# **random**: An R package for true random numbers

Dirk Eddelbuettel

August 2006

## 1 Introduction

Simulation techniques are a core component of scientific computing and, more specifically, computational statistics. All simulation methods—Monte Carlo methods, bootstrapping, estimation by simulation to name but a few—rely on 'computer-generated randomness' (more on this below). In practice, this means sequences of random numbers. Generating 'good' (for a suitable metric) random numbers is therefore of critical importance for scientific software environments. As a concrete example, consider that the current version of the R environment (R Development Core Team, 2006) contains no fewer than six random number generation algorithms all of which are available to the user at run-time, and refers to a total of twelve publications as further references.

Deterministically generated random numbers—i.e. numbers obtained by means of executing a computer program that implements a particular algorithm—are commonly referred to as *pseudo random numbers*. The adjective pseudo stresses the implicit contradiction between the deterministic manner in which 'randomness' is being produced. The algorithms and computer programs that produce these random number are generally referred to as pseudo-random number generators, or PRNGs for short. An distinction is sometimes made between pseudo- and quasi-random number generators (QRNGs). The latter are preferable for certain simluation studies due to better convergence properties, and which leads these methods also being called low-discrepancy sequences. For our purposes, the distinction between QRNGs and PRNGs is not as important, and we will use the term PNG to refer to both types.

The Comprehensive R Archive Network (CRAN) contains numerous packages that provide additional random number generators for R. The **gsl** package (Hankin, 2005) provides the battery of generators for pseudo- and quasi-random number generators from the GNU Scientific Library, or GSL (Galassi et al., 2006). The **randaes** package (Lumley, 2005) provides a cryptographically secure RNG based on Ferguson and Schneier's Fortuna generator. Package **SuppDists** (Wheeler, 2006) provides another RNG. Quasi-random number generators are provided by the Rmetrics component package **fOptions** (Wuertz, 2006). Lastly, RNGs suitable for distributed computing are available via **rsprng** (Li, 2006) and **rstream** (Leybold, 2006).

All of these packages have one major commonality: they produce pseudo random numbers determined by 'software': an algorithm implemented as executable computer code. As such, given the initial conditions of the algorithm and implementation, typically one or more so-called numbers acting as 'seeds', parts or the entire sequence can be replicated. In other words, these software-generated random numbers are entirely determinisitic. In many cases, this is in fact a desirable property as it provides replicability, a key feature for reproducing scientific results.

On the other hand, so-called *true* random numbers require actual unpredictability arising from genuinely physical processes such as radioactive decay, photon emissions or atmospheric noise. An overview of hardware random number generators is provided by Wikipedia (2006). Davies (2000) discusses testing hardware generators.

There are a number of situations in which it is desirable to use *non-deterministically* determined random numbers. Examples include

- to seed distributed computing on different nodes with truly indepedent seeds;
- to obtain portable initializations for RNGs that do not depend on particular operating system or hardware features;
- to validate simulation results using non-deterministic random numbers;
- to provide indeterministic seeds used for lottery drawings or games: imagine playing R's **sudoku** (Brahm and Snow, 2006) seeded by true randomness!

Until now, obtaining non-deterministic random numbers was not (easily) possible in R.[1] The **random** package aims to fill this gap by offering to supply random integers with possible replacement, random integer sequences without replacement, and random bytes from the random.org (Haahr, 1999, 2006) web service.

The rest of this paper is organised as followes. The next section introduces random.org. This is followed by a section on testing the properties of the generator. Next, we disucss the implementation of the **random** package, and provide a simple example. Potential problems are briefly discussed in the following section, before a summary concludes.

## 2 About random.org

As discussed above, generating random number via computer programs is critically important for simulation methodologies. Yet at the same time, it is a contradiction in terms as there is nothing 'random' about generating sequences in a deterministic fashion. Given an initial condition, the entire sequence can (at least in therory) be predicted.

