Using Random Variables to Manipulate and Summarize Simulations in R

Jouni Kerman
Department of Statistics
Columbia University
kerman@stat.columbia.edu

Andrew Gelman
Department of Statistics
Columbia University
gelman@stat.columbia.edu

November 15, 2005

Abstract

A fully Bayesian computing environment calls for the possibility of defining vector and array objects that may contain both random and deterministic quantities, and syntax rules that allow treating these objects much like any variables or numeric arrays. Working within the statistical package R, we introduce a new object-oriented framework based on a new random variable data type that is implicitly represented by simulations.

We seek to be able to manipulate random variables and posterior simulation objects conveniently and transparently and provide a basis for further development of methods and functions that can access these objects directly. This new environment is fully Bayesian in that the posterior simulations can be handled directly as random variables.

Keywords: Bayesian inference, object-oriented programming, posterior simulation, random variable objects

1 Introduction

In practical Bayesian data analysis, inferences are drawn from an $L \times k$ matrix of simulations representing L draws from the posterior distribution of a vector of k parameters. This matrix is typically obtained by a computer program implementing a Gibbs sampling scheme or other Markov chain Monte Carlo (MCMC) process. Once the matrix of simulations from the posterior density of the parameters is available, we may use it to draw inferences about any function of the parameters.

In the Bayesian paradigm, any quantity is modeled as random; observed values (constants) are but realizations of random variables, or are random variables that are almost everywhere constant. In this paper, we demonstrate how an interactive programming environment that has a random variable (and random array) *data type* makes programming for Bayesian data analysis considerably more intuitive. Manipulating simulation matrices and generating random variables for predictions is of course possible using software that is already available. However, an intuitive programming environment makes problems easier to express in program code and hence also easier to debug.

Our implementation integrates seamlessly into the R programming environment and is *transparent* in the sense that functions that accept numerical arguments will also accept random arguments. There are also no new syntax rules to be learned.

1.1 Our programming environment

Convenient practice of Bayesian data analysis requires a programmable computing environment, which, besides being able to write a program to draw the posterior simulations, allows for random variate generation and easy manipulation of simulations.

Finding posterior simulations is an essential part of any Bayesian programming environment, but in this paper we concentrate in the post-processing phase of Bayesian data analysis, and in our examples we assume that we have access to an application that draws posterior simulations. Such programs include MCMCPack (Martin and Quinn, 2004), JAGS (Plummer, 2004), OpenBUGS (Thomas and O'Hara, 2004).

Once the simulations have been obtained, we post-process them using R (R Development Core Team,

2004). The combination of BUGS and R has proven to be a powerful combination for Bayesian data analysis. R is an interactive, fully programmable, object-oriented computing environment originally intended for data analysis. R is especially convenient in vector and matrix manipulation, random variable generation, graphics, and common programming.

R was not developed with Bayesian statisticians in mind, but fortunately it is flexible enough to be modified to our liking. In R, the data are stored in vector format, that is, in variables that may contain several components. These vectors, if of suitable length, may then have their dimension attributes set to make them appear as matrices and arrays. The vectors may contain numbers (numerical constants) and symbols such as `Inf` (∞) and the missing value indicator `NA`. Alternatively, vectors can contain character strings or logical values (`TRUE`, `FALSE`). Our implementation extends the definition of a vector or array in R, allowing any component of a numeric array to be replaced by an object that contains a number of simulations from some distribution.

1.2 A simple example

We illustrate the "naturalness" of our framework with a simple example of regression prediction. Suppose we have a class with 15 students of which all have taken the midterm exam but only 10 have taken the final. We shall fit a linear regression model to the ten students with complete data, predicting final exam scores y from midterm exam scores x,

$$y_i|\beta_1, \beta_2, x_i, \sigma \sim \mathrm{N}(\beta_1 + \beta_2 x_i,\, \sigma^2)$$

and then use this model to predict the final exam scores of the other five students. We use a noninformative prior on $(\beta, \log(\sigma))$, or $(\beta, \sigma) \propto 1/\sigma^2$.

