DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

JDiehard: An implementation of Diehard in Java

Balasubramanian Narasimhan
Department of Statistics
Stanford University
Stanford, CA 94305-4065

Abstract

We describe JDiehard, an implementation of the stringent battery of tests for random number generators created by George Marsaglia. The original implementation of Diehard (in Fortran and/or C) is command-line driven and not very user-friendly. JDiehard uses features of a modern language like Java to present a Graphical User Interface (GUI) to Diehard. Facilities are provided for easy addition of new defined generators and tests provided they implement some simple interfaces.

The current implementation of Diehard is meant for use on a single system. However, the extensive use Java interfaces means that the design is flexible enough to distribute computation using Java RMI or CORBA. Newer versions of JDiehard will make such features easily available.

1 Introduction

George Marsaglia's Diehard battery of tests are widely used to certify random number generators as being worthy of use in serious research. They became well-known after Marsaglia's keynote address [3]. Until then, several generators that had been used passed other simple tests, but failed the Diehard tests. The original implementation of Diehard was in Fortran and/or C. Since 1990, they have also been distributed on the internet and on a CDROM.

JDiehard is an implementation of Diehard in a modern language like Java. The goals of this implementation are the following:

User friendliness We use Java and Swing to provide a point and click interface to the generators and tests.

Ease of customization By using property files and demand loading of classes, we provide easy ways for the user to add his or her own generators and tests.

Online Generator Testing There are several companies today that provide physical devices that produce random bits. One can easily envision a world where such bits are made available online via a URL for driving online games etc. Since a language like Java is network-aware, we get this ability for free!

Multiple precision and Cryptographic Generators The Java arbitrary precision library allows for easy implementations of the new generators used for example in Advanced Encryption Standard (AES) of NIST.

Distributable Distributed Computing is the technology of the future. Since JDiehard makes extensive use of interfaces, there is potential to distribute tests and generators across the network. A generator and a test are merely interfaces and the actual location of the object implementing the interface is irrelevant as long as it is available.

2 A Quick Tour

The requirements for running JDiehard are the following:

- Java 2 platform.
- Element Construction Kit (ECS) from `java.apache.org`.

Figure 1 shows the Diehard GUI. The tab named **RNG** provides the user access to the available random number generators. The generators are organized in factories so as to lessen the possibility of name clashes. Users can seed and sample integers and doubles. One can also time generation of integers and doubles. Generators that are to be tested can be chosen via the **Select** button.

The tab named `Tests` lists the available generators organized in factories. Tests parameters can be changed from the default settings. Tests to be run on generators can be selected via the **Select** button.

Once the user has chosen tests and generators, the tab named **Launch Pad** allows users to run all possible combinations or specific ones. All results are formatted in HTML and shown in the output window.

3 What is a Random Number Generator?

Any class that implements the following methods is a Random Number Generator to JDiehard.

```
public String getName();
public void doSetUp();
public boolean returnsZero();
public boolean returnsOne();
public double nextDouble();
public int nextInt();
```

