A Comparison of Four Pseudo Random Number Generators Implemented in Ada[*]

William N. Graham
Applied Research Laboratories
The University of Texas at Austin
bill@titan.tsd.arlut.utexas.edu

Abstract

Four random number generators implemented in Ada are compared in terms of their equidistribution, independence, speed and period. Each generator is described and it's code presented. The tests used to evaluate the generators were simple ones. First, a stream of random numbers generated were checked for equidistribution by means of a chi-square test. Next, the numbers were checked for independence by means of a serial test. The code was then timed for a certain number of iterations. Finally, the reported periods of the generators are compared.

1. INTRODUCTION

The objective is to examine pseudo random number generators implemented in Ada and select one that exhibited good characteristics in several categories. A random number generator should most importantly generate numbers which are "close" to being uniformly distributed and where subsequent numbers generated "appear" to be independent from previous numbers generated. The generator should also be as fast and efficient as possible. Finally, the generator should have a sufficiently large period for the application in which it will be used. For background and further information on random number generators and their evaluation, Molloy (Molloy, 89) gives a good introduction to the topic while Fishman (Fishman, 78) and Law and Kelton (Law, Kelton, 82) give a more in-depth treatment of the topic.

All of the generators examined had source documentation. However, not all of the generators had the same quality of documentation available. Some sources presented the generator's underlying theoretical foundation as well as substantial empirical testing results. Other sources discussed theory or testing only briefly, if at all.

The amount of CPU time a generator uses in order to achieve randomness and the effort expended by the tester to test for randomness depends on what level of statistical randomness is required by the intended application. The generators examined were not subjected to a battery of statistical tests nor were they subjected to any theoretical proofs other than those tests and proofs described in the source documentation. The statistical testing done here, excluding the testing described in the documentation, was quite simple and certainly not comprehensive.

The statistical quality of the generators was tested in two ways. The first test applied to the random number generators was the chi-square test. This test involved producing 1,000 random integers between 1 and 100. The differences between the observed and expected frequencies of the integers were then used to compute a chi-square statistic. This test is used to determine how well a generator's output is uniformly distributed.

The serial test was the second statistical test applied to the generators. This test involved producing streams of random integers between 1 and 10. Groups of N consecutive random integers produced in a stream are then grouped

[*] This work was sponsored by the ARL:UT Independent Research and Development Program.

into N element vectors. These vectors are mapped to points in N-dimensional space. The expected frequency of occurrence of each vector is then compared with the actual frequency of occurrence of each vector in order to compute a chi-square statistic. This test is used to determine the independence of a generated random number from the previously generated random numbers in the random stream produced by a generator.

The random number generators were also timed to determine how fast their code executed and thus determine their average speed. Each generator was timed for 1,000,000 iterations.

Finally, the reported periods of the random number generators are given. A random number generator's period is the length of the longest sequence of numbers generated before the sequence begins to repeat again. These sequence periods are not independently proven, but simply taken from the documentation and assumed to be correct.

With both the speed of a random number generator on a particular computer known and that random number generator's sequence period known, it is possible to calculate how much time it would take that generator to iterate through it's period on that particular computer. These time durations are calculated by dividing the sequence period by the observed speed.

These tests were conducted on a Sun 3[†] workstation which contained a 68030 processor, a 25 MHZ clock, and a MC68881 floating point coprocessor. The workstation was running a SunOS 4.0.3 UNIX[‡] operating system. The random number generators were compiled with a VADS 6.0[**] Ada compiler.

2. THE RANDOM NUMBER GENERATORS

Four pseudo random number generators are examined. The first one was developed by B. A. Wichmann and I. D. Hill (Wichmann, Hill, 87) and coded in Ada by Bill Sugden at General Dynamics (Angel, Juozitis, Sabuda, 89). The second one was developed by G. J. Mitchell and D. P. Moore and is described by D. E. Knuth (Knuth, 81) and Bill Allen. The third was developed by G. Marsaglia and implemented in Ada by M. G. Harmon and T. P. Baker (Harmon, Baker, 88). The last generator was developed by P. L'Ecuyer (L'Ecuyer, 88).

2.1 WICHMANN AND HILL

Wichmann and Hill (Wichmann, Hill, 87) briefly discuss the algorithm's theoretical foundation, period and testing results while giving a good description of implementing a multiplicative random number generator while avoiding overflow. In another report Wichmann and Hill (Wichmann, Hill, 82) describe the generator's test results in detail.

Bill Sugden implemented the algorithm in Ada for the TASKIT simulator (Angel, Juozitis, Sabuda, 89). Sugden's random number generator is a part of his Random_Number_Services package which provides several different random distributions. Each distribution is connected to the underlying random number generator via a stream pointer. The generator combines three multiplicative linear congruential generators (MLCGs) in order to produce a generator that has better randomness and longer period than the component MLCGs. An array of 100 well selected initial seeds are also provided. The random number generator stream can be initialized by either one or three of these suggested seeds. One seed is either provided or generated for each of the three component MLCGs. The fiftieth element in the initial seed array or 11207 was used for the testing. This generator is designed to run on any 16 bit or larger machine.

Sugden's Random_Number_Services package was obtained from the Department of Defense STARS (Software Technology for Adaptable, Reliable Systems) Foundations Repository. Relevant parts of this package are included in the appendix.

