Fully Bayesian Computing

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

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

April 16, 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.

We illustrate the use of this new programming environment with several examples of Bayesian computing, including posterior predictive checking and the manipulation of posterior simulations. 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. Now we have plenty of computing power and some convenient applications to use in our work, but the current implementations are not fully up to the requirements of Bayesian data analysis, especially when the user wants to make predictions or do other things with the parameters beyond simple summaries such as posterior means, standard deviations, and uncertainty intervals.

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,

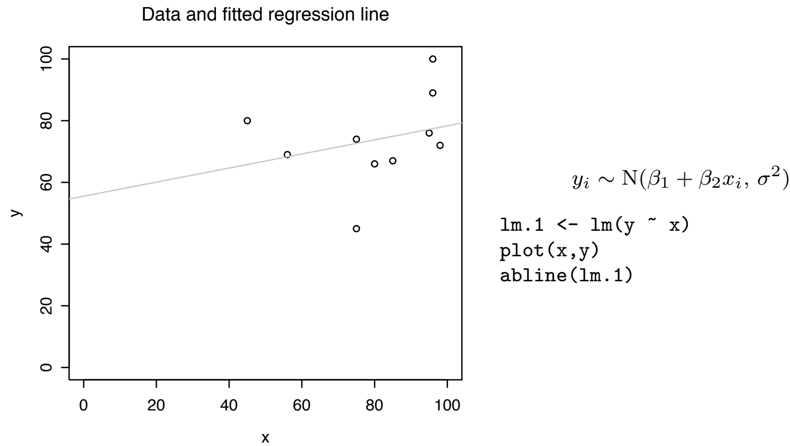

Figure 1: Predicting the final examination scores: fitted regression line.

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,

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

```
> 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 2: Quick summary of the posterior predictive distribution of y is obtained by typing the name of the vector (`y.pred`) on the console.

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))
```

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 2.

We now can impute the predicted values. Our object-oriented framework allows us to combine constants with random variables: `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. This particular step would be impossible in standard R: each component `y.pred` is represented by possibly thousands of simulations, but the user only sees a variable `y` that has the same length as before.

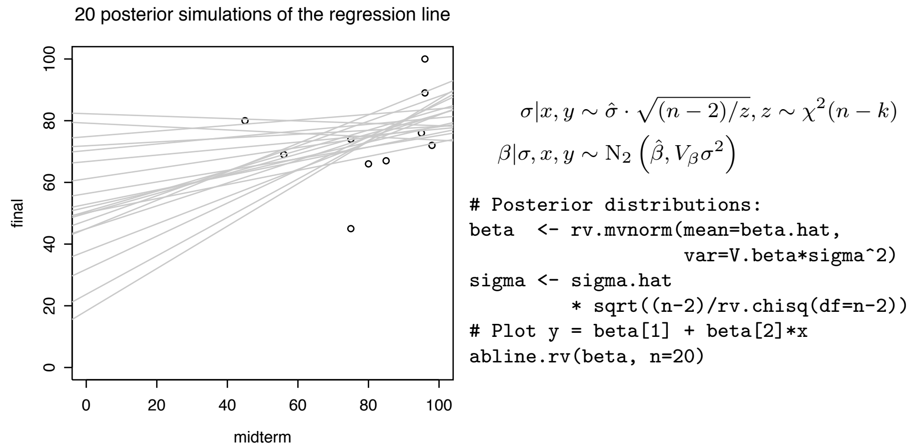

Figure 3: Predicting the final examination scores: 20 simulated regression lines $y = \beta_1 + \beta_2 x$ from the posterior distribution of (β_1, β_2).

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 )
```

and 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.

The random-variate generating functions such as `rv.norm`, `rv.chisq` generate a fixed number of simulations. This number is determined by the value assigned to a global variable. It is denoted by `L` in the sample code and by L in the mathematical notation.

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

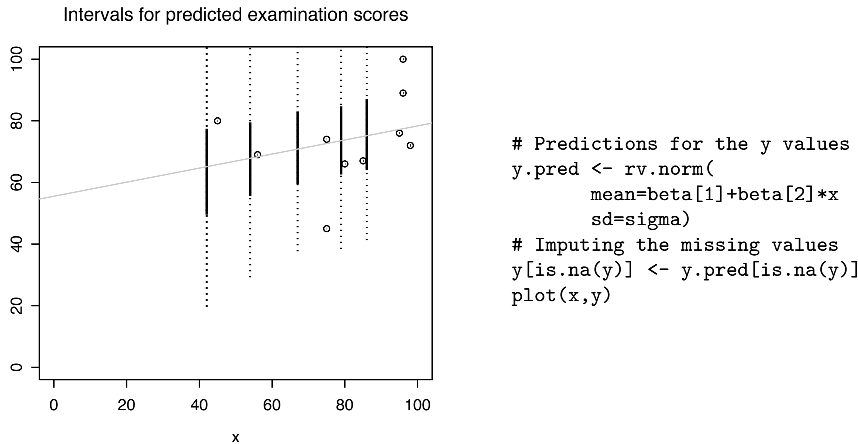

Figure 4: Predicting the final examination scores: uncertainty intervals of the five predicted final exam scores. The 50% intervals are shown as solid vertical lines and the 95% intervals as dotted vertical lines. The observed values are shown as circles.

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.

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] The unspecified dimension, for example in `beta[,2]`, refers to the dimension containing the vector simulations of β_2. This is not only confusing, but forces us to work in a lower level of abstraction: we would rather like to refer to the variable β_2 as `beta[2]`, that is, think about it as a random variable and not as a dimension of a matrix consisting of simulations.

For another example, suppose we want to work with a rotated vector $\gamma = R\beta$, where $R = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{pmatrix}$ where θ is a random variable taking values in $[0, 2\pi)$, represented the scalar random variable object `theta`; R is therefore a random matrix. To generate R, we write

```
R <- rv.matrix(c(cos(theta),sin(theta),sin(theta),-cos(theta)),c(2,2))
```

which appears to be a 2×2 random matrix but internally is a $L \times 2 \times 2$ matrix of simulations; each component in the matrix is a random variable object. To multiply R by a 2×1 random vector β, we can then simply write

[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.

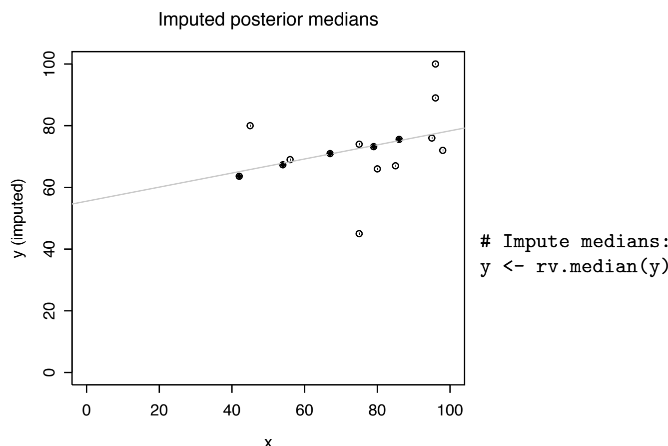

Figure 5: Predicting the final examination scores: imputed posterior medians (shown as black dots) of the predicted values.