The posterior predictive distribution of y is obtained by simulating $\beta = (\beta_1, \beta_2)$ and σ from their joint posterior distribution and then generating the missing elements of y from the normal distribution above. Assume that we have obtained the classical estimates $(\hat{\beta}, \hat{\sigma})$ along with the unscaled covariance matrix V_β using the standard linear fit function `lm()` in R. The posterior distribution of σ is then

$$\sigma|x, y \sim \hat{\sigma} \cdot \sqrt{(n-2)/z}, \quad \text{where} \quad z \sim \chi^2(n-k).$$

Using our random variable package for R, this mathematical formula translates to the statement,

```
sigma <- sigma.hat*sqrt((n-2)/rv.chisq(df=n-2))
```

```
> y.pred
     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 1: Quick summary of the posterior predictive distribution of y is obtained by typing the name of the vector (`y.pred`) on the console.

which is remarkably similar to the corresponding mathematical notation. The posterior distribution of β is $\beta|\sigma, x, y \sim \mathrm{N}\left(\hat{\beta}, V_\beta\sigma^2|x, y, \sigma^2\right)$, simulated by

```
beta  <- rv.mvnorm(mean=beta.hat, var=V.beta*sigma^2)
```

The predictions for the missing y values are obtained by

```
y.pred <- rv.norm(mean=beta[1]+beta[2]*x[is.na(y)], sd=sigma)
```

and quickly summarized by typing the name of the variable on the console. This is shown in Figure 1.

We now can impute the predicted values. Our object-oriented framework allows us to combine constants with random variables to produce a "mixed" vector: `y[is.na(y)] <- y.pred` replaces the missing values (indicated by the symbol `NA`) by the predicted values, which are implicitly represented by simulations. The predictions can be plotted along with the observed (x, y) pairs using the command `plot(x, y)` which shows the determinate values as points, and the random values as intervals.

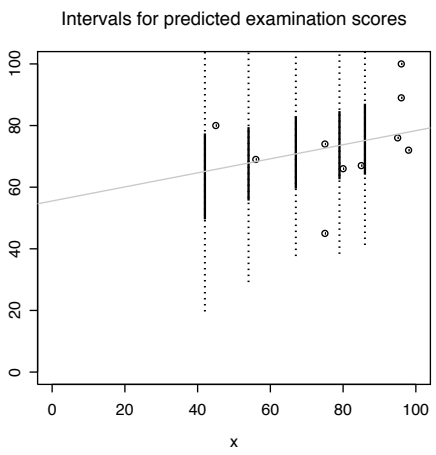
Any function of the random variables is obtained as easily as a function of constants. For example, the distribution of the mean score of the class is `mean(y)`,

```
> mean(y)
    mean   sd    Min 2.5%  25%  50% 75% 97.5%  Max
[1] 70.9 5.22 ( 50.7 60.5 67.9 70.8  74  80.9 98.5 )
```

which is the mean of ten constants and five random variables. The probability that the average is more than 80 points is given by `Pr(mean(y)>80)` which comes to 0.04 in this set of simulations.

1.3 Motivation

The motivation for a new programming environment has grown from our practical needs. Suppose, for example, that we have obtained posterior simulations for the coefficients $\beta = (\beta_1, \ldots, \beta_k)$ of a large regression model $y \sim \mathrm{N}(X\beta, \sigma^2)$. The posterior simulations are then (typically) stored in an $L \times k$ -dimensional




```
# Predictions for the y values
y.pred <- rv.norm(
       mean=beta[1]+beta[2]*x
       sd=sigma)
# Imputing the missing values
y[is.na(y)] <- y.pred[is.na(y)]
plot(x,y)
```

Figure 2: Predicting the final examination scores: uncertainty intervals of the five predicted final exam scores. The 50% intervals are shown as solid vertical lines and the 95% intervals as dotted vertical lines. The observed values are shown as circles.

matrix `beta` with L simulations per parameter. Our wish list for a "fully Bayesian" programming language includes the following features:

Hiding unnecessary details. Once we have obtained posterior simulations for the quantities of interest, we wish to concentrate on the essential and not to think about such details as which dimension of the matrix `beta` contains the simulations for some β_j. Suppose we want to work with a rotated vector $\gamma = R\beta$ where $R = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1\sqrt{2} \end{pmatrix}$. We can then simply write `gamma <- R %*% beta` and not

```
gamma <- array(NA,c(L,length(beta[1,])))
for (i in 1:L) {
   gamma[i] <- R %*% beta[i,]
}
```

For another example, to compute the posterior distribution of β_1/β_2, we wish to write `ratio <- beta[1]/beta[2]` and not `ratio <- beta[,1]/beta[,2]`.[1]

Intuitive programming. To draw inferences from the posterior distribution, we wish that our program code be as close to mathematical notation as possible. Ideally, we should be able to express the statement $y = X\beta$ as `y <- X %*% beta` in the same way we express matrix multiplication for numeric vectors.

To implement this for a random (or constant) matrix X and random vector β in "traditional" program code would require at least one loop. While writing loops and other typical programming structures are certainly not difficult things to do, nevertheless it takes our mind away from the essence of our work: Bayesian data analysis, or more specifically, computation of posterior inferences and predictions. We also believe that any program code that does *not* resemble mathematical notation is but an attempt to emulate such notation. For instance, draws from the posterior predictive distribution $y^{pred}|X, \beta, \sigma, y \sim \mathrm{N}(X\beta, \sigma^2 \,|\, y)$ should be accessible using a statement that resembles the mathematical expression as much as possible:

