Internal Note SUF–PFY/94–02
Stockholm, February 24, 1994

ON UNIFORM RANDOM NUMBER GENERATORS

by
Christian Walck
Particle Physics Group
Department of Physics
University of Stockholm
(e-mail: WALCK@VANA.PHYSTO.SE)

1 Introduction

In an earlier report [1] we described methods to obtain pseudorandom numbers from various statistical distributions as well as more general methods. The underlying generators giving uniformly distributed random number between zero and one was only briefly described. In this report we try to summarize some of the very interesting work in this area which has taken place over the past years. It is only intended as a short summary and is not trying to duplicate work in other excellent reviews in this area [2].

All the methods described are computer algorithms and normally we attach the prefix *pseudo* as in *pseudorandom numbers* to indicate that in reality these numbers are not at all truly random. Indeed, in a philosophical sense it is completely wrong to talk about randomness since the sequence is exactly predictable at each step. However, what is implied is that for a naïve observer who does not know the algorithm the sequence should *appear* to be random. In other words the random number sequence generated should pass all possible rigorous statistical tests[1].

It is well known that poor quality pseudorandom numbers may create systematic errors in Monte Carlo simulations. It is thus not unimportant to use good "state of the art" generators especially since good alternatives exist and are easily coded in high level languages. Versions of such generators has been included into the CERN library and alternative implementations have been made by the author.

1.1 Requirements

A good computer algorithm for a random number generator should meet several important requirements (the following list is basically adopted from reference [3]):

- Randomness: as stated above tests for randomness of a computer algorithm is somehow a strange concept. What one wants to achieve is a sequence which appears random when scrutinized by all kinds of statistical tests.
- Long period: With modern computers of today and the very big Monte Carlo simulations performed it is important to have a very long period before which the sequence exactly repeats. As a guideline one should never exhaust more than a few percent of a random number sequence in order to assure a good behaviour. This is a problem with the classical generators still often being used but as we will see there are nowadays excellent alternatives with extremely long periods for which this is no longer any problem.
- Efficiency: Although the big simulations used in *e.g.* particle physics experiments of today this is not a big problem there are still good reasons to try to be efficient as concerns speed and memory of the algorithm. In smaller applications still a considerable amount of time is spent in actually generating the random numbers.

[1]Nb not too well but in a sound statistical way. Eg a uniform distribution from n random numbers should show statistically reasonable fluctuations and neither be too flat nor significantly deviate from a true uniform distribution.

- Repeatability: Although this may seem to contradict randomness it is a virtue that, using the same initial conditions, the sequence exactly repeats itself. Truly random numbers would make it virtually impossible to *e.g.* debug any simulation program.
- Portability: It is necessary to make the implementation in such a way that the algorithm is easily installed on *any* type of computer including everything from PC's up to and including big main frames. Past generators were often written in assembler language in order to be as fast as possible and sometimes also in order to make use of special tricks. More recent random number generators are easily coded in any high level language such as Fortran, Pascal or C. Such code may often be made in a completely machine independent manner such that portability is trivial.
- Homogeneity: Not only the random number sequence itself but all possible subsets of bits of each number should behave in a completely random way.
- Independent sequences: A useful and nice feature of some of the most recent generators is to make it possible to simultaneously generate independent sequences. This requires the possibility to easily obtain independent non-overlapping subsequences from the basic generator.
- Restarting: If is useful to be able to restart the sequence at a certain point *e.g.* to debug a Monte Carlo simulation which have failed after a long run. For this purpose all variables, classically the *seeds* of the sequence, can be saved at specific points during a run. Beware, however, that care has to be taken when coding to assure that such restarts exactly restores the situation. Any pending random numbers left in a buffer or likewise would ruin this.

2 Multiplicative Congruential Methods

The classical method to obtain uniformly distributed pseudorandom numbers is using the so called *multiplicative linear congruential generators* (sometimes abbreviated MLCG below). These are of the type *linear recurrence relations* where subsequent seeds are obtained by the formula

$$x_{n+1} = \lambda_0 x_n + \lambda_1 x_{n-1} + \cdots + \lambda_j x_{n-j} + \mu \bmod P$$

where neither the coefficients (multipliers) λ_i and P nor the start x-values have any common divisor. A convenient choice for P on a binary computer is a power of 2, *e.g.* 2^{32} on a computer with 32-bit word length.

2.1 Additive method

The so called Fibonacci generators

$$x_{n+1} = x_n + x_{n-1} \bmod P$$

would be a fast and simple method. For $P = 2^\beta$ it has a period of $2^{\beta-1}$ regardless of the initial values for x_0 and x_1 (unless both are zero) but within this there are subperiods which severly limits the method.

