Generating good pseudo-random numbers

B. A. Wichmann
National Physical Laboratory[*], Teddington, UK.
I. D. Hill
Chorleywood, UK.

December 5, 2005

This is the long version of this paper with marginal references which refer to notes in an additional appendix.

Abstract

In 1982, the authors produced a pseudo-random number generator that has been widely used, but now 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

Version 0.96 of this paper, documentation and related software.

Randomness and chaos are anathema to the mathematician.
Music of the Primes: Marcus du Sautoy, 2003.

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

[*]and The Open University

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(1982)] resulted.

It has had a 'good innings' but its cycle length of about 7×10^{12} is now considered inadequate for some purposes, and it has been reported [McCullough and Wilson(2005)] as having failed some tests at a probability level of less than 10^{-15}.

Old gen., page 14

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. In view of the widespread use that the original version has enjoyed, it seems wise to retain the same underlying plan, and to regard the passing of a suitable barrage of empirical tests as adequate justification. There are many different constants that could have been put into the plan with success and we make no claim that those we recommend are necessarily better than others that might have been selected, but only that they have survived stringent testing.

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(2000)] and TestU01, [L'Ecuyer(2005)]. [McCullough and Wilson(2005)] report 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 hours to execute, depending mainly upon the speed of the generator being tested.

Testing, page 15

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 $[10^{-15}..1-10^{-15}]$ range) — these should clearly not arise with any high quality generator

Big Crush, page 15

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 $[10^{-15}..1 - 10^{-15}]$ 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 Greenwood's statistic ([Greenwood(1946)]) 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 three 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.

Test 3-cycle, page 15

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: 2147483579, 2147483543, 2147483423 and 2147483123.

Primes, page 14

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: 11600, 47003, 23000 and 33000 respectively.

Multipliers, page 14

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

```
$ix := 11\_600 \times ix \bmod 2147483579$;
$iy := 47\_003 \times iy \bmod 2147483543$;
$iz := 23\_000 \times iz \bmod 2147483423$;
$it := 33\_000 \times it \bmod 2147483123$;
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.

We avoid 64-bit arithmetic in the same way as with the old generator. For the first constituent, we have $2147483579/11600 = 185127.89\ldots$ and $2147483579 - 185127 \times 11600 = 10379$. Hence the resulting computation becomes:

$$ix := 11600 \times (ix \bmod 185127) - 10379 \times (ix \div 185127)$$

However, if this result is negative, then 2147483579 must be added.

The algorithm in the variant suitable for 32-bit arithmetic is: Logic, page 14

```
$ix := 11\_600 \times (ix \bmod 185\_127) - 10\_379 \times (ix \div 185\_127)$;
$iy := 47\_003 \times (iy \bmod 45\_688) - 10\_479 \times (iy \div 45\_688)$;
$iz := 23\_000 \times (iz \bmod 93\_368) - 19\_423 \times (iz \div 93\_368)$;
$it := 33\_000 \times (it \bmod 65\_075) - 8\_123 \times (it \div 65\_075)$;
**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**
    $it := it + 2\_147\_483\_123$;
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:

$$2147483579 - 1 = 2 \times 1073741789,$$

$$2147483543 - 1 = 2 \times 3137 \times 342283,$$

$$2147483423 - 1 = 2 \times 7 \times 557 \times 275389,$$

$$2147483123 - 1 = 2 \times 1073741561.$$

This implies that the period of the generator is: Factors, page 16

$$\begin{aligned}
&2 \times 1073741789 \times 3137 \times 342283 \times 7 \times 557 \times 275389 \times 1073741561 \\
&= 2658454842761624389388266709412111698 \\
&\approx 2.65 \times 10^{36} \\
&\approx 2^{121}
\end{aligned}$$

The operations $\times m_i \bmod p_i$ for each of the four 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: Inverse, page 16

$$\frac{11600}{2147483579} = 1/(185127 + 1/(1 + 1/(8 + 1/(1 + 1/(1 + \frac{1}{610})))))$$

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

$$\frac{11600}{2147483579} \approx \frac{19}{3517430}$$

or

$$19 \times 2147483579 - 1 = 11600 \times 3517430$$

Hence the inverse of 11600 is $2147483579 - 3517430 = 2143966149$. The other three inverses are 197144682, 981586662 and 1289335852.

