Stat Comput (2007) 17: 235–244
DOI 10.1007/s11222-007-9020-4

Manipulating and summarizing posterior simulations using random variable objects

Jouni Kerman · Andrew Gelman

Received: 16 November 2005 / Accepted: 7 June 2007 / Published online: 14 July 2007

Abstract Practical Bayesian data analysis involves manipulating and summarizing simulations from the posterior distribution of the unknown parameters. By manipulation we mean computing posterior distributions of functions of the unknowns, and generating posterior predictive distributions. The results need to be summarized both numerically and graphically.

We introduce, and implement in R, an object-oriented programming paradigm based on a random variable object type that is implicitly represented by simulations. This makes it possible to define vector and array objects that may contain both random and deterministic quantities, and syntax rules that allow to treat these objects like any numeric vectors or arrays, providing a solution to various problems encountered in Bayesian computing involving posterior simulations.

We illustrate the use of this new programming environment with examples of Bayesian computing, demonstrating missing-value imputation, nonlinear summary of regression predictions, and posterior predictive checking.

Keywords Bayesian inference · Bayesian data analysis · Object-oriented programming · Posterior simulation · Random variable objects

J. Kerman (✉)
Statistical Methodology, Novartis Pharma AG, 4002 Basel, Switzerland
e-mail: jouni.kerman@novartis.com

A. Gelman
Department of Statistics, Columbia University, New York, USA
e-mail: gelman@stat.columbia.edu

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, for example using WinBUGS (Lunn et al. 2000) and the R package R2WinBUGS (Sturtz et al. 2005). 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, unknown quantities have probability distributions and are thus random variables. Observed values are just realizations of random variables, and constants may be thought of as random variables with point mass distributions. In mathematical notation, we deal with objects that are random variables, but in practice these objects are approximated by vectors of numbers, that is, simulations. Consequently, when programming for manipulating simulations of unknown quantities, we must write code to manipulate arrays of numeric constants.

Arrays of simulations are cumbersome objects to work with. Functions that work with vectors will not in general work with matrices, so special versions of the functions need to be written to accommodate matrices of simulations as arguments. For example, a scalar-valued random variable becomes a vector of simulations, and a random vector becomes a matrix of simulations.

This gives rise to a question why our computing environment is not equipped to handle objects that correspond directly to the mathematical random variables. Do we really have to deal with arrays of simulations? Do we gain anything if we try to introduce such an object class in our programming environment?

We demonstrate how an interactive programming environment that has a random variable (and random array) data type makes programming involving simulations considerably more intuitive and powerful. This is especially true for Bayesian data analysis. Along with new possibilities, introduction of such a data type raises some new questions, for example, what is a *mean* of a random vector of length n? Is it the *distribution* of the arithmetic average, a scalar quantity, or is it the *expectation* of the individual components, a vector of n constants? If we apply a comparison operator such as "$>$" to two random variables, what kind of an object is created? What does a scatterplot of a random vector look like? How should we plot a histogram of a random vector of length n?

Common programming languages are not equipped to handle random variable objects by default, not even R (R Development Core Team 2004), which is especially suited for statistical computing. However, we can create random variables in object-oriented programming languages by introducing a new class of objects. Manipulating simulation matrices is of course possible using software that is already available, but an intuitive programming environment that allows us to formulate problems in terms of random variable objects instead of arrays of numbers makes statistical problems easier to express in program code and hence also easier to debug.

1.1 A new programming environment

We have written a working prototype of a random-variable-enabled programming environment in R, which 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. We suspect that our ideas could also be implemented in other statistical environments such as Xlisp-Stat (Tierney 1990) or Quail (Oldford 1998).

In R, numeric data objects are stored as vectors, that is, in objects 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, allowing any component of a numeric array to be replaced by an object that contains a number of simulations from some distribution. Internally, a random vector is represented by a list of vectors of simulations, but the user sees them as a single vector, and is also able to manipulate it as such without thinking of the individual simulation draws and such details as how many draws are included per random scalar. Random variables and vectors are thus integrated transparently into the programming environment.

There are no new syntax rules to be learned: built-in numeric functions work directly with random vectors, returning new random vectors. Most user-defined numeric functions that manipulate vectors will also work with these objects directly without any modification.

2 Manipulating posterior simulations

Once the model has been fit and posterior simulations for the unknown parameter vector, say θ, obtained from a model-fitting program, the Bayesian data analyst typically needs to compute all or some of the following tasks:

1. Posterior interval and point estimates of the components of θ, such as means, medians, 50%, 80%, and 95% posterior intervals and the standard deviation which summarizes the uncertainty in θ.
2. Posterior interval and point estimates of functions of θ. For example, if θ is a vector of length 50 consisting of some measures for all fifty U.S. states, we may be interested in the distribution of the mean of the fifty random quantities, $\frac{1}{50}\sum_{i=1}^{50}\theta_i$.
3. Graphical summaries of the quantities mentioned above, for example plots that show point estimates and intervals.
4. Posterior probability statements such as $\Pr(\theta_1 > \theta_2|y)$.
5. Histograms and density estimates of components of θ.
6. Scatterplots and contourplots showing the joint posterior distribution of two-dimensional random quantities.
7. Simulations from the posterior predictive distribution of future data y.
8. Bayesian p-values and graphical data discrepancy checks, using functions of parameters θ, replicated data y^{rep}, and observed data y.