2.2 Multiplicative method

The multiplicative congruential method

$$x_{n+1} = \lambda x_n \text{mod } P$$

is very commonly used. For $P = 2^\beta$ and $\beta \geq 3$ the maximum period is $2^{\beta-2}$ which is obtained for λ mod $8 = 3$ or λ mod $8 = 5$ and all odd start values. The sequences generated are permutations of the following values:

Type	λ mod 8	Start value	Values in sequence	
I	3	1,3,9,11 (mod 16)	$8\nu+1, 8\nu+3$	$\nu = 0, 1, \ldots, 2^{\beta-3}-1$
II	3	5,7,13,15 (mod 16)	$8\nu+5, 8\nu+7$	$\nu = 0, 1, \ldots, 2^{\beta-3}-1$
III	5	1 (mod 4)	$4\nu+1$	$\nu = 0, 1, \ldots, 2^{\beta-2}-1$
IV	5	3 (mod 4)	$4\nu+3$	$\nu = 0, 1, \ldots, 2^{\beta-2}-1$

2.3 Mixed method

In the mixed congruential method

$$x_{n+1} = \lambda x_n + \mu \text{ mod } P$$

For $P = 2^\beta$ this generator has the maximum period 2^β when λ mod $4 = 1$ and μ mod $2 = 1$ independently of the start value for the sequence. This generator produces a permutation of the numbers $0, 1, \ldots, 2^\beta - 1$.

2.4 Combination of MLCG's

The simple multiplicative congruential methods was gradually found to be insufficient, not only for subtile quality aspects, but because their period became far to small as the speed of computers increased. It is never recommendable to exhaust more than a few percent of the full period of a random number generator.

An interesting proposal to achieve longer periods by the combination of two multiplicative linear congruential generators ere made by l'Ecuyer in 1988 [4]. In this work several alternatives are presented the one proposed for 32-bit computers combining two multiplicative linear congruential generators. A nice feature is the proposed code which makes the integer arithmetics in a manner which avoids integer overflow[2].

For a multiplicative linear congruential generator with multiplier a and modulus m satisfying $a^2 < m$ each subsequent random number is obtained by

$$s_i = a\ s_{i-1} \text{ mod } m$$

Decomposing $m = aq+r$ with q the integer part of m/a, written as $q = \lfloor m/a \rfloor$, and thereby $r < a$ we may use

$$a\ s \text{ mod } m = \{a(s \text{ mod } q) - \lfloor s/q \rfloor\ r) \text{ mod } m$$

[2]Classically multiplicative linear congruential generators utilizes the fact that many computers either do not care or can be told not to care about integer overflow. In this way *e.g.* using the modulus 2^{32} is implicit on a 32-bit computer.

In this equation every intermediate value will remain within $-m$ to m during the computation. As an example from the article [4] the code for the first generator with $m_1 = 2\,147\,483\,563 = 2^{31} - 85$ and $a_1 = 40014$, giving $q = 53668$ and $r = 12211$, updating the seed `iseed` becomes

```
k = iseed / 53668
iseed = 40014 * ( iseed - k * 53668 ) - k * 12211
if ( iseed .lt. 0 ) iseed = iseed + 214748563
```

This generator is combined with another one having $m_2 = 2\,147\,483\,399$ and $a_2 = 40692$ to obtain the final algorithm. The two MLCG's are combined by a simple subtraction adding 2 147 483 562 if the difference is zero or negative. The period of the combined generator becomes $(m_1 - 1) \cdot (m_2 - 1)/2 \approx 2.31 \cdot 10^{18}$.

3 Shift Register Generators

The shift register generators are closely related to lagged Fibonacci sequences generating individual bits. These bits are collected into a word using a shift register.

One well known generator within the Particle Physics community, RNDM2, is based on a shift register generator with period $4\,292\,868\,097 = 2^{32} - 2^{21} - 2^{11} + 1$ combined with a MLCG with multiplier 69069 modulus 2^{32} having a period of 2^{30}. The overall period is thus $\approx 4.6 \cdot 10^{18}$.

These generators have been heavily criticized by G. Marsaglia. There are, however, quite good generators of this kind on the market. Indeed RNDM2 is not too bad a choice with its quite long period (remember, however, that this is not a pure shift register generator). Anyway, these generators neither seem to compete with the recent lagged Fibonacci and Subtract-with-borrow generators described below nor meet all the requirements given in section 1.1.