An alternative to these deterministic software generators are truly non-deterministic hardware based generators. One of the longer-running freely available services is the random.org service created by Mads Haahr (Haahr, 2006), and described briefly, along with some motivation and background, in Haahr (1999).[2]

According to Haahr (1999), the original motivation for random.org was to generate random numbers for the prototype of an on-line gambling site. The basic idea is to gather atmospheric noise via a radio-receiver card tuned to an unused frequency and connected to a computer where it is sampled and digitized. A skew-correction algorithm, originally developped by John von Neumann in the 1940s and discussed by Davies (2000), is then applied to the bit stream.

The random.org site offers random numbers in several different formats: as integer random numbers with possible replication (similar to rolling die), as re-arranged integer sequences (without replication and similar to random reshuffling of a given set) and as random bits in several common formats (hexadecimal, decimal, octal and binary). Lastly, a covenience function is offered to query

[1] Under Linux, access to `/dev/random` provides hardware entropy, but only in a non-portable manner.

[2] This esay is also included in the **random** package as the vignette *random-essay*.

the state of the generator. It is recommended to delay requests for larger sets of random numbers if the generator is indicating less than 20% capacity.

These random numbers can be retrieved using three different interfaces. The oldest, and original, interface is a standard CGI script accessible via the HTTP protocal used by webservers and browsers. The second is using the (somewhat more complex) Corba networking protocal and infrastructure, and the third is using SOAP, an exchange of files in XML markup that also uses HTTP as a transport.

Use of the random.org service, as reported on the site, is varied. It covers actual lotteries (both for outright prices as well for assignments of, say, students to classrooms), games, arts as well as conventional programming with random numbers. It is not immediately apparent which of these uses are truly improved due to their use of a non-deterministic random number generator.

## 3 Testing random.org data

Given the critical importance of random numbers for scientific computing via simulation methods, fairly rigorous approaches for measuring and ensuring the quality of random numbers have been developed (Davies, 2000). One of the better-known test suites is the so-called **diehard** collection of tests by Marsaglia (1995), which has been re-implemented and extended by Brown (2006a) under the name **dieharder**.

The basic idea of testing a random number generator is as follows (and this follows the discussion by Brown (2006a) in the extended documentation by of **dieharder**). For a given suite of $n$ uniform random numbers

$$u_n = u_1, u_2, \ldots \quad \text{where} \quad u_n \sim U(l, h)$$

that are distributed uniformly between the bounds $l$ and $h$, we have analytical results for the expected mean and variance given by

$$E(u) = \frac{h - l}{2}$$

and

$$V(u) = \frac{(h - l)^2}{12}.$$

For a sample of uniformly distributed random numbers from a random number generator we wish to test, as well as knowledge of the theoretical sample mean and variance, we can undertake a classic hypothesis test to compute a marginal probability (or $p$-value) for obtaining the given sample by chance from a (hypothetical) perfect generator. Any one such $p$-value from a single sample is likely to be non-informative. However, we can repeat the test for series of sets of random generators. This provides us with a series of associated $p$-values that satisfy the common assumptions of independent and identically distributed statistics.

Now, under the null of a perfect random generator, this series of $p$-values ought to be uniformly distributed over the range of possible probabilities between zero and one. This property is itself testable, using for example using a Kolmogorov-Smirnov or Anderson-Darling test. Test suites such as diehard and dieharder provide a variety of different tests each of which applies this basic principle of going from one $p$-value from a testable expression to testing a suite of such $p$-values from one particular test against the uniform distribution.

Dieharder provides 24 tests in its current version 1.4.24. Besides the original diehard test, the set comprises tests from NIST as well as new tests, see Brown (2006a). The aggregate number of random numbers required to run all these tests is considerable. In fact, it is much, much larger than what random.org can provide as a live service (leading to the author's IP address being blacklisted and blocked by random.org for a few days). In order to test the generator, we had to rely on a set of fifteen pre-generated binary files of 10 megabytes size each, which we combined into one large file of 150 megabytes. Given the standard (four byte) size of an unsigned integer, and the adjustment factor of $2^{10}/10^3$ stemming from the colloquial 'kilobyte' being 1024 bytes, the dieharder tests are provided with $150/4 \times 1024^2 \times 10^{-6} = 39,321,600$ unsigned integers. The output log provided by the program reports that the binary input file was rewound a total of 297 times. It follows that a lower bound for the total number of random numbers consumed by dieharder in testing the random.org generator is $39.3 \times 10^6 \times 297$ or approximately 11.6 billion random numbers.

