain meta information that allows the module loader to verify that the loaded module provides the services required by the client. It is also an extremely safe language because of its very strong type system and automatic heap management (garbage collection).

CP software typically consists of many unlinked modules plus a small executable or dynamic link library that is able to load modules as required. The modules are arranged as a directed acyclic graph under the import (make use of) relation. Loading a module causes all modules in the sub-graph to be loaded. Module initialization code is executed when a module is loaded. Modules are grouped into subsystems with the subsystem name used as a prefix to the module name. Physically modules are represented by files with the location and name of the file derived from the module name. Each subsystem is kept in a separate subdirectory while the executable (or dynamic link library) is kept in the root directory.

The BlackBox Component Builder comes with several subsystems of modules which make the development of graphical user interfaces simple. More

[3]http://www.oberon.ch/

novel is the idea of a compound document, an editable text document that is able to contain graphics views. Graphics views can be developed by extending a view class. Graphics views can be made editable and special purpose drawing tools such as DoodleBUGS can be easily developed. About one quarter of the modules comprising OpenBUGS implement the graphical user interface and various graphics views used for output.

These GUI modules are only available for 32-bit Windows. The package **BRugs** does not make use of any GUI module. All other modules in the OpenBUGS distribution can be used under Linux on x86 based platforms as well.

Metaprogramming

Metaprogramming is self awareness for software. Software can ask itself questions. For example does module `Foo` export an item called `Bar`? What sort of item is `Bar`? Can such a thing be done with `Bar`? More formally we can ask if a particular module is loaded. If the module is loaded we can examine its metadata and then query this metadata. For example we could ask if a module `Foo` is loaded and if not load the module. Then we could ask if module `Foo` contains a procedure `Bar` with say no parameters and if so to call (execute) this procedure. Note that this process is safe: we do not just hope that `Foo` contains a `Bar` of the right sort (with a crash if this is not so).

BUGS makes use of metaprogramming in many places. These uses of metaprogramming fall into two broad groups: program configuration and interfacing. In the first group are support for the BUGS language, loading sampling algorithms and loading data reading algorithms. In the second group construction of GUI interfaces, implementing a scripting language and interfacing to R.

Each time the BUGS language parser comes across the name of a distribution it uses metaprogramming to load the module that implements this distribution. The link between distribution name and module name is stored in a configuration file called 'grammar'. A list of modules implementing sampling algorithms is stored in a file. When BUGS starts up this file is read and the appropriate modules are loaded. Currently BUGS can read data in two formats: the S-PLUS (Insightful Corporation, 2004) format and rectangular format. Again the modules that implement reading these formats are loaded at program start up. Other data reading option such as from SQL tables could be added.

Metaprogramming makes construction of the widgets typical of a GUI simple. For example a button is just a region of a window which responds to a mouse click by executing a procedure (without parameters). A string containing the module and procedure names is associated with the button and when the mouse is clicked metaprogramming is used to load the module and execute the procedure. Note in this approach no code is written to represent the button.

In a scripting language, typing a command at a prompt causes the system to execute some action. This involves some sort of interpreter. This is easily written using metaprogramming. The command in the scripting language is a string which is mapped into a series of procedures in the CP language. Metaprogramming is then used to load and execute these procedures. For example the command `modelCheck(^0)` in the BUGS scripting language gets mapped to

```
BugsCmds.SetFilePath('^0');
BugsCmds.ParseGuard;
BugsCmds.ParseFile
```

where `^0` is a holder for a string. The mapping between commands in the BUGS scripting language and the corresponding CP procedure is stored in a file, making the language extensible.

BRugs: Interfacing to R

The R interface to OpenBUGS is realized by a very small dynamic link library 'brugs.dll' corresponding to the 'WinBUGS.exe'. It exports a couple of `.C()` entry points, among those several for direct access to the BUGS scripting language. This way, it is possible to realize R functions that are very similar to commands in the BUGS scripting language, not only sharing the same names (e.g. `modelCheck()`) but also sharing almost the same (order of) arguments. Therefore, it was possible to implement a huge number of R functions that allow almost full control of OpenBUGS in R.

Commands and some data are passed directly from R to OpenBUGS by `.C()` calls. Some information is passed back from OpenBUGS to R as the value from these calls, as it is common practice in R programming and interfaces. Unfortunately, we still have to pass back some other information and results of sampling using temporary text files that are imported into R by `readLines()`, `read.table()`, `scan()` and friends. Transparently reporting error messages from OpenBUGS to the R user is another topic that needs further improvement – currently we are sometimes relying on a good guess for generating error messages.

BRugs provides at least five kinds of functions:

- basic functions (such as `modelCheck()`) corresponding to the BUGS scripting language mentioned above,
- functions (e.g. `write.datafile()`) to prepare R data and inits (in the form of dataframes, for example) for OpenBUGS adapted from the **R2WinBUGS** package (Sturtz et al., 2005),

- high level functions such as `BRugsFit()` which run a whole simulation using only one function call,
- functions (e.g. `buildMCMC()`) to prepare the data for output analysis using the **coda** package (Plummer et al., 2005), and
- some internal help functions to read the temporary buffer file, for example.

Using these functions, it is possible to run an interactive sampling and analysis session in R where you can sample, calculate some (intermediate) results and make convergence diagnostics, and sample further on if required.

For example, Weihs and Ligges (2006) used this capability of **BRugs** for some MCMC optimization in the following manner. In principle, after each 50 or 100 iterations (of OpenBUGS), the convergence of the error rate of the underlying model was calculated using linear regression (in R). If the coefficient was no longer significantly negative (i.e. convergence of the error), the extremely computational expensive iterations could be stopped, otherwise iterations continued in OpenBUGS again.