---

† Sun is a trademark of Sun Microsystems Inc.
‡ UNIX is a trademark of AT&T.
** VADS is a trademark of Verdix Inc.

2.2 MITCHELL AND MOORE

There was good documentation for this generator's algorithm in Donald E. Knuth's "THE ART OF COMPUTER PROGRAMMING, VOLUME 2" (Knuth, 81). Knuth states that this generator is the unpublished work of Mitchell and Moore in 1958. This generator is described as being very fast and producing good test results since 1958. Knuth also briefly describes the theoretical foundation which exists for the algorithm. The period for the generator has been clearly proven, however there is a scarcity of theoretical foundation for the generator's randomness.

This generator is an additive congruential generator (ACG). This generator uses an array of the 55 previously generated random integers to generate a new random integer. The array can be initialized with any set of 55 integer seeds as long as they are not all even. Therefore, in the implementation tested here, an odd integer is placed in the first position of the array and a simple linear congruential generator (LCG) like the one described by Sedgewick (Sedgewick, 83) is used to initialize the rest of the array. For the testing, (50*2)+1 or 101 was used as the initial seed.

In 1990, a Modula-2 implementation of this generator was posted in the comp.lang.modula2 news group by William C. Allen of UCLA. In his posting, Allen describes this generator as working very well. The Ada implementation of this generator was developed by the author and is given in the appendix.

2.3 MARSAGLIA

In Harmon's and Baker's paper (Harmon, Baker, 88), Marsaglia's "Universal" Random Number Generator is briefly described and it's implementation in Ada is presented. This generator is a lagged-Fibonacci sequence generator. Marsaglia designed the generator to be "universal" and therefore intended that it should generate exactly the same sequence of 24-bit real numbers when it is ported to different computers. The generator's theoretical foundation is not discussed by Harmon and Baker. However, they do state that the generator has passed tests for randomness. This generator provides a very long period and can be implemented on any machine with at least 16-bit integer arithmetic and 24-bit floating point arithmetic. For the testing, the four default initial seeds of 12, 34, 56 and 78 were used. Harmon and Baker's Ada implementation is given in the appendix.

2.4 L'ECUYER

This random number generator is well documented in L'Ecuyer's CACM paper (L'Ecuyer, 88). In his paper, L'Ecuyer describes the underlying theoretical foundation of this generator, it's implementation and testing. This generator is based on combining two multiplicative linear congruential generators (MLCGs). A simple MLCG is described first. Then the theory of the combination generator is presented. Both 16-bit and 32-bit Pascal implementations of the combination generator are provided. These Pascal implementations are said to be portable and producing the same sequence on different machines.

The generator was tested extensively by L'Ecuyer. There were 9 different types of statistical tests that were applied to the generator with the same type of test often repeated with different parameters. The statistical tests were the chi-square test, serial test, gap test, poker test, coupon collector's test, permutation test, runs-up test, maximum-of-t test and the collision test. Each of these tests were repeated and the empirical results were compared with the predicted results using a Kolmogorov-Smirnov test. In these tests, the 16-bit combined generator outperformed the simple MLCG while the 32-bit combined generator performed best of all. The results of the 32-bit combined generator were very good. It passed all of the statistical tests.

The Ada implementation of this 32-bit combined generator was developed and tested by the author. For the testing, the initial seeds of 100 and 200 were used. The source code for this implementation is given in the appendix.

3. COMPARISON OF THE GENERATORS

3.1 CHI-SQUARE TEST

This test involved producing sequences of 1,000 random integers between 1 and 100. This is accomplished by generating real numbers greater than 0.5 but less than 100.5 and then rounding the fractional part. If this generator produces uniformly distributed numbers, one would expect to have about 10 occurrences of the integer 1, 10 occurrences of the integer 2 and so on up to 10 occurrences of the integer 100. Therefore, the expected frequency would be 10 for each integer 1 to 100. The actual frequency of each integer produced by the generator is the observed frequency for that integer. The difference between the observed and the expected frequency of each integer can be used to compute the chi-square statistic as follows,

$$\chi^2 = \sum_{i=1}^{R} \frac{(O_i - E_i)^2}{E_i}$$

where R is the number of different random integers possible, O_i is the observed frequency of occurrence for the random integer i and E_i is the expected frequency of occurrence for the random integer i. Because the distribution of integers is expected to be uniform, the expected frequencies of occurrence for each random integer are equal. If N is the total number of observations, then the following equation can be used to compute E_i.

$$\forall i : E_i = N / R$$

The E_i can then be replaced by the constant N/R, producing the following chi-square equation form.

$$\chi^2 = \frac{R}{N}\left(\sum_{i=1}^{R} O_i^2\right) - N$$

This equation provides a more efficient method of computing the chi-square. For this particular test, the chi-square can be more specifically described by the following equation.

$$\chi^2 = \frac{1}{10}\left(\sum_{i=1}^{100} O_i^2\right) - 1000$$

where R=100 and N=1000.

A large chi-square statistic indicates the random numbers are not uniformly distributed. In general, a smaller chi-square is considered preferable to a larger chi-square. The one exception to this is when a chi-square statistic is very close to 0. This case may indicate that the random numbers are artificially too uniformly distributed.