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 three cycles could be combined to re-write the generator in the form:

$$X_{n+1} = 16555425264690 \times X_n \bmod 27817185604309$$

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 11600, 47003, 23000 and 33000 and the four primes 2147483579, 2147483543, 2147483423, and 2147483123. 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. Single LCG, page 17

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.

FLP, page 17

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

Timing, page 18

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, the old generator took 0.85 ms on the PDP11 of its day [Hill and Wichmann(1982)]. This implies that the old generator would repeat after 187 years. The new generator, based upon the timings above, would repeat in about 8,000 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.)

5 A generator package

Programming languages and implementations typically provide a random number facility. In the case of the C language ([C(1999)]), this provides integers

C, page 18

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 ([Ada(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 $2^{31} - 2$.

Ada 95, page 18

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 that 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.

Abstraction, page 19

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 [Java(2002)]. Here, the one generator has methods for providing uniform random values of all the major Java types, and 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. Java, page 20

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(2000)]. 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.

6 Generating many sequences

Consider the problem of undertaking a Monte Carlo simulation on a highly parallel system with a hundred 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 three 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. Many seq., page 20

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. Poor, page 20

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:

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

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.

Columns, page 20

The two new multipliers are 46340 and 22000 to give the cycles:

$$ix := 46340 \times ix \bmod 2147483579$$

and

$$iy := 22000 \times iy \bmod 2147483543$$

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
        $ix := ix + 2\_147\_483\_579;$
if $iy < 0$ then
        $iy := iy + 2\_147\_483\_543;$
```

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 generators running in parallel on different processors, using this method, will then 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 10000 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.

Correlation, page 22

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):

Table, page 21

Name	*Period*	*Lines of code*	*Size of state (bytes)*	*Relative timing*
ISAAC	$\geq 2^{40}$	97	1024	1.0
AES	??	85	16	2.1
Mersenne twister [Matsumoto(1998)]	$2^{19937}-1$	48	2,500	2.3
MRG32k3a [L'Ecuyer(1999)]	$\approx 2^{191}$	31	48	2.7
Knuth, TAOCP [Knuth(2002)]	$\approx 2^{129}$	90	404	4.9
CLCG4 [L'Ecuyer(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 would not necessarily wish to advocate our generator, but rather any generator which satisfies our criteria for passing Big Crush and has a means of handling highly parallel systems.

8 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

[Ada(1995)] ISO/IEC 8652: 1995. Programming languages — Ada. ISO. Geneva.

[Barnes(2002)] Barnes, J., High Integrity Software — the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[C(1999)] ISO/IEC 9899: 1999. Programming languages — C. ISO. Geneva.

[Matsumoto(1998)] 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.

[Greenwood(1946)] Greenwood, M. The statistical study of infectious diseases. *J.R.Statist.Soc.,* A, 109, 85–109. 1946.

[Hill and Wichmann(1982)] Hill I. D, and Wichmann B. A., A Pseudo-Random Number Generator. NPL Report, DITC 6/82 May 1982.

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

[Java(2002)] Class Random. See:
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html.

[Jenkins(2002)] Bob Jenkins Jr. ISAAC (Indirection, Shift, Accumulate, Add, and Count) generator. http://burtleburtle.net/bob/rand/isaacafa.html

[Knuth(2002)] Knuth, D.E., 1981. *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms. Section 3.2.1. AddisonWesley, Reading, MA. (The 2002 printing is required.)

[L'Ecuyer(1994)] L'Ecuyer, P., 1994. Uniform random number generation. *Ann. Oper. Res.* 53, 77–120.

[L'Ecuyer(1997)] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, 44:99–107, 1997.

[L'Ecuyer(1999)] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[L'Ecuyer(2005)] L'Ecuyer, P., and Simard, R. 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(2000)] 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`

[McLeod(1985)] McLeod A. I. Remark on Algorithm AS 183, *Appl. Statist.,* 34, 198–200, 1985.

[McCullough and Wilson(2005)] McCullough, B. D., and Wilson, Berry. On the Accuracy of Statistical Procedures in Microsoft Excel 2003. *Computational Statistics & Data Analysis.* 49(4), 1244–1252, 2005.

[McCullough(2006)] McCullough, B D (2006), A Review of TESTU01, *Journal of Applied Econometrics* (to appear)

[NIST(2001)] 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/.

[Wichmann(1982)] 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: Correction: algorithm AS 183: an efficient and portable pseudo-random number generator. *Appl. Statist.* 33, 123.

Reprinted with the correction in Griffiths P. and Hill I. D. *Applied Statistics Algorithms*, Ellis Horwood. 1985.