4 Lagged Fibonacci sequences

A lagged Fibonacci sequence, denoted $F(r, s, \bullet)$, with lags r and s (with $s < r$) using the operation $\bullet$ starts with r initial elements and obtain successive elements by

$$x_n = x_{n-r} \bullet x_{n-s}$$

As an example the classical Fibonacci sequence for integers is given by $F(2, 1, +)$ *i.e.* $x_n = x_{n-2} + x_{n-1}$ normally starting with the two first positive integers 1 and 2 (*i.e* the sequence is $1, 2, 3, 5, 8, 13, \ldots$).

Some possible lagged Fibonacci sequences are [5]:

- Using 32-bit integers and the operation subtraction or addition. Such generators, $F(r, s, \pm \bmod)$ have a period of $2^{32}(2^r - 1)$ for suitable r and s.
- With 32-bit odd integers and $\bullet$ being multiplication mod m we get $F(r, s, * \bmod m)$ generators with a period of $2^{30}(2^r - 1)$ for suitable choices of r and s. Here we recognize an ordinary multiplicative linear congruential generator as a special case.

- Using k-bit computer words and the exclusive-or operation $\oplus$ we get $F(r, s, \oplus)$ generators with a period of $2^r - 1$ regardless of k but for suitable choices of r and s.

- With floating-point numbers with 24-bit fractions another, very interesting, alternative is obtained. This will be discussed in some detail in the subsection below.

4.1 Universal Generator

The last type of generator mentioned above was proposed by George Marsaglia and Arif Zaman [6] as a "universal generator" in 1987. Their first report was used for different implementations described later in this document and later on their work were published, together with a third author Wai Wan Tsang [3].

The algorithm used is a combination of a lagged Fibonacci sequence and an arithmetic sequence giving a very long period as well as good behaviour when tested for "randomness". For a lagged Fibonacci sequence with lags r and s subsequent values x_n are given by

$$x_n = x_{n-r} \bullet x_{n-s}$$

where the operation $\bullet$ used is defined as

$$x \bullet y = \{\text{if} \quad x > y \quad \text{then} \quad x - y \quad \text{else} \quad x - y + 1\}.$$

The authors choose $r = 97$ and $s = 33$ for which the period of the sequence is $(2^{24}-1)\cdot 2^{96} \approx 1.3 \cdot 10^{36}$. The sequence is very good in most respects when tested for randomness but fails one test made. Because of this, and also since there are strong theoretical support for combining two different generators, the authors proposes the addition of a simple arithmetic sequence modulus $2^{24} - 3$ where

$$y_n = y_{n-1} \circ d$$

The operation $\circ$ is defined by

$$c \circ d = \left\{\text{if} \quad c \geq d \quad \text{then} \quad c - d \quad \text{else} \quad c - d + \tfrac{16\,777\,213}{16\,777\,216}\right\}$$

This sequence has a period of $2^{24} - 3$ and is not very good by itself but when combined with the lagged Fibonacci sequence it provides a sequence that meets all requirements. The combination of the two sequences are made by $x_n \bullet y_n$. Throughout these calculations can be made with exact 24-bit fractions.

5 Add-with-carry and Subtract-with-borrow

5.1 Subtract-with-borrow Generators

A new type of generators, once again proposed by George Marsaglia and coworkers [5], use a technique similar to this of a lagged Fibonacci sequence of type $F(r, s, - \bmod m)$. These generators are called Add-with-carry or Subtract-with-borrow generators and the main difference as compared to the Fibonacci generators is the use of a "carry" or "borrow" bit

(c). In the Subtract-with-borrow case, discussed in this subsection, subsequent values of the sequence are obtained by

$$x_n = x_{n-s} - x_{n-r} - c \bmod m$$

With an initial value *e.g.* $c = 0$ the next carry bit is set to 1 or 0 according to whether the subtraction produced a negative value to which m had to be added.

Generally the maximum period possible for any random number generator would be achieved if all possible seed values were in the sequence. Using r seeds each with k bits one could theoretically obtain a period of 2^{kr}. As an example the authors give a lagged Fibonacci generator $F(5, 2, - \bmod 10)$ which has a maximum period of 6448. Adding the carry bit the new sequence has a period of 99900 *i.e.* virtually all of the 10^5 possible seed values are included.

In fact the Subtract-with-borrow generator $x_n = x_{n-s} - x_{n-r} - c \bmod b$ has a period of $b^r - b^s$ if $b^r - b^s + 1$ is a prime and has b as a primitive root. Using 32-bit integers $b = 2^{32}$ is a naturally choice on binary computers. However, the condition cannot be fulfilled for this b-value. A close by value $b = 2^{32} - 5$ meets the requirements for lags $r = 43$ and $s = 22$. That this b-value itself happen to be a prime is, however, not important. The generator thus becomes

$$x_n = x_{n-22} - x_{n-43} - c \bmod b \quad \text{with} \quad b = 2^{32} - 5$$

using as the 43 start values 32-bit integers in the range 0 to $2^{32} - 6$ and an initial carry bit (0 or 1). The period of this generator is thus $b^{43} - b^{22} \approx 2^{1376} \approx 10^{414}$.