Ten chi-square statistics were produced for each of the four random number generators. Each of the 10 chi-square statistics were computed from different sample sequences of random numbers generated. The same seed was used for all 10 trials. The different sample sequences were generated by first calling the random number generator for a specified number of “warm up” iterations and then begin using the next 1,000 random numbers generated to compute the chi-square. For example, the first 1,000 numbers were used for the first trial. For the second trial, 100 warm up iterations were generated from the beginning of the sequence and the next 1,000 numbers were used to compute the chi-square. For the third trial, 200 warm up iterations were generated from the beginning of the sequence and the next 1,000 numbers were used to compute the chi-square. The samples were taken at 0, 100, 200, 500, 1000, 2000, 3000, 4000, 5000 and 10000 warm up iterations into the sequence of random numbers.

3.1.1 Chi-square Results

CHI-SQUARE for 1000 observations and 100 categories				
# of warm ups	**Wichmann and Hill**	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
0	91.8	$\boxed{135.4}$	107.8	111.6
100	97.2	101.6	111.8	112.2
200	102.4	118.0	95.2	96.4
500	107.4	111.4	91.0	86.4
1000	89.4	100.6	100.8	91.2
2000	104.4	90.2	88.0	91.8
3000	103.0	110.2	119.0	92.4
4000	102.8	96.6	101.0	$\boxed{71.6}$
5000	89.6	110.4	85.2	118.2
10000	104.0	102.2	100.0	87.4
AVG	99.2	107.7	100.0	95.9
90% Confidence interval range = 77 to 123				

All of the generators seemed to produce acceptable chi-square statistics. Most of the chi-squares were within the 90% confidence interval of between 77 to 123 for 99 degrees of freedom. The number of degrees of freedom is simply the number of categories minus one.

The first chi-square value for the Mitchell and Moore algorithm looked a little high. This seemed to confirm the manual examination of the first few hundred in the Mitchell and Moore sequence where there appeared to be some initial clustering of random numbers. Clustering occurs when several consecutive random numbers in the generated stream are very close in value. Nevertheless, the chi-squares for the Mitchell and Moore generator improved after the first hundred or so in the sequence.

Also, the L'Ecuyer generator had one chi-square value below the 90% confidence interval. However, this is statistically acceptable because you would expect one out of 10 to be outside of the 90% confidence interval.

3.2 SERIAL TEST

This test involved producing streams of random integers between 1 and 10. Groups of N consecutive random integers produced are grouped into N element vectors or N-tuples. These vectors are then mapped to points in N-dimensional space with the first integer in the vector taken as the first coordinate and the second integer in the vector taken as the second coordinate and so on up to the N^{th} integer in the vector taken as the N^{th} coordinate. For example, the first random integer through the N^{th} random integer produced by a generator are mapped to a point in N-space, then the $(N+1)^{st}$ through $(2N)^{th}$ random integers produced by the generator are mapped to a point in the same N-space. This is continued for enough trials to sufficiently fill the N-space with vector occurrences. If these vector occurrences are uniformly distributed in the N-space, then an expected frequency can be computed for each point or vector in the N-space. The expected frequency is then compared with the actual frequency of occurrence of each vector in order to compute a chi-square statistic.

The serial test chi-square statistic is a measure of how uniformly the vectors are distributed in N-space. If these vectors are not uniformly distributed, then this is an indication that a generated random number is not independent from the previously generated random numbers in the stream. Therefore, if a large chi-square statistic is produced in the serial test, then the random number generator will be rejected for not exhibiting statistical independence.

For the generators described here, there were three categories of serial tests performed. The first set of tests mapped 1,000 pairs of random numbers into 10^2 or 100 points of two-dimensional space. If the pairs are uniformly distributed in two dimensional space, then it would be expected that there would be close to 10 pairs mapped to each of the 100 points. The differences between the observed and expected frequencies of each possible coordinate pair were then used to compute a chi-square statistic. This statistic will have 99 degrees of freedom and have a 90% confidence interval of approximately 77 to 123.

The second set of tests mapped 10,000 vectors of three random numbers into 10^3 or 1,000 points of three-dimensional space. Therefore, the expected frequency is 10 vector occurrences per point. The resulting chi-square statistic will have 999 degrees of freedom and have a 90% confidence interval of approximately 970 to 1,073.

The third set of tests mapped 100,000 vectors of four random numbers into 10^4 or 10,000 points of four dimensional space. Again, the expected frequency is 10 vector occurrences per point. The resulting chi-square statistic will have 9,999 degrees of freedom and have a 90% confidence interval of approximately 9,768 to 10,232.

In general, the larger the vector or the more dimensions used in the serial test, the more trials or vectors must be produced in order to fill the N-space. For example, in order to have 10 occurrences per point in five-dimensional space, it would require one million five-component vectors to be produced. This would be five million iterations of the random number generator for just one test. Therefore, at some point the serial test becomes less convenient.

3.2.1 Serial Test Results

SERIAL TEST for 1000 vectors mapped to 100 points in 2 dimensions				
# of warm ups	**Wichmann and Hill**	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
0	**114.8**	**114.4**	**110.2**	**90.4**
2000	**92.6**	**79.4**	**123.0**	**107.2**
4000	**96.6**	$\boxed{\textbf{126.8}}$	**90.0**	**106.4**
6000	**111.4**	**102.6**	**78.0**	**102.4**
8000	**114.0**	**105.2**	**90.6**	**98.0**
AVG	**105.9**	**105.7**	**98.4**	**100.9**
90% Confidence interval range = 77 to 123				

For the serial test in two dimensions, the Mitchell and Moore generator had one chi-square value higher than the 90% confidence interval. However, it's other chi-squares were within the interval. All of the chi-squares produced by the other generators were within the 90% confidence interval.

SERIAL TEST for 10,000 vectors mapped to 1000 points in 3 dimensions				
# of warm ups	**Wichmann and Hill**	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
0	**1024.2**	$\boxed{\textbf{964.0}}$	**1031.0**	$\boxed{\textbf{963.4}}$
30,000	**970.4**	$\boxed{\textbf{925.8}}$	$\boxed{\textbf{946.8}}$	**976.4**
60,000	$\boxed{\textbf{950.2}}$	**1037.4**	**989.0**	**970.6**
90,000	$\boxed{\textbf{1079.2}}$	**1019.2**	$\boxed{\textbf{1077.2}}$	**1019.6**
120,000	$\boxed{\textbf{934.8}}$	$\boxed{\textbf{919.2}}$	**977.4**	**1006.2**
AVG	**991.8**	**973.1**	**1004.3**	**987.2**
90% Confidence interval range = 970 to 1073				

For the serial test in three dimensions, the Wichmann and Hill generator had one chi-square higher than the 90% confidence interval and two chi-squares below the 90% confidence interval. The Mitchell and Moore generator had three chi-squares lower than the 90% confidence interval. The Marsaglia generator had one chi-square lower and one chi-square higher than the 90% confidence interval. The L'Ecuyer generator had the most chi-squares in the 90% confidence interval with just one value lower than the 90% confidence interval.

.

SERIAL TEST for 100,000 vectors mapped to 10,000 points in 4 dimensions				
# of warm ups	**Wichmann and Hill**	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
0	**9,977.0**	**10,149.8**	**9,974.2**	**9,850.2**
400,000	**10,030.8**	**9,911.2**	**9,962.8**	**10,079.0**
800,000	**9,981.8**	**9,952.0**	**9,949.6**	**9,883.6**
1,200,000	**9,865.4**	**9,846.8**	**10,126.2**	**9,951.8**
1,600,000	**10,000.2**	**10,112.6**	**10,001.6**	**10,120.0**
AVG	**9,971.0**	**9,994.5**	**10,002.9**	**9,976.9**
90% Confidence interval range = 9,768 to 10,232				

For the serial test in four dimensions, all of the generators produced acceptable chi-square statistics. All of chi-square values were within the 90% confidence interval.

3.3 CODE TIMING

Each random number generator was timed for one million iterations. First, the function CLOCK from the standard Ada CALENDAR package was called to determine the start time. Then the generator was called repeatedly within a FOR loop. After the loop, the function CLOCK was called again to determine the end time. Finally, the start time was subtracted from the end time to determine the number of seconds required for the generator. Once a time length in seconds is determined, it is also possible to compute the time per iteration and the number of iterations per second, if desired.

3.3.1 Timing Results

Time for 1,000,000 Iterations in seconds			
Wichmann and Hill	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
162.039	**28.779**	**75.479**	**52.179**

The absolute values of the timing durations for the generators are not as important as the relative differences of the values. A generator will run at different speeds when it is run on different computers or different operating systems. A generator will even run at different speeds on the same computer and same operating system. However, when all of the compared generators are run on the same system, the relative speeds of the generators can be estimated.

The Mitchell and Moore generator was by far the fastest of the generators. The L'Ecuyer generator was a close second and was 81% slower than the Mitchell and Moore generator. The Marsaglia generator came in third and was 162% slower than Mitchell and Moore. The Wichmann and Hill generator finished last and was 463% slower than Mitchell and Moore. If an application requires many calls to a random number generator, then a slowdown of 463% in a random number generator could significantly impact the application's speed.

3.4 PERIODS

3.4.1 Reported Sequence Periods

Sequence Period			
Wichmann and Hill	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
6.9×10^{12}	**3.6×10^{16}**	**2.2×10^{43}**	**2.3×10^{18}**

The generator's sequence periods are presented with two significant digits and using scientific notation. In the case of the Mitchell and Moore generator, the sequence period is described in the literature (Knuth, 81) as falling somewhere within the range 3.6×10^{16} to 3.8×10^{25} if the modulus is a power of 2. However, in this implementation, the modulus may not be a power of 2. It can only be assumed to be even. For this situation, the lower bound of the period range is given in the table because the actual period is guaranteed to be at least this large.

3.4.2 Calculated Period Durations

A sequence period, by itself, may not convey sufficient information to determine if it is large enough for an application. However, a generator's sequence period and the speed of the generator on a particular computer can be combined to determine how much time it would take that generator to cycle through it's period on that particular computer. The period's time duration can then be compared with the intended application to determine if the generator has a sufficiently large period. Here, the period durations are calculated using the same generator timings that were given above.

Period Duration in years			
Wichmann and Hill	**Mitchell and Moore**	**Marsaglia**	**L'Ecuyer**
35	**32,000**	**5.2×10^{31}**	**3,800,000**

These period durations turned out to be so large for a small computer that the times are given in years. For all practical purposes, these generators have an infinite period for any small computer. The Marsaglia generator has also been reported to have been tested on various mainframes and on an ETA Supercomputer (Harmon, Baker, 88). Therefore, the much larger Marsaglia period should be long enough to be used for supercomputer applications as well.