[Wichmann(2000)] Wichmann, B. A., Some Remarks about Random Testing. Download from: `http://www.npl.co.uk/ssfm/download/stress.pdf`

[Zeisel(1986)] Zeisel H. Remark on Algorithm AS 183, *Appl. Statist.,* 35, 89, 1986.

A Implementation notes

The marginal text in the main paper refers to the notes here. These notes give additional information not appropriate to a published paper, but would be useful for those wanting to use the new generator or to understand it in greater detail. Many of these notes refer to files available with the longer form of the paper from the NPL web site.

Those wishing to code the generator in a language could start from an Ada 95 version (the files `Version6.ads` and `Version6.adb`) or from 'C' (the files `Version6.h` and `Version6.c`). Please read these notes before coding. Some notes on implementing this generator appear in Appendix C.

A.1 Introduction

The old generator was tested by McCullough [McCullough(2006)] and we include here the test script and the results for completeness. The formulation can be undertaken in two ways using the facilities inside TestU01 without actually recoding the generator. The first formulation uses a three-cycle generator specification, see the code in `wich1.c` and the output `wich1.out`. The second formulation uses a method of composing three generators, see the code in `wich2.c` and the output `wich2.out`. Both fail catastrophically the Birthday Spacings test, the results being essentially identical.

Old gen.,

A.2 Constructing a revised generator

The primes were calculated with the aid of a short Ada 95 program. The program source text is the file `find.adb` and the output produced, the file `find.out`. The output file has been annotated with suitable primes based upon the factorization of $p - 1$.

Primes

The multipliers were calculated using the program `mult.adb` and details of its use are recorded in the file `mult.out`. The output is collected from many runs of the program and annotated to give the suitable values. The logic used to avoid overflow of 32-bits requires that the multipliers are less than the square root of the prime. (An alternative logic is possible, but we have retained the logic used in the old generator.)

Multipliers

When this paper was refereed, it was pointed out (correctly) that the Spectral Test should have been applied to check the choice of the primes and multipliers. See Section 3.3.4 of [Knuth(2002)] for details of this. We did not have software to undertake this test and we restricted ourselves to the testing reported here. However, since we know that the 3-cycle version was very near to passing Big Crush, we feel confident that the 4-cycle version is indeed adequate. The generator [L'Ecuyer(1997)] is very similar to ours and has been chosen using the Spectral Test and can usefully be reviewed by those interested.

The logic is identical to that of the original generator — the changes being in the choice of the constants and the use of four cycles rather than three. Logic

The logic is not given in a specific language since that seems to be potentially confusing when the reader may not be familiar with the language used. Implementations in Ada 95 and C are provided.

It is important to note that although the basic logic given here may be adequate in some contexts, a complete random package should be implemented, again available in Ada 95 and C.

Note that the final value returned is the fractional part of `W` and hence is never equal to 1.0. (However, producing 1.0 should not necessarily matter — code should not rely upon that.)

Further notes appear in section C.

A.3 Testing the generator

Since we knew that the old generator passed DIEHARD but failed TestU01, it seemed that the right thing to do was to aim at passing TestU01 with the new generator. It also seemed that TestU01 was more comprehensive than DIEHARD, indeed TestU01 has been described as DIEHARD with steroids! Testing
TestU01 has been revised and updated over the years, but this testing refers to version 6.0 dated 14th January 2005.

At first, BW could not get TestU01 to work on his Apple G5 which runs BSD Unix. Fortunately, Richard Simard managed to solve this problem even without his institution having an Apple. Bruce McCullough's review [McCullough(2006)] was useful in indicating the range of tests that could be undertaken with TestU01, although it was for an earlier version than that used here.

It is important to note that TestU01 requires that a generator to be tested be coded in 'C' in a specific way — for instance, the function must produce a `double`. For this generator, it is available with two different starting seeds as `Version6a.c` and `Version6b.c`. The new generator was coded as an 'external' generator, rather than being constructed from facilities internal to TestU01. The initial value of the seeds is set directly into the code. For the timing tests to be meaningful, the external generator must be compiled with the option `-O2`.

TestU01 has a built-in system for combining generators which can be used to construct our old generator from the three cycles. It was not possible to repeat this with the new generator since each individual cycle uses 64-bit arithmetic when coded as a simple multiply and add (no add in our case).

The 3-cycle generator was not quite adequate. The results from `Version4a.c` are in the file `biga.out` (and similarly for `Version4b`). Since some results Test 3-cycle
were outside the [0.01..0.99] range, further testing was done via the program `extra1.c`, the results of which are in `extra1.out` (which shows the failure to pass the criteria we have set).

The full results are in file `big6a.out`. The full results from `Version6b` are in `big6b.out`, and two summaries are: Big Crush