```
y.pred <- rv.norm(mean=X %*% beta, sd=sigma)
```

This statement, which features *random* arguments, generates a normally distributed (column) vector of length equal to the length of $X\beta$.

Transparency. In a fully Bayesian computing environment, program code should not be dependent on the nature of the parameters: the same code should apply to both random variables and constants. For example, `p <- invlogit( X %*% beta )` (where `invlogit` is the inverse logit function `1/(1+exp(-x))`) works for constant `beta` vectors and compatible matrices `X`, but ideally it should also work if `beta` is a mixed vector of random variables and constants. If any of the arguments are random, the result should be the *distribution* of `p`.

In R, many functions adapt automatically to the length of the argument n: if $x = (x_1, \ldots, x_n)$ is a vector of length n, then such a "vectorized" function f returns $(f(x_1), \ldots, f(x_n))$ of length n. Combined with this convenient feature, it is possible to write code that does not depend on the length of vectors *and* that does not depend on the nature of arguments.

Faster development and debugging cycle. Short, compact expressions are more readable and easier to understand than traditional code with looping structures and awkward matrix indexing notation. Such code is less prone to contain typographical errors and other mistakes.

[1] `ratio <- beta[,1]/beta[,2]` with output equivalent to `for (i in 1:L) ratio[i] <- beta[i,1]/beta[i,2]` where `ratio` will then contain the componentwise ratio of the simulations, that is, the distribution of the ratio of β_1 and β_2.

2 The implementation

Our implementation of the ideal Bayesian programming environment is based on putting all numerical quantities on a conceptually equal level: any numerical vector component is either a random variable or is missing. Missing values are represented by the special value `NA`. We refer to this new data type as *random variable* and instances of random variables as random variable *objects* or simply as random variables, vectors, or arrays.

A random variable is internally represented by *simulations*, that is, random draws from its distribution. Typically these are obtained either from an MCMC process or generated using built-in random number generators. For compatibility, pure constants are allowed in any component of a random vector.

A random variable $x = x_1$ is represented internally by a numerical column vector of L simulations:

$$x_1 = (x_1^{(1)}, x_1^{(2)}, \cdots, x_1^{(L)})^T$$

The number of simulations L is user-definable, typically to a value such as 200 or 1,000 (Gelman et al., 2003, pp. 277–278). We refer to x_1 as a *vector of simulations*; this is not usually visible to the user, although it is possible to retrieve the simulations and manipulate them directly. The user only sees a random variable `x[1]`: the index `[1]` is the subscript 1 in x_1.

Let n be a fixed number. A random vector $x = (x_1, \ldots, x_n)$ being by definition an n-tuple of random variables, is represented internally by n vectors of simulations. Conceptually, these n column vectors form an $L \times n$ *matrix of simulations*

$$M = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \cdots & x_n^{(3)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(L)} & x_2^{(L)} & \cdots & x_n^{(L)} \end{pmatrix}$$

Each row $x^{(\ell)}$ of the matrix M is a random draw from the *joint distribution* of x. The components of $x^{(\ell)}$ may be dependent or independent. In our implementation, each column j of the matrix is stored separately in the slot allocated for random variable x_j.

In general, we may allow random vectors to have *random length.* For example, suppose that n is a Poisson-distributed random variable. Then n is internally represented by the vector of L simulations $(n^{(1)}, n^{(2)}, \cdots, n^{(L)})^T$. A random vector x_n of random length n is then represented by a ragged array (a list) consisting of rows $x^{(\ell)}$ where row ℓ has length $n^{(\ell)}$. This array is stored as $N = \max_\ell\{n^{(\ell)}\}$ column vectors of length L. The missing values are represented by `NA`: $x_j^{(\ell)}$ equals `NA` if $j > n^{(\ell)}$. In particular, if $n^{(\ell)} = 0$, all components in row i are `NA`s.

Each vector $x^{(\ell)} = (x_1^{(\ell)}, \ldots, x_n^{(\ell)})$ may be thought of as the beginning of an infinitely long vector with the rest of the vector "missing." The missing part of the vector will then consist of all `NA`s, but the tail of the vector is of course not stored in memory.

2.1 Vector operations

A function $f : \mathbb{R}^n \mapsto \mathbb{R}^k$ taking a random vector x as its argument yields a new random vector of length k; $f(x)$ is thus equivalent to the $L \times k$ matrix of simulations consisting of rows $f(x^{(\ell)})$, $\ell \in \{1, \ldots, L\}$. where $x^{(\ell)}$ is the ℓth row of the matrix of simulations of x.

For example, the summation function "+" taking two arguments, say x_1, x_2, will in effect yield a random variable represented by a matrix of simulations with each row ℓ equal to $x_1^{(\ell)} + x_2^{(\ell)}$. In our implementation, given that the random vector `x` is defined, this summation can be performed using the natural statement `y <- x[1] + x[2]`.

If `x.sim` is the corresponding matrix of simulations, this code is then equivalent to `y <- x.sim[,1] + x.sim[,2]`. This statement adds two vectors of length L componentwise; this produces the same output as the loop