4. CONCLUSION

All of the pseudo-random number generators passed the chi-square test. In the serial test, all of the generators appeared to do well in two and four dimensions and to do worse in three dimensions. The generators seemed to vary the most in their speed and in their periods. The Wichmann and Hill generator was designed to be used on 16-bit machines and, therefore, had a smaller modulus and thus a smaller period. Another disadvantage of the Wichmann and Hill generator was that it combined three generators and as a result had the slowest speed. The Mitchell and Moore generator had the fastest speed while the Marsaglia generator had the longest period. The L'Ecuyer generator had the second fastest speed combined with a long period. Another advantage of the L'Ecuyer generator was that it was well documented and tested by L'Ecuyer (L'Ecuyer, 88).

5. APPENDIX - SOURCE CODE

5.1 Wichmann and Hill

```
--::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
-- Random Number Services Package
--::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
-- Prepared By: General Dynamics
-- Data Systems Division/Western Center
-- P.O. Box 85808
-- San Diego, CA 92138
-- Phone: 619-573-3800
--
-- Component Name: Random_Number_Services
-- Version: 2.0, Date: 10/10/88
-- Purpose: Generate pseudo-random numbers and sample
-- probability distributions
-- Author: Bill Sugden
--
-- Description:
-- This package provides pseudo random number generators and
-- functions to generate samples from various probability distributions.

with MATH, Text_IO; use MATH, Text_IO;

package Random_Number_Services is
 -- GLOBAL DATA TYPES
type RANDOM_STREAM_RECORD_TYPE is limited private;
type RANDOM_STREAM_ACCESS_TYPE is
access RANDOM_STREAM_RECORD_TYPE;
-- Access type pointing to a random number stream. A pointer of this type
-- is created using the Create_Stream function, and required for subsequent
-- calls the random number generators and/or distribution functions.

type SEED_ARRAY_TYPE is array (1..100) of integer;

SEED_ARRAY : SEED_ARRAY_TYPE :=
( 25789, 1728, 21691, 17139, 21479, 1276, 7005, 20760, 2600, 26767, 26667,
9660, 14174, 17486, 14324, 21041, 212, 5996, 27937, 2207, 5398, 29052,
8023, 17072, 11988, 11081, 5730, 23082, 1419, 14759, 412, 8837, 31727,
14994, 19681, 16516, 17757, 7836, 2292, 19316, 12507, 19600, 28266, 10440,
14402, 14742, 6089, 19588, 14303, 11207, 847, 14516, 27422, 4189, 9499,
21180, 27678, 2746, 16599, 18929, 24264, 8174, 1281, 16958, 28251, 21389,
28122, 29999, 14510, 5878, 11225, 22639, 15593, 10928, 13588, 27968,
31800, 11289, 21233, 17979, 11881, 27286, 31174, 4913, 25408, 27020,
15485, 26008, 20686, 12851, 5248, 23909, 9419, 6251, 7188, 3540, 20792,
10052, 31067, 22161 );
-- SEED_ARRAY:
-- This array holds 100 unique initial seed values. This array is not used
-- by this package but is provided as a convenient source of initial seed
-- values. When creating a random number stream pointer, the user can
-- extract values from this array, or supply his/her own initial values.

function Random
(STREAM_PTR : in RANDOM_STREAM_ACCESS_TYPE) return float is

-- Description:
-- Random number generator, returns a random number, in the range 0 to 1.
-- This random number generator was originally proposed by B. Wichmann
-- and D. Hill, and presented in Byte, March 1987. It generates a random
-- number using three independent generators and combining the results. This
-- generator is portable to any 16 bit or larger machine, and is reported to have
-- a period of 6.9E12. Three initial seed values are required for this random
-- number generator. The user has an option of supplying all three, or supplying
-- one and have the other two automatically generated.
-- Parameters:
-- STREAM_PTR:
-- Indicates which stream the random number is to be generated from.

TEMP : float;

begin
    -- Check stream pointer
    if (STREAM_PTR = null) then
        raise INVALID_PARAMETER_EXCEPTION;
    end if;
```

```
        -- First generator
        STREAM_PTR.SEED_1 := 171 * (STREAM_PTR.SEED_1 mod 177)
            - 2 * (STREAM_PTR.SEED_1 / 177);
        if (STREAM_PTR.SEED_1 < 0) then
            STREAM_PTR.SEED_1 := STREAM_PTR.SEED_1 + 30269;
        end if;
        -- Second generator
        STREAM_PTR.SEED_2 := 172 * (STREAM_PTR.SEED_2 mod 176)
             - 35 * (STREAM_PTR.SEED_2 / 176);
        if (STREAM_PTR.SEED_2 < 0) then
            STREAM_PTR.SEED_2 := STREAM_PTR.SEED_2 + 30307;
        end if;
        -- Third generator
        STREAM_PTR.SEED_3 := 170 * (STREAM_PTR.SEED_3 mod 178)
             - 63 * (STREAM_PTR.SEED_3 / 178);
        if (STREAM_PTR.SEED_3 < 0) then
            STREAM_PTR.SEED_3 := STREAM_PTR.SEED_3 + 30323;
        end if;
        -- Combine seeds; TEMP will probably be some number larger than 1.0;
        -- if so, the part to the left of the decimal is subtracted off, leaving
        -- the result between 0.0 and 1.0
        TEMP := float (STREAM_PTR.SEED_1) / 30269.0 +
            float (STREAM_PTR.SEED_2) / 30307.0 +
            float (STREAM_PTR.SEED_3) / 30323.0;
        -- The algorithm, as written by Wichmann and Hill, call for this as the
        -- next step: return (TEMP - float (integer (TEMP - 0.5)))
        -- which should work assuming the conversion to an integer rounds as the
        -- LRM says it should. However, not all compilers round, we have found a
        -- compiler during our portability testing that truncates when converting
        -- to an integer, thus the following code was added which will produce the
        -- correct results regardless of how the conversion to integers works.
        TEMP := TEMP - float (integer (TEMP - 0.5));
        if (TEMP > 1.0) then return (TEMP - 1.0);
        else return (TEMP); end if;
    exception
        when INVALID_PARAMETER_EXCEPTION =>
            new_line;
            put_line ("Exception raised in function Random");
            put_line ("Invalid parameter; null pointer passed as a parameter");
            raise INVALID_PARAMETER_EXCEPTION;
        when others =>
            new_line;
            put_line ("Exception raised in function Random");
            raise RANDOM_NUMBER_EXCEPTION;
    end Random;
end Random_Number_Services;
```

5.2 Mitchell and Moore

```
-- Coded by: William N. Graham
-- Applied Research Laboratories
-- The University of Texas at Austin
-- Date: 2-27-91

with TEXT_IO; use TEXT_IO;

Package MITCHELL_MOORE_KNUTH_RANDOM is
    subtype SEED_INDEX is INTEGER range 1..100;
    procedure INITIALIZE(SI : in SEED_INDEX);
    -- Initialize array of initial seeds.
    function RANDOM return FLOAT;
    -- Return another random number between 0 and 1.
end MITCHELL_MOORE_KNUTH_RANDOM;

package body MITCHELL_MOORE_KNUTH_RANDOM is

MAX_ARRAY_SIZE:constantPOSITIVE:=55;
-- Number of initial seeds required in an array.
MAX_INDEX:constantNATURAL:=MAX_ARRAY_SIZE-1;
-- Last index in the array.
MODULUS : INTEGER; -- Modulus number, should be large and even.
MODULUS_INV : FLOAT;
-- Modulus inverse = 1 / modulus, this will be a float.
X : array (0..MAX_INDEX) of NATURAL;
-- Array of initial seeds and previous sequence numbers generated.
-- The initial seeds must not all be even.
```

```
FIRST : NATURAL := 0;
-- Array index of first number in previous sequence.
-- This is the index of the oldest number in the sequence.

procedure INITIALIZE( SI : in SEED_INDEX ) is
    M : constant INTEGER := 100000000; -- Modulus used for linear method.
    M1 : constant INTEGER := 10000; -- Square root of M.
    B : constant INTEGER := 31415821;
    -- Multiplier used for linear method. This should be of length 1 digit
    -- less than M and end in the sequence ...821 .

    function MULT( P : INTEGER; Q : INTEGER ) return INTEGER is
    -- Multiply two large integers without overflow.
    -- p*q = (10000*p1 + p0) * (10000*q1 + q0)
    -- p*q = 100000000*p1*q1 + 10000*(p1*q0+p0*q1) + p0*q0
    P1 : INTEGER;
    P0 : INTEGER;
    Q1 : INTEGER;
    Q0 : INTEGER;
    begin
        P1 := P / M1;
        P0 := P mod M1;
        Q1 := Q / M1;
        Q0 := Q mod M1;
        return( (((P0*Q1+P1*Q0) mod M1) * M1 + P0 * Q0) mod M );
    end MULT;

    function LINEAR_RANDOM( N : INTEGER ) return INTEGER is
    -- Linear Congruential random integer.
    -- N := (N*B+1) mod M.
    begin
        return( (MULT(N,B) + 1) mod M );
    end LINEAR_RANDOM;

begin
    MODULUS := INTEGER'LAST / 2;
    -- Half of max integer will ensure against overflow.
    -- Ensure MODULUS is even.
    if (MODULUS mod 2) /= 0 then -- MODULUS Odd.
        MODULUS := MODULUS - 1; -- Make even.
    end if;
    MODULUS_INV := 1.0 / FLOAT(MODULUS);
    -- Compute inverse of modulus.
    -- The initial seeds must not all be even.
    -- Initialize array with seeds by a weak method.
    -- Use the linear congruential method to initialize array.
    -- Set first seed from seed index.
    X(0) := (SI * 2) + 1; -- Ensure first seed is odd.
    for I in 0..MAX_INDEX-1 loop -- Until array filled.
        -- Invoke linear generator to fill out array.
        X(I+1) := LINEAR_RANDOM( X(I) );
    end loop;
end INITIALIZE;

function RANDOM return FLOAT is
    -- Will return a random Float number that is greater than
    -- or equal to 0.0 and less than 1.0 .
    FIRST_PLUS_1 : NATURAL;
    -- The next index of the oldest number in the sequence.
    -- This is the index of the first addend.
    FIRST_PLUS_32 : NATURAL;
    -- This is the index of the second addend.
    RANDOM_INT : INTEGER; -- New integer generated;

begin
    FIRST_PLUS_1 := FIRST + 1;
    -- Compute next index of the oldest number in the sequence.
    -- This is also the index of the first addend.
    -- Check for index out of bounds.
    if FIRST_PLUS_1 > MAX_INDEX then -- Adjust.
        FIRST_PLUS_1 := FIRST_PLUS_1 - MAX_ARRAY_SIZE;
    end if;

    FIRST_PLUS_32 := FIRST + 32;
    -- Compute index of the second addend.
    -- Check for index out of bounds.
    if FIRST_PLUS_32 > MAX_INDEX then -- Adjust.
        FIRST_PLUS_32 := FIRST_PLUS_32 - MAX_ARRAY_SIZE;
        end if;

        RANDOM_INT := X(FIRST_PLUS_1) + X(FIRST_PLUS_32);
        -- Compute new random integer.
        -- Check for integer out of bounds.
        if RANDOM_INT >= MODULUS then -- Adjust.
                RANDOM_INT := RANDOM_INT - MODULUS;
        end if;

        X(FIRST) := RANDOM_INT; -- Assign new integer to the array.

        FIRST := FIRST_PLUS_1;
        -- Set new index of the oldest number in the sequence.

        return( FLOAT(RANDOM_INT) * MODULUS_INV );
        -- Normalize random number and return it.

end RANDOM;

begin
        INITIALIZE(1); -- Initialize array with seeds.
end MITCHELL_MOORE_KNUTH_RANDOM;
```