```
============== Summary results of BigCrush ==============

 Generator:        Generator Version6a
 Number of tests:  90
 Total CPU time:   25:52:44.55
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 39  RandomWalk1 C (L = 1000)       5.5e-3
 ----------------------------------------------
 All other tests were passed

============== Summary results of BigCrush ==============

 Generator:        Generator Version6b
 Number of tests:  90
 Total CPU time:   23:07:59.61
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  5  MultinomialBitsOver            0.9959
 13  BirthdaySpacings (t = 8)       0.9972
 ----------------------------------------------
 All other tests were passed
```

To pass our criteria, we need to repeat the three tests to ensure that the potential problem does not arise too often. These extra tests are the files `extra2.c`, `extra4.c` and `extra5.c`, with the results in `extra2.out`, `extra4.out` and `extra5.out`.

As expected, the individual cycles of the generator perform poorly as freestanding generators — they fail Small Crush catastrophically. The six cycles used — four in the main generator and two for the column generator and tested in the program `four.c`, the cycles in `cycles.c` and the results in `cycles.out`.

A.4 Some properties

See the file `find.out` which is annotated with the factorization. Factors

It is a straightforward but tedious calculation to find the inverses this way. The Unix tool `bc` and a Python program was used. The relationship between continued fractions and Euclid's algorithm which underlies this calculation is explained in [Knuth(2002)], section 4.5.3. Given the inverse, it is simple to check it! Inverse

Express 23000 / 2147483423 as a continued fraction

$$1/(93368 + 1/(1 + 1/(5 + 1/(2 + 1/(3 + 1/(14 + 1/(3 + 1/(5 + \frac{1}{2}))))))))$$

Removing the $\frac{1}{2}$ and multiplying up we get:

$$\frac{23000}{2147483423} \approx \frac{10513}{981586662}$$

$$2147483423 \times 10513 + 1 = 23000 \times 981586662$$

Hence the inverse is: 981586662.
Express 33000 / 2147483123 as a continued fraction

$$1/(65075 + 1/(4 + 1/(15 + 1/(1 + 1/(100 + 1/(1 + 1/(1 + \frac{1}{2})))))))$$

Removing the $\frac{1}{2}$ and multiplying up we get:

$$\frac{33000}{2147483123} \approx \frac{13187}{858147271}$$

$$33000 \times 858147271 + 1 = 2147483123 \times 13187$$

Hence the inverse of 33000 is $2147483123 - 858147271 = 1289335852$

Rather hard to solve the equations, although an explicit solution can be written down as: Single LCG

$$a = a_1 \times (p_2 \times p_3 \times p_4)^{p_1 - 1} + a_2 \times (p_1 \times p_3 \times p_4)^{p_2 - 1} + a_3 \times (p_1 \times p_2 \times p_4)^{p_3 - 1} + a_4 \times (p_1 \times p_2 \times p_3)^{p_4 - 1} \bmod p_1 \times p_2 \times p_3 \times p_4$$

The above formula has been programmed in Python (`single.py`) and gives:

$$X_{n+1} = 120333009958606346118146497013089037762 \times X_n \bmod 212676387817070635609756481954556615 13$$

The three-cycle generator gives:

$$X_{n+1} = 3957351460688778375479726849 \times X_n \bmod 9903518474220420479167438931$$

Our generator does not produce the bits directly from the integer calculation and hence relies on a floating point calculation to deliver a value in the [0.0..1.0] range. Hence the precision of the floating point is vital here. In McLeod's case, the floating point is less precise than the integer calculation. In this case, the final calculation can produce 0.0 even though algebraically it is not zero. (Note that the coding should ensure that the value 1.0 is not obtained, although this is not essential for most uses of the generator.) FLP