```
gamma <- R %*% beta
```

which produces a 2×1 random vector `gamma`. If instead we worked with a 3-dimensional matrix of simulations `R` and a 2-dimensional vector simulations `beta`, we would have to write code that looks like

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

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.

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*

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

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.

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. [2] The gain may seem to be slight, reducing the expression only by omitting the commas. Suppose now, however, that we have a function `f` that

[2] In R there are other ways to write this loop. For example, `y <- apply(x.sim[,1:2], 1, '+')` applies the summation function `+` to each of the rows of the first two columns of the simulation matrix `x.sim`.

computes the ratio of the two first components of a vector: `f <- function(x) x[1]/x[2]`. We can then get a meaningful result by calling `f(x)` when `x` is a random vector object, a numeric vector of constants, or a vector with some components numbers and some random variable objects. However, `f(x.sim)` will cause a syntax error, since `x.sim` is a two-dimensional matrix and not a one-dimensional vector.

The true gain of introducing random variable objects is not only the simplification of notation, although this by itself would be a worthy goal. The true gain is the higher level of abstraction, which allows us to think in terms of more natural Bayesian entities: random variables.

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)) # Allocate an L*k*m matrix
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 vector `z` of n iid standard normal random variable objects, we will write,

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

where `z` 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 accessed via the method `sims(z)` if needed.[3]

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:

$$p(z) := \int\int \mathrm{N}(z|\mu,\sigma)p(\mu,\sigma)\,\mathrm{d}\mu\,\mathrm{d}\sigma.$$

where $p(\mu,\sigma)$ is the joint density of the 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.

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

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. We aim to work at a higher conceptual level, and apply user-level functions directly to random variable objects, not to the simulations.

The 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.

For convenience we have supplied some special componentwise functions. These functions are prefixed by `rv`: `rv.mean(x)` is equivalent to `simapply(x,mean)` and thus gives componentwise means for each component of a random vector `x`. Similarly, `rv.sd(x)` gives the componentwise standard deviations and `rv.quantile` gives componentwise quantiles. Since the componentwise mean is used quite often, we have also supplied aliases for the command `rv.mean(x)`, namely `E(x)` and `Pr(x)`. `E(x)` denotes the expectation and indeed does return the expectation of the vector x. `Pr()` stands for probability and is recommended to be used when the argument is an event, e.g. `Pr(x>0)` which indeed returns the probability of the event $\{x > 0\}$.

The conceptual difference of the mean `mean(x)` and the expectation `E(x)` may be hard to grasp at first, but `E` works actually as the expectation operator $\mathbb{E}(\cdot)$ in probability theory, returning the expectations of each component. Also, `mean(x)` indeed takes a vector and computes the arithmetic mean. If any component is random, we will get a random variable. If the variances of the components of `x` approach zero, `mean(x)` will approach the arithmetic mean of the limit. The true confusion here is caused by the term “mean” which is used to refer to both arithmetic mean and the expectation.

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 2.

2.5 Graphical summaries

We have provided some basic graphical summary methods that work on the random vector objects. `plot(x)`, given a numeric vector `x`, plots the components against the index as dots. If `x` contains random components, `plot(x)` will then draw uncertainty intervals for each random component of `x`. Thus if the variances of random components reduce to zero, the uncertainty intervals reduce to points and the command reduces to the basic `plot` function. An example is shown in Figure 4.

`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.

3 Examples

We illustrate with two examples, one from probability and one from Bayesian statistics.

3.1 Simulating a stochastic process

Let us consider the problem of simulating the price path S_t of a financial asset. We use the usual assumption that the returns S_{t+d}/S_t are identically and independently distributed over all nonoverlapping intervals $(t, t+d)$. Further, assuming a lognormal distribution for the returns and an arbitrage-free market (Karatzas and Shreve, 1998), we may write the price S_t on day t as

$$S_t = S_0 \exp(X_t).$$

S_0 is today's market price of the asset, and X_t is a Gaussian random process with mean $(r - \frac{1}{2}\sigma^2)t$ and standard deviation $\sigma\sqrt{t}$, where r is the daily risk-free interest rate. The standard deviation σ is called the daily volatility.

The formula for the random process X_t is easily expressed as a function `X(t)`:[4]

```
X <- function (t) {
  rv.norm(mean=t*(r-sigma^2/2), sd=sigma*sqrt(t))
}
```

[4] `X(t)` in this example just produces the simulations for a random variable X_t; `X(1:10)` will not produce the path for ten days, although such a program can be easily written.

For simplicity we assume r and σ are global variables, which may be random or fixed; they may be draws from some prior or posterior distribution. The final asset price S_t for time t given a starting value S_0 is then simulated by the statement