5.3 Marsaglia

```
---------------------------------------------------------------------------
-- The following is an implementation of a "universal" random number
-- generator algorithm developed by Dr. George Marsaglia of the
-- Supercomputer Computations Research Institute (SCRI) at Florida
-- State University. This generator has a period of ~2**144 and has
-- been tailored for reproducibility in all CPU's with at least
-- 16 bit integer arithmetic and 24 bit floating point. This algorithm
-- does not generate random numbers < 2**-24.
-- This code appeared in the March/April publication of SIGAda's
-- Ada Letters and is considered public domain. PCK
---------------------------------------------------------------------------
```

```
package U_Rand is
M1 : constant := 179 ;
M2 : constant := M1 - 10 ;

subtype SEED_RANGE_1 is integer range 1..M1-1 ;
subtype SEED_RANGE_2 is integer range 1..M2-1 ;

Default_I : constant SEED_RANGE_1 := 12 ;
Default_J : constant SEED_RANGE_1 := 34 ;
Default_K : constant SEED_RANGE_1 := 56 ;
Default_L : constant SEED_RANGE_1 := 78 ;

procedure Start (  New_I : SEED_RANGE_1 := Default_I ;
                   New_J : SEED_RANGE_1 := Default_J ;
                   New_K : SEED_RANGE_1 := Default_K ;
                   New_L : SEED_RANGE_2 := Default_L ) ;

function Next return FLOAT ;
end U_Rand ;

package body U_Rand is
M3 : constant := 97 ;
Init_C : constant := 362436.0/16777216.0 ;
CD : constant := 7654321.0/16777216.0 ;
CM : constant := 16777213.0/16777216.0 ;

subtype RANGE_1 is INTEGER range 0..M1-1 ;
subtype RANGE_2 is INTEGER range 0..M2-1 ;
subtype RANGE_3 is INTEGER range 1..M3 ;

I, J, K : RANGE_1 ;
NI, NJ : INTEGER ;
L : RANGE_2 ;
C : FLOAT ;
U : array(RANGE_3) of FLOAT ;

procedure Start (  New_I : SEED_RANGE_1 := Default_I ;
                   New_J : SEED_RANGE_1 := Default_J ;
                   New_K : SEED_RANGE_1 := Default_K ;
```

```
                        New_L : SEED_RANGE_2 := Default_L ) is
S, T : FLOAT ;
M : RANGE_1 ;
begin
I := New_I ; J := New_J ; K := New_K ; L := New_L ;
NI := RANGE_3'last ;
NJ := (RANGE_3'last/3) + 1 ;
C := Init_C ;

for II in RANGE_3 loop
      S := 0.0 ; T := 0.5 ;
      for JJ in 1..24 loop
            M := (((J * I) mod M1) * K) mod M1 ;
            I := J ; J := K ; K := M ;
            L := (53 * L + 1) mod M2 ;
            if ((L * M) mod 64) >= 32 then
                  S := S + T ;
            end if ;
            T := 0.5 * T ;
      end loop ;
      U(II) := S ;
end loop ;
end Start ;

function Next return FLOAT is
Temp : FLOAT ;
begin
Temp := U(NI) - U(NJ) ;
if Temp < 0.0 then
      Temp := Temp + 1.0 ;
end if ;
U(NI) := Temp ;
NI := NI - 1 ;
if NI = 0 then
      NI := RANGE_3'last ;
end if ;
NJ := NJ - 1 ;
if NJ = 0 then
      NJ := RANGE_3'last ;
end if ;
C := C - CD ;
if C < 0.0 then
      C := C + CM ;
end if ;
Temp := Temp - C ;
if Temp < 0.0 then
      Temp := Temp + 1.0 ;
end if ;
return Temp ;
end Next ;

begin
Start ; -- initialize table U
end U_Rand ;
```