When the floating point is *more* precise than the integer calculation, then the integer values can be recovered from the floating point values. (Floating point will be more precise if the old generator produces a IEEE double length result.) This might mean trying a few values, since the integer part of the floating point value is not known. Assuming this can be done, then the next value in the sequence can be found. This implies that the algorithm is cryptographically weak and would not be appropriate for some applications. There are many papers about randomness with cryptographic strength, but this is not the concern here.

Timing

The table of the timing was produced from the Ada program `test_rand.adb`, for the first five generators written in Ada, and by the 'C' program `main.c`. The 'C' program uses the 3-cycle and 4-cycle versions of our generator for that language which is in the files `Version4.h`, `Version4.c` and `Version6.h`, `Version6.c`.

The GNAT generator is from L. Blum, M. Blum, and M. Shub, *SIAM Journal of Computing*, Vol 15. No 2, May 1986. No details of the quality of this generator could be located.

The 'C' clock measures CPU usage, while the Ada program uses elapsed time, which seems to account for most of the difference.

A 64-bit 'C' version could be produced, but it hardly seems worthwhile — it is clear the 32-bit version is best, at least on current machines.

A.5 A generator package

C

The C facility is so poor one wonders why it is included. This is partly due to the size of `int`. It is surely a trap for the unwary.

Ada 95

Although Ada 95 has two comprehensive packages, it is not without problems. Ada 95 as a standard is really in two parts: the main language and then some annexes which are optional for an implementation. Here, the vital one is the Numerics Annex, since additional requirements are placed upon an implementation when this annex is implemented.

The Numerics Annex places quite tight requirements on the precision of the mathematical functions which clearly implies that high quality is expected. Not so with the random number package requirements in two major ways:

1. The period only need be $2^{31} - 2$.

2. The basic floating point results only need have 6 digits of precision.

Here, we have taken the recommendation that the period of a quality generator should be at least 2^{60}, see [L'Ecuyer(1994)]. It has been stated that requiring a longer period would lead to an untestable requirement, but this is inconsistent with having any accuracy requirements on numeric operations.

The second reason above is just as important. Few numerical calculations these days are done to only 6 digits for good reasons. The TestU01 package requires results which are `double`, ie, 11 digits. Hence an Ada package cannot be tested with TestU01 (or rather, it would always fail).

The Ada standard (A.5.2 (44)) makes are interesting claim: *No one algorithm for random number generation is best for all applications.* It could be argued that the generator proposed here is at least *good* for all applications. Reasons for not being the best are:

1. Even among those that pass Big Crush, it is not the fastest.
2. With reduced statistical quality, a much faster generator would be possible.

Abstraction

A random number package should have a structure reflecting the underlying requirements rather than the specifics of the implementation. The language also affects the structure. The restrictions on Ada functions make for some difficulties, unless one uses procedures instead. The Ada 95 package uses two types for the generator, while the formulation here uses just one.

Our package specification is designed to give a specification suitable for any random number generator (although the value of Image_Width might need changing). However, in this paper, we are actually concerned with the internals and properties of the generator given in the private type Generator and in the package body (not listed here).

The version of Reset which has an integer parameter uses the 'column generator' technique to ensure that when Reset is called with values from 1 upwards, the first random number produced has the properties noted above, ie, passes Small Crush.