To implement these tasks as computer programs, they must be reinterpreted in terms of posterior simulations. A scalar random variable, say θ_1, is represented internally by a numerical column vector of L simulations:

$$\theta_1 = (\theta_1^{(1)}, \theta_1^{(2)}, \ldots, \theta_1^{(L)})^{\mathsf{T}}.$$

The number of simulations L is typically a value such as 200 or 1,000 (Gelman et al. 2003, pp. 277–278). We refer to $y_1^{(\ell)}$, $\ell = 1, \ldots, L$, as a *vector of simulations.*

Let k be a positive integer. A random vector $\theta = (\theta_1, \ldots, \theta_k)$ being by definition an k-tuple of random variables, is represented internally by k vectors of simulations. These k column vectors form an $L \times k$ *matrix of simulations*

$$\Theta = \begin{pmatrix} \theta_1^{(1)} & \theta_2^{(1)} & \cdots & \theta_k^{(1)} \\ \theta_1^{(2)} & \theta_2^{(2)} & \cdots & \theta_k^{(2)} \\ \theta_1^{(3)} & \theta_2^{(3)} & \cdots & \theta_k^{(3)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_1^{(L)} & \theta_2^{(L)} & \cdots & \theta_k^{(L)} \end{pmatrix}.$$

Each row $\theta^{(\ell)} = (\theta_1^{(\ell)}, \ldots, \theta_k^{(\ell)})$ of the matrix Θ represents a random draw from the *joint distribution* of θ. The components of $\theta^{(\ell)}$ may be dependent or independent.

2.1 The currently-standard approach

The usual approach now is to use loops and vector-matrix computations to manipulate the matrix Θ of posterior simulations. This approach is general but awkward and far from transparent, as we now discuss.

Typically, depending on the problem at hand, we name the simulation vectors and matrices according to the random variables in our model such as β and σ, and manipulate several of these array objects in our programs.

Means, medians, quantiles, and standard deviations of scalar components are obtained by applying the corresponding numerical function such as `mean`, `mean`, `quantile`, `sd` to each of the columns of a matrix of simulations. This typically requires a loop or application of a special function such as `apply` in R, which implements the loop more efficiently. The intervals and standard deviations summarize uncertainty in the parameter. These are examples of quantities that are computed by applying a summary function "columnwise" to a matrix of simulations. For example, in R, the posterior mean of θ_1 is computed by `mean(theta[,1])`. Here `theta` is an $L \times k$ matrix and `theta[,1]` refers to the first column of the matrix.

Distributions of functions of random variables are obtained by applying the function to each *row* in the matrix of simulations. The resulting object is a vector or a matrix with L rows: it represents thus the distribution of the function applied to a random variable. Again, this requires that a loop is written or that a function such as `apply` is used. This time the function is applied, to the rows of the matrix of simulations. For example, if we have a vector of unknowns $\theta = (\theta_1, \ldots, \theta_{50})$ that is represented by an $L \times 50$ matrix of simulations, the distribution of the arithmetic mean of the components of θ is represented by the L numbers $\frac{1}{50}\sum_{i=1}^{50} \theta_i^{(\ell)}$, $\ell = 1, \ldots, L$.

For a simpler example, we may be interested in the distribution of $\theta_1 + \theta_2$. This is represented by a vector of length L with components $\theta_1^{(\ell)} + \theta_2^{(\ell)}$. In R, this would be computed by `theta[,1]+theta[,2]`, adding the two vectors together, yielding a new vector of length L.

Random matrices can be thought of random vectors with a dimension attribute. If θ is a 2×2 random matrix, it is represented by an $L \times 2 \times 2$ matrix of simulations. We can for example find the distribution of the determinant of θ by applying the determinant function to each of the L two-by-two matrices of simulations. Again, this requires a loop or the application of `apply`. For example, in R, the multiplication of two random matrices $A = \Theta\Sigma$ is accomplished by,

```
A <- array (NA,c(L,k,m))
# Allocate an L*k*m matrix
for (i in 1:L) {
    A[i,,] <- Theta[i,,] %*% Sigma[i,,]
}
```

where `k` is the number of rows in `Theta[i, ,]` and `m` is the number of columns in `Sigma[i, ,]`.

These are examples of functions that are applied *rowwise* to a matrix of simulations, yielding an array with the first dimension of size L.

Posterior probability statements such as $\Pr(\theta_1 > \theta_2|y)$ are computed by taking the proportion of the simulations which satisfy the condition "$\theta_1^{(\ell)} > \theta_2^{(\ell)}$," that is, by computing $\frac{1}{L}\sum_{i=1}^{L} \mathbf{1}_{\{\theta_1^{(\ell)} > \theta_2^{(\ell)}\}}$, where $\mathbf{1}_A$ is an indicator function of an event A. In R, `theta[,1]>theta[,2]` yields a *logical* vector of length L, that is, a vector of `TRUE` and `FALSE` values. Since R coerces the `TRUE` values into ones and `FALSE` values into zeros whenever necessary, we can compute the probability value by `mean(theta[,1]>theta[,2])`.