A BRugs session

For demonstration of the use of **BRugs** we use a normal hierarchical model for the rats data that is used throughout the WinBUGS manual (Spiegelhalter et al., 2005). The example is originally taken from section 6 of Gelfand and Smith (1990).

The WinBUGS manual is available in HTML format documentation from within R by calling `help.WinBUGS()`. Analogously, `help.BRugs()` starts up the **BRugs** manual (Thomas, 2004). For references on R functions, the usual help files such as `?help.BRugs` for function `help.BRugs()` itself are available.

After loading the **BRugs** package by

```
R> library(BRugs)
```

we change the working directory to simplify file specification in the next steps:

```
R> oldwd <- getwd()
R> setwd(system.file("OpenBUGS", "Examples",
+                     package = "BRugs"))
```

To initialize a model, the user types functions corresponding to the BUGS scripting language instead of clicking buttons. First, the model has to be checked. The model file for the rats model is given by 'ratsmodel.txt':

```
R> modelCheck("ratsmodel.txt")
```

Of course, it is also possible to specify the file by the absolute path (using forward slashes). Afterwards, data have to be loaded by `modelData()`. This function takes a file name as argument. R objects (named list of data or a vector or list of object names) can be written to such a file using `bugsData()`. For example:

```
R> data(ratsdata)
R> modelData(bugsData(ratsdata))
```

If data are stored in more than one file, the argument can be a vector of files as well or the function has to be called successively.

Now it is time to compile the model. In this example, we use three chains to run the MCMC simulation.

```
R> modelCompile(numChains = 3)
```

Initial values can be specified by calls to the function `modelInits()`. For more than one chain, one can either call `modelInits()` with a character vector of more than one filename (one for each chain) or call the function successively for each file containing initial values. For random effect nodes, the function `modelGenInits()` can generate appropriate inits.

In order to write files that contain initial values as accepted by OpenBUGS and the `modelInits()` function, the function `bugsInits()` can be used. Its argument is a list with one element for each chain. Each element of this list is itself a list of starting values for the OpenBUGS model, *or* a function creating (possibly random) initial values.

Therefore, we demonstrate the use of these three different approaches to specify initial values, one for each chain:

```
R> data(ratsinits)
R> modelInits("ratsinits.txt")
R> modelInits(bugsInits(list(ratsinits),
+   fileName = tempfile()))
R> initfoo <- function() {
+   list(
+     alpha = rnorm(30, mean = 250, sd = 1),
+     beta = rnorm(30, mean = 6, sd = 1),
+     alpha.c = runif(1, 140, 160),
+     beta.c = 10,
+     tau.c = 1,
+     alpha.tau = 1,
+     beta.tau = 1)
+ }
R> modelInits(bugsInits(initfoo,
+                       fileName = tempfile()))
```

The model is initialized now and we start with 1000 updates as a burn-in period:

```
R> modelUpdate(1000)
```

By default, sampled parameter values are discarded by WinBUGS after each iteration unless the user explicitly requests that the values are stored for later use. This is done with the `samplesSet()` function before running the simulation for further 2000 iterations.

```
R> samplesSet(c("alpha", "beta"))
R> modelUpdate(2000)
```

To analyse the results of this simulation, we can take a look at the summary statistics, similar to clicking `stats` in the Sample Monitor tool within WinBUGS.

```
R> samplesStats("*")
```

An asterisk (`"*"`) can be entered instead of a node name as shorthand for all the stored samples.

All these calls can be performed conveniently by a single call to the meta function `BRugsFit()`:

```
R> BRugsFit(data = ratsdata,
+    inits = initfoo,
+    para = c("alpha", "beta"),
+    nBurnin = 1000, nIter = 2000,
+    modelFile = "ratsmodel.txt", numChains = 3,
+    working.directory =
+      system.file("OpenBUGS", "Examples",
+                  package = "BRugs"))
```

It returns a list containing the summary statistic as `samplesStats()` as well as the Deviance Information Criterion (DIC, Spiegelhalter et al., 2002), a Bayesian extension of the Akaike Information Criterion to hierarchical models. The DIC can also be imported into R by the low level functions `dicSet()` (for setting) and `dicStats()` (for getting).

`BRugsFit()` is only one wrapper function summarizing a couple of functions from the whole **BRugs** framework. Users might want to come up with their own wrapper functions fitting their own purposes, or some plot functions appropriate for their analyses, for example based on the code of `BRugsFit()`.

Plots known from WinBUGS are also provided by **BRugs**, for example history of the simulation (`samplesHistory()`), plots of autocorrelations (`samplesAutoC()`), plots of smoothed kernel density estimates (`samplesDensity()`), etc. Of course, graphical parameters may be passed as additional arguments to these plot functions.

As an example, we plot the smoothed kernel density estimates for the first 6 components of node `"alpha"` (figure 1):

```
R> samplesDensity("alpha[1:6]")
```

To finish this example session, we reset the working directory by

```
R> setwd(oldwd)
```

Outlook

OpenBUGS has made modern Bayesian inference software available in an Open Source package. The software is also open in the sense that it has been designed so that new features such as distributions, sampling methods, and user interfaces can be easily added. An OpenBUGS user has already contributed Component Pascal modules to implement the generalised extreme value and generalised pareto distributions.

The R package **BRugs** links components of OpenBUGS into R. This allows users to combine the strengths of both applications and make use of them interactively.

Unfortunately, currently **BRugs** is only available for Windows. We hope to provide a Linux version of BRugs shortly along with Component Pascal development tools for Linux. Until these tools become available to the public there is no alternative to distributing binary versions of the package for Windows and Linux.

Acknowledgments

The work of Uwe Ligges has been supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 475.