```
package Version6 is

   type Generator is private;

   subtype Uniformly_Distributed is Long_Float range 0.0 ..  1.0;

   procedure Random (Gen:  in out Generator; Value:  out Uniformly_Distributed);
      -- Value gives the next random number, Gen is advanced

   procedure Reset (Gen :  out Generator;
         Initiator :  in Positive);
      -- uses Initiator to reset Gen

   procedure Reset(Gen :  out Generator);
      -- uses the clock to reset Gen

   Image_Width :  constant := 44;

   subtype String_State is String(1..Image_Width);
      -- The character encoding of the state

   function Image (Gen :  in Generator) return String_State;
      -- Gives the character encoding of the generator.

   function Value (Coded_State:  String_State) return Generator;
      -- Gives the value of the Generator corresponding to the string

   function Next_Sequence(Gen :  in Generator) return Generator;
      -- Produces a new starting value for a generator for which a sequence
      -- of starting values produces a random sequence.

   private
      type Generator is record
            ix, iy, iz, it:  Integer := 1;
         end record;

end Version6;
```

A problem to be faced with an abstraction is the relative size of the parameter used to set the seeds and the size of the state-space. The other two generators compared in the conclusions have a very much larger state-space.

The Java implementation uses a simple (48-bit) linear congruence generator. The documentation refers to Knuth, but it does *not* use the lagged Fibonacci generator recommend by Knuth which passed Big Crush (see 10).

Python uses the old Hill/Wichmann generator, but have recently added the Mersenne Twister as well [Matsumoto(1998)].

A.6 Generating many sequences

The Mersenne twister and Knuth's generator have both been modified so that when using the values $1..n$ to initialize the generators, a random sequence is obtained. However, it is unclear how many calls can be made before the sequences could overlap. Of course, since both have long periods, the probability of an overlap is surely small. Many seq.

The 'random' sequence of n numbers produced by using the seeds (x, y, z, n) are so poor that the TestU01 system loops! However, this method of generating multiple sequences could still be recommended when there is no ordering between the sequences.

The additional multipliers also appear in the file `mult.out` as before. The code for producing another sequence in this manner can be seen from the function `Next_Sequence` in the package body file `Version4.adb`. This is tested using Small Crush in the files: `Version4k.c` source text of the column generator, `smallk.c` test program to all the Small Crush battery of tests, and `smallk.out` for the results of the tests. Note that the column generator advances ix and iy column-wise and then repeats the of the main generator. The result passes Small Crush. Columns

The coding of the version of Reset which uses an integer parameter follows the column method. The values of iz and it are fixed, and the values of ix and iy use the column logic. The number of changes made to ix and iy is equal to the value of the integer parameter.

The generator [L'Ecuyer(1997)] is very similar to ours but does not have the equivalant of the column generator and hence, as implemented, is not suitable for highly parallel applications.

The full implementation is to be found in the package body in file `Version6.adb`.

A.7 Conclusions

The period of the Knuth generator does not seem to be in [Knuth(2002)], but was found elsewhere.

The lines of code have been worked out by trying to lay out the code to be the same 'standard' as C. Blank and comment lines have been ignored. For Knuth, the lines in the file without the main (test) program — the comment and blank lines being off-set by loops and conditionals on one line. Table

The timing has been taken from the program `time4.c` which produced the output `time4.out`. The code for the other generators are available within the TestU01 system.

Apart from our generator presented here, the other generators that do *not* perform bit manipulation operations are CLCG4, MRG32k3a and MRG63k3a — the last two being recommended by Pierre L'Ecuyer.