Simulations from the posterior predictive distribution of the n-dimensional data vector y are computed by generating random variates distributed according to the model of y, given the posterior simulations of the unknown parameters. For example, suppose that $y \sim \mathrm{N}(\mathsf{X}\beta, \sigma^2)$ where X is an $n \times k$ matrix of observed covariates and β is an unknown (random) vector $\beta = (\beta_1, \ldots, \beta_k)$ and σ is an unknown scalar-valued variable. If we have posterior simulations for $\beta^{(\ell)}$ (β is a k-vector, so $\beta^{(\ell)}$ is also a k-vector of simulations) and for the scalar $\sigma^{(\ell)}$, the posterior predictive distribution of y, called the *replication* distribution y^{rep} is represented by simulations such that $y^{\mathrm{rep}(\ell)}$ is an n-vector generated so that

$$y^{\mathrm{rep}(\ell)} \sim \mathrm{N}(\mathsf{X}\beta^{(\ell)}, \sigma^{(\ell)}).$$

In R, this step is implemented by the program,

```
y.rep <- matrix (NA, L, n)
for (i in 1:L) {
    y.rep[i,] <- rnorm(nrow(X),
    mean=X %*% beta[i,], sd=sigma[i])
}
```

Similarly, we can generate posterior predictive distributions of the unknown parameters, θ^{rep}, if desired.

Replicated data distributions y^{rep} are required when computing Bayesian p-values and posterior predictive checks. These can be used to assess the fit of the model. A Bayesian p-value is computed by the tail-area probability

$\Pr(T(y^{\text{rep}}, \theta) \geq T(y, \theta))$

for some appropriate test statistic function T. y is here the observed data vector and y^{rep} is the replicated vector, The estimated p-value is the proportion of the L simulations for which the test quantity equals or exceeds its realized value: that is, for which

$T(y^{\text{rep}(\ell)}, \theta^{(\ell)}) \geq T(y, \theta^{(\ell)})$

for all $\ell = 1, \ldots, L$ (Gelman et al. 2003, pp. 162–163).

To compute this in R is straightforward if T is a function that accepts vectors of simulations and knows how to repeat the calculation over a given vector for each simulation $1, \ldots, L$. For most scalar-valued numerical functions this poses no great problem in R; we can plug in a vector of simulations and expect a vector of simulations back. However, if T is a numeric R function that acts on a vector, such as the standard deviation function, it will not be able to accept a matrix of simulations and repeat the computation automatically for each of the L simulations. Again a loop must be written or the `apply` function applied.

2.2 Toward a more natural programming environment

As we have seen, thanks to the convenient vectorized programming language of R, most of the simplest programming tasks in posterior simulation manipulation are not too difficult to implement. Some of them take only one line to write, although they do not look intuitive: the distribution of the standard deviation of a random vector θ would be computed by applying the sample standard deviation function to the rows of a matrix of simulations `theta` by executing `apply(theta,1,sd)`, which returns a vector of simulations of length L. Ideally, we would like to write "`sd(theta)`" where `theta` is not a matrix of simulations, but rather a vector of random variable objects. The result should be a *scalar* random variable object, containing the simulations computed by `apply(theta,1,sd)`.

However, the main motivation here is not only further simplification of code, but the need to work in a more natural programming environment, where objects to be manipulated are random variables that are in the same conceptual level as the random variables in mathematical notation. Manipulation of individual simulations should be kept out of the view, in the background, and the user should be able to focus on telling the computer what function of the *random variables*—not that of the *simulations*—to compute.

For another example, suppose that we wish to compute the distribution of the ratio of two random scalars, say θ_1/θ_2. We would like to have a random vector object `theta` available, of some fixed length k, and be able to write `r <- theta[1]/theta[2]` to obtain another random variable object `r` whose simulations consist of the ratios $\theta_1^{(\ell)}/\theta_2^{(\ell)}$. Currently we are must keep in mind that the object `theta` is not a random vector object but a matrix of numbers, and to compute the ratio, we will have to write `theta[,1]/theta[,2]`, obtaining a vector of numbers (simulations), not a random scalar object. The awkward subscript indexing notation is confusing, and forces us to work in a lower level of abstraction: we would rather like to refer to the variable θ_1 as `theta[1]`, that is, think about the indices as a indices referring to random variables and not as a indices referring to dimensions of a matrix consisting of simulations.

Further, to express the multiplication of two random matrices Θ and Σ, we would like to write `Theta %*% Sigma` and not a loop such as the one shown above. The result of such an expression should be a random matrix object.

While writing loops and using other programming structures and specialized functions are certainly not difficult things to do, nevertheless it takes our mind away from the essence of our work: Bayesian data analysis. Program code intended to implement manipulation of posterior simulations while it does not resemble mathematical notation is but an attempt to emulate such notation.

As Bayesian data analysts and programmers, we think of constants as special cases of random variables: they are just variables having point-mass distributions and represented by a single simulation draw. We do not wish to treat constants and random variables as two separate incompatible classes of objects. Our programming environment should be able to treat both constants and random variable objects equally: whatever one can do with constants one should be able to do with random variables. More specifically, any function that accepts a numeric vector (or matrix) should be able to accept a random vector (or matrix) instead, producing a new random variable object as the result. Therefore, for instance, to compute the distribution of the sample standard deviation statistic of a random vector θ, one should be able to write `sd(theta)` and obtain a scalar-valued random variable—that summarizes the uncertainty in the distribution of the function `sd`, that is, $\sqrt{\frac{1}{k-1}\sum_{i=1}^{k}\theta_i}$.

