rv: a simulation-based random variable class
Version 1.1.0

Jouni Kerman

January 31, 2011

1 Introduction

rv is an implementation of a simulation-based random variable object class for R, originally introduced in Kerman and Gelman [2007]. *rv* implements a new class of vectors that contain a hidden dimension of simulations in each scalar component. The rv objects can be manipulated much like any numeric vectors, but the arithmetic operations are performed on the simulations, and summaries are calculated from the simulation vectors.

rv is convenient for manipulating posterior simulations obtained from MCMC samplers, for example using Umacs [Kerman, 2006] or R2WinBUGS [Sturtz et al., 2005] (the package provides a coercion method to convert `bugs` objects to `rv` objects.)

The paper by Kerman and Gelman [2007] introduces the principles of the design of random variable objects. This document is a short overview of some of the commands provided by the package `rv`. At the end of the document there is a short description of the implementation. A short version of the paper is available as a vignette:

```
    vignette("rv-paper")
```

1.1 Installation

Install the package 'rv' using the Package Installer command in R (from the menu). The version should be 0.920 or later, and load the package using,

```
Package rv loaded.
```

2 A quick tour

The rv objects (or, "random vectors") that we manipulate usually come from a Markov chain sampler. To introduce some commands quickly, we will instead use some random vectors generated by *random-vector generating functions* which sample from a given distribution.

Number of simulations. First, we will set the number of simulations we use. We choose 2500 simulations per each scalar component of a random vector:

```
> rvnsims(2500)
```

```
[1] 1
```

We will not usually change this value during our session, unless we want to repeat our analysis with more (or fewer) simulations. 2500 is also the default value so this is not a necessary step to do every time we start the package.

A Normally distributed random vector. To draw a random Gaussian (Normal) vector of length 5 with corresponding means $1, 2, 3, 4, 5$ and s.d. 1,

```
> x <- rvnorm(mean = 1:5, sd = 1)
```

In effect, the object `x` now contains five vectors of length 1000, drawn (internally) using `rnorm`, but we see `x` as a *vector of length 5.*

The length of the vector is derived from the length of the mean vector (and the sd vector), and it is not necessary to specify a parameter "`n`".

Quick distribution summary. To summarize the distribution of `x` by viewing quantiles, means, and s.d.'s, we only type the name of the object at the console:

```
> x
```

```
    mean   sd    1%    2.5%  25% 50% 75% 97.5% 99% sims
[1] 0.96 0.98 -1.37 -0.9659 0.30 1.0 1.6   2.9 3.2 2500
[2] 1.98 0.99 -0.41  0.0025 1.30 2.0 2.7   3.9 4.3 2500
[3] 3.04 0.99  0.77  1.0405 2.39 3.1 3.7   5.0 5.3 2500
[4] 4.02 1.02  1.64  1.9409 3.36 4.0 4.7   6.0 6.3 2500
[5] 4.97 0.96  2.80  3.1197 4.32 5.0 5.6   6.8 7.2 2500
```

Similarly we can draw from Poisson (`rvpois`) Gamma, (`rvgamma`), Binomial (`rvbinom`).

Componentwise summaries. To extract the means, we use `rvmean`, the s.d.'s, we use `rvsd`, the minimum, `rvmin`, the maximum `rvmax`, and the quantiles, we use `rvquantile`. The componentwise medians are also obtained by `rvmedian`:

```
> rvmin(x)

[1] -3.19 -1.19 -0.24  0.51  2.03

> rvmean(x)

[1] 0.96 1.98 3.04 4.02 4.97

> rvsd(x)

[1] 0.98 0.99 0.99 1.02 0.96

> rvmin(x)

[1] -3.19 -1.19 -0.24  0.51  2.03

> rvquantile(x, c(0.025, 0.25, 0.5, 0.75, 0.975))

        2.5%  25% 50% 75% 98%
[1,] -0.9659 0.30 1.0 1.6 2.9
[2,]  0.0025 1.30 2.0 2.7 3.9
[3,]  1.0405 2.39 3.1 3.7 5.0
[4,]  1.9409 3.36 4.0 4.7 6.0
[5,]  3.1197 4.32 5.0 5.6 6.8

> rvmax(x)

[1] 4.2 5.7 6.4 7.1 8.3

> rvmedian(x)

[1] 1.0 2.0 3.1 4.0 5.0
```

For convenience, there is an alias `E(...)` for `rvmean(...)` which gives the "expectation" of a random vector.