B List of files

These are in two tables, both in alphabetical order. Table 1 contains the main files associated with the new generator (and its testing). Table 2 contains other files, such as those concerned with the 3-cycle generator which was inadequate. The notation x.ads/adb refers to two files x.ads and x.adb (the specification and body of the Ada package x). Similarly, the notation extra2.c/out refers to a program extra2.c (this one in C) and the output it produces extra2.out.

All the text files are in Unix line format.

C Implementing the generator in other languages

It would be best to implement a package or module corresponding to the Ada 95 package `Version6.ads` and `Version6.adb`. If you are more familiar with 'C', you could start from the corresponding code in `version6.h` and `version6.c`. Having coded this up, it would be best to write a simple test program like `test_rand.adb` (or `main.c`) in which the new package is called. Some results can be checked by eye. Also a timing comparison is made with any facility built-into the language being used. (The 'C' test program does not attempt to time the random functions in the language due to their poor specification.)

The package specification for the Ada 95 version is the file `version6.ads`. The corresponding implementation is the file `Version6.adb`.

The test program `test_rand.adb` includes several different generators, including the one defined in the Ada standard. Versions 1 and 2 refer to the old generator and 3 and 4 to the 3-cycle one which was found to be inadequate. The one recommended is version 6 due to the speed and absence of 64-bit integer arithmetic on many current machines/languages. It may be the case that Test_rand

Name	*Reference*	*Description*
big6.c	Page 15	C code for Big Crush — seed b (seed a is similar)
big6a.out	Page 15	TestU01 results from Big Crush — seed a
big6b.out	Page 15	TestU01 results from Big Crush — seed b
ctest.out	Page 21	Output from test of C implementation
cycles.c/out	Page 16	Test for single-cycle generators
extra2.c/out	Page 15	New 4-cycle generator: extra test 2
extra4.c/out	Page 15	New 4-cycle generator: extra test 4
extra5.c/out	Page 15	New 4-cycle generator: extra test 5
find.adb/out	Page 14	Program to find primes and result (with factorization of $(p-1)$)
four.c	Page 16	Test for single-cycle generators—main program
LICENCE.PDF	—	Software End-User Licence Agreement
long.pdf	—	This paper
main.c	Page 21	C test program for implementation
mult.adb/out	Page 14	Program to find suitable multipliers and results
random6.pas/out	Page 10	Program to compute correlation between columns
short.pdf	—	Submitted for publication
single.py	Page 17	Python program to compute the single LCG values for some combined generators
smallk.c/out	Page 20	Code to test column generator with Small Crush
test_rand.adb/out	Page 21	Ada test program for completed generator
time4.c/out	Page 21	Code to time several generators being compared
version6.ads/adb	Page 21	New 4-cycle generator, 32-bit version
version6a.c	Page 15	C code new generator for testing — seed a
version6b.c	Page 15	C code new generator for testing — seed b
version6c.c	Page 15	C code column generator for testing
version6.h	Page 21	C header file for implementation
version6.c	Page 21	C implementation
wich1.c/out	Page 14	Code using TestU01 to test the old generator (1)
wich2.c/out	Page 14	Code using TestU01 to test the old generator (2)

Table 1: Main files

Name	*Reference*	*Description*
biga.out	Page 15	New 3-cycle generator: Big Crush output — seed a
bigb.out	Page 15	New 3-cycle generator: Big Crush output — seed b
extra1.c	Page 15	New 3-cycle generator: extra test (which failed)
extra1.out	Page 15	New 3-cycle generator: Failure of extra test
version1.ads/adb	Page 21	ALGORITHM AS 183 APPL. STATIST. (1982) VOL.31, NO.2, 32-bit version
version2.ads/adb	Page 21	ALGORITHM AS 183 APPL. STATIST. (1982) VOL.31, NO.2, 16-bit version
version3.ads/adb	Page 21	New 3-cycle generator, 64-bit version
version4.ads/adb	Page 21	New 3-cycle generator, 32-bit version
version4a.c	Page 15	New 3-cycle generator, 32-bit version for testing — seed a
version4b.c	Page 15	New 3-cycle generator, 32-bit version for testing — seed b