Table 1 summarizes the results of running version 1.4.24 of dieharder over the binary data retrieved from random.org. With the caveat that we do not actually know whether these were in fact generated by the random.org generator, we note that the generator passes a large majority of tests. Diehard, and by extension dieharder, are known as rigorous test suites, and passing a large majority of their component tests is a satisfying outcome.[3]

Brown (2006b) provides additional information on some of the failed tests. The 'RGB Bit Distributioen Test' in known to fail at $n = 6$ for every generator. The 'Diehard Overlapping 5-Permutations' may be prone to type I errors so the failure may be disregarded. On the other hand, the 'Diehard 32x32 Binary Rank Test' result is a genuine test failure: better generators pass this test. The 'Diehard Bitstream Test' is also somewhat doubtful as known 'good' generators can fail this test yet pass the related OPSO test. 'Marsaglia and Tsang GCD Test' is a well-respected that the best generators pass, so failure here in also informative. These comments notwithstanding, the test outcome does not lead us to recommend against use of random.org.

A comparison from running **dieharder** over random number streams from the six generators provided by R in its base package is left for future research.

## 4 Implementation of the random package

The combination of excellent networking support in R and the simple HTTP interface at random.org allows for a fairly light implementation. The standard data-reading function `read.table` in R is able to operate directly on a remote file, or URL. It is hence straightforward to connect R to CGI scripts providing data. In fact, the **tseries** package (Trapletti and Hornik, 2006) has had the ability to download data for financial instruements directly off Yahoo! Finance for at a least a half-decade.

Consequently, the main requirement for functions to access random.org are validity checks for the function arguments, construction of the URL query string, fetching the data and possibly reformatting the data before retuning it to the user. This high-level description summarizes the functions provided by the **random** package.

Let us consider a simple example where we illustrate simple data acquisition from random.org. To avoid needless stress the server on re-runs of this document, we cache data, once downloaded, in a file and skip the download if the cache file is still present.

[3] We should note that at least one of the tests is still considered experimental, and that, strictly speaking, the test suite itself has not been validated.

| Test | Outcome |
|---|---|
| RGB Timing Test | 4.45 million rands/second |
| RGB Bit Persistence Test | Passed at 5% |
| RGB Bit Distributioen Test | Passed at 5% for n=1,2,3,4,5 |
| | Failed at $< 0.01\%$ for n=6 |
| Diehard Birthdays test (mod.) | Passed at 5% |
| Diehard Overlapping 5-Permutations | Failed at $< 0.01\%$ |
| Diehard 32x32 Binary Rank Test | Failed at $< 0.01\%$ |
| Diehard 6x8 Binary Rank Test | Passed at 5% |
| Diehard Bitstream Test | Failed at $< 0.01\%$ |
| Diehard Overlapping Pairs (OPSO) | Passed at 5% |
| Diehard Overlapping Quadruples (OQSO) | Passed at 5% |
| Diehard DNA Test | Passed at 5% |
| Diehard Count the 1s (stream) (mod.) | Passed at 5% |
| Diehard Count the 1s (byte) (mod.) | Passed at 5% |
| Diehard Parking Lot Test (mod.) | Passed at 5% |
| Diehard Minimum Distance Test | Passed at 5% |
| Diehard 3d Sphere Test | Passed at 5% |
| Diehard Squeeze Test | Possibly weak |
| Diehard Sums Test (up) | Passed at 5% |
| Diehard Sums Test (down) | Passed at 5% |
| Diehard Craps Test (mean) | Passed at 5% |
| Diehard Craps Test (freq) | Passed at 5% |
| Marsaglia and Tsang GCD Test | Failed at $< 0.01\%$ |
| Marsaglia and Tsang Gorilla Test (preli | Passed at 5% |
| STS Monobit Test | Passed at 5% |
| STS Runs Test | Passed at 5% |
| Dieharder User Example Lagged Sums | Passed at 5% |

Table 1: Results of **dieharder** for `random.org`

```
> ## cached data to not depend on a) a network connection
> ## and b) data at random.org
> if ( !(file.exists("random.Rdata")) ) {
+   randomOrg <- randomNumbers(n=5000, min=1, max=1e6, col=2)/1e6
+   save(randomOrg, file="random.Rdata")
+ } else {
+   load("random.Rdata")
+ }
> #
```

This shows that the call to obtain data from random.org is straightforward: we specify how many draws (with replication) over which interval we desire, and how many columns we would like in the return matrix. Exploratory data analysis using moments and quantiles, not shown here but

available on request, reveal nothing suspicous that would indicate that the data is not distributed according to the requested $U(0,1)$ distribution.

```
> summary(randomOrg)
> apply(randomOrg, 2, function(X) quantile(X,c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.9
> #
```

Further analysis using a simple scatter plot of half the uniform random numbers against the other, as well as a histogram of the entire data set, confirm the intial analysis.

```
> plot(randomOrg, ylab="", xlab="", main="5000 random,org U(0,1) draws", pch='.')
> hist(matrix(randomOrg, ncol=1), xlab = "", ylab = "", main="Histogram")
```

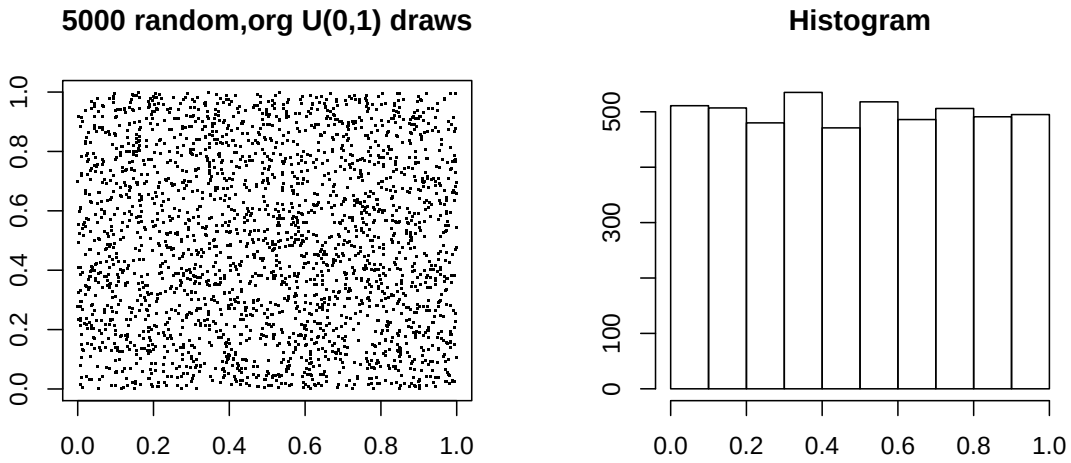


Figure 1: Scatter plot and histogram

## 5 Potential problems

A few potential problems with the non-deterministic random numbers provided by random.org should be mentioned.

First, there is no actual service guarantee for the random numbers. Deterministic random numbers can always be produced requiring 'just code': given a suitable compiler and computer, one should always be able to produce deterministic random number. And, given the so-far continued progression of hardware according to Moore's Law, more and more deterministic random numbers can be produced per unit of time. On the other hand, random.org is a web-based service. While the website has been on the Internet since 1999, and seemingly without any incident or break in the service, nothing guarantees that the site will be available at any given future point in time. This could block serious production-level implementation using random.org.

Second, the data transfer from the service is entirely *in the open*. There is no provision for either a simple login or other forms of authentication, or an encryption layer covering the data. This could blocks any serious applications, in particular those involving cryptography. However, it it probably safe to assume that cryptography was never a focus of random.org.