```
y <- array(NA,L)
for (i in 1:L)
  y[i] <- x.sim[i,1] + x.sim[i,2]
```

Besides the basic arithmetic operators, also the elementary functions such as `exp()`, `log()`, `sin()`, `cos()`, etc. have been adapted to accept random vectors. For example, if `x` is a random vector of length n, then `exp(x)` returns a random vector of length n consisting of components `exp(x[i])` for $i = 1, \ldots, n$, corresponding to a $L \times n$ matrix of simulations.

Logical operations such as $<$, $\leq$, produce indicators of events. For example, `x>y` yields an indicator random variable of $\{x > y\}$. We may naturally apply functions involving indicators and other random

variables: `(x-y) * (x>y)` yields a random variable with the distribution of $(x-y)\cdot \mathbf{1}_{\{x>y\}} \equiv (x-y)^+$.

2.2 Matrix and array operations

A *random matrix* is implemented as a random vector possessing a dimension attribute. This corresponds to the way matrices are implemented in R. To multiply two (compatible) random matrices, say X and Y, one can simply write `Z <- X %*% Y` which in effect is equivalent to the code

```
Z <- array(NA,c(L,k,m))
for (i in 1:L)
    Z[i,,] <- X[i,,] %*% Y[i,,]
```

where `k` is the number of rows in `X` and `m` is the number of columns in `Y`.

It is possible to define any matrix function to work with random variables. For example `det(M)` returns the distribution of the determinant of the matrix `M`.

The user can be oblivious to the way the simulations are manipulated "behind the scenes." Usually there is no need to access the matrices of simulations; most of the functions that work with numerical arrays also work with random arrays.

2.3 Random variate generation

In practice, we often need to generate random variates to generate simulations. In R, this is typically done using built-in functions. A single n-dimensional draw of independent standard normal random variates is generated by the statement
`z <- rnorm(n,mean=0,sd=1)`.

Our implementation features a number of functions that return independently and indentically distributed (iid) simulations as random vector objects; let us call such functions *random variable-generating* functions. For example, to generate a n-dimensional standard normal random vector object, the random vector `z` generated by the statement

```
z <- rv.norm(n,mean=0,sd=1)
```

is internally represented by n column vectors of simulations (draws from $\mathrm{N}(0,1)$) of length L each. The user sees `z` as an n-dimensional object. The matrix of simulations is accessible via the method `sims(z)` if needed.[2]

[2] `as.matrix(z)` would instead return the $n \times 1$ random matrix object `z` and *not* the simulations.

Consider another example. To simulate the distribution of the random variable $y = \sum_{i=1}^{n} z_i \mathbf{1}_{\{z_i>0\}}/n$ we can write `y <- mean(z*(z>0))` where `mean` returns the distribution of the arithmetic average. This code will also work with a numerical vector.

To compute marginal distributions of quantities integrated over random parameters, we also need to pass random arguments to random variable-generating functions. Take for example the random variable $z \sim \mathrm{N}(\mu, \sigma^2)$, where μ and σ are themselves random variables. We wish to draw simulations from the marginal density of z. two parameters. This is obtained by drawing $z^{(\ell)} \sim \mathrm{N}\left(\mu^{(\ell)}, (\sigma^{(\ell)})^2\right)$ where $(\mu^{(\ell)}, \sigma^{(\ell)})$ are simulations from the joint density of μ and σ. If `mu` and `sigma` are each random variables (random scalars), then
`z <- rv.norm(mean=mu, sd=sigma)` will be a random variable whose vector of simulations consists of components with the distribution of z.

The parameters may also be of arbitrary length. If `mu` and `sigma` are vectors of length k, then