```
S.0 * exp(X(t))
```

A portfolio of assets. Now suppose `sigma` and `S.0` are vectors of length k. $\sigma = (\sigma^{(1)}, \ldots, \sigma^{(k)})$ is a k-vector of constants or random variables of volatilities of the corresponding k assets in a portfolio, and $S_0 = (S_0^{(1)}, \ldots, S_0^{(k)})$ is a k-vector of the current prices of the assets.

The statement `S.0 * exp(X(20))` would then simulate the k closing prices at the end of the 20th trading day (say, end of the month). `X(20)` is a vector of k iid normal random variables since `sigma` is a k-vector. The length of the vectors has changed, but the code stays the same.[5]

The total value of the portfolio at time t is $P_t = \sum_{i=1}^{k} S_t^{(i)}$, given by

```
sum(S.0 * exp(X(t)))
```

which is the sum of k random variables.

The probability that the we lose at least 20% of the value of the portfolio by the end of the month ($t = 20$) is $\mathbb{P}\{P_{20}/P_0 < 0.80\}$ can be computed by the short program

```
P.0  <- sum(S.0)
P.20 <- sum(S.0*exp(X(20)))
print(Pr(P.20/P.0<0.80))
```

Value-at-risk. The d-day *value-at-risk at* α% is defined as the number VaR such that $\mathbb{P}\{P_d < P_0 - \text{VaR}\} = \alpha$; that is, the (negative of) the α-quantile of the distribution of $P_d - P_0$. VaR is therefore a portfolio manager's "worst-case scenario" for a given time frame.

Take $\alpha = 0.01$ and $d = 20$. The 20-day Value-at-risk at 1% of our portfolio is obtained by

```
VaR <- -rv.quantile(P.20-P.0, 0.01)
```

using the definitions of `P.20` and `P.0` above.

Financial option pricing. Many kinds of financial options (contingency claims) have complex payoff schemes and thus have no closed-form solutions for their present values. Consider a *European barrier call*

[5] This example assumes independence of the assets, which is unrealistic, but illustrates the convenience of creating iid random variables.

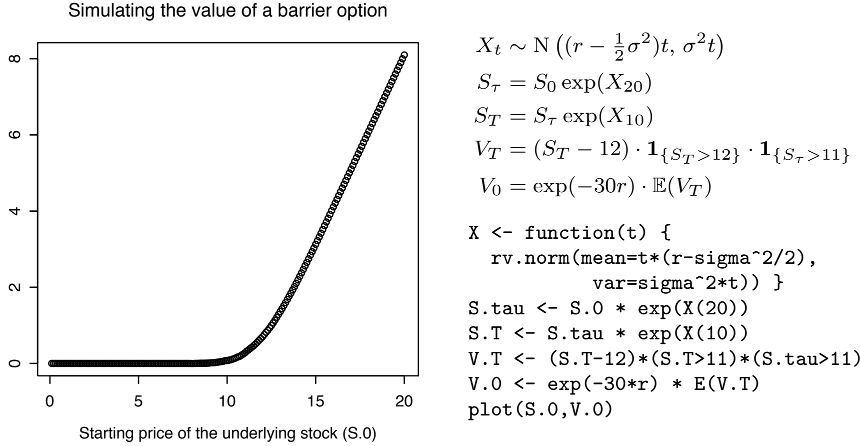

Figure 6: Simulating the value of an option: plotting the present value V_0 of the option as a function of current values S_0. The corresponding program code closely matches the mathematical formulas.

option, whose value depends on the event that the underlying asset exceeds a certain preset level at some preset time.

The value of the option is denoted by V_t at time point t. The option activates if the price S_t of the underlying asset (say, a stock) exceeds a given level α at a given time point $\tau < T$. At $t = T$, the option expires; then the holder of the option has the right to claim the payoff $(S_T - \beta)^+$, but only if the option is activated. For a numerical illustration, let us set $\tau = 20$, $T = 30$ and $\alpha = 11.00, \beta = 12.00$.

At expiry ($T = 30$) the option has the value

$$V_T = (S_T - 12)^+ \cdot \mathbf{1}_{\{S_\alpha > 11\}} = (S_T - 12) \cdot \mathbf{1}_{\{S_T > 12\}} \cdot \mathbf{1}_{\{S_\alpha > 11\}}$$

The indicator variables translate directly into the Bernoulli random variable objects `(S.T > 12)` and `(S.tau > 11)`. S_T is now given by $S_T = S_\tau \cdot \exp(X_{T-\tau})$. [6]

```
S.tau <- S.0 * exp(X(20))
S.T   <- S.tau * exp(X(10))
V.T   <- (S.T-12)*(S.T>11)*(S.tau>11)
```

The present value V_0 of the option is then given by the "risk-neutral expectation" $V_0 = \exp(-rT) \cdot \mathbb{E}(V_T)$:

```
V.0 <- exp(-r*30) * E(V.T)
```

[6] One must not make the mistake of generating a new random variable $S_0 \cdot \exp(X_T)$, since this almost surely not equal to S_T; the realization of the random process must pass through S_τ.

If `S.0` is a vector containing m different possible prices for the underlying asset, `V.0` will be a vector of m constants giving the present values of the option given the different starting values. This is shown in Figure 6.

Here again the Bayesian paradigm of treating any quantity as random is easily applied: our code does not need to incorporate the knowledge of specific quantities being constant or random, or of certain length; we may write a function returning the value of a quantity, and later we can decide which inputs are constant and which are random and what length the vectors should possess.

3.2 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.[7] 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:

$$p_i^t = \begin{cases} 1 & \text{if } v_i^{t-2} > .5 \\ -1 & \text{if } v_i^{t-2} < .5 \end{cases} \quad \text{for } t \in \{1984, 1986\}.$$

The data frame is

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

The model for the proportion of the Democratic vote has two predictors: the outcome in the previous

[7]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).

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

Posterior 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 $(\beta, \sigma) \propto 1/\sigma^2$, the posterior distribution of σ is, just like in our data imputation example, $\sigma|V \sim \hat{\sigma} \cdot \sqrt{(n-2)/x}, \quad \text{where} \quad x \sim \chi^2(n-k)$, while the posterior distribution of β, given σ, is normal with mean $\hat{\beta}$ and covariance matrix $\sigma^2 V_\beta$ is given by $\beta|\sigma, V_\beta \sim \mathrm{N}\left(\hat{\beta}, V_\beta \sigma^2 | \sigma, V_\beta\right)$. The next step is to generate posterior simulations for the parameters β and σ. Our implementation imitates the mathematical expression as much as possible:

```
x     <- rv.chisq(df=n-2)
sigma <- sigma.hat  * sqrt((n-2)/x)
beta  <- rv.mvnorm(mean=beta.hat, var=V.beta*sigma^2)
```

where the function `rv.chisq()` returns an independent chi-square random variable; `rv.mvnorm()` returns a normally distributed random vector of the same length as `beta.hat`, given a covariance matrix. These objects are both internally represented by L simulations for each vector component.

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-k)/rchisq(1,n-k))
  beta[sim,] <- mvrnorm (1, beta.hat, V.beta*sigma[sim]^2)
}
```

That "traditional" way of writing 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.

Point estimates for the estimated parameters are obtained using the predefined functions, for example, the posterior mean for β is given by `E(beta)`, and the medians of β_1, β_2 by `rv.median(beta)`. A comprehensive summary of the simulation distributions is most conveniently obtained by entering the name of the vector

on the console, as explained above.

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

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

which is a normal random vector of length $n = 80$. The indicator of incumbency in 1988 p^{88} is the indicator $\mathbf{1}_{\{v^{86}>0.5\}}$. Our program code resembles the mathematical notation as much as possible:

```
mu      <- beta[1]+beta[2]*vote.88+beta[3]*inc.88
pred.88 <- rv.norm(mean=mu, sd=sigma)
```

The mean and standard deviation may be random.

A corresponding "traditional" version of the code is but a crude attempt to emulate the simplicity of the above program:

```
xlen <- length(vote.88)
pred.88 <- array(NA,c(L,xlen))
for (i in 1:xlen) {
 for (sim in 1:L) {
  mu <- beta[sim,1]+beta[sim,2]*vote.88[i]+beta[sim,3]*inc.88[i]
  pred.88[sim,i] <- rnorm(1, mu, sigma[sim])
 }
}
```

Moreover, this version does not work in cases where one or more parameters are constants, whereas the Fully Bayesian version works for both numerical, random, and mixed vectors.

Making forecasts. The posterior predictive distribution can be used to compute various forecasts. For example, the predicted average Democratic district vote in 1988 is

$$\bar{v}^{pred88} = \frac{1}{n}\sum_{i=1}^{n} v_i^{pred88}.$$

This is the average of the predictive distribution:

```
avg.pred.88 <- mean(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(pred.88>0.5)`.

4 Toward a fully Bayesian computing environment

BUGS and other programs that produce posterior simulations are environments for obtaining Bayesian inferences. So far we have introduced a new extended programming environment within R where posterior simulations can be manipulated and predictions produced in a more intuitive way. Bayesian data analysis involves not only fitting models but also building confidence in them. In the future, a fully Bayesian computing environment should integrate Bayesian model building, model checking, and software validation. As always, we can already do everything with our current programming tools (or, for that matter, with any Turing machine), but in practice, there is a lot of computational overhead involved, as shown in the case of manipulation of simulations. The random variable data type solves the problem of computational overhead for manipulation of posterior simulations, and it should be of great help in Bayesian model checking and validation.

Let us have a look at the common Bayesian model validation techniques, posterior predictive checking, computation of test quantities and Bayesian p-values, and software validation using simulated observations (also called “fake data”).

4.1 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, Chapter 6). Discrepancies between the observed and simulated values indicate possible model deficiency.

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]*vote.84+beta[3]*inc.84
rep.84 <- rv.norm(mean=mu, sd=sigma)
```

where `mu` and `sigma` are random variables.

In general, we may consider the linear regression model $y|X, \beta, \sigma \sim \mathrm{N}(X\beta, \sigma^2)$, 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 conveniently obtain *either* predictions *or* replications by the statement

```
rv.norm(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

```
rep.84 <- rv.norm(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:[8]

```
X.84 <- cbind(1, vote.84, inc.84)
```

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: $T^{switch}(v, p) = \sum_{i=1}^{n} \left|\mathbf{1}_{\{v_i > 0.5\}} - p_i\right|$. The corresponding R code is simply

[8] The standard R function `cbind()` returns a matrix with given columns.

`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:

$$p\text{-value} := \mathbb{P}\mathrm{r}\left\{T^{switch}(v^{rep}, p^{84}) \geq T^{switch}(v^{84}, p^{84})\right\}$$

The corresponding program code is again obvious:

```
p.value <- Pr(T.switch(rep.84,inc.84) >= T.switch(vote.84,inc.84))
```

The p-value comes to 0.873 in this set of simulations. $T^{switch}(v^{rep}, p^{84})$ is summarized by

```
mean   sd   Min 2.5% 25% 50% 75% 97.5% Max
2.31 1.64 (   0    0   1   2   3     6   9 )
```

4.2 Software validation using fake data

By *fake data* we mean computer-simulated observations generated from the model with arbitrary distributions for the parameters. Validation for Bayesian software has been recently studied by Cook et al. (2004). Fake data have the same structure as the observations but potentially a completely different probability distribution. In the previous example, the random variate generating function

```
rv.norm(X %*% beta, sd=sigma)
```

can also be used to generate fake data if we use some set of arbitrary inputs X, β, σ as arguments.