Third, the random.org service, like other hardware generators, is comparatively slow—an order of magnitude slower than software-based generators even on modest hardware. In fact, Davies (2000) already suggested to use files to store the output from hardware random number generators, and to then access these files for access at higher speeds.

Fourth, while the result of the **dieharder** test suite are encouraging, they have been performed using static binary files. And, simply put, we just do not know for sure that the data was in fact generated by the random.org generator. A replicated study with a better guarantee concerning the origin of a sufficient amount of test data would be welcomed.

## 6 Summary

The **random** package complements the existing *deterministic* random number generators in R by providing a first *non-deterministic* source based on the random.org service (Haahr, 2006). The **random** package is portable and cross-platform. This paper also provides the first rigorous tests of the random.org service using the **dieharder** battery of tests (Brown, 2006a) which indicate that the generated random numbers are of sufficiently high quality. We hope that **random** package presented here will prove to be useful for users of the R enviroment.

## References


David Brahm and Greg Snow. **sudoku***: Sudoku puzzle generator and solver*, 2006. URL `http://cran.r-project.org/src/contrib/Descriptions/sudoku.html`. R package **sudoku**, version 2.0.

Robert G. Brown. *`dieharder`: A Random Number Test Suite*, 2006a. URL `http://www.phy.duke.edu/~rgb/General/dieharder.php`. C program archive **dieharder**, version 1.4.24.

Robert G. Brown. Personal communication. Via email, August 2006b.

Robert B. Davies. *Hardware random number generators*, 2000. Presented at the 5th Australian Statistics Conference, July 2000, and the 51st conference of the NZ Statistical Association in September, 2000. Published as http://www.robertnz.net/hwrng.htm (Accessed 10-August-2006).

Mark Galassi, Brian Gough, Gerald Jungman, James Theiler, Jim Davies, Michael Booth, and Fabrice Rossi. *The GNU Scientific Library Reference Manual*, 2006. URL `http://www.gnu.org/software/gsl`. ISBN 0954161734.

Mads Haahr. *`random.org`: Introduction to Randomness and Random Numbers*, 1999. Published as `http://random.org/essay.html` and also available as vignette `random-essay` in R package **random**.

Mads Haahr. random.org: True random number service, 2006. URL `http://random.org`. (accessed 6-August-2006).

Robin Hankin. **gsl***: A wrapper for the Gnu Scientific Library*, 2005. URL http://cran.r-project.org/src/contrib/Descriptions/gsl.html. R package **gsl**, version 1.6.6.

Josef Leybold. **rstream***: Streams of random nummbers*, 2006. URL http://statistik.wu-wien.ac.at/arvag/. R package **rstream**, version 1.2.

Na Li. **rsprng***: R interface to SPRNG (Scalable Portable Random Number Generatos)*, 2006. URL http://www.biostat.umn.edu/~nali/SoftwareListing.html. R package **rsprng**, version 0.3-3.

Thomas Lumley. **randaes***: Random number generator based on AES cipher*, 2005. URL http://cran.r-project.org/src/contrib/Descriptions/randaes.html. R package **randaes**, version 0.1.

George Marsaglia. *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*, 1995. URL http://stat.fsu.edu/pub/diehard. (accessed 6-August-2006).

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL http://www.R-project.org/. ISBN 3-900051-07-0.

Adrian Trapletti and Kurt Hornik. **tseries***: Time Series Analysis and Computational Finance*, 2006. URL http://cran.r-project.org/src/contrib/Descriptions/tseries.html. R package **tseries**, version 0.10-3.

Bob Wheeler. **SuppDists***: Supplementary distributions*, 2006. URL http://www.bobwheeler.com/stat. R package **SuppDists**, version 1.1-0.

Wikipedia. Hardware random number generator. In *Wikipedia, The Free Encyclopedia*, 2006. URL http://en.wikipedia.org/w/index.php?title=Hardware_random_number_generator&oldid=68022952. (accessed 6-August-2006).

Diethelm Wuertz. **Rmetrics***: An Environment and Software Collection for Teaching Financial Engineering and Computational Finance*, 2006. URL http://www.Rmetrics.org/. R package **fOptions**, version 221.10065.