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.

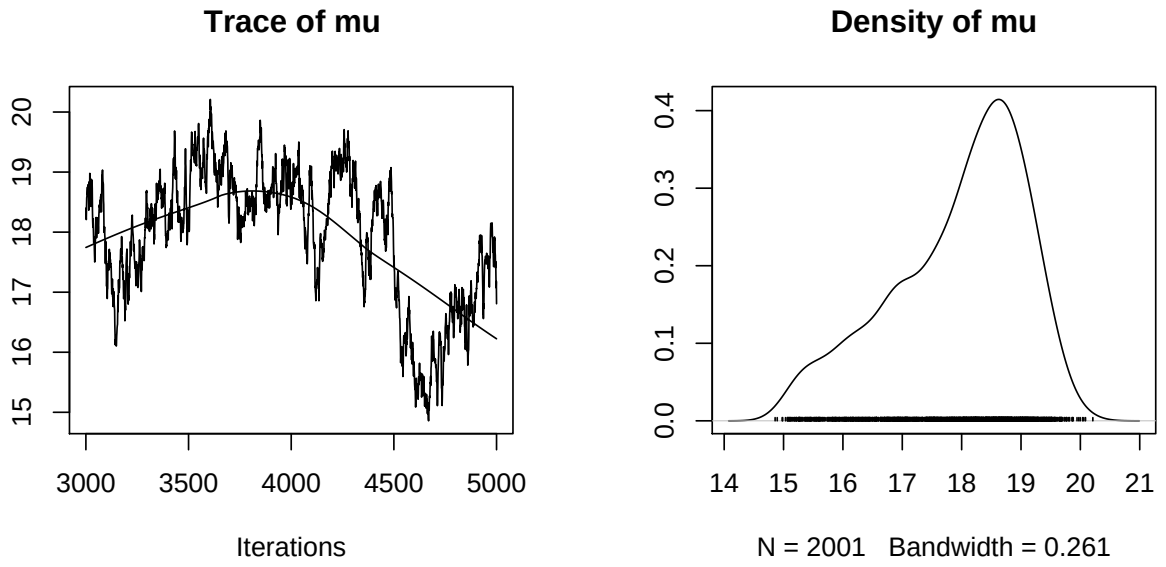

Figure 1: Example of a trace plot and density plot produced by the plot method for `mcmc` objects.

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**.

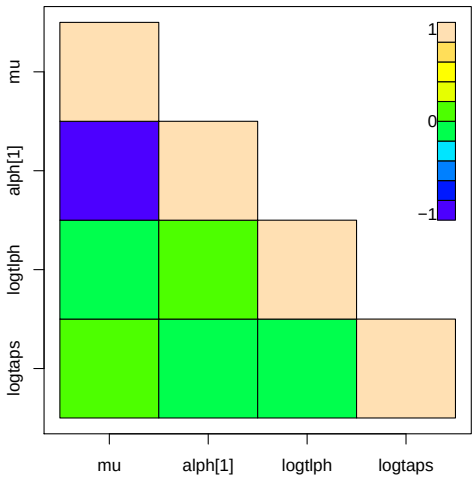

Figure 2: Example output from the function `crosscorr.plot`

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.