The very fact that constants are just very special random variables (with point-mass distributions) hints at us that the syntax involving random variable objects should be no different from that involving regular numeric variables. If we compute a function of a random vector or matrix, we obtain a random vector or matrix; if we "let the variance of the components go to zero," the computation should be equivalent to a computation with numeric vectors or matrices, returning a (constant) numeric result.

Imputation In a programming environment that treats numeric constants and random variables as equals, it should be natural that random variable objects can be embedded in vectors that contain constants. The resulting vector is a "mixed" random vector object, where constants appear as if they were random variables with zero variance. Constant components behave as if they consisted of L draws from a degenerate distribution, although only one number is stored in memory.

This would make possible the straightforward *imputation* of missing values with random variables. Since such mixed vectors are random variable objects, they should be accepted as arguments by any numeric function.

Generating replications Without random variable objects, generating replications y^{rep} is done by repeated calls to a random variate generating function (in R, the loop is executed automatically if vectors are given as arguments); see the example above. Naturally, these functions are designed to work only with numbers, so we need to write new functions that can take random vectors and arrays as arguments and return new random variable objects containing newly generated simulations.

Let us return to the example above where the matrix of simulations of y^{rep} was made by generating $y^{\text{rep}(\ell)} \sim \text{N}(\mathsf{X}\beta^{(\ell)}, \sigma^{(\ell)})$. By introducing a new "normal random variable generating function," call it `rvnorm`, we will be able to obtain the random variable object representing the distribution of y^{rep} by the function call,

```
y.rep <- rvnorm(mean=X %*% beta,
sd=sigma)
```

where both the mean and standard deviation parameters are random variable objects. The mean is a vector and standard deviation is a scalar, applying to all components of the mean. For non-normal models we will need to write other random variable generating functions that draw from other distributions.

Graphical summaries Graphical summaries for random variable objects, such as interval plots, histograms, scatterplots and contourplots of simulations need to be implemented by writing new, special functions that accept random variable objects.

In many cases we may extend existing methods. Take for example `plot(x,y)`, which plots a scatterplot of two numeric vectors. A scatterplot of two random scalars (x, y) should produce a cloud of points representing draws (simulations) from the joint distribution of (x, y). If the x-coordinate of a pair of scalars (x, y) is a constant and the y-coordinate is a random variable object, we would expect to see an uncertainty *interval* of y as a vertical line. Similarly if y is constant but x is random, we expect to see a horizontal uncertainty interval. If we want we could plot the median or the mean as a point on top of the interval. Moreover, using colors or varying thickness of the line, we can plot for example 50% and 80% intervals on top of each other. A scatterplot of a pair of a vectors of random variables (x, y) would then amount to several clouds of points, but a vector of constants x and random variables y would be displayed as a series of vertical uncertainty intervals.

Other functions Many random-variable specific functions need to be written, for example a function returning means, quantiles, medians, and standard deviations of each component of a random vector or matrix.

Distinguishing rowwise and columnwise operations It is important to realize that the "means of the simulations of the components of a random vector" are quite different from the "mean of a random vector." The former is a "columnwise" operation on a matrix of simulations, returning constants, and the latter is a rowwise operation, returning a vector or a matrix of simulations. In other words, the columnwise operations summarize the uncertainty of each scalar random variable by a number, but the rowwise operations are functions of the random vectors themselves, returning a new random vector or matrix.

If `theta` is a random vector object, the function call `mean(theta)` must return a random variable object, since `mean` is defined to take a vector and return a scalar. Giving `mean` a random vector must return a random scalar, which then represents the distribution of the random variable $\frac{1}{k}\sum_{i=1}^{k}\theta_i$. Internally the result of `mean(theta)` must contain the simulations $\frac{1}{k}\sum_{i=1}^{k}\theta_i^{(\ell)}$ for $\ell = 1, \ldots, L$.

The same logic goes for all numeric functions such as the variance (`var`), covariance (`cov`), and any user-defined numeric function. "Randomness in, randomness out." In some cases the output may of course be a constant, but as explained before, they are just special cases of random variables.

However, if we want to return numerical summaries of the simulations such as the posterior means and standard deviations, we need to write new functions. For example, a function `rvmean(theta)` would return the posterior means for each component of θ; `rvsd(theta)` the standard deviations, `rvmedian(theta)` the medians, and so forth.

3 Implementation

Taking the above ideas into consideration, we have written a working prototype of a random-variable enabled programming environment within R. This collection of functions is implemented as an R "package" which extends the functionality of R. Once the package is loaded and enabled, random variable objects can be generated and used as arguments in almost any numerical functions. A collection of random-variable specific functions that implement graphical and numerical summaries and functions that generate new random variable objects from various distributions are available.

A random variable object is implemented internally as a list of vectors of simulations. The simulations are not kept in matrices but each "column" is kept separate for convenience; one vector of simulations corresponds to one component of a random vector or a matrix. Since random matrices are just random vectors with dimension attribute, and random scalars are just random vectors of length 1, in the following whatever is referred to as "random vectors" will also apply to random scalars and random matrices.

Applying arithmetic operations and elementary functions on a random vector object produces a new random vector object that consists of a list of appropriate vectors of simulations. If a plain numeric vector object is imputed as a random scalar or a vector, the resulting object is a new random vector with the remaining numeric components being treated as "vectors of simulations of length 1."