Note. Since the random vectors are all represented by simulations, the expectation and all other functions that we compute are just numerical approximations. Generating a "standard normal random variable" with `z <- rvnorm(n=1, mean=0, sd=1)` will not have an expectation exactly zero. Our main purpose here is to handle simulations, so the answers will be approximate and necessarily involve a simulation error.

Extracting and replacing. Since rv objects work just like vectors, we can extract and replace components by using the bracket notation. Here we replace the 3rd and 4th components with random variables having (an approximate) binomial distributions:

```
> x[3:4] <- rvbinom(size = 1, prob = c(0.1, 0.9))
> x[3:4]
```

```
    mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1] 0.11 0.31  0    0   0   0   0     1   1 2500
[2] 0.89 0.31  0    0   1   1   1     1   1 2500
```

The "mean" column now shows the estimate of the expectation of the two indicator functions we generated.

Imputing into regular vectors. It is possible to "impute" a random vector in a regular numeric vector:

```
> y <- 1:5
> y[3:4] <- x[3:4]
> y
```

```
    mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1] 1.00 0.00  1    1   1   1   1     1   1    1
[2] 2.00 0.00  2    2   2   2   2     2   2    1
[3] 0.11 0.31  0    0   0   0   0     1   1 2500
[4] 0.89 0.31  0    0   1   1   1     1   1 2500
[5] 5.00 0.00  5    5   5   5   5     5   5    1
```

The regular numeric vector is coerced into an rv object with the non-random components appearing as "constants," or in other words, random variables with point-mass distributions (and therefore having a zero variance).

Summaries of functions of random vectors. Standard numerical functions can be applied directly to random vectors. To find a summary of the distribution of the function $1/(1 + \exp(-x_1))$, we would write,

```
> 1/(1 + exp(-x[1]))

    mean   sd  1% 2.5%  25%  50%  75% 97.5%  99% sims
[1] 0.69 0.18 0.2 0.28 0.57 0.73 0.84  0.95 0.96 2500
```

Or of the function of almost anything we like:

```
> 2 * log(abs(x[2]))

    mean  sd   1% 2.5%  25% 50% 75% 97.5% 99% sims
[1]  1.0 1.5 -4.4 -2.8 0.53 1.4 2.0   2.7 2.9 2500
```

Order statistics. To simulate the order statistics of a random vector `x`, we can use `sort(x)`, `min(x)`, `max(x)`.

```
> x <- rvpois(lambda = 1:5)
> x

    mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1]  1.0 0.98  0    0   0   1   2     3   4 2500
[2]  2.0 1.41  0    0   1   2   3     5   6 2500
[3]  2.9 1.75  0    0   2   3   4     7   8 2500
[4]  4.0 1.95  0    1   3   4   5     8   9 2500
[5]  4.9 2.20  1    1   3   5   6    10  10 2500

> sort(x)

    mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1] 0.63 0.70  0    0   0   1   1     2   3 2500
[2] 1.63 0.91  0    0   1   2   2     4   4 2500
[3] 2.74 1.07  1    1   2   3   3     5   5 2500
[4] 3.98 1.30  2    2   3   4   5     7   7 2500
[5] 5.89 1.81  3    3   5   6   7    10  11 2500

> min(x)

    mean  sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1] 0.63 0.7  0    0   0   1   1     2   3 2500
```

```
> max(x)

    mean  sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1]  5.9 1.8 3    3   5   6   7    10  11 2500
```

Note: the `order` method is not implemented.