The authors is not content with this generator alone but combines it with another generator. The same was done for the lagged Fibonacci generator described earlier [3]. The argument is the same namely the theoretical support that combining two generators will produce better results, or at least not worse, than either of the component generators.

The second generator used is a so called Weyl generator

$$x_n = x_{n-1} - k \bmod m$$

H. Weyl used an irrational constant for k but here an integer relatively prime to m is used. The final choice is $m = 2^{32}$ and $k = 362\,436\,069$. The period of this generator is 2^{32} and the two generators are combined by subtraction mod 2^{32}. The final period of the combined sequence thus becomes the quite enormous number

$$2^{32}(2^{32 \cdot 43} - 2^{32 \cdot 22}) \approx 7.1 \cdot 10^{423}$$

.

5.2 Add-with-carry Generators

Very similar to the case of subtract-with-borrow generators add-with-carry generators evaluate subsequent values of the sequence by

$$x_n = x_{n-s} + x_{n-r} + c \bmod m$$

where the carry bit is put to 1 or zero according to whether the addition produced a value greater than m or not.

5.3 CERN implementation

In the review article F. James suggests a simple Subtract-with-borrow generator using $b = 2^{24}$, $r = 24$, and $s = 10$ to produce 24-bit fractions for 32-bit floating point numbers [2]. The proposed generator has a period of $(2^{576} - 2^{240})/48 \approx 5.2 \cdot 10^{171}$. If follows closely the recipes proposed by G. Marsaglia *et al* but does not combine with a second generator.

A generator of similar quality to this could have been achieved using a add-with-carry generator. In this case some care has to be taken in order to avoid possible rounding errors [7] if working with 24-bit floating point fractions. M. Lüscher has suggested the following procedure to update the seed in steps as

$$\begin{aligned}
x_n &= x_{n-r} - 1.0 \\
x_n &= x_n + x_{n-s} + c \\
x_n &= x_n + 1.0 \quad \text{and} \quad c = 0 \quad \text{if} \quad x_n < 0.0 \\
c &= 2^{-24} \quad \text{otherwise}
\end{aligned}$$

For versions programmed with 24-bit integers this is no problem, however, and the code may be simplified again.

5.4 Luxury generators

Further theoretical details on the generator proposed by F. James [2] may be found in a subsequent work by Martin Lüscher [8]. The latter proposes improved versions of the generator skipping $p - r$ random numbers for every r accepted. The integer $p \geq r$ is a fixed parameter which in the original case were equal to r *i.e* no random numbers were skipped. For high-quality random numbers passing all rigorous tests Lüscher proposes $p = 223$ or even $p = 389$.

The CERN implementation, soon to be released, is called RANLUX and is CERN library entry V115. In the description of the RANLUX routine [9] (see also below) it is claimed, without going into details on the tests being referred to, that

- $p = 24$ fails many tests,
- $p = 48$ passes the gap test but still fails spectral test,
- $p = 97$ passes all tests but theoretically still defective,
- $p = 223$ any theoretical possible correlation have very small chance of being observed (default used), and
- $p = 389$ highest "luxury" all 24 bits chaotic.

One may think that throwing away up to 365 (for $p = 389$) random numbers for each 24 used would lower the period of the generator by a corresponding factor. However, Lüscher has shown that if the total number generated and skipped per cycle is prime the period is maintained. The period is anyway large enough that this is hardly any problem.

6 Implementations

In this section we mention a few pseudorandom number generators mainly from the CERN library or of our own design. Others exist too and many computer manufacturers supply random number routines as part of the system library. Beware, however, that these are not always of the best quality.

6.1 Multiplicative Congruential Generators

6.1.1 RNDM

The classical implementation of a multiplicative congruential generator used for decades in the field of particle physics is the CERN library routine RNDM (entry V104). It is still used by many application programs although its period and quality is very poor for most modern simulation problems. This routine uses a multiplier of 69069 (although for some reason it used to be 690069 for many years) and a default start value of 12345. The period is only 2^{30} and is easily exhausted in a limited time on modern computers.

6.1.2 NRAN

Another old CERN library is NRAN (entry V105) returning a vector of uniform random numbers between zero and one. It uses exactly the same MLCG as RNDM but the sequence is independent of the latter.