In an interactive computing environment, we may type the name of an object (variable) on the console and view immediately its value. It is obvious what we expect to see when typing the name of a numerical vector or an array, but how should a random variable object be displayed? Since a random variable object consists of independent draws from a distribution, it is natural to view quantiles and such numerical summaries as the mean and standard deviation of the simulations. The mean of the simulations is an estimate of the expectation of the random variable.

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, and the 1%, 2.5%, 25%, 50%, 75%, 97.5%, and 99% quantiles. Most functions can be adapted easily to accept random variable objects as arguments.

The best way to illustrate how the new programming environment works is to give some examples.

4 Examples

4.1 Prediction and multiple imputation

We illustrate the use of our programming environment with a simple example of regression prediction using posterior simulations. 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 $p(\beta, \sigma) \propto 1/\sigma$.

4.1.1 Posterior predictive distribution

Our goal is to impute the missing values with draws from the posterior predictive distribution of y. We do this 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 model.

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

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

In our programming environment, this mathematical formula translates to the statement,

```
z     <- rvchisq(df=n-2)
sigma <- sigma.hat*sqrt((n-2)/z)
```

The function `rvchisq` returns a random variable object consisting of L independent and identically distributed simulations from the chi-square distribution. The quantities `sigma.hat` and `n` statement are fixed scalars. Since `sigma` is a constant divided by a random variable `z`, it is a random variable object. we treat this variable in our program as if it were the actual random variable of the mathematical model. The posterior distribution of β is $\beta|\sigma, x, y \sim \mathrm{N}(\hat{\beta}, V_\beta\sigma^2|x, y, \sigma^2)$, which can be simulated by,

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

where `rvnorm` ("normal random variable") is a function returning a vector of (multivariate) Gaussian random variables, with each component consisting of simulations. The argument `var` accepts a variance matrix, which in this case depends on the random variable `sigma`. Thus we are actually drawing from a distribution of β, averaging over the uncertainty of σ.

The length of `beta` is determined by the arguments: in this case `beta` has the same length as the mean vector, `beta.hat`.

Our implementation imitates the corresponding mathematical expressions as much as possible; however, an equivalent program written in plain R could look like,

```
sigma <- array(NA,L) # Allocate a vector
beta  <- array(NA,c(L,2))
# Allocate a matrix
for (sim in 1:L) {
    sigma[sim] <- sigma.hat*sqrt((n-2)/
      rchisq(1,n-2))
    beta[sim,] <- mvrnorm(1, beta.hat,
      V.beta*sigma[sim]^2)
}
```

The "traditional" way of writing R code is longer and less robust: besides having to implement a looping structure, one must constantly mind the dimensions of the simulation matrices and the slightly awkward index notation where `beta[sim,]` refers to the simulation number `sim` from the joint distribution of $(\beta_1, \beta_2, \beta_3)$; in contrast, `beta[,1]` is the vector of L simulations for β_1.

According to the model, y is normal with mean $\mathsf{X}\beta$ and standard deviation σ. The predictions for the missing y values are obtained by the natural statement,

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

where `rvnorm` returns independent and identically distributed normal variables, but here the two arguments, the mean and standard deviation, are both random variables; thus we are drawing y from its marginal distribution, averaging over the uncertainty of both σ and β. `x[is.na(y)]` simply picks the covariates for the missing five students. Since `beta[1]` and `beta[2]` are scalar-valued random variables, the mean will be a vector of the same length as `x[is.na(y)]`, that is, five. The resulting vector `y.pred` is also a vector of length five.

Point estimates for the estimated parameters are obtained using predefined functions, for example, the posterior mean (expectation) for β is given by `rvmean(beta)`, and the medians of β_1, β_2 by `rvmedian(beta)`.

The distributions of the predicted values are quickly summarized by typing the name of the variable on the console:

```
> y.pred
     name mean sd       1% 2.5%  25%  50%  75% 97.5%  99%   sims
[1] Alice 68.9 19.5 ( 17.8 28.7 56.9 69.2 81.0   106  116 ) 1000
[2]   Bob 73.6 17.6 ( 28.3 38.8 63.0 73.5 84.1   108  118 ) 1000
[3] Cecil 65.7 20.7 ( 17.3 22.7 52.2 66.0 79.1   105  115 ) 1000
[4]  Dave 70.3 17.5 ( 24.4 34.4 59.5 70.7 81.5   103  109 ) 1000
[5] Ellen 74.8 18.0 ( 34.0 40.9 64.3 73.8 85.2   113  122 ) 1000
```

4.1.2 Imputing the predicted values

Our object-oriented framework allows for combining constants with random variables into a single vector or array. Thus we can impute the random variables into the original vector `y` which used to hold only constants and missing values. This is done by the R statement,

```
y[is.na(y)] <- y.pred
```

which replaces the missing values (indicated by the symbol `NA`) by their corresponding predictions.