Random matrices and arrays. *rv* objects behave like numerical vectors in R; thus you can set their dimension attributes to make them appear as arrays, and also use the matrix multiplication operator.

```
> p <- runif(4)
> y <- rvbinom(size = 1, prob = p)
> dim(y) <- c(2, 2)
> y

       mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1,1] 0.094 0.29  0    0   0   0   0     1   1 2500
[2,1] 0.679 0.47  0    0   0   1   1     1   1 2500
[1,2] 0.791 0.41  0    0   1   1   1     1   1 2500
[2,2] 0.890 0.31  0    0   1   1   1     1   1 2500
```

```
> y %*% y

      mean   sd 1% 2.5% 25% 50% 75% 97.5% 99% sims
[1,1] 0.63 0.58  0    0   0   1   1     2   2 2500
[2,1] 0.67 0.58  0    0   0   1   1     2   2 2500
[1,2] 0.78 0.56  0    0   0   1   1     2   2 2500
[2,2] 1.43 0.59  0    0   1   1   2     2   2 2500
```

The componentwise summary functions such as `E` (`rvmean`) and `rvsd` return the summaries with the correct dimension attribute set:

```
> E(y)

      [,1] [,2]
[1,] 0.094 0.79
[2,] 0.679 0.89
```

Creating indicator functions with logical operations. Applying logical operators gives indicators of events. If `z` is a standard normal random variable the indicator of the event $\{z > 1\}$ is given by the statement `z>1`:

```
> z <- rvnorm(1)
> z > 1

    mean   sd sims
[1] 0.16 0.36 2500
```

We can also use the convenience function `Pr(...)` to compute the estimates of the expectation of these indicators:

```
> Pr(z > 1)

[1] 0.16
```

Of course, we can find joint events as well and computer their probabilities similarly. To find the probability that $Z_1 > Z_2^2$, where both Z_1 and Z_2 are independent standard normal, we'd type

```
> z <- rvnorm(2)
> Pr(z[1] > z[2]^2)

[1] 0.27
```

We can even compute probabilities of intersections or unions of events,

```
> Pr(x[1] > x[2] & x[1] > x[4])

[1] 0.020

> Pr(x[1] > x[2] | x[1] > x[4])

[1] 0.20
```

Functions of several random variables. We can use random vectors, regular vectors, standard elementary functions, logical operations in any combination as we wish.

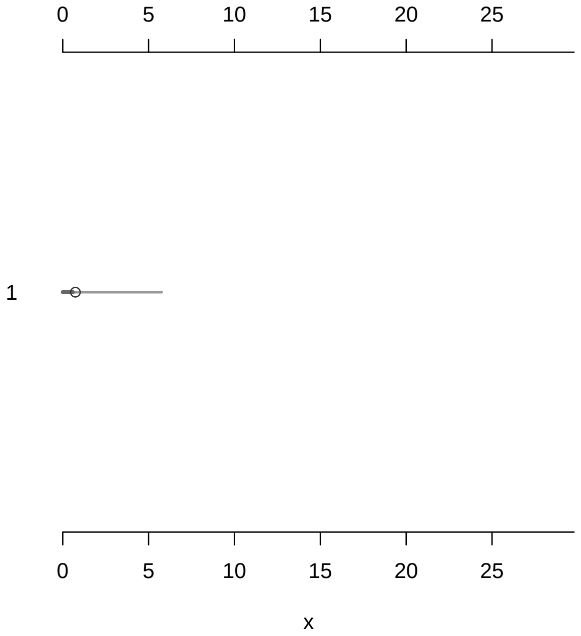
Example. Let z_1, z_2 be standard normal, and let $y_1 = \exp(z_1), y_2 = y_1 \exp(z_2)$. Compute the expectation of $x = (y_1 - 1)1_{y_1>1}1_{y_2>1}$ and find the probability $\Pr(x > 1)$.

```
> z <- rvnorm(n = 2, mean = 0, sd = 1)
> y <- exp(z)
> y[2] <- y[2] * y[1]
> x <- (y[1] - 1) * (y[1] > 1) * (y[2] > 1)
> E(x)

[1] 0.74

> Pr(x > 1)

[1] 0.20
```

Graphical summaries Graphical summaries are still in development, but it is now possible to plot a scatterplot with a regular vector against a random vector, showing the 50% and 95% *uncertainty intervals* along with the median, using `plot(y,x,...)`, where `y` is not random but `x` is. or we can show two random scalars plotted as a 2-dimensional scatterplot with `plot(x[1],x[2],...)`.

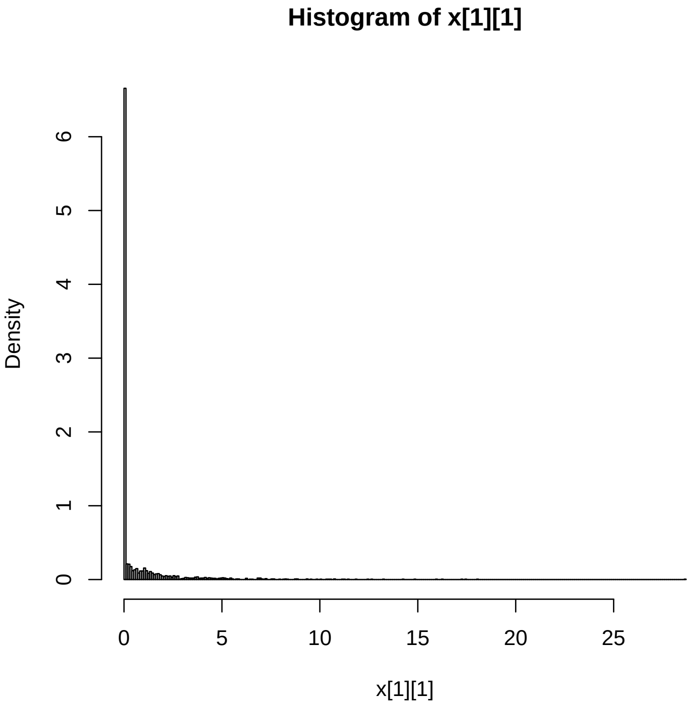
Or, we can show a random vectors as horizontal intervals using `mlplot`:

```
> mlplot(x)
```



The histogram of the simulations of a random scalar `x[1]` can be plotted with

```
> rvhist(x[1])
```



Posterior simulations from a classical regression model. We can generate posterior simulations from a classical regression model, using the standard assumptions for the priors. For convenience there is a function `postsim` to do this.

```
> x <- 1:30
> y <- rnorm(30, mean = x/10)
> fit <- lm(y ~ x)
> s <- postsim(fit)
```

Now `s["sigma"]` contains the variable σ with simulations from the posterior distribution $p(\sigma|y)$:

```
> s["sigma"]
```

```
     name mean   sd   1% 2.5% 25% 50% 75% 97.5% 99% sims
[1] sigma  1.1 0.17 0.85 0.88 1.0 1.1 1.2   1.5 1.6 2500
```

and the other components are the coefficient estimates β with the joint distribution $p(\beta|y)$.

Creating replicated simulations. Continuing the previous example, we'll resample from the sampling distribution of y using the posterior simulations we got. We can use the function `rvnorm` to do this, since it accepts *random vectors as arguments.* Rather than think `rvnorm` to draw normal random vectors, it rather "samples from the normal model." The vector will be normal *given* (constant) mean and s.d., but if the mean and s.d. are not constants, the resulting vector will not be normal.

```
> sigma <- s[1]
> betas <- s[-1]
> X <- model.matrix(fit)
> y.rep <- rvnorm(mean = X %*% betas, sd = sigma)
> mlplot(y.rep)
```