6.1.3 RN32

The CERN library routine RN32 (entry V106) is created in order to give the same sequence on different computers. The generator uses as the two previous routines the multiplier 69069 and by default the start seed 65539. After each multiplication it uses the 31 rightmost bits (which becomes the new seed), masks off the 8 last bit, and multiplies the resulting number by 2^{-31}.

We want to warn, however, that such machine independence is often uninteresting inasmuch as the different floating point representation of different systems normally implies that the tested program very fast get out of sequence if *e.g.* reject-accept techniques are being used.

6.1.4 RANECU

This routine is the CERN library implementation of the technique proposed by Pierre L'Ecuyer [4] (CERN library entry V114). This routine is implemented in the GEANT simulation package but we find it less elegant than *eg* the universal generator (eg the CERN library routine RANMAR or our version RNUNIF). One technical disadvantage is to achieve independent subsequences which is solved in the CERN implementation by preparing a number of start points accessible through the initialization routine.

6.2 Shift Register Generators

6.2.1 RNDM2

Another well known generator in the CERN library was RNDM2 (entry V107). Using a combination of a multiplicative congruent generator modulus 2^{32} with multiplier 69069 and a period of 2^{30} and a shift register generator with period 4 292 868 097 it has a total period of about $4.6 \cdot 10^{18}$. It is still a decent generator for many purposes but is more difficult to program in a machine independent way.

6.3 Lagged Fibonacci Generators

6.3.1 RANMAR

One of the best and most flexible generators on the market is still the one proposed by George Marsaglia, Arif Zaman and Wai Wan Tsang [3]. It is of the type $F(97, 33, - \bmod 1)$ and is briefly described above. This routine is implemented as CERN library entry V113.

6.3.2 RNUNIF

This is our implementation of the Fibonacci random number generator by G. Marsaglia *et al.* It is described in the internal report [10]. The package includes some useful utilities for the handling of random number sequences.

A special version of this generator was first implemented into the simulation program for the DELPHI experiment at CERN by the author and Nick van Eijndhoven in 1989 even before RANMAR existed. The author also made a special library for this routine including some useful utility routines to be used in conjunction with the old random number library created during many years and described in reference [1]. A test program to check this routines on different computers accidentally also became a humble benchmark tests. The routines along with the test results, the latter still being updated from time to time, are described in reference [10].

6.4 Subtract-with-borrow Generators

6.4.1 RCARRY

The original implementation of the technique proposed by G. Marsaglia *et al* as presented by F. James [2]. This is a predecessor to RANLUX in which further options are built in.

6.4.2 RANLUX

This is the most recent CERN library routine including many options to obtain either the generator originally proposed by F. James [2] or using "luxury" skipping of random numbers a la M. Lüscher [8]. Five so called luxury levels ranging from 0 to 4 are supported corresponding to $p = 24$ (no numbers skipped), $p = 48$, $p = 97$, $p = 223$, and $p = 389$, respectively. The default value is luxury level 3 using $p = 223$. The routine also allows the use of any value for p between 24 and 2000 to be used. This routine is soon to be released as CERN library entry V115.

6.4.3 RNSWB

This is our implementation of the Subtract-with-borrow random number generator originating from G. Marsaglia *et al.* It is done in the same spirit as RANLUX but is quite different in some respects. The implementation also has big similarities with the code for the lagged Fibonacci generator RNUNIF.

This is our basic generator but we also supply a subroutine which fills a vector with many random numbers. When applicable this may considerably speed up the time taken per random number generated.

6.4.4 RNAWC

This is our implementation of a Add-with-carry random number generator made in a very similar way as RNSWB above. It includes some tricks due to M. Lüscher to avoid round off errors as described in reference [7]. This generator should have similar qualities to RNSWB but we prefer the latter in order to follow the recipes of the great gurus in the field.

As for RNSWB we also supply a subroutine RNAWCV which fills a vector with many random numbers.

7 Summary

The latest development regarding random number generation of uniform pseudorandom numbers are presented. Many excellent new options exist. The most luxury is Subtract-with-borrow generators with skipping of random numbers (as RANLUX) but the lagged Fibonacci generator (as RANMAR or the implementation of the author RNUNIF) is still a very good choice. For a description of the most recent development of the CERN library versions see the article by F. James [9].

The new methods are easily coded in a high level language like FORTRAN and any user may make his own routines to his or her taste. For convenience we have made special packages for the two most interesting methods, the lagged Fibonacci generator (RNUNIF) [10] and the Subtract-with-borrow generator (RNSWB) [11], both of which are proