Available online at www.sciencedirect.com
ScienceDirect
Computational Statistics & Data Analysis 51 (2006) 1614–1622
COMPUTATIONAL STATISTICS & DATA ANALYSIS
www.elsevier.com/locate/csda

Generating good pseudo-random numbers

B.A. Wichmann[a,*,1], I.D. Hill[b]

[a] *National Physical Laboratory, Teddington, TW11 0LW, UK*
[b] *Chorleywood, UK*

Received 9 June 2005; received in revised form 6 March 2006; accepted 25 May 2006
Available online 21 June 2006

Abstract

A widely used pseudo-random number generator has been shown to be inadequate by today's standards. In producing a revised generator, extensive use has been made of a test package TestU01 for random number generators. Using this, criteria have been devised for the revised generator—also other high-quality generators have been identified. Facilities have been devised to allow the new generator to be used in a highly parallel environment, which is likely to be a feature of many future applications.

Keywords: Pseudo-random numbers; Tests for randomness; Non-overlapping sequences; Parallel applications

1. Introduction

In the early 1980s there seemed to be a need for a pseudo-random generator that would have good statistical properties, could easily be implemented in any programming language, would give the same results on any computer, and could run on 16-bit computers without overflow problems.

With suitably chosen constants, multiplicative congruential generators were known to do well, but with the 16-bit restriction, it would be difficult to find any constants that would give good statistical properties, so we investigated whether it would be possible to combine more than one, relatively poor, generator in some way that would give better properties than those of the individual components.

Having found two such generators, we tried combining them by adding their results and taking the fractional part of the answer. Although still inadequate, the results were sufficiently encouraging as to suggest that if a third component were added, it would give what we were seeking. Algorithm AS 183, Hill and Wichmann (1982) and Wichmann and Hill (1982) resulted.

It has had a 'good innings' but its cycle length of about 7×10^{12} must now be considered inadequate. It has been reported (McCullough and Wilson, 2005) as having failed some tests at a probability level of less than 10^{-15}, which surely is indicative of a major failing.

Computing developments over the last quarter of a century now make a better version both possible and desirable. In particular, there does not now seem to be a need for the 16-bit restriction, as 32-bit availability is almost universal.

* Corresponding author. Tel.: +44 1483 766729.
E-mail address: Brian.Wichmann@bcs.org.uk (B.A. Wichmann).
[1] Also at The Open University.

0167-9473/$ - see front matter
doi:10.1016/j.csda.2006.05.019

In view of the widespread use that the original version has enjoyed, it seems wise to retain the same underlying plan, see Section 3.

2. Testing a generator

In 1982, the work required to test the generator was very much larger than that required to write it. Fortunately, there are now publicly available test suites for random number generators, and that substantially reduces the effort involved. Moreover, the choice of statistical tests has been made independently of ourselves.

Two such test suites are: DIEHARD, (Marsaglia, 1985) and TestU01 (L'Ecuyer and Simard, 2005). It has been reported (McCullough and Wilson, 2005) that our old generator passed DIEHARD, but failed the more recent tests in TestU01. The TestU01 package is very comprehensive with many individual tests but also three batteries of test: Small Crush, Crush, and Big Crush. Our aim with any generator is to 'pass' the Big Crush tests. A review of these tests (McCullough, 2006) was particularly helpful. Big Crush uses 2^{38} random values and can take over 24 h to execute, depending mainly upon the speed of the generator being tested.

Big Crush reports P-values for all its tests, and signals those that come outside the [0.01..0.99] range. As it produces 124 P-values altogether, 1 or 2 would be expected to fall outside this range even for perfect randomness, though it must be remembered that not all the tests will be independent. In addition, TestU01 indicates catastrophic failures (defined as outside the $[10^{-15} \ldots 1 - 10^{-15}]$ range)—these should clearly not arise with any high-quality generator.

The Big Crush test should be run at least twice, with different seeds, as an insurance against an exceptional single run. Big Crush does not specify a 'pass criterion' as such. It is not sufficient merely to have few results outside the [0.01 . . . 0.99] range. For a reasonable degree of randomness, the P-values should themselves be more-or-less uniformly distributed between 0 and 1. So to judge a new generator, we actually use three requirements which must all be satisfied:

1. There must be no catastrophic failure: (P value outside the $[10^{-15} \ldots 1\text{–}10^{-15}]$ range).
2. For any component test with a value outside the [0.01 . . . 0.99] range, we repeat the test a further 4 times. Of these further four runs, we require not more than one to indicate a value outside the above range. This criterion was suggested by Richard Simard, one of the Big Crush authors.
3. For each run of Big Crush we consider the distribution of the P-values by calculating the Greenwood (1946) statistic and finding its approximate tail-area probability using the technique given in Hill (1979). We require the two-tailed value to come within the [0.01 . . . 0.99] range. Preferably it should come within the [0.1 . . . 0.9] range, but even for perfect randomness, such limits would be violated on 20% of occasions, so it would be unreasonable to insist upon it.

We are now in a position to know when we have an acceptable generator (which need not be our own).

3. Constructing a revised generator

The obvious way to proceed was simply to enhance the existing generator by using three components with suitable constants for 32-bit arithmetic rather than the 16-bit arithmetic we used in 1982. Unfortunately, such a generator failed according to the criteria above. Firstly, a multinomial distribution test failed with Big Crush with a P-value outside the [0.01 . . . 0.99] range. When repeated 4 times, the same test failed a further 3 times. Secondly, the observed values of the Greenwood statistic gave $P = 0.076$ on a first test and $P = 0.050$ on a second, not actually failing but too low for comfort. In consequence, we decided to add a further cycle to make our new generator a 4-cycle system.

Using the same design method, we need four primes p_1, p_2, p_3 and p_4, such that $p_i < 2^{31}$. In order to ensure that the generator has the maximum period, we select the $p_i - 1$ to have no common factor other than 2. It is straightforward to write a program to find suitable primes, the candidate ones being: 2 147 483 579, 2 147 483 543, 2 147 483 423 and 2 147 483 123. The period of the generator is now substantially more than the 2^{60} recommended by L'Ecuyer (1994).

The referees consider we should have undertaken some theoretical tests on the generator. Specifically, the spectral test (Knuth, 1981) (Section 3.3.4) could have been used to select the primes as was done in L'Ecuyer and Andres (1997). Because the present generator is simply the extension of our previous generator, we did not consider theoretical testing, but if we were developing another random number generator from scratch, we certainly would do theoretical testing, such as the spectral test. However, since the generator was nearly adequate with only 3 cycles, we feel confident that the proposed 4-cycle generator will prove adequate in practice.

The constituent linear congruence generators have no additive constant, but we need to choose multipliers with a range of values up to $\sqrt{p_i - 1}$ such that each value is a primitive root of $p_i - 1$. Suitable values can again be found with the aid of a short program. The candidate multipliers are: 11 600, 47 003, 23 000 and 33 000, respectively.

Combining these four generators in a simple way would then require 64-bit integer arithmetic, which is as follows:

$ix := 11\,600 \times ix \bmod 2\,147\,483\,579;$

$iy := 47\,003 \times iy \bmod 2\,147\,483\,543;$

$iz := 23\,000 \times iz \bmod 2\,147\,483\,423;$

$it := 33\,000 \times it \bmod 2\,147\,483\,123;$

$\mathrm{W} := ix/2\,147\,483\,579.0 + iy/2\,147\,483\,543.0 + iz/2\,147\,483\,423.0 + it/2\,147\,483\,123.0;$

return $W - \lfloor W \rfloor$.

Note that the four constituent generators are combined by taking each as a fraction of its prime, summing them and taking the fractional part of the result.

Sixty four-bit arithmetic can be avoided in the same way as with the old generator. For the first constituent, we have $2\,147\,483\,579/11\,600 = 185\,127.89\ldots$ and $2\,147\,483\,579 - 185\,127 \times 11\,600 = 10\,379$. Hence the resulting computation becomes:

$$ix := 11\,600 \times (ix \bmod 185\,127) - 10\,379 \times (ix \div 185\,127).$$

However, if this result is negative, then 2 147 483 579 must be added.

The algorithm in the variant suitable for 32-bit arithmetic is

$ix := 11\,600 \times (ix \bmod 185\,127) - 10\,379 \times (ix \div 185\,127);$
$iy := 47\,003 \times (iy \bmod 45\,688) - 10\,479 \times (iy \div 45\,688);$
$iz := 23\,000 \times (iz \bmod 93\,368) - 19\,423 \times (iz \div 93\,368);$
$it := 33\,000 \times (it \bmod 65\,075) - 8\,123 \times (it \div 65\,075);$
if $ix < 0$ **then**
$ix := ix + 2\,147\,483\,579;$
if $iy < 0$ **then**
$iy := iy + 2\,147\,483\,543;$
if $iz < 0$ **then**
$iz := iz + 2\,147\,483\,423;$
if $it < 0$ **then**
$it := it + 2\,147\,483\,123;$
$\mathrm{W} := ix/2\,147\,483\,579.0 + iy/2\,147\,483\,543.0 + iz/2\,147\,483\,423.0 + it/2\,147\,483\,123.0;$
return $W - \lfloor W \rfloor$.

This generator passed the Big Crush test according to our criteria. Using two runs, with different seeds, there was only one value outside the [0.01..0.99] range, and repeating the particular test four more times, the value was within the range each time. The Greenwood statistic gave $P = 0.22$ on the first occasion and $P = 0.52$ on the second, which can be regarded as highly satisfactory.

4. Some properties

The four $p_i - 1$ are

$$2\,147\,483\,579 - 1 = 2 \times 1\,073\,741\,789,$$

$$2\,147\,483\,543 - 1 = 2 \times 3137 \times 342\,283,$$

$$2\,147\,483\,423 - 1 = 2 \times 7 \times 557 \times 275\,389,$$

$$2\,147\,483\,123 - 1 = 2 \times 1\,073\,741\,561.$$

This implies that the period of the generator is

$$\begin{aligned}
&2 \times 1\,073\,741\,789 \times 3137 \times 342\,283 \times 7 \times 557 \times 275\,389 \times 1\,073\,741\,561 \\
&\quad = 2\,658\,454\,842\,761\,624\,389\,388\,266\,709\,412\,111\,698, \\
&\quad \approx 2.65 \times 10^{36}, \\
&\quad \approx 2^{121}.
\end{aligned}$$

The operations $\times m_i \bmod p_i$ for each of the 4 cycles have an inverse of the same form. To find the inverse of m_1, we compute the continued fraction for m_1/p_1:

$$\frac{11\,600}{2\,147\,483\,579} = 1/\left(185\,127 + 1/\left(1 + 1/\left(8 + 1/\left(1 + 1/\left(1 + \frac{1}{610}\right)\right)\right)\right)\right).$$

Removing the $\frac{1}{610}$ and multiplying up we get

$$\frac{11\,600}{2\,147\,483\,579} \approx \frac{19}{3\,517\,430}.$$

or

$$19 \times 2\,147\,483\,579 - 1 = 11\,600 \times 3\,517\,430.$$

Hence the inverse of 11 600 is 2 147 483 579 − 3 517 430 = 2 143 966 149. The other three inverses are 197 144 682, 981 586 662 and 1 289 335 852.

Using these four inverses as multipliers, a new generator could be constructed which would have essentially the same statistical properties as the original one.

For our old generator, Zeisel (1986) pointed out that the 3 cycles could be combined to re-write the generator in the form:

$$X_{n+1} = 16\,555\,425\,264\,690 \times X_n \bmod 27\,817\,185\,604\,309.$$

Similarly, the new generator has a single-cycle version in which the modulus is the product of the four primes. To find the multiplier a we need to solve the four equations: $a = a_i \bmod p_i$ for the four individual multipliers 11 600, 47 003, 23 000 and 33 000 and the four primes 2 147 483 579, 2 147 483 543, 2 147 483 423, and 2 147 483 123. These equations can be solved using the Chinese Remainder Theorem, as pointed out by Zeisel. Producing an explicit solution would be of no real benefit since it is impractical to compute the random numbers this way. The NPL web site (Wichmann and Hill,) gives the explicit solution which has a modulus of 39 digits. Using multi-length arithmetic to undertake the computation would be substantially slower than the 4-cycle implementation provided.

McLeod has pointed out that the precision of the floating point arithmetic influences the values that can be produced (McLeod, 1985). His analysis was concerned with obtaining 0.0 with a computer with only 23 mantissa bits. It is doubtful that a precision as low as this should be used for serious computation, but the analysis is indicative in other ways. The old generator produced essentially 48 ‘bits’ of randomness by combining three 16-bit generators. If the old generator was used to produce IEEE double length values which have 53 bits in the mantissa, then the three integers in the seeds could be computed from the result. This implies that the next value can be computed! For this reason, our generator cannot be regarded as cryptographically strong. With the new 4-cycle generator, the number of random bits is roughly 121 implying that no problems should arise with IEEE double length arithmetic—although, as McLeod noted, the value 0.0 can be produced.

Timings have been made of this generator on an Apple 1.6 GHz Power PC G5 as follows:

Generator	Millions of calls per second
Ada GNAT	1.31
Old generator, 32-bit	1.91
Old generator, 16-bit	1.76
3-Cycle generator, 64-bit	0.81
3-Cycle generator, 32-bit	1.93
4-Cycle generator, 64-bit	0.65
4-Cycle generator, 32-bit	1.57
C coding of new 32-bit version	3.97

The C generator times are not strictly comparable with the others as the timing methods were different—it seems that the C generator is about 20% faster than the Ada ones. The 16-bit old generator and the 32-bit new 3-cycle one are roughly comparable, the only difference being the size of the operands. The timing shows that even when 64-bit integer arithmetic is available, the 32-bit version can be significantly faster. Of course, our statistical testing implies that only the 4-cycle generators are acceptable—with the 32-bit one being the fastest (at least in this case).

In 1982 (Hill and Wichmann, 1982), the old generator took 0.85 ms on the PDP11 of its day. This implies that the old generator would repeat after 187 years. The new generator, based upon the timings above, would repeat in about 8000 times the age of the earth! In other words, the increase in the period of the new generator seems to be adequate to cater for the likely increase in computer speeds over the next 20 years. (In contrast, the old generator on an Apple G5 machine can execute the entire sequence in 49 days, which shows that the period is indeed inadequate.)

An interesting question is the fraction of the period p which can safely be used for a generator. Ripley (1990) proposes that if the number used is n, then $p \gg 200n^2$. For the generator proposed here this limits the number n to less than about 2^{56}. We thank a referee for this observation.

5. A generator package

Programming languages and implementations typically provide a random number facility. In the case of the C language (ISO/IEC 9899, 1999), this provides integers only in the 0..`RAND_MAX` range, and with a means of resetting the seed. Since the integers have type `int`, the range need only be 16 bits.

In contrast, Ada 95 (ISO/IEC 8652, 1995) provides two comprehensive packages for random numbers. These are very similar, one being for type `Float` and the other generic for any discrete type. In both cases, several sequences can be used, and the state of any generator can be saved or restored. For handling the setting of seeds from external information, the state can be saved or restored to/from a string. The standard makes the observation ‘No one algorithm for random number generation is best for all applications’. Two problems with the existing Ada facilities are worth noting: random numbers of type Long_Float are not available, and the required period for the generator when the numerics annex is implemented is only $2^{31} - 2$.

The Ada packages can provide a random number by means of a function. However, an Ada function cannot change its parameter, which implies that the side-effect which the function must perform to advance the cycle must be undertaken indirectly. For those concerned with program proof, typically for highly critical situations, such behaviour is not allowed. Hence the SPARK Ada subset (Barnes, 2002) could not be used to write a random number generator in the functional style. These considerations led to the formulation in Ada different from that in the standard library. Other changes from the Ada 95 standard specification is to produce a result of type Long_Float and to have the Initiator value to the Reset procedure to be positive. The reason for the latter change is for the initialization to align to the proposals for handling multiple sequences.

Abstractly, one would like to hide the state, which in Ada is achieved by means of a private type. However, one does need to set the seeds and hence some form of visibility is needed, which is undertaken by means of conversion to a string. The ‘size’ of the state is given by the length of the string, which for the generator here is given by the four integers in decimal.

Both Ada and Java provide a simple mechanism to set the seeds. The Ada GNAT implementation uses a 32-bit integer value to set the seed, although this is not adequate to produce all values of the state (the period is about 2^{49}). (The string facility can be used to cover all values.) With Java, the situation is reversed with only 48 of the 64 bits of the value provided being effective in setting the seed. Note that changing the actual algorithm for random numbers could easily alter the relationship between the seed size and the state size.

In Ada 95 and Java it would be possible to undertake an implicit initialization, perhaps using the clock, on the declaration of a generator. We have not done this with the Ada 95 implementation, since we will show later that when many sequences are required, additional care is required with the initialization.

Java shares with Ada the need to provide random numbers in the presence of tasking which implies that the state data must be separated from the code and be able to be placed within the data associated with a task. `Random` is a constructor class, while the methods are of the form `next...`. In fact, the Java class provides very extensive facilities, see (Class Random,). Here, the one generator has methods for providing uniform random values of all the major Java types, as well as a Gaussian for `double`. (There is another random facility in the class `Math` which we do not consider here.)

One property of Java is that of strict portability. For instance, the *sin* function must produce the nearest approximation to the mathematical result. For the random function, this implies that when initialized with a specific value, the sequence is determined. Hence, a strict implementation cannot change the algorithm for the sequence generation—this is unfortunate since the simple linear congruence generator used could otherwise be replaced by a generator passing Big Crush.

Producing random numbers is not quite the same as producing repeatable, random-like values in a sequence. Strict repeatability, as in Java, is useful in applying a technique of generating random test cases for software (Wichmann,). Each test, no matter how complex when generated, can be recorded merely by the seeds. Retesting can be undertaken by regeneration and regression testing by regenerating just those tests which failed. Strict repeatability is the basis of all science, though, where a result depends upon the use of random numbers, it is necessary to show that conclusions are unchanged if a different set of numbers are used. An enhancement to a random number generator library might improve its statistical properties but could make previous code strictly non-repeatable, but, of course, the conclusions should still be valid.

6. Generating many sequences

Consider the problem of undertaking a Monte Carlo simulation (Cox et al., 2003) on a highly parallel system with 100 or more processors. One needs hundreds of different sequences which should not overlap at all.

Given an existing long period generator, even with a randomly chosen seed, there is a small risk that two sequences will overlap. What is the best approach to take under such circumstances? Should one accept the risk, which would certainly be small with the generator presented here?

In fact, for the generator proposed here, the solution appears to be quite simple. We assume each parallel process is given a unique number n. For each simulation, fixed values x, y, z are taken for the first three seeds of the generator. The fourth seed is set to n. For any sequence to overlap, the first three integers must be x, y and z, but this can only arise after about 2^{90} calls of the generator. In other words, we are splitting the generator by means of starting from fixed points on the first 3 cycles.

It is not always possible to obtain the same effect with the other generators which pass the Big Crush tests. One needs a means of splitting the entire sequence into subsequences which are further apart than the likely number of calls made to the generator.

Unfortunately, our simple solution above has a flaw. Assuming we have 100 parallel processes, when they start execution, the first random number produced will be very similar! We need therefore to devise a method such that the sequence of random numbers given by the first number from each of our processes themselves pass specific tests for randomness. This property may not be needed by some applications but could be important for others.

6.1. A list of sequences

The generation of multiple sequences is a special case of a more general problem of producing a matrix of random numbers

$$\begin{array}{cccc} s_{11} & s_{12} & s_{13} & \cdots \\ s_{21} & s_{22} & s_{23} & \cdots \\ s_{31} & s_{32} & s_{33} & \cdots \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \end{array}$$

Our usual sequence of random numbers is represented by the rows. Of course, the rows are much longer than any likely use of the random values.

The solution given above was to set s_{n1} to (x, y, z, n), which we know is not adequate in some circumstances. It is inadequate because the columns do not give a statistically random sequence.

In 2001, a researcher in Spain, Pedro Gimeno, reported a problem to Knuth which showed the Knuth generator as giving unacceptable results. Using the notation above, the issue was that the sequence s_{n1} was not random. Does this matter? This is exactly the problem of producing a matrix of random values so that the columns as well as the rows are random.

If one requires a number of *independent sequences*, each one of which is random (say, passing Big Crush), then the proposal above is fine. Here, only the rows are relevant. However, if the application only uses a few random values from each sequence, and the ordering of the sequences is important, then the problem that was reported to Knuth may be critical.

Can we therefore adapt the generator to produce a list of sequences? The properties we require is that each row and column should be statistically sound and that none of these should overlap.

Using our four primes p_i, and the existing generator which gives the rows above, how can we produce the columns? The answer is simple. For two of the primes, say p_1 and p_2, we produce another two multipliers distinct from those used in the original generator (and their inverses). The method of obtaining the next row is by applying the multiplier to the first two values while leaving the other two fixed. (The operations of moving along the row or going down the column are commutative.) By using two new multipliers we ensure that the column sequences pass at least Small Crush, and, of course, each individual row passes Big Crush as before.

The two new multipliers are 46 340 and 22 000 to give the cycles

$$ix := 46\,340 \times ix \bmod 2\,147\,483\,579$$

and

$$iy := 22\,000 \times iy \bmod 2\,147\,483\,543.$$

Taking the seeds for s_{11} in the matrix above as ix, iy, iz and *it*, we compute the seeds corresponding to s_{21} by

$ix := 46\,340 \times (ix \bmod 46\,341) - 41\,639 \times (ix \div 46\,341);$
$iy := 22\,000 \times (iy \bmod 97\,612) - 19\,543 \times (iy \div 97\,612);$
if $ix < 0$ **then**
$\qquad ix := ix + 2\,147\,483\,579;$
if $iy < 0$ **then**
$\qquad iy := iy + 2\,147\,483\,543;$

where *iz* and *it* are unchanged. Repeating this operation we can compute the seeds for s_{31}, and so on.

Keeping *iz* and *it* fixed ensures that no overlap occurs for over 2.3×10^{18} values at the very minimum.

Since such generators running in parallel on different processors, will, using this method, have two of the four components in common, it might seem likely that they would suffer from greater correlation between them than if all four were varying separately on each. In the event, this is not so. Taking 10 cases of 10 000 pairs of numbers from non-overlapping parts of the sequence with all four components varying separately, 95% confidence limits for the mean correlation were −0.044 to 0.011. Doing the same with only two components varying separately and the other two varying together, using the technique for deriving seeds given above, 95% limits were −0.003 to 0.010. These both include zero, which is satisfying, and the latter limits are marginally narrower than the former ones. We do not for one moment suggest that the latter would actually give less correlation in general, but there is certainly no evidence here of it being greater.

7. Conclusions

We know of several generators which pass the Big Crush battery of statistical tests. We think that only such generators can be recommended for general use. These can be compared for basic properties (fastest first):

Name	Period	Lines of code	Size of state (bytes)	Relative timing
ISAAC (Jenkins)	$\geqslant 2^{40}$	97	1024	1.0
AES (NIST, 2001)	not known	85	16	2.1
Mersenne twister (Matsumoto and Nishimura, 1998)	$2^{19\,937}-1$	48	2500	2.3
MRG32k3a (L'Ecuyer, 1999)	$\approx 2^{191}$	31	48	2.7
Knuth, TAOCP (Knuth, 1981)	$\approx 2^{129}$	90	404	4.9
CLCG4 (L'Ecuyer and Andres, 1997)	$\approx 2^{121}$	34	16	9.2
This paper—4-cycle	$\approx 2^{120}$	26	16	10.0
MRG63k3a (L'Ecuyer, 1999)	$\approx 2^{377}$	40	48	14.3

The last three columns should only be taken as an indication of the basic characteristics since the generators operate in rather different ways which makes direct comparison problematic.

Our combined 4-cycle generator can be recommended for the following reasons:

1. Our generator is easy to code in any programming language. It does not depend upon bit manipulation used by several of the other generators.
2. The state is small and easy to handle.
3. It is possible to use the generator to provide multiple sequences needed for highly parallel applications.

We wish to advocate any generator which satisfies our criteria for passing Big Crush. A generator should also be able to produce at least 1000 subsequences separated by at least 2^{50} values to handle highly parallel applications. In addition, wherever possible, appropriate theoretical analysis of the generator properties should be undertaken.

Acknowledgements

In 1982, our original generator was produced by us almost in isolation. In contrast, the revision has benefited substantially from the published literature and help from the following: Bruce McCullough (Drexel University, Philadelphia), Pierre L'Ecuyer and Richard Simard (Université de Montréal) and support from the National Physical Laboratory under the Software Support for Metrology programme. The Open University electronic library was invaluable in obtaining details of many of the generators referenced here.

References

Barnes, J., 2002. High Integrity Software—the SPARK Approach to Safety and Security. Addison-Wesley, Reading, MA.

Class Random. See: ⟨http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html⟩.

Cox, M.G., Harris, P.M., Siebert, B.R.L., 2003. Evaluation of measurement uncertainty based on the propagation of distributions using Monte Carlo integration. Izmeritelnaya Technika 9, 9–14.

Greenwood, M., 1946. The statistical study of infectious diseases. J. Roy. Statist. Soc., Ser. A 109, 85–109.

Hill, I.D., 1979. Approximating the distribution of Greenwood's statistic with Johnson distributions. J. Roy. Statist. Soc., Ser. A 142, 378–380 (See also Hill, I.D., Corrigenda. J. Roy. Statist. Soc., Ser. A 144, 388. 1981).

Hill, I.D., Wichmann, B.A., 1982. A pseudo-random number generator. NPL Report, DITC 6/82.
ISO/IEC 8652, 1995. Programming languages—Ada. ISO. Geneva.
ISO/IEC 9899, 1999. Programming languages—C. ISO. Geneva.
Jenkins Jr. J. ISAAC (Indirection, Shift, Accumulate, Add, and Count) generator. ⟨http://burtleburtle.net/bob/rand/isaacafa.html⟩.
Knuth, D.E., 1981. The Art of Computer Programming, vol. 2, Seminumerical Algorithms. Section 3.2.1. Addison-Wesley, Reading, MA. (The 2002 printing is required.)
L'Ecuyer, P., 1994. Uniform random number generation. Ann. Oper. Res. 53, 77–120.
L'Ecuyer, P., 1999. Good parameters and implementations for combined multiple recursive random number generators. Oper. Res. 47 (1), 159–164.
L'Ecuyer, P., Andres, T.H., 1997. A random number generator based on the combination of four LCGs. Math. Comput. Simulation 44, 99–107.
L'Ecuyer, P., Simard, R., 2005. TestU01: A software library in ANSI C for empirical testing of random number generators. Laboratoire de simulation et d'optimisation. Université de Montréal IRO. Version 6.0, dated January 14, 2005. ⟨http://www.iro.umontreal.ca/∼simardr/⟩.
Marsaglia, G., 1985. The Marsaglia random number CDROM, with the Diehard battery of tests of randomness, produced at Florida State University under a grant from The National Science Foundation. Available at ⟨http://www.cs.hku.hk/∼diehard⟩.
Matsumoto, M., Nishimura, T., 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8 (1), 3–30.
McCullough, B.D., 2006. A Review of TESTU01. J. Appl. Econom., to appear.
McCullough, B.D., Wilson, B., 2005. On the accuracy of statistical procedures in microsoft excel 2003. Comput. Statist. Data Anal. 49 (4), 1244–1252.
McLeod, A.I., 1985. Remark on Algorithm AS 183. Appl. Statist. 34, 198–200.
NIST. Specification for the advanced encryption standard (aes). NIST special publication 197 (FIPS-197), National Institute of Standards and Technology (NIST), 2001. See, ⟨http://csrc.nist.gov/encryption/aes/⟩.
Ripley, B., 1990. Thoughts on pseudorandom number generators. J. Comput. Appl. Math. 31, 153–163.
Wichmann, B.A., Some Remarks about random testing. Download from: ⟨http://www.npl.co.uk/ssfm/download/abstracts.html#60⟩.
Wichmann, B.A., Hill, I.D., 1982. Algorithm AS 183: an efficient and portable pseudo-random number generator. Appl. Statist. 31, 188–190. See also: Wichmann, B.A., Hill, I.D., 1985. Correction: algorithm AS 183: an efficient and portable pseudo-random number generator. Appl. Statist. 33, 123. Reprinted with the correction In: Griffiths, P., Hill, I.D. Applied Statistics Algorithms, Ellis Horwood, Chichester, UK.
Wichmann, B.A, Hill, I.D., Generating good random numbers. See ⟨http://www.npl.co.uk/ssfm/download/abstracts.html#196⟩ for implementations of algorithm presented here in Ada and C, with further documentation.
Zeisel, H., 1986. Remark on Algorithm AS 183. Appl. Statist. 35, 89.