The function call

```
> X %*% betas
```

```
     name    mean   sd     1%   2.5%    25%     50%  75% 97.5%  99% sims
 [1]    1 -0.0913 0.41 -1.063 -0.918 -0.350 -0.0876 0.17  0.72 0.88 2500
 [2]    2  0.0077 0.39 -0.932 -0.779 -0.239  0.0054 0.26  0.78 0.92 2500
 [3]    3  0.1066 0.37 -0.795 -0.636 -0.127  0.1062 0.35  0.84 0.99 2500
 [4]    4  0.2056 0.35 -0.661 -0.496 -0.015  0.2021 0.44  0.90 1.04 2500
 [5]    5  0.3045 0.33 -0.508 -0.352  0.095  0.2997 0.53  0.96 1.10 2500
 [6]    6  0.4034 0.31 -0.356 -0.215  0.206  0.4003 0.61  1.01 1.15 2500
 [7]    7  0.5024 0.29 -0.232 -0.092  0.315  0.5013 0.70  1.08 1.20 2500
 [8]    8  0.6013 0.28 -0.093  0.038  0.422  0.6005 0.79  1.15 1.25 2500
 [9]    9  0.7003 0.26  0.049  0.174  0.527  0.7003 0.88  1.21 1.32 2500
[10]   10  0.7992 0.25  0.188  0.296  0.635  0.7993 0.97  1.28 1.38 2500
[11]   11  0.8982 0.24  0.326  0.423  0.742  0.8996 1.05  1.36 1.44 2500
[12]   12  0.9971 0.23  0.453  0.547  0.852  0.9983 1.14  1.45 1.53 2500
[13]   13  1.0961 0.22  0.538  0.654  0.957  1.0949 1.24  1.52 1.61 2500
[14]   14  1.1950 0.22  0.641  0.771  1.057  1.1945 1.33  1.62 1.71 2500
[15]   15  1.2940 0.21  0.761  0.864  1.158  1.2944 1.43  1.72 1.80 2500
[16]   16  1.3929 0.21  0.859  0.961  1.256  1.3930 1.52  1.82 1.91 2500
[17]   17  1.4919 0.22  0.965  1.058  1.356  1.4896 1.62  1.92 2.02 2500
[18]   18  1.5908 0.22  1.065  1.153  1.450  1.5888 1.73  2.04 2.14 2500
[19]   19  1.6898 0.23  1.163  1.240  1.541  1.6882 1.83  2.15 2.26 2500
[20]   20  1.7887 0.24  1.237  1.336  1.632  1.7874 1.94  2.27 2.38 2500
[21]   21  1.8877 0.25  1.303  1.407  1.722  1.8852 2.04  2.39 2.50 2500
[22]   22  1.9866 0.26  1.366  1.488  1.812  1.9872 2.15  2.51 2.63 2500
```

```
[23]    23  2.0856 0.27  1.434  1.554  1.901  2.0870 2.26  2.64 2.75 2500
[24]    24  2.1845 0.29  1.496  1.617  1.990  2.1865 2.37  2.77 2.88 2500
[25]    25  2.2835 0.31  1.563  1.689  2.079  2.2861 2.48  2.90 3.01 2500
[26]    26  2.3824 0.32  1.625  1.753  2.164  2.3856 2.60  3.02 3.15 2500
[27]    27  2.4813 0.34  1.685  1.819  2.251  2.4856 2.70  3.16 3.28 2500
[28]    28  2.5803 0.36  1.729  1.886  2.339  2.5854 2.82  3.30 3.43 2500
[29]    29  2.6792 0.38  1.777  1.944  2.427  2.6825 2.93  3.44 3.57 2500
[30]    30  2.7782 0.40  1.835  1.997  2.511  2.7797 3.04  3.58 3.72 2500
```

returns a random vector of length 30, and `sigma` is also random. Thus all the uncertainty in the mean estimate $X\beta$ and the residual s.d. estimate σ is propagated when the replicated vector y^{rep} is generated. In effect, this single line of code thus will in fact draw from the distribution $p(y^{\mathrm{rep}}|y) = \int\int \mathrm{Normal}(y^{\mathrm{rep}}|\mu,\sigma)p(\mu,\sigma|y)\mathrm{d}\mu\mathrm{d}\sigma$.