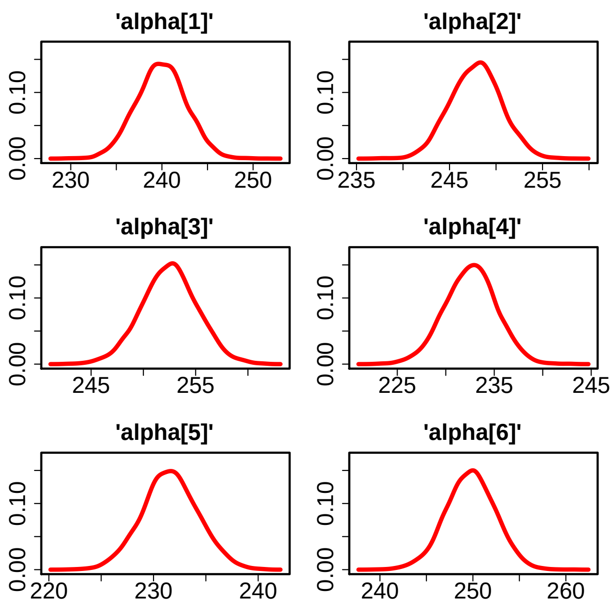

Figure 1: Density plot for the first 6 components of node `"alpha"`.

Bibliography

A. Gelfand and A. Smith. Sampling-based Approaches to Calculating Marginal Densities. *Journal of the American Statistical Association*, 85:398–409, 1990. 15

Insightful Corporation. *S-PLUS 6.2*. Insightful Corporation, Seattle, WA, USA, 2004. URL http://www.insightful.com. 14

M. Plummer. *JAGS: Version 0.90 manual*, 2005. URL http://www-ice.iarc.fr/~martyn/software/jags/. 13

M. Plummer, N. Best, K. Cowles, and K. Vines. *coda: Output Analysis and Diagnostics for MCMC*, 2005. R package version 0.10-3. 15

D. Spiegelhalter, N. Best, B. Carlin and A. van der Linde. Bayesian Measures of Complexity and Fit. *Journal of the Royal Statistical Society/B*, 64:583–639, 2002. 16

D. Spiegelhalter, A. Thomas, N. Best, and D. Lunn. *WinBUGS: User Manual, Version 2.10*. Medical Research Council Biostatistics Unit, Cambridge, 2005. 12, 15

S. Sturtz, U. Ligges, and A. Gelman. R2WinBUGS: A Package for Running WinBUGS from R. *Journal of Statistical Software*, 12(3):1–16, 2005. URL http://www.jstatsoft.org/. 13, 14

A. Thomas. *BRugs User Manual, Version 1.0*. Dept of Mathematics & Statistics, University of Helsinki, 2004. 12, 15

C. Weihs and U. Ligges. Parameter Optimization in Automatic Transcription of Music. In M. Spiliopoulou, R. Kruse, A. Nürnberger, C. Borgelt, and W. Gaul, editors, *From Data and Information Analysis to Knowledge Engineering*, pages 740–747, Berlin, 2006. Springer-Verlag. 15

Andrew Thomas, Bob O'Hara
Department of Mathematics & Statistics
University of Helsinki, Finland
ant@rni.helsinki.fi, bob.ohara@helsinki.fi

Uwe Ligges, Sibylle Sturtz
Fachbereich Statistik, SFB475
Universität Dortmund, Germany
<ligges,sturtz>@statistik.uni-dortmund.de

The BUGS Language

by Andrew Thomas

The BUGS language is a computer language not unlike the S language (Becker et al., 1988) in appearance, but it has a very different purpose.

Statistical models must be described before they can be used. A language to describe statistical models is needed by both the users of the model and the software that makes inference about the model. The language should be a formal language with well defined rules which can be processed automatically. It should not be concerned with the technology used to make inference about the model. We have developed a model description language called the BUGS language because of its use in the **B**ayesian inference **U**sing **G**ibbs **S**ampling (OpenBUGS) package. However, the BUGS language can be used outside the OpenBUGS software. For example, it is used in the JAGS package (Plummer, 2005) and has influenced other packages such as Bassist (Toivonen et al., 1999) and AUTOBAYES (Fisher and Schumann, 2003).

We choose to describe statistical models in terms of a joint probability distribution. Model description in terms of a joint probability distribution is both very general and very explicit. We consider these good points. We do not consider it a good idea to have a patchwork of specialized (maybe very elegant) notations for different types of model. We want to be able to combine small submodels to build larger models using a consistent notation. A small change to a model should not lead to a large change in the way that model is described. Examples of small changes to the model are: choice of sampling distribution, form of regression, covariate measurement error, missing data, interval censoring, *etc*. Explicitness is important in a model description language. There should be no doubt if two models are the same.

We hope the BUGS language will be useful to anyone who uses complex statistical models, and even to people who do not want to use the OpenBUGS package to make inference. Is the BUGS language really about statistics? OpenBUGS has many users who do not think of themselves primarily as statisticians, who are mainly interested in the deterministic skeleton of a model. We think that if a probabilistic model is used to explain observations, given this deterministic skeleton, that this is a form of statistics.

Influences

Formal languages have rules both for syntax and for semantics. For the BUGS language, syntax has been influenced by the S language (Becker et al., 1988) and semantics by graphical models. A model described in the BUGS language looks like a piece of S code but the meaning is completely different. The BUGS language is declarative. It describes static relations beween quantities, not how to do calculations.

Many joint probability distributions can be written as a product of factors. This leads to a graphical notation for describing joint probability distributions. Each factor in the joint probability distribution is a function of several variables. It is possible to order these variables for each factor so that the factor is represented by a node in a directed acyclic graph (DAG) labeled by one of the variables of the factor with the remaining variables being parents of the node in the graph. Describing the DAG is equivalent to describing the joint probability distribution. A DAG can be described by specifying the parents of each node. In the simplest case, the factors are probability distribution functions whose parameters are given by the values of the node's parents. In the more general case the parameters of the distribution will be functions of the values of the node's parents.