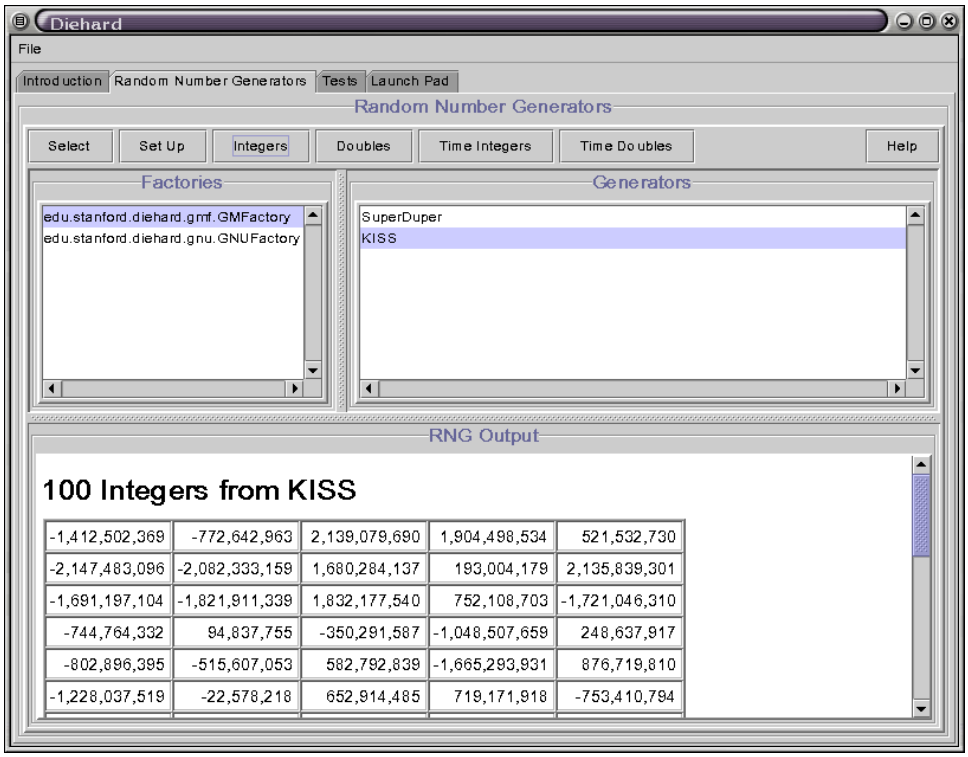

Figure 1: JDiehard GUI

The `getName` method should merely return an identifier string. The `doSetup` method is meant for initialization. The next two `returnsZero` and `returnsOne` are needed by some tests to detect when they apply mathematical functions like `log` etc. The methods `nextInt` and `nextDouble` return the next integer and the next double in the sequence. It should be noted that java uses *signed* integers. The methods `saveState` and `restoreState` work in tandem to store and restore the state of the generator.

In extending the definition of a generator to online ones, it is clear that a distinction must be made between *seekable* generators, those that can restart from any point in their sequence and *non-seekable*, those that cannot. Seekable Generators implement two additional methods below.

```
public RandomNumberGeneratorState getState();
public void setState(RandomNumberGeneratorState state);
```

Most conventional random number generators are, of course, seekable.

4 What is a Test?

Any class that extends implements the following methods is a test to JDiehard.

```
public String getName();
public void doSetUp();
public String getOutput();
public void setRNG(RandomNumberGenerator rng);
public void run();
```

The methods of importance here are `getOutput` and `run`. The latter is where the test is actually run and is a result of the test extending the `Runnable` class. Note that since the test is actually run in a thread, the `getOutput` method must be thread-safe in accessing the output of the test. The output is expected in HTML format, although easy extensions to XML are possible. Indeed, JDiehard uses the *Element Construction Kit* (ECS) from the Apache group for generating dynamic output. This kit handles both HTML and XML.

5 Available Generators

We currently have many of the generators proposed by George Marsaglia implemented. We also hope to implement the following families of generators.

- Congruential: $x_n = ax_{n-1} + c \bmod m$.
- Lagged Fibonacci: $x_n = x_{n-r} \oplus x_{n-k}$
- Subtract-with-borrow and Add-with-carry: $x_n = x_{n-r} \pm x_{n-k} \pm c \bmod m$.
- Multiply-with-carry: $x_n = x_{n-r} * x_{n-s} + c \bmod m$.

6 Available Tests

We have implemented all the tests in the original version of Diehard. They are briefly described below. A couple of newly developed tests by Marsaglia are also available.

6.1 Birthday Spacings

The Birthday Spacings Test is described in [3].

Choose m "birthdays" in a "year" of n days. List the spacings between the birthdays. Let J be the number of values that occur more than once in that list. Then, J is asymptotically Poisson distributed with mean $m^3/(4n)$.

Experience shows n must be quite large, say $n \geq 2^{18}$, for comparing the results to the Poisson distribution with that mean. The default set up uses $n = 2^{24}$ and $m = 2^{10}$ so that the underlying distribution for J is taken to be Poisson with $\lambda = 2^{30}/2^{26} = 16$. A sample of 200 J's is taken, and a chi-square goodness-of-fit test provides a p-value.

In our implementation of the test, we do the following. Assume the default settings of the parameters above. The first test uses bits 1–24 (counting from the left) from the currently chosen random number generator. Then the bits 2-25 of the *same sequence* are used to provide birthdays, and so on to bits 9–32. After we exhaust the 32 bits, we begin with bits 1–24 again using a *new sequence*. Each set of bits provides a p-value and at least 20 such p-values are used in a Kolmogorov-Smirnov test.