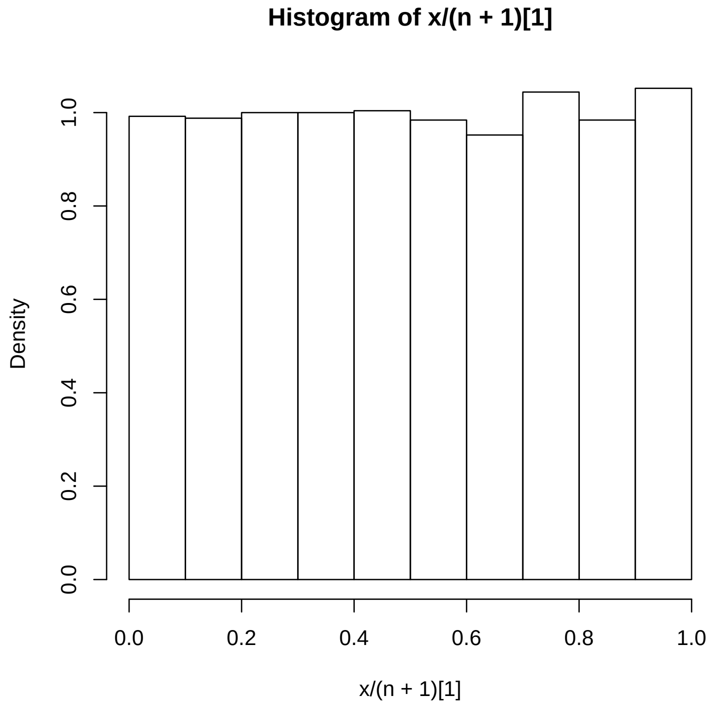
Example: Simulating Pólya's Urn. This code simulates 200 iterations of the well-known Pólya's urn problem. The parameter `x/(n+1)` for the Bernoulli-variate-generating function `rvbern(...)` is random: we can generate random variables using random parameters without much trickery; our code looks therefore more natural.

The model:

$$\begin{aligned} X_0 &= 1 && (1) \\ X_n - X_{n-1}|X_{n-1} &\sim \mathrm{Bernoulli}(X_{n-1}/(n+1)) && (2) \end{aligned}$$

The R code:

```
> x <- 1
> for (n in 1:100) {
+     x <- x + rvbern(n = 1, prob = x/(n + 1))
+ }

> rvhist(x/(n + 1))
```



We can see that the distribution is close to uniform, which is the limiting distribution in this case.

3 Details

Obtaining the simulation matrix. To extract the simulation matrix embedded in an rv object, use `sims`:

```
> s <- sims(y.rep)
> dim(s)
```

```
[1] 2500   30
```

It is our convention to have the columns represent the random vector and the rows represent the draws from the joint distribution of the vector.

Converting matrices and vectors of simulations to rv objects. A matrix or a vector of simulations is converted into an rv object by `rvsims`. Continuing the above example, we'll convert the matrix back to an rv object.

```
> y <- rvsims(s)
```

You can verify that `all(sims(y)==s)` returns TRUE. Also note that `dim(y)` gives , since `y` is "just a vector."

Coercing vectors and matrices. The function `as.rv(x)` coerces objects to rv objects. However, this does not mean that matrices of simulations are turned into rv objects—this is done with `rvsims`, as explained above. `as.rv(rnorm(1000))` would return a random vector of length 1000, where each component has zero variance (and one single simulation). You probably mean `rvsims(rnorm(1000))`, but the correct way to generate this object is `rvnorm(1)`.