version5.ads/adb	Page 21	New 4-cycle generator, 64-bit version

Table 2: Supplementary files

Version 5 would be faster on machines which properly support 64-bit integer arithmetic (and it is certainly simpler).

For those not familiar with Ada who would like to code this is another language, the following might be useful:

1. `Gen` is a record with the four 32-bit (`ix, iy, iz, it`) integers which are advanced by the procedure.

2. The calculation of `W` in the main logic uses multiplication rather than division, since it is faster and gives essentially identical results. (This change made the Ada program execute about 15% faster.)

3. `Value` is the result of type `Long_Float`, but constrained to be in the range 0.0..1.0. (If the value were outside this range, an exception would be raised.) However, the value 1.0 should not be obtained, but this would not give rise to an exception.

4. Ada allows underscores in literals which is used here for readability. It is very important that the integer literals have the correct values.

5. There are two procedures called `Reset` which is called overloading in Ada. They are distinguished (at compile-time) by the parameters.

6. The final computation of `Value` removes the integer part of `W`. It can be undertaken by means of the text which is commented out. The while loop is actually faster and is probably clearer.

7. The version of `Reset` with two parameters uses the same basic logic as `Next_Sequence`. However, to ensure reasonable speed, it uses 64-bit integer arithmetic. One alternative would be to use a loop for `Initiator` number of times — rather slow for large values.

8. Resetting the generator with the clock can only be undertaken really effectively if the characteristics of the clock are known in detail.

It is very important to *check* that the four cycles perform the same integer calculation as printed by `test_rand.adb` in the file `test_rand.out`.

A key issue with the coding is the use of 32 and 64-bit integer arithmetic. The main generator uses only 32-bit integer arithmetic as specified in the original goal. However, the procedure Reset which logically performs the column generator many times uses 64-bit integer arithmetic as coded in Ada 95 and 'C'. If 64-bit integer arithmetic is not available, then either this form of Reset can be omitted, or the specification can be retained by means of multi-length working. Of course, for small values of Initiator, it is possible to apply the logic in Next_Sequence repeatedly.

If you compare the test result in `test_rand.out` and `testc.out`, you will notice that the rounding of the floating point values differs occasionally — this is to be expected. The integers for the three cycles *must* be the same. The number of values less than 0.5 might be different due to small differences in the computation (say, just 1 or 2 different). On the same machine, one would expect the values to be the same, as here with the Ada 95 and 'C' versions.

If you only implement the basic algorithm rather than all of the Ada 95/'C' package, then you should initialise the seeds to 1, 1, 1 and 1, and check the first two random numbers produced are 0.00005336618663 and 0.84487665211815. This check is essential to show that the constants have been correctly set.

If you produce a new version for another language, please email Brian Wichmann with the details, specifically the result corresponding to `test_rand.out` and any details you have of the built-in generator.

D Document details

1. First complete version (0.91) with related files: 12th April 2005.
2. Version 0.92. some typos corrected and two files added: 16th April 2005.
3. Version 0.93. Set for a Journal, but this makes the left-hand marginal notes appear in the wrong place! Cannot be set with article style due to other changes: April 28th 2005.
4. Version 0.94. Corrected marginal note error and other minor typos. 5th May 2005.
5. Version 0.95. Very minor typos corrected. Ada clock testing code moved and comment added to C test code to ensure they both produce same number less than 0.5. 3rd August 2005.
6. Version 0.96. Updated to reflect comments from two referees (main paper). Additional remarks also added to the appendix. November 29th 2005.