If we choose constants for the parameters β, σ, then the distribution of $v^{fake} \sim \mathrm{N}(X\beta, \sigma^2)$ is then completely known. This fact can be exploited in *software validation*, that is, verifying that the program code works the way we expect it to be. Here we are interested in verifying that the program that produces the estimates does indeed produce inferences based on the model.

First, we draw v^{fake} from a known distribution. Second, we run our program and get the estimates for the parameters β and σ; this can be done using BUGS or a customized MCMC sampling program. Third, we compare the estimates and the actual parameter values. Errors in the program code will tend to manifest themselves as discrepancies between the estimates and "true" parameter values.

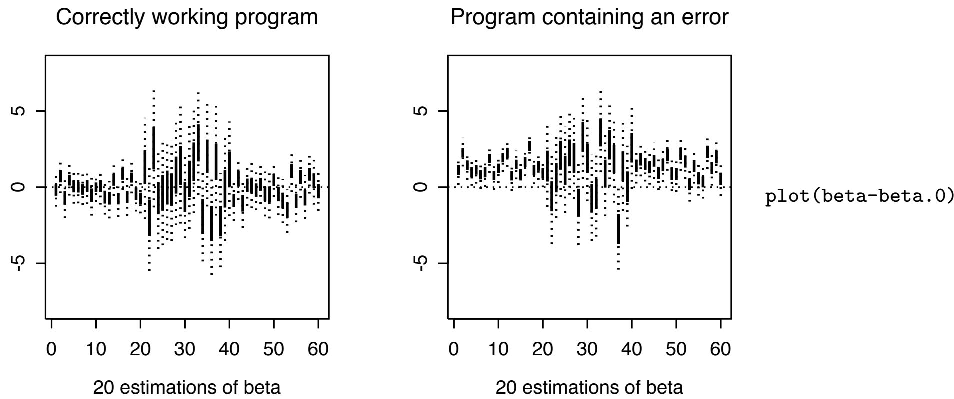

Figure 7: Software validation with fake data: comparing the “true” parameter values with the estimates obtained from the program for the election example. The plot on the left shows the differences between the true parameter β_0 and the estimated values. The plot shows 20 estimations of β for each component β_1, β_2, and β_3. The plot on the right shows 20 estimations by a program that has an error; the estimated 95% intervals (dotted vertical lines) should contain zero 95% of the time, but in the plot on the right we can see that too many intervals do not contain zero. This procedure is a quick device which usually indicates if there are errors in the simulation-generating program, but sometimes errors do not show this clearly.

We draw a set of observations from the prior distribution of (β, σ). Since the actual prior is noninformative, for convenience we choose values near zero and one.

```
beta.0 <- rnorm(3)
sigma.0  <- exp(rnorm(1))
v.fake <- sample( rv.norm(mean=X.84 %*% beta.0, sd=sigma.0) )
```

The function `sample()` draws one set of observations from the distribution of v^{fake}. Next, we obtain the posterior distributions of β, σ: `beta` and `sigma`. To compare the estimates and the true values β_0, σ_0, we may for example draw a scatterplot of the differences. Since `sigma` is random, `sigma-sigma.0` is also random; our version of the `plot()` method for random vectors draws credible intervals (say, 95%) for each component. If repeated, we expect that these intervals include zero 95% of the cases.

Having only four parameters is too few; we may repeat this experiment sufficiently many times (say 20 times) and then look for discrepancies in the estimates. This is illustrated in Figure 7.

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 simulates 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 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, replications, and fake data. 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 users' 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.

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 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

Samantha Cook, Andrew Gelman, and Donald B. Rubin. Validation of software for Bayesian models. Technical report, Department of Statistics, Columbia University, 2004.

Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, London, 2nd edition, 2003. ISBN 1-58488-388-X.

Andrew Gelman and Gary King. A unified model for evaluating electoral systems and redistricting plans. *American Journal of Political Science*, 38:514–554, 1994.

Ioannis Karatzas and Steven E. Shreve. *Methods of Mathematical Finance*. Springer-Verlag, New York, 1998.

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.