5.4 L'Ecuyer

```
package LECUYER is

-- This package provides a pseudo random number generator that produces
-- FLOAT numbers that are uniformly distributed between 0.0 and 1.0 .
-- The random number generator can also be initialized with two positive
-- integer seeds. The only requirement of these seeds is that they fall
-- in the specified ranges 1..2147483562 and 1..2147483398. This algorithm
-- was developed by Pierre L'ecuyer. The ADA implementation
-- of this algorithm is taken from the code example given in L'ecuyer's
-- EFFICIENT AND PORTABLE COMBINED RANDOM NUMBER
-- GENERATORS paper in the Communications of the ACM, June 1988,
-- pp. 742-774. This code is appropriate for any 32 bit or larger machine.
-- The period for this generator is reported to be ~ 2.3 * 10**18 .
--
-- Coded by: William N. Graham
-- Applied Research Laboratories
-- The University of Texas at Austin
-- Date: 2-11-91
```

```
subtype SEED_RANGE_1 is INTEGER range 1..2147483562;
subtype SEED_RANGE_2 is INTEGER range 1..2147483398;

procedure INITIALIZE (SEED1 : in SEED_RANGE_1; -- First Seed.
                      SEED2 : in SEED_RANGE_2 ); -- Second Seed.
-- Initialize the random number generator with two seeds.

function UNIFORM return FLOAT;
-- Random number generator will return a number between 0.0 and 1.0 .
-- This random number generator combines two Multiplicative
-- Linear Congruential Generators (MLCG's). A MLCG can be
-- characterized as F(S) := (A*S) mod M.

end LECUYER;

package body LECUYER is


-- F(S) := (A*S + C) mod M
-- For a Multiplicative Linear Congruential Generator (MLCG),
-- C will be 0.

M1 : constant POSITIVE := 2147483563; -- First Modulus.
A1 : constant POSITIVE := 40014; -- First Multiplier.
Q1 : constant POSITIVE := 53668; -- First Quotient.
R1 : constant POSITIVE := 12211; -- First Remainder.
M2 : constant POSITIVE := 2147483399; -- Second Modulus.
A2 : constant POSITIVE := 40692; -- Second Multiplier.
Q2 : constant POSITIVE := 52774; -- Second Quotient.
R2 : constant POSITIVE := 3791; -- Second Remainder.
S1 : INTEGER := 1; -- The first current state variable.
S2 : INTEGER := M2 / 2; -- The second current state variable.

M1_MINUS_1 : constant POSITIVE := M1 - 1; -- M1 - 1 .
NORMALIZE : constant FLOAT := 4.656613E-10; -- To normalize result.

procedure INITIALIZE( SEED1 : in SEED_RANGE_1; -- First Seed.
                      SEED2 : in SEED_RANGE_2) is -- Second Seed.
-- Initialize the random number generator with two seeds.

begin

    S1 := SEED1; -- Assign first seed to first state variable.
    S2 := SEED2; -- Assign second seed to second state variable.

end INITIALIZE;

function UNIFORM return FLOAT is
-- Random number generator will return a number between 0.0 and 1.0 .
-- This random number generator combines two Multiplicative
-- Linear Congruential Generators.

    Z : INTEGER; -- Random integer result.
    K : INTEGER; -- Quotient.

begin

    K := S1 / Q1;
    S1 := A1 * (S1 - K * Q1) - K * R1;
    if S1 < 0 then S1 := S1 + M1; end if;

    K := S2 / Q2;
    S2 := A2 * (S2 - K * Q2) - K * R2;
    if S2 < 0 then S2 := S2 + M2; end if;

    Z := S1 - S2;
    if Z < 1 then Z := Z + M1_MINUS_1; end if;

    return( FLOAT(Z) * NORMALIZE );

end UNIFORM;

end LECUYER;
```

Acknowledgments. I would like to thank Dr. Jeffrey Brumfield at The University of Texas at Austin Computer Science Department and Steven Glicker, Lawrence Maturo and Gene Joe at The University of Texas at Austin Applied Research Laboratories for their helpful technical assistance and advice.

References

Ada (1983). “Reference Manual for the Ada Programming Language,” ANSI/MIL-STD-1815A-1983, United States Department of Defense.

Angel, M., P. Juozitis and J. Sabuda (1989). “TASKIT: An Ada Alternative for Activity Oriented Simulations,” Proceedings of the 1989 Summer Computer Simulation Conference (Austin, Texas, July 24-27), The Society for Computer Simulation, 631-635.

Fishman, G.S. (1978). “Principles of Discrete Event Simulation,” John Wiley & Sons, New York, 345-391.

Harmon, M.G. and T.P. Baker (1988). “An Ada Implementation of Marsaglia’s UNIVERSAL Random Number Generator,” ACM SIGAda, Ada Letters, VIII(2), 110-112, March/April.

Knuth, D.E. (1981). “The Art of Computer Programming: Seminumerical Algorithms,” vol 2, 2nd ed., Addison-Wesley, Reading, Mass., 1-177.

Law, A.M. and W.D. Kelton (1982). “Simulation Modeling and Analysis,” McGraw-Hill, New York.

L’Ecuyer, P. (1988). “Efficient and Portable Combined Random Number Generators,” Communications of the ACM, 31(6), 742-774, June.

Molloy, M.K. (1989). “Fundamentals of Performance Modeling,” Macmillan, New York, 84-95.

Sedgewick, R. (1983). “Algorithms,” Addison Wesley, Reading, Mass. 33-42.

Wichmann, B.A. and I.D. Hill (1982). “A Pseudo-Random Number Generator,” National Physical Laboratory Report, DITC, June.

Wichmann, B.A. and I.D. Hill (1987). “Building a Random Number Generator,” Byte, March, 127-128.