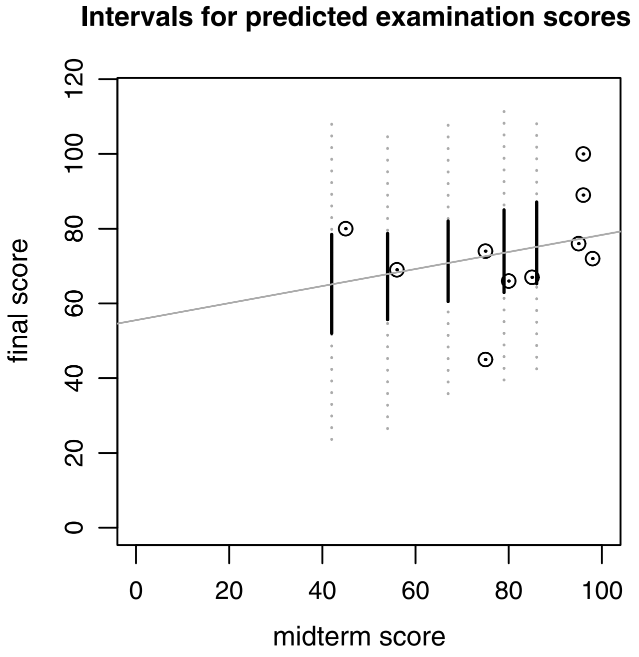

Fig. 1 Predicting the final examination scores: uncertainty intervals for the five predicted final exam scores. This is a scatterplot of the vector pair (x, y) where the components x are all constant and y includes ten constant and five random components. This is done by a generic function call "plot(x, y)"; the five pairs (x_i, y_i) where y_i is random are automatically drawn as intervals (the 50% intervals are shown as solid vertical lines and the 95% intervals as dotted vertical lines.) The observed pairs (x_i, y_i) are shown as *circles*.

This particular step would be impossible in standard R: each component `y.pred` is internally represented by a matrix of possibly thousands of rows of simulations, but the left-hand side of the assignment, `y[is.na(y)]`, is a numeric vector of length 5.

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.

4.1.3 Computing functions of random vectors

A 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       1% 2.5%  25%  50%  75% 97.5% 99%   sims
[1]  72.8 3.39 ( 64.2   66 70.7 72.8 74.8  79.2  81 ) 1000
```

The "mean" here is not the expectation of the random vector y, but the distribution of the arithmetic average of the 15 components of the vector y.

4.1.4 Computing probabilities of events

In R, if we use comparison operators with numeric vectors we obtain true/false values; using them with random vectors we obtain distributions of indicators of events. The probability of such an event (or, equivalently, the expectation of the corresponding random variable) is computed using the function `Pr`. For example, 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.

4.2 Regression forecasting

We illustrate the power and convenience of random-variable computations with a nonlinear function of regression predictions. We shall first fit a classical regression, then use simulation objects to summarize the inferences and obtain predictions. As we shall see, it is then easy to work directly with the random vector objects to get nonlinear predictions.

Our data consist of the percentage of the vote of the Democratic party's share of the two-party vote in legislative elections in California in years 1982, 1984, and 1986: v^{82}, v^{84}, v^{86}, respectively. (Missing values were imputed the value 0.5, and vote proportions of uncontested elections (zeros and ones) were imputed 0.25 and 0.75, respectively. This is a simplified version of a standard model for state legislative elections (Gelman and King 1994).)

These are all vectors of length $n = 80$, corresponding to the 80 election districts. We also included predictors that indicate the incumbent party in the district; the values for years 1984 and 1986 were simply computed from the previous election results:

$$p_i^t = \begin{cases} 1 & \text{if } v_i^{t-2} > .5, \\ -1 & \text{if } v_i^{t-2} < .5 \end{cases} \qquad \text{for } t \in \{84, 86\}.$$

The data frame is,

$$\mathsf{V} = \begin{pmatrix} p_1^{88} & v_1^{86} & p_1^{86} & v_1^{84} & p_1^{84} & v_1^{82} \\ p_2^{88} & v_2^{86} & p_2^{86} & v_2^{84} & p_2^{84} & v_1^{82} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ p_n^{88} & v_n^{86} & p_n^{86} & v_n^{84} & p_n^{84} & v_n^{82} \end{pmatrix}.$$

The model for the proportion of the Democratic vote has two predictors: the outcome in the previous election and the incumbent party:

$$v_i^{86}|\theta \sim \mathrm{N}(\beta_1 + \beta_2 v_i^{84} + \beta_3 p_i^{86}, \sigma^2), \quad i = 1, \ldots, n,$$

where $\theta = (n, \mathsf{V}, \beta, \sigma)$ and $\beta = (\beta_1, \beta_2, \beta_3)$.

4.2.1 Posterior predictive distribution

We fit the linear regression model and obtain the least-squares estimates $\hat\beta = (\hat\beta_1, \hat\beta_2, \hat\beta_3)$, the classical unbiased variance estimate $\hat\sigma^2$, and the (unscaled) covariance matrix V_β. Again using a noninformative prior $p(\beta, \sigma) \propto 1/\sigma$, the posterior distribution of σ is, just like in our data imputation example, $\sigma|\mathsf{V} \sim \hat\sigma \cdot \sqrt{(n-3)/z}$, where $z \sim \chi^2(n-3)$, while the posterior distribution of β, given σ, is $\beta|\sigma, \mathsf{V}_\beta \sim \mathrm{N}(\hat\beta, \mathsf{V}_\beta\sigma^2|\sigma, \mathsf{V}_\beta)$. The next step is to generate posterior simulations for the parameters β and σ. These are generated by identical statements as those in the previous example.

We now predict the outcome for the following election year, 1988. The posterior predictive distribution is,

$$v^{pred88}|\theta, y \sim \mathrm{N}(\beta_1 + \beta_2 v^{86} + \beta_3 p^{88}, \sigma^2)$$

which is a normal random vector of length $n = 80$. The indicator of incumbency in 1988, p^{88}, is the indicator of the event $\{v^{86} > 0.5\}$.

```
mu         <- beta[1]+beta[2]*v.86
              +beta[3]*p.88
v.pred.88 <- rvnorm(mean=mu, sd=sigma)
```

4.2.2 Making forecasts

The posterior predictive distribution can be used to compute various forecasts. For example, the predicted average Democratic district vote in 1988 is $\bar v^{pred88} = \frac{1}{n}\sum_{i=1}^n v_i^{pred88}$, that is, the arithmetic average of the predicted values `pred.88`:

```
avg.pred.88 <- mean(v.pred.88)
```

Any linear or nonlinear function of the predictions is obtained in a straightforward way. For example, the proportion of seats obtained by the Democrats is $\bar s^{pred88} = \frac{1}{n}\sum_{i=1}^n \mathbf{1}_{\{v_i^{pred88} > 0.5\}}$, which translates to

```
s.pred.88 <- mean(v.pred.88>0.5)
```

4.3 Posterior predictive checking

Posterior predictive checking is a Bayesian model validation technique where we draw simulations from the posterior predictive distribution of the observed data y and compare them to the observed values (Gelman et al. 2003, Chap. 6). Discrepancies between the observed and simulated values indicate possible model deficiency.

4.3.1 Drawing replicated data

We illustrate with the election example, where the posterior predictive distribution in the election example is the hypothetical distribution of the observations in year 1986; that is, a draw from the posterior predictive distribution of y given the (original) predictors of 1984. These simulations are called *replications* and obtained by,

```
mu        <- beta[1]+beta[2]*v.84
             +beta[3]*p.86
v.rep.86 <- rvnorm(mean=mu, sd=sigma)
```

where `mu` and `sigma` are random variables representing the posterior distributions of μ and σ.

In general, we may consider the linear regression model $y|\mathsf{X}, \beta, \sigma \sim \mathrm{N}(\mathsf{X}\beta, \sigma^2)$, where X is the predictor matrix, β is the k-vector of coefficients and σ is the standard deviation of y. If we include the relevant predictors in a matrix `X`, then we can obtain *either* predictions *or* replications by the statement,

```
rvnorm(mean=X %*% beta, sd=sigma)
```

with a different prediction matrix `X`. The above statement returns a normally distributed random vector given a predictor matrix `X`, the coefficient vector `beta`, and the standard deviation `sigma`. These parameters may contain both constants and random components. The replications can be obtained by the function call,

```
v.rep.86 <- rvnorm(mean=X.84 %*% beta,
                   sd=sigma)
```

where `beta` and `sigma` contain the posterior simulations of β and σ, and X_{84} is the predictor matrix of the year 1984 vote proportions and incumbency indicators, along with the constant predictor column.[1]

4.3.2 *Computing test quantities*

A test quantity, or "discrepancy measure" is a scalar summary of parameter and data that is used as a standard when comparing data to predictive simulations (Gelman et al. 2003). It is straightforward to compute test quantities from the replicated data: one needs to write a function $T(y, \theta)$, where y is a vector that has a similar distribution as the observed values, and θ is the vector containing any given predictors and parameter values.

Arithmetic operations and elementary numeric functions will also work with random variables, so in practice, almost any R function that accepts a *numerical* vector y of length n can be used to compute the distribution of $T(y^{rep}, \theta)$.

In the regression example, one suitable test quantity could be for example the number of "switches" occurred in the $n = 80$ districts of California between consecutive years, that is, number of districts where the incumbent party was defeated: $T^{switch}(v, p) = \sum_{i=1}^{n} |\mathbf{1}_{\{v_i > 0.5\}} - p_i|$. The corresponding R code is `T.switch <- function (v,p) sum(abs((v>0.5)-p))`. This code works with constant vectors `v, p`, but also with random vectors.

Lack of fit of the data with respect to the posterior predictive distribution can be expressed by the Bayesian p-value, which is defined as the expected value of the replicated test quantity being at least extreme as the observed test quantity:

$$p\text{-value} := \mathbb{P}\mathrm{r}\{T^{switch}(v^{rep86}, p^{86}) \geq T^{switch}(v^{86}, p^{86})\}.$$

[1]`X.84 <- cbind(1, v.84, p.86)`. The standard R function `cbind` returns a matrix with given columns.

The corresponding program code is again obvious:

```
p.value <- Pr(T.switch(v.rep.86,p.86)
        >= T.switch(v.86,p.86))
```

The p-value comes to 0.61 in this set of simulations. $T^{switch}(v^{rep86}, p^{86})$ is summarized by

```
     mean   sd   1% 2.5%  25%  50%  75% 97.5% 99%   sims
[1]   4.2 1.89 ( 0    1    3    4    5     8   9 ) 1000
```

5 Discussion

5.1 Summary of advantages

From the Bayesian viewpoint, it is natural to expect the computing environment to accept inputs as either constants or random: we essentially treat all quantities as random variables, constants being just given realizations of them. The absence of the random variable data type forces us to write code that somehow emulates the existence of such a data type, as shown in the examples above. Unless we develop a common framework that will actually implement this data type, we will end up writing similar code over and over again. This will result in longer, more complicated program code that is more likely to contain errors. With random vector objects, our program code becomes compact, easy to read, and easy to debug, since in many cases it will resemble mathematical notation.

To summarize the advantages:

1. *Transparency.* Program code written for numerical vectors can be often used for mixed vectors without modification. No looping structures to emulate simulation-by-simulation computation need to be written.
2. *Flexibility and robustness.* Program code works with both random and mixed input parameters. There is no need to rewrite separate code for generating predictions and replications. The code accepts different (random or constant) types of input.
3. *Intuitive appeal.* Program code resembles more like mathematical notation than "traditional" computer programs. Ugly technical details have been hidden from user's view.
4. *Improved readability.* Short, compact expressions are more readable and easier to understand than traditional code with looping structures and awkward matrix indexing notation. This is especially helpful when one needs to interpret code written by someone else.
5. *Productivity gain.* Compact, intuitive program code is relatively easy to write, easy to debug, and easy to maintain. We can concentrate on Bayesian data analysis and program in a more natural syntax.

All of these advantages stem from the conceptual advantage of thinking in terms of random variable objects rather than just in terms of arrays of simulations.

5.2 Disadvantages

Despite a host of obvious advantages, the object-oriented approach has an obvious disadvantage compared to the traditional way of programming: we do not have the opportunity to *optimize* the code generating the random variates to the specific task.

For example, the multiplication of an indicator (usually obtained by applying a logical operation such as "$>$") by another random variable is implemented by first drawing the L simulations for the indicator, and then drawing the L simulations for the other random variable, and finally multiplying the simulations componentwise. An optimized code would be able to skip the computation of the expression for the second variable if the indicator produced a zero. Customized code would also be able to combine nested function calls, but in the object-oriented method the computer must compute values for function calls one by one.

However, we feel that the gain in speed in most cases may be negligible compared to the effort needed to write, edit, and debug optimized code. Truly speed-critical applications require machine-compiled code written, e.g., in C or Fortran. If possible, such optimization should be done in the "system level" so that the end-users need not worry about such technical details.

5.3 Conclusion

Computing functions of simulations and drawing graphs of them involves manipulating arrays of numbers by writing code that imitates the manipulation of objects that behave like random variables. By introducing a special object class representing random variables and arrays, we can create a programming environment that makes manipulating simulations intuitive and effective.

Our programming environment proves to be especially helpful for Bayesian data analysis, where routinely considering all uncertain quantities as random variables is natural. Although researchers need to understand fully the underlying mechanism of manipulating simulations, we believe that writing code to emulate the manipulation of simulation-based random variable objects is a wasted effort if an application to simplify the programming syntax is available. Our approach should help to make writing, reading, and understanding programs more efficient.

Simulation-based random variable objects can also be used in conjunction with semi-analytical methods for Bayesian inference ("Rao-Blackwellization"; see Gelfand and Smith 1990) such as used in computing Bayes factors (e.g., Chib 1995; Chib and Jeliazkov 2001) and inferences for very small probabilities (e.g. Gelman et al. 1998). In addition, random variable objects can be used for direct probability calculations (Kerman 2005).

Just as vector and matrix computations allow the user to operate at a higher level of abstraction (compare, for example, code in Fortran 77 to Fortran 90), we believe that random variable objects have the potential to facilitate a more "statistical" framework for computing in the presence of uncertainty.

Acknowledgements We thank the National Science Foundation for partial support of this research.

References

Chib, S.: Marginal likelihood from the Gibbs output. J. Am. Stat. Assoc. **90**, 1313–1321 (1995)

Chib, S., Jeliazkov, I.: Marginal likelihood from the Metropolis-Hastings output. J. Am. Stat. Assoc. **96**, 270–281 (2001)

Gelfand, A.E., Smith, A.F.M.: Sampling-based approaches to calculating marginal densities. J. Am. Stat. Assoc. **94**, 247–253 (1990)

Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B.: Bayesian Data Analysis, 2nd edn. Chapman & Hall/CRC, London (2003)

Gelman, A., King, G.: A unified model for evaluating electoral systems and redistricting plans. Am. J. Political Sci. **38**, 514–554 (1994)

Gelman, A., King, G., Boscardin, W.J.: Estimating the probability of events that have never occurred: when does your vote matter? J. Am. Stat. Assoc. **93**, 1–9 (1998)

Kerman, J.: Using random variable objects to compute probability simulations. Technical Report, Department of Statistics, Columbia University (2005)

Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS—a Bayesian modelling framework: concepts, structure, and extensibility. Stat. Comput. **10**, 325–337 (2000)

Oldford, R.W.: The Quail project: a current overview. Invited paper, 30th Symposium on the Interface, Minneapolis (1998). http://www.stats.uwaterloo.ca/~rwoldfor/papers/Interface1998/paper.pdf

R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna (2004)

Tierney, L.: LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics. Wiley, New York (1990)

Sturtz, S., Ligges, U., Gelman, A.: R2WinBUGS: a package for running WinBUGS from R. J. Stat. Softw. **12**(3), 1–16 (2005). ISSN 1548-7660