6.2 Ranks of Binary Matrices

From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6×8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0, 1, 2, 3 are rare; their counts are pooled with those for rank 4. Ranks are found for $100,000$ random matrices, and a chi-square test is performed on counts for ranks 6,5 and $0, \ldots, 4$ pooled together.

6.3 Bitstream Test

Consider the output from an generator as a stream of bits, $b_1, b_2, b_3, \ldots$. Consider an alphabet of 2 "letters" 0 and 1 and think of the stream as a succession of 20-letter overlapping words. Thus the first word is $b_1 b_2 \ldots b_{20}$ and the second $b_2 b_3 \ldots b_{21}$ and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^{21} overlapping 20-letter words. There are 2^{20} possible 20-letter words. For a truly random string of $2^{21} + 19$ bits, the number of missing words J should be (very close to) normally , distributed with mean $\mu = 141909$ and standard-deviation $\sigma = 428$. Thus $(J - 141909)/428$ should be a standard normal variate, that leads to a uniform $[0, 1]$ p-Value. We repeat the test a number of times and perform a Kolmogorov-Smirnov test on the p-Values.

Actually, in this test, we allow the number of words sampled, n to be one of 2^{20} or 2^{21} or 2^{22}.

6.4 Craps

The test plays $n \geq 200000$ games of craps and counts the number of wins and the number of throws necessary to end each game. The number of wins should be very close to normal with mean np and variance $np(1-p)$ where $p = 244/495$. Throws necessary to complete the game can vary from 1 to ∞, but all throws $>= 21$ are lumped together. A χ-squared test is made on the number-of-throws cell counts.

6.5 Minimum Distance

Choose $n = 8000$ random points in a square of side 10000. Find d, the minimum distance between the $(n^2 - n)/2$ pairs of points. If the points are truly independent uniforms, then d^2, the square of the minimum distance should be very close to being exponentially distributed with mean $.995$. Thus $1 - \exp(-d^2/.995)$ should be uniform on $[0, 1)$ and a Kolmogorov-Smirnov test on the resulting uniform values serves as a test of uniformity for random points in the square.

6.6 Monkey

The monkey tests are described in [2] and originally discussed in [3]. It includes tests designated as OPSO (Overlapping-Pairs Sparse Occupancy), OQSO (Overlapping-Quadruples Sparse Occupancy) and the DNA tests. The Bit Stream test is also a special case of the monkey test.

Consider the output from a generator as a sequence of overlapping 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from the generator. We generate 2^{21} (overlapping) 2-letter words (from $2^{21} + 1$ "keystrokes") and count J, the number of missing words—that is, the number of 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean $\mu = 141909$ and $\sigma = 290$. The standard deviation and mean are exact. Thus $(J - 141909)/290$ should be a standard normal variable.

Our implementation of the OPSO test starts from the 10 leftmost bits of the generator and computes the p-Value for the normal statistic. Then, it restarts the same sequence and uses the next group of 10 bits by moving over one bit to the right, and repeats the computation. It repeats this a number of times and the p-Values are used for a Kolmogorov-Smirnov test.

The OQSO test is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the generator. The mean number of missing words in a sequence of 2^{21} four-letter words, ($2^{21} + 3$ "keystrokes"), is again 141909, with $\sigma = 295$. The mean is based on theory; σ comes from extensive simulation.

The DNA test considers an alphabet of 4 letters: C, G, A, T, determined by two designated bits in the sequence generated by the generator. It uses 10-letter words, so that as in OPSO and OQSO, there are 2^{20} possible words, and the mean number of missing words from a string of 2^{21} over-lapping 10-letter words ($2^{21} + 9$ "keystrokes") is 141909. The standard deviation $\sigma = 339$ was determined as for OQSO by simulation.

6.7 Overlapping Permutations

These tests are overlapping m-tuple tests for which elements of the overlapping m-tuples are not independent, or even successive states of a Markov chain. Let $u_1, u_2, u_3, \ldots$ be uniform variates produced by a RNG. Each of the overlapping 3-tuples (u_1, u_2, u_3), (u_2, u_3, u_4), (u_3, u_4, u_5),..., is in one of six possible states:

$$S_1\text{: } x < y < z; \quad S_2\text{: } x < z < y; \quad S_3\text{: } y < x < z;$$
$$S_4\text{: } y < z < x; \quad S_5\text{: } z < x < y; \quad S_6\text{: } z < y < x.$$

Thus overlapping triples of u's lead to a sequence of states such as

$$3, 3, 2, 5, 1, 4, 3, ..., 3, 2, 5, \ldots.$$

If w_{ijk} represents the number of times that the successive states i, j, k appear in the state sequence, then

$$\sum (w_{ijk} - \mu_{ijk}) c^{-}_{ijk,rst} (w_{rst} - \mu_{rst})$$

will have, asymptotically, a χ^2 distribution. The means, covariance matrix C, and any weak inverse C^- must be found.