```
z <- rv.norm(mean=mu,sd=sigma)
```

will generate a random vector of length k where the jth component `z[j]` has mean `mu[j]` and standard deviation `sigma[j]`. Thus the matrix of simulations of `z` contains components `z[j]` distributed as $z_j^{(\ell)} \sim \mathrm{N}\left(\mu_j^{(\ell)}, (\sigma_j^{(\ell)})^2\right)$. If the lengths of the vectors do not match, the shorter vector is "recycled" as necessary. Typically `mu` has k different means, but σ is a scalar, so $z_j \sim \mathrm{N}(\mu_j, \sigma^2)$ for $j = 1, \ldots, k$.

2.4 Numerical summaries

Once we have the desired distributions (that is, random vectors or arrays), we need to summarize the results. The distributions being represented by vectors of simulations, the only thing we need to access is the simulations. We have provided a method `sims(x)` to access the matrix of simulations of a random vector object `x`; the user may summarize these any way she or he likes. However, this is not the preferred way of doing things in an object-oriented computing environment. It is usually more productive to write a function (method) that accesses the simulations of the argument objects, and produces the desired results.

Our function `simapply()` applies a given function to each column of the matrix of simulations of a random vector, returning an array of numbers. For example, to find the mean (expectation) of the random

variable x_1, we need obtain the simulations $x_1^{(1)}, \ldots, x_1^{(L)}$ and compute $\sum_{\ell=1}^{L} x_1^{(\ell)}/L$. This is accomplished by `simapply(x[1], mean)` which is equivalent to `mean(sims(x[1]))`, the arithmetic mean of the simulations of the first component of the random vector `x`, x_1. `simapply(x, mean)` applies the `mean()` function to the simulation vectors of each component `x[1],...,x[n]` and thus yields a numeric vector of length n.

On the other hand, if `x` is a random vector, the function `mean(x)` returns the average of the individual random components `x[1],...,x[n]`, that is, `sum(x)/length(x)` which is a (one-dimensional) random variable, internally represented by a vector of L simulations from the distribution of $\sum_{i=1}^{n} x_i/n$.

We can imagine `mean(x)` taking the *rowwise* mean of the matrix of simulations `sims(x)` and `simapply(x, mean)` taking the *columnwise* means of the individual components. The former yields a column vector of length L, that is, a random variable. The latter yields a row vector of length n consisting of constants. In the same fashion, `var(x)` gives the sample variance; if any of the components of `x` is a random variable, the result will be the a random variable with L random draws from the distribution of the random variable $\sum_{i=1}^{n}(x_i - \bar{x})^2/(n-1)$, but `simapply(x, var)` gives the n componentwise variances for the n-dimensional vector `x`.

Several familiar functions taking numerical vectors as arguments have been adapted to accept random vector objects: for example, `quantile()` for finding quantiles, `sort()` for creating distributions of the order statistics, `var()` for the sample variance, `sd()` for the sample standard deviation. Given random variables as arguments, these functions return always random variables. The argument can of course be a mixed vector constants and random variables.

The most often used summaries can be viewed most conveniently by entering the name of the random vector on the console; the default printing method returns the mean, standard deviation, minimum value, maximum, median, and the 2.5% and 97.5% quantiles. This output routine is customizable. See Figure 1.

2.5 Graphical summaries

We have provided some basic graphical summary methods that work on the random vector objects. `plot(x)` draws a scatterplot with credible intervals drawn for each random component of x. If all components are constants, the command reduces to `plot` function. `rv.hist(x)` draws a grid of histograms of simulations, each grid cell containing one histogram for each component of x. Many other functions are being developed. Most functions can be adapted easily to accept random variable objects as arguments.

2.6 Toward fully Bayesian computing

We believe that we have managed to lay the foundation of an essential component in an ideal, fully Bayesian computing environment. The next challenge is to integrate a Bayesian (probabilistic) modeling language to R. Ideally, this language should be part of the R syntax and not just a module that parses BUGS-like models saved in a text file: this way of programming introduces *redundancy*. We need to express our statistical model in a language that BUGS understands, but also in R to draw replications and predictions.

Since computation is an essential part of practical Bayesian data analysis, we wish that making Bayesian programming easier will make Bayesian data analysis methods more effective by routinely considering all uncertain quantities as random variables.

References

Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, London, 2nd edition, 2003.

Andrew D. Martin and Kevin M. Quinn. `MCMCpack` 0.5-2. `http://mcmcpack.wustl.edu/`, 2004.

Martyn Plummer. JAGS: Just Another Gibbs Sampler. `http://www-fis.iarc.fr/~martyn/software/jags/`, 2004.

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. URL `http://www.R-project.org`.

Andrew Thomas and Robert B. O'Hara. OpenBUGS. `http://mathstat.helsinki.fi/openbugs/`, 2004.