Obtaining simulations from R2WinBUGS R2WinBUGS [Sturtz et al., 2005] is an interface for calling WinBUGS within R, and obtaining the simulations as an R matrix (that is embedded in a "`bugs`" object). If `bugsobj` is the bugs object returned by the `bugs(...)` function call, then `as.rv` will coerce it into a random vector: `y <- as.rv(bugsobj)` Now `y` is a vector with the `names` attribute set. Usually we want to access the components such as `theta[1],...,theta[8]` and `sigma` etc. by their names, directly. You can split a random vector into a list of named subvectors by `splitbyname`:

```
> x <- rvnorm(6)
> names(x) <- c(paste("theta[", 1:4, "]", sep = ""), paste("mu[",
+     1:2, "]", sep = ""))
> x
```

```
         name    mean   sd   1% 2.5%   25%    50%  75% 97.5% 99% sims
[1] theta[1] -0.0178 1.00 -2.3 -2.0 -0.70 -0.018 0.67   1.9 2.3 2500
[2] theta[2]  0.0048 0.98 -2.3 -1.9 -0.65  0.012 0.66   1.9 2.3 2500
[3] theta[3]  0.0102 0.98 -2.2 -1.8 -0.68 -0.024 0.66   2.0 2.4 2500
[4] theta[4]  0.0293 1.00 -2.4 -1.9 -0.62  0.015 0.70   2.0 2.4 2500
[5]    mu[1]  0.0138 0.98 -2.3 -2.0 -0.65  0.025 0.68   1.9 2.2 2500
[6]    mu[2] -0.0100 1.01 -2.5 -2.0 -0.66 -0.019 0.66   2.0 2.5 2500
```

```
> rv::splitbyname(x)

$mu
     name   mean   sd   1% 2.5%   25%    50%  75% 97.5% 99% sims
[1] mu[1]  0.014 0.98 -2.3 -2.0 -0.65  0.025 0.68   1.9 2.2 2500
[2] mu[2] -0.010 1.01 -2.5 -2.0 -0.66 -0.019 0.66   2.0 2.5 2500

$theta
        name    mean   sd   1% 2.5%   25%    50%  75% 97.5% 99% sims
[1] theta[1] -0.0178 1.00 -2.3 -2.0 -0.70 -0.018 0.67   1.9 2.3 2500
[2] theta[2]  0.0048 0.98 -2.3 -1.9 -0.65  0.012 0.66   1.9 2.3 2500
[3] theta[3]  0.0102 0.98 -2.2 -1.8 -0.68 -0.024 0.66   2.0 2.4 2500
[4] theta[4]  0.0293 1.00 -2.4 -1.9 -0.62  0.015 0.70   2.0 2.4 2500
```

Obtaining simulations from Umacs. Umacs facilitates the construction of a Gibbs/Metropolis sampler in R [Kerman, 2006], and returns the simulations wrapped in an "`UmacsRun`" object. Again, the coercion method `as.rv` will convert the Umacs object, say `obj`, into a list of namedsubvectors: `y <- as.rv(obj)`.

4 Some implementation details

rv is written in "S3" style object-oriented R rather than using the `methods` ("S4") package. The main reason was speed, the secondary consideration was the ease of writing new functions.

The main class is called "rv". Most functions expecting an rv object have names starting with "rv". for example `rvnorm`, `rvmean`, etc.

The package also features rv-specific methods extending the basic numeric vector classes, e.g. `c.rv`, `plot.rv`, etc. However, the method-invoking routine is not perfect in R: for example the concatenation function `c(...)` will not call `c.rv` for example in the following case: suppose that `x` is an object of class `rv`. Then `c(10, x)` will not call `c.rv` since the method-dispatch mechanism only looks at the first element. As a temporary measure, the current version of *rv* overwrites some of the functions in the base classes to create the illusion of proper method-dispatch mechanism. (Ideally, the simulation dimension should be part of R itself.) To temporarily disable the overwritten basic functions, execute `rvcompatibility(1)`.

5 Disclaimer

This program is a work in progress, and it may contain bugs. Many new features will be eventually (and hopefully) added.

For information about random variables in R, please refer to Kerman and Gelman [2007].

The web site `http://www.stat.columbia.edu/~kerman/` contains links to the articles and to the software.

References

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

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

Jouni Kerman and Andrew Gelman. Manipulating and summarizing posterior simulations using random variable objects. *Statistics and Computing* 17:3, 235–244.

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