6.8 Overlapping Sums

Let $m \geq 100$ be a fixed integer. Take a sequence of iid $U(0,1)$ random variables $U_1, U_2, \ldots$, and form the overlapping sums $S_1 = U_1 + U_2 + \cdots + U_m$, $S_2 = U_2 + U_3 + \cdots + U_{m+1}$, and so on. The S_i's, $i = 1, 2, \ldots, m$ are virtually normal with a covariance matrix. The covariance matrix is easy to calculate. Clearly $E(S_i) = m/2$, and $V(S_i) = m/12$, $i = 1, 2, \ldots, m$. Furthermore, if $1 \leq i < j \leq m$, then S_i and S_j have a sum S of $m - j + i$ uniform values in common with $X = S_i - S$, S, and $Y = S_k - S$ being mutually independent. Therefore, $Cov(S_i, S_j) = \frac{m-j+i}{12}$.

Thus, if C denotes the m-by-m covariance matrix of the S_i's, the matrix $12C$ is Toeplitz with diagonals m, $m-1$,...,1. A cholesky factorization yields $C = VV^T$, where V is lower triangular. Since V^{-1}, the inverse of a lower triangular matrix is easily computed, we can convert the vector s of S_i's to independent normals via the linear transformation $x = V^{-1}s$ which can be tested for normality or uniformity after converting to uniforms via the normal cdf.

6.9 Park

Let each point in m-space be the center of a cubic or spherical "car", of specified size, and suppose we park "by ear" (as many people do). If $c_1, c_2, c_3, \ldots$ are non-overlapping cars, already parked, we try to park randomly until we succeed with a car that does not hit any of those already parked, then add the new car to the list. Out of n tries, we will have a list of $k(n)$ cars successfully parked. The distribution of $k(n)$ is not known, but simulation with a good RNG gives the mean and variance accurately enough for comparison with other generators.

6.10 Runs

The runs test is a classic test described in many books. See for example [1].

We implement the runs test by computing the statistics for runs-up and runs-down in sequences of length n. This is repeated a number of times and a Kolmogorov-Smirnov test is done on the p-values.

6.11 Spheres

The three-dimensional spheres test goes as follows.

Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is very close to being exponentially distributed with mean $120\pi/3$. Thus, the radius cubed is exponential with mean 30. The mean is obtained by extensive simulation. The test generates 4000 such spheres a number of times. Each minimum radius r, cubed leads to a uniform variable by means of the transformation $1 - \exp(-r^3/30)$, then a Kolmogorov-Smirnov test is done on the uniforms.

6.12 Squeeze

The Squeeze test uses the random uniform values in $[0, 1)$. Starting with $k = 2^{32}$, the test finds J, the number of iterations necessary to reduce k to 0 using the reduction $k = \lfloor kU \rfloor$, where U is a random uniform. A sample of $100,000$ J's is used for a χ^2 test of the cell frequencies.

7 Adding Tests and Generators

We have tried to make it easy to add generators in tests. To add a generator, a user must implement the methods described in section 3. Similarly for a test, the methods in section 4 must be implemented. Once these Java classes compile, they need to placed in the `CLASSPATH`. Then a property file—a simple text file—which lists available generators and tests has to be edited; the new tests and generators must be added to the file.

That's it. No recompilation of the entire application is needed. JDiehard will pick up the new generators and tests upon intialization.

It is also possible to add generators written in languages such as C and C++ via the Java Native Interface. The steps are more involved.

8 Future plans

We hope to exploit the RS-Java interface to make JDiehard available to R and S users.

References

[1] *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., 1998.

[2] G. Marsaglia and A. Zaman. Monkey tests for random number generators. *Computers and Mathematics with Applications, Part A,* 26(9):1–10, 1993.

[3] George Marsaglia. A current view of random number generators. In *Computer Science and Statistics: Proceedings of the 16th Symposium on the Interface*.