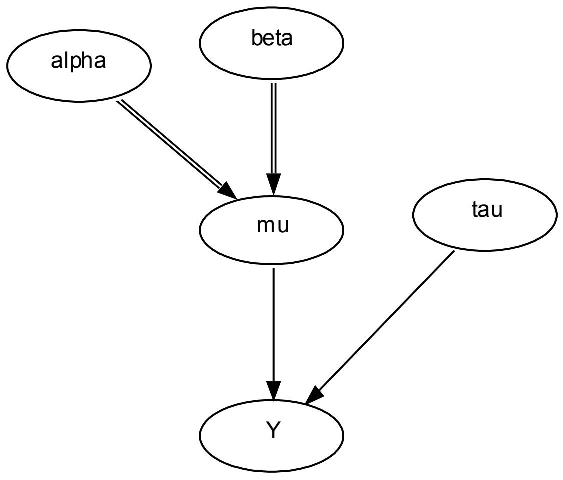

Figure 1: A simple directed acyclic graph

Consider the small graph in figure 1. This represents the joint probability distribution

```
P1(Y|mu, tau) P2(alpha) P3(beta) P4(tau)
```

where P1, P2, P3 and P4 are probability distributions associated with nodes in the graph. Nodes with solid arrows pointing into them represent stochastic relations and those with hollow arrows logical relations. Hence mu is some function of the values of alpha and beta.

Example 1: Growth curve in rats

An example of a simple model is the hierarchical linear growth curve model considered by Gelfand and Smith (1990). This model has a simple representation as a DAG (drawn with the DoodleBUGS editor), shown in figure 2. The plate, a rectangular box with four parallel lines along its bottom and right edges, is used as a metaphor for repetition.

To describe the DAG in the BUGS language, a textual language, we need two types of relation: stochastic relations and logical relations. The stochastic relations tell which probability distribution function is associated with which node in the model. The logical relations define how to calculate the values of the parameters of the probability distribution functions in terms of the values of the node's parents. For stochastic relations we use the tilde (~) as the relational operator and for logical relations the left pointing arrow (<-). A final element in the BUGS language is a notation for repetition. We use the notation

```
for (i in M:N) { ... }
```

where the statements between the braces are duplicated with the place holder i replaced by the integer values M through N. Comments in the BUGS language are any characters that follow the hash sign (#) up to the end of the line.

Written in the BUGS language, our example is

```
model
{
    for (i in 1:N) {
        for (j in 1:T) {
          Y[i,j] ~ dnorm(mu[i,j],tau.c)
          # linear growth curve
          mu[i,j] <- alpha[i]+beta[i]*(x[j]-xbar)
        }
        alpha[i] ~ dnorm(alpha.c,alpha.tau)
        beta[i] ~ dnorm(beta.c,beta.tau)
    }
    tau.c ~ dgamma(0.001,0.001)
    sigma <- 1/sqrt(tau.c)
    alpha.c ~ dnorm(0.0,1.0E-6)
    alpha.tau ~ dgamma(0.001,0.001)
    beta.c ~ dnorm(0.0,1.0E-6)
    beta.tau ~ dgamma(0.001,0.001)
    alpha0 <- alpha.c-xbar*beta.c
}
```

We make some comments about this model. The data Y consists of the weight of N rats measured at T time points. A linear model is fitted for each rat. The slope and intercept for each rat are drawn from normal distributions with unknown hyper parameters alpha.c, alpha.tau, beta.c and beta.tau. These hyper parameters are given vague priors.

The parameterization used by each distribution must be documented. For example, the dnorm distribution parameterizes the normal distribution in terms of its mean and precision (the reciprocal of the variance), not the standard deviation. Logical nodes

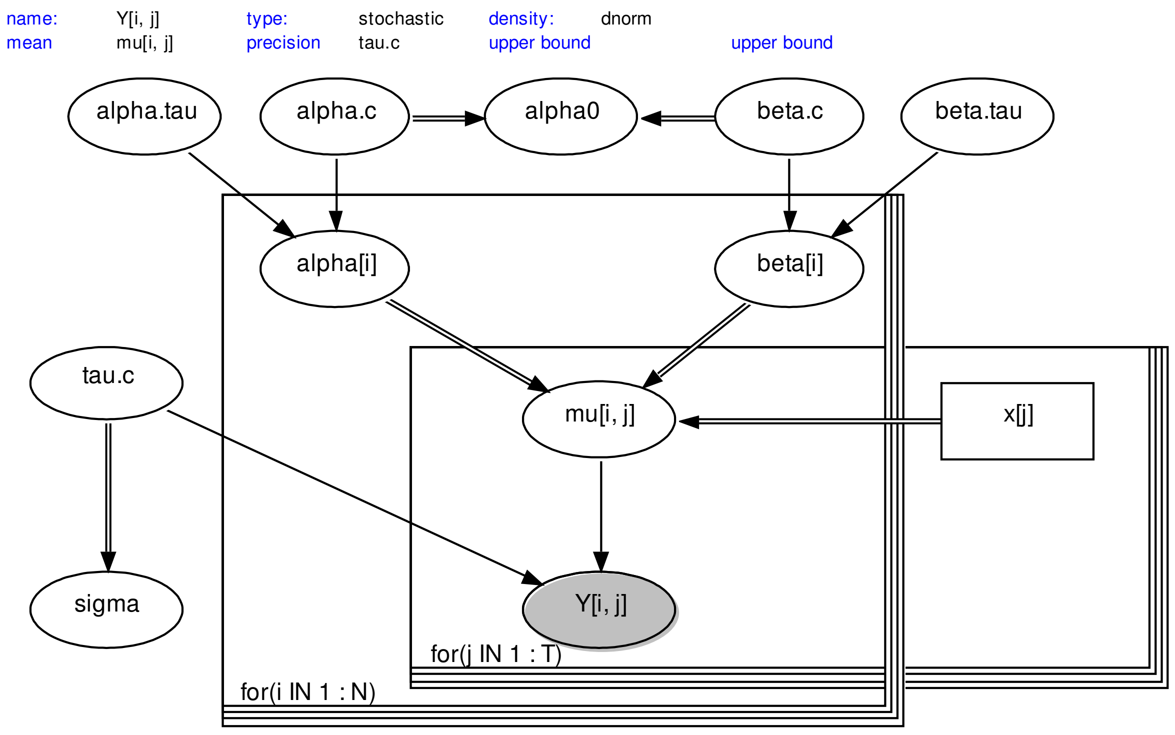

Figure 2: Directed Acyclic Graph for a hierarchical linear growth model

can be added to the model to calculate functions of stochastic nodes, for example `sigma` and `alpha0` in this model. We can easily change the model: the distribution of the data `Y` can be changed from the normal (`dnorm`) to, say, the *t* distribution (`dt`) to allow for outliers, or the linear growth curve relation for `mu` could be changed to a non-linear one, *etc*.

Example 2: Biopsy data

A slightly more complex model is the biopsy data considered by Spiegelhalter and Stovin (1983). In this model, the state of an internal organ (the heart) is probed by taking tissue samples with a hollow needle. The true (latent) state of the organ is at least as bad as the worst category of the tissue sample taken. Multiple tissue samples are taken from each organ giving rise to multinomial data. The probability vector of proportions in the multinomial is modeled as a mixture of Dirichlet distributions with the constraint that elements of the error matrix above the leading diagonal are zero (no false positives). This model is quite difficult to draw as a graph using the DoodleBUGS editor but easy to write in the BUGS language.

```
model
{
    for (i in 1:ns){
        nbiops[i] <- sum(biopsies[i,])
        true[i]  ~ dcat(p[ ])
        biopsies[i,1:4] ~
            dmulti(error[true[i],],nbiops[i])
    }
    error[2,1:2] ~ ddirch(prior[1:2])
    error[3,1:3] ~ ddirch(prior[1:3])
    error[4,1:4] ~ ddirch(prior[1:4])
    error[1,1] <- 1 error[1, 2] <- 0
    error[1,3] <- 0 error[1, 4] <- 0
    error[2,3] <- 0 error[2, 4] <- 0
    error[3,4] <- 0
    # prior for p
    p[1:4] ~ ddirch(prior[ ])
}
```

Note the use of variable indexing in the relation for biopsies: the variable `true[i]` takes a value in $\{1, 2, 3, 4\}$ and picks which row of the error matrix is used in the multinomial distribution. In the BUGS language, a variable index must always be a named quantity in the model. If the index is non variable, then an expression that evaluates to a constant can also be used. Nested indexing is allowed.

Data

Usually some of the quantities in the statistical model have fixed values: they are data. Within the BUGS language there is no distinction between quantities that are data and quantities about which inference is required. We put quantities with fixed values in a data set. The syntax we have chosen for data sets is the S list format containing scalars, vectors and multi dimensional arrays (in the form of structures).

If a quantity has both fixed components and components that need estimating, then the later will be represented as `NA`s in the data set. The OpenBUGS software processes both the model description language and the associated data sets to build the joint probability distribution.

Correctness

Using the BUGS language it is easy to write down a complex statistical model. But is the model correct? The model must be both syntactically and semantically correct. Checking syntactic correctness is quite easy: parsing the model will detect any errors and produce clear error messages. It is much harder to check that the description of a model in the BUGS language plus a data set (or data sets) defines a complete and consistent model. We take a constructive approach to this problem. OpenBUGS tries to compile the BUGS language into a detailed graph that represents the joint probability distribution. The completeness and consistency of this graph are then checked. This approach can detect many errors. However, users have found the error messages produced somewhat cryptic. Some typical cases of lack of consistancy are:

1. The data set defines the length of a vector quantity to be say `L` and the model uses a component of this vector quantity with a index greater than `L`.

2. Multiple definitions of a node in the model are given, for example the statement

   ```
   for(i in 1:10){ x ~ dnorm(0, 1) }
   ```

Computation on the graph

A detailed representation of the graph of the model allows us to easily calculate the joint probability distribution. It also makes it easy to calculate the conditional distribution of a single node in the model, holding all other nodes fixed, in an efficient way. These single node conditional distributions are the basic building blocks of inference algorithms based on an extreme divide-and-conquer approach. Conditional distributions of blocks of nodes can be derived from the single node conditional distributions. These multi-node conditional distributions are useful for inference algorithms when the divide approach is not taken to extremes. The deviance of the model can be calculated from the distributions associated with data nodes (including censored observations) in the model. The OpenBUGS software tries to classify the functional form of the single node conditional distributions. The more detailed the classification of these single node conditional distributions, the wider the choice of algorithms that can be proved valid for statistical inference on the model. Markov Chain Monte Carlo (MCMC) simulation makes heavy use of the calculation of conditional distributions. This fact makes MCMC simulation a natural choice of inference technology to combine with the BUGS language. However other approaches to inference could be added to the OpenBUGS software.

Outlook

The BUGS language provides a uniform way of specifying complex statistical models. It allows a model to be worked on and shared by several people. The BUGS language can be used by other software that makes inference about complex models. The OpenBUGS software provides source code level access to the lexical and parsing tools used to process the BUGS language.

Different inference algorithms, for example the EM algorithm (Dempster et al., 1977), the variational algorithm (Jaakkola and Jordan, 2000) or particle filters (Doucet et al., 2001), could be built on top of the OpenBUGS software. We hope that the separation of model specification and parameter inference become more common in the future development of statistical software.

In many statistical packages, the idea of a model stays in the background; the emphasis is on fitting data. This makes interfacing the OpenBUGS software with, say, R conceptually difficult. R has data objects and functions for doing computation on data objects but not model description objects. **BRugs**, the R interface to OpenBUGS, has to read the model description from a file. This is less than ideal. In outline, the ideal way of interfacing R and OpenBUGS would be to have model description objects that could be translated into compiled model objects. Compiled model objects could then be passed to inference algorithms (MCMC etc) to give fitted model objects (for MCMC, something like an `mcmc` object from the **coda** package (Plummer et al., 2005)). Standard R functions could then be applied to the fitted model object to compute any derived quantity of interest. This is a long term program.

Bibliography

R. A. Becker, J. M. Chambers, and A. R. Wilks. *The new S language: A programming environment for data analysis and graphics*. Wadsworth & Brooks/Cole, Pacific Grove, Calif, 1988. 17

A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39:1, 1977. 20

A. Doucet, N. de Freitas, and N. Gordon, editors. *Sequential Monte Carlo in Practice*. Springer-Verlag, 2001. ISBN 0-387-95146-6. 20

B. Fisher and J. Schumann. Autobayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3): 483–508, May 2003. 17

A. Gelfand and A. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85:398–409, 1990. 18

T. Jaakkola and M. I. Jordan. Bayesian parameter estimation via variational methods. *Statistics and Computing*, 19:25–37, 2000. 20

M. Plummer. *JAGS Version 0.90 Manual*. International Agency for Research on Cancer, Lyon, France, September 2005. http://www-ice.iarc.fr/~martyn/software/jags. 17

M. Plummer, N. Best, K. Cowles, and K. Vines. *CODA: Output analysis and diagnostics for MCMC*, 2005. R package version 0.10-3. 20

D. J. Spiegelhalter and P. G. I. Stovin. An analysis of repeated biopsies following cardiac transplantation. *Statistics in Medicine*, 2:33–40, 1983. 19

H. Toivonen, H. Mannila, J. Seppänen, and K. Vasko. Bassist user's guide for version 0.8.3. Technical Report C-1999-36, Dept of Computer Science, University of Helsinki, 1999. http://www.cs.helsinki.fi/research/fdk/bassist/. 17

Andrew Thomas
Department of Mathematics & Statistics
University of Helsinki, Finland
ant@rni.helsinki.fi

Bayesian Data Analysis using R

by Jouni Kerman and Andrew Gelman

Introduction

Bayesian data analysis includes but is not limited to Bayesian inference (Gelman et al., 2003; Kerman, 2006a). Here, we take *Bayesian inference* to refer to posterior inference (typically, the simulation of random draws from the posterior distribution) given a fixed model and data. *Bayesian data analysis* takes Bayesian inference as a starting point but also includes fitting a model to different datasets, altering a model, performing inferential and predictive summaries (including prior or posterior predictive checks), and validation of the software used to fit the model.

The most general programs currently available for Bayesian inference are WinBUGS (BUGS Project, 2004) and OpenBUGS, which can be accessed from R using the packages **R2WinBUGS** (Sturtz et al., 2005) and **BRugs**. In addition, various R packages exist that directly fit particular Bayesian models (e.g. **MCMCPack**, Martin and Quinn (2005)). or emulate aspects of BUGS (JAGS). In this note, we describe our own entry in the "inference engine" sweepstakes but, perhaps more importantly, describe the ongoing development of some R packages that perform other aspects of Bayesian data analysis.

Umacs

Umacs (Universal Markov chain sampler) is an R package (to be released in Spring 2006) that facilitates the construction of the Gibbs sampler and Metropolis algorithm for Bayesian inference (Kerman, 2006b). The user supplies data, parameter names, updating functions (which can be some mix of Gibbs samplers and Metropolis jumps, with the latter determined by specifying a log-posterior density function), and a procedure for generating starting points. Using these inputs, Umacs writes a customized R sampler function that automatically updates, keeps track of Metropolis acceptances (and uses acceptance probabilities to tune the jumping kernels, following Gelman et al. (1995)), monitors convergence (following Gelman and Rubin (1992)), summarizes results graphically, and returns the inferences as random variable objects (see **rv**, below).

Umacs is customizable and modular, and can be expanded to include more efficient Gibbs/Metropolis steps. Current features include adaptive Metropolis jumps for vectors and matrices of random variables (which arise, for example, in hierarchical regression models, with a different vector of regression parameters for each group).

Figure 1 illustrates how a simple Bayesian hierarchical model (Gelman et al., 2003, page 451) can be fit using Umacs: $y_j \sim \mathrm{N}(\theta_j, \sigma_j^2)$, $j = 1, \ldots, J$ ($J = 8$), where σ_j are fixed and the means θ_j are given the prior $t_\nu(\mu, \tau)$. In our implementation of the Gibbs sampler, θ_j is drawn from a Gaussian distribution with a random variance component V_j. The conditional distributions of θ, μ, V, and τ can be calculated analytically, so we update them each by a direct (Gibbs) update. The updating functions are to be specified as R functions (here, `theta.update`, V.update, mu.update, etc.). The degrees-of-freedom parameter ν is also unknown, and updated using a Metropolis algorithm. To implement this, we only need to supply a function calculating the logarithm of the posterior function; Umacs supplies the code. We have several Metropolis classes for efficiency; SMetropolis implements the Metropolis update for a scalar parameter. These "updater-generating functions" (Gibbs and SMetropolis) also require an argument specifying a function returning an initial starting point for the unknown parameter (here, theta.init, mu.init, tau.init, etc.).

```
s <- Sampler(
  J       = 8,
  sigma.y = c(15, 10, 16, 11,  9, 11, 10, 18),
  y       = c(28,  8, -3,  7, -1,  1, 18, 12),
  theta   = Gibbs(theta.update,theta.init),
  V       = Gibbs(V.update, V.init),
  mu      = Gibbs(mu.update,mu.init),
  tau     = Gibbs(tau.update, tau.init),
  nu      = SMetropolis(log.post.nu, nu.init),
  Trace("theta[1]")
)
```

Figure 1: *Invoking the Umacs* Sampler *function to generate an R Markov chain sampler function* s(...). *Updating algorithms are associated with the unknown parameters ($\theta, V, \mu, \tau, \nu$). Optionally, the non-modeled constants and data (here J, σ, y) can be localized to the sampler function by defining them as parameters; the function* s *then encapsulates a complete sampling environment that can be even moved over and run on another computer without worrying about the availability of the data variables. The "virtual updating function"* Trace *displays a real-time trace plot for the specified scalar variable.*

The function produced by Sampler runs a given number of iterations and a given number of chains; if we are not satisfied with the convergence, we may resume iteration without having to restart the chains. It is also possible to add chains. The length of the burn-in period that is discarded is user-definable and we may also specify the desired number of simulations to collect, automatically performing thinning as the sampler runs.

Once the pre-specified number of iterations are done, the sampler function returns the simulations wrapped in an object which can be coerced into a plain matrix of simulations or to a list of random variable objects (see **rv**, below), which can be then attached to the search path.

rv

rv is an R package that defines a new simulation-based *random variable* class in R along with various mathematical and statistical manipulations (Kerman and Gelman, 2005). The program creates an object class whose instances can be manipulated like numeric vectors and arrays. However, each element in a vector contains a hidden dimension of simulations: the rv objects can thus be thought of being approximations of random variables. That is, a random scalar is stored internally as a vector, a random vector as a matrix, a random matrix as a three-dimensional array, and so forth. The random variable objects are useful when manipulating and summarizing simulations from a Markov chain simulation (for example those generated by Umacs). They can also be used in simulation studies (Kerman, 2005). The number of simulations stored in a random variable object is user-definable.

The rv objects are a natural extension of numeric objects in R, which are conceptually just "random variables with zero variance"—that is, constants. Arithmetic operations such as + and ^ and elementary functions such as exp and log work with rv objects, producing new rv objects.

These random variable objects work seamlessly with regular numeric vectors: for example, we can impute random variable z into a regular numeric vector y with a statement like y[is.na(y)] <- z. This converts y automatically into a random vector (rv object) which can be manipulated much like any numeric object; for example we can write mean(y) to find the distribution of the arithmetic mean function of the (random) vector y or sd(y) to find the distribution of the sample standard deviation statistic.

The default print method of a random variable object outputs a summary of the distribution represented by the simulations for each component of the argument vector or array. Figure 2 shows an example of a summary of a random vector z with five random components.

```
> z
     name mean   sd      Min   2.5%  25%  50%  75% 97.5% Max
[1] Alice 59.0 27.3 ( -28.66   1.66 42.9 59.1 75.6   114 163 )
[2]   Bob 57.0 29.2 ( -74.14  -1.98 38.3 58.2 75.9   110 202 )
[3] Cecil 62.6 24.1 ( -27.10  13.25 48.0 63.4 76.3   112 190 )
[4]  Dave 71.7 18.7 (   2.88  34.32 60.6 71.1 82.9   108 182 )
[5] Ellen 75.0 17.5 (   4.12  38.42 64.1 75.3 86.2   108 162 )
```

Figure 2: *The* print *method of an* rv *(random variable) object returns a summary of the mean, standard deviation, and quantiles of the simulations embedded in the vector.*

Standard functions to plot graphical summaries of random variable objects are being developed. Figure 3 shows the result of a statement plot(x,y) where x are constants and y is a random vector with 10 constant components (shown as dots) and five random components (shown as intervals).

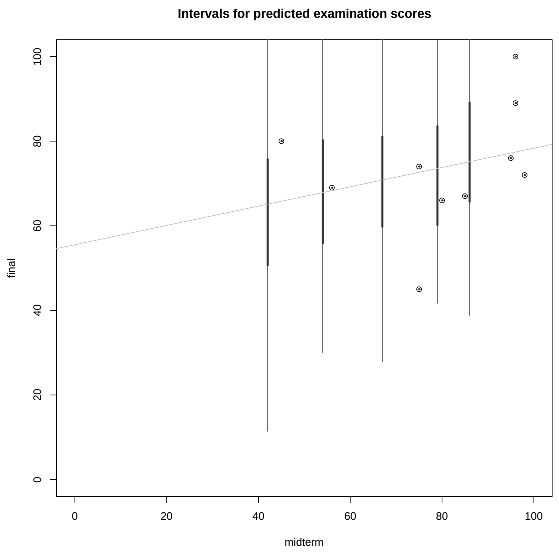

Figure 3: *A scatterplot of fifteen points* (x,y) *where five of the components of* y *are random, that is, represented by simulations and thus are drawn as intervals. Black vertical intervals represent the 50% uncertainty intervals and the gray ones the 95% intervals. (The light grey line is a regression line computed from the ten fixed points).*

Many methods on rv objects have been written; for example E(y) returns the individual means (expectations) of the components of a random vector y. A statement Pr(z[1]>z[2]) would give an estimate of the probability of the event $\{z_1 > z_2\}$.

Random-variable generating functions generate new rv objects by sampling from standard distributions, for example rvnorm(n=10, mean=0, sd=1) would return a random vector representing 10 draws from the standard normal distribution. What makes these functions interesting is that we can give them parameters that are also random, that is, represented by simulations. If y is modeled as $\mathrm{N}(\mu, \sigma^2)$ and the random variable objects mu and sigma represent draws from the joint posterior distribution of (μ, σ)—we can obtain these if we fit the model with **Umacs** or BUGS for example—then a simple statement like rvnorm(mean=mu, sd=sigma) would generate a random variable representing draws from the posterior predictive distribution of y. A single line of code thus will in fact perform Monte Carlo integration of the joint density of $(y^{\text{rep}}, \mu, \sigma)$, and draw from the resulting distribution $p(y^{\text{rep}}|y) = \int\int \mathrm{N}(y^{\text{rep}}|\mu, \sigma)p(\mu, \sigma|y)\,\mathrm{d}\mu\,\mathrm{d}\sigma$. (We distinguish the observations y and the unobserved random variable y^{rep}, which has the same conditional distribution as y).

R & B

The culmination of this research project is an R environment for Bayesian data analysis which would allow inference, model expansion and comparison, model checking, and software validation to be performed easily, using a high-level Bayesian graphical modeling language "*B*" adapted to R, with functions that operate on R objects that include *graphical models*, *parameters* (nodes), and *random variables*. *B* exists now only in conceptual level (Kerman, 2006a), and we plan for its first incarnation in R (called *R & B*) to be a simple version to demonstrate its possibilities. *B* is not designed to be tied to any particular inference engine but rather a general interface for doing Bayesian data analysis. Figure 4 illustrates a hypothetical interactive session using *R & B*.

```
## Model 1: A trivial model:
NewModel(1, "J", "theta", "mu", "sigma", "y")
Model(y)          <- Normal(J, theta, sigma)
Observation(y)    <- c(28,8,-3,7,-1,1,18,12)
Hypothesis(sigma) <- c(15,10,16,11,9,11,10,18)
Observation(J)    <- 8
Fit(1)
# Look at the inferences:
print(theta)
## Model 2: A hierarchical t model
NewModel(2, based.on.model=1, "V", "mu", "tau")
Model(theta) <- Normal(J, mu, V)
Model(V)     <- InvChisq(nu, tau)
Fit(2)
# Look at the new inferences:
plot(theta)
# Draw from posterior predictive distribution:
y.rep1 <- Replicate(y, model=1)
y.rep2 <- Replicate(y, model=2)
## Use the same models but
##   a new set of observations and hypotheses:
NewSituation()
Hypothesis(sigma) <- NULL # Sigma is now unknown.
Fit()
...
```

Figure 4: *A hypothetical interactive session using the high-level Bayesian language "B" in R (in development). Several models can be kept in memory. Independently of models, several "inferential situations" featuring new sets of observations and hypotheses (hypothesized values for parameters with assumed point-mass distributions) can be defined. Fitting a model launches an inference engine (usually, a sampler such as* **Umacs** *or BUGS) and stores the inferences as random variable objects. By default, parameters are given noninformative prior distributions.*

Conclusion

R is a powerful language for statistical modeling and graphics; however it is currently limited when it comes to Bayesian data analysis. Some packages are available for fitting models, but it remains awkward to work with the resulting inferences, alter or compare the models, check fit to data, or validate the software used for fitting. This article describes several of our research efforts, which we have made into R packages or plan to do so. We hope these packages will be useful in their own right and also will motivate future work by others integrating Bayesian modeling and graphical data analysis, so that Bayesian inference can be performed in the iterative data-analytic spirit of R.

Acknowledgements

We thank Tian Zheng, Shouhao Zhao, Yuejing Ding, and Donald Rubin for help with the various programs and the National Science Foundation for financial support.

Bibliography

BUGS Project. BUGS: Bayesian Inference Using Gibbs Sampling. http://www.mrc-bsu.cam.ac.uk/bugs/, 2004. 21

A. Gelman and D. Rubin. Inference from iterative simulation using multiple sequences (with discussion). *Statistical Science*, 7:457–511, 1992. 21

A. Gelman, G. Roberts, and W. Gilks. Efficient Metropolis jumping rules. In J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, editors, *Bayesian Statistics 5*. Oxford University Press, 1995. 21

A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, London, 2nd edition, 2003. 21

J. Kerman. Using random variable objects to compute probability simulations. Technical report, Department of Statistics, Columbia University, 2005. 22

J. Kerman. An integrated framework for Bayesian graphic modeling, inference, and prediction. Technical report, Department of Statistics, Columbia University, 2006a. 21, 23

J. Kerman. Umacs: A Universal Markov Chain Sampler. Technical report, Department of Statistics, Columbia University, 2006b. 21

J. Kerman and A. Gelman. Manipulating and summarizing posterior simulations using random variable objects. Technical report, Department of Statistics, Columbia University, 2005. 22

A. D. Martin and K. M. Quinn. `MCMCpack` 0.6-6. http://mcmcpack.wustl.edu/, 2005. 21

S. Sturtz, U. Ligges, and A. Gelman. R2WinBUGS: A package for running WinBUGS from R. *Journal of Statistical Software*, 12(3):1–16, 2005. ISSN 1548-7660. 21

Jouni Kerman, Andrew Gelman
Department of Statistics
Columbia University, NY, USA

Erratum

The article "BMA: An R package for Bayesian Model Averaging" that appeared in R News volume 5(2) contained an error in the second paragraph on page 4. When describing the probability that the variable was not in the model, the text gave the value **0.445** when in fact the correct value was **0.555**. This error was corrected in the online version of R News 5(2) on 16/12/05.

Thanks to Antti Pirjeta for pointing out the error.

R News is a publication of the R Foundation for Statistical Computing, communications regarding this publication should be addressed to the editors. All articles are copyrighted by the respective authors. Please send submissions to regular columns to the respective column editor, all other submissions to the editor-in-chief or another member of the editorial board (more detailed submission instructions can be found on the R homepage).

R Project Homepage:
http://www.R-project.org/

This newsletter is available online at
http://CRAN.R-project.org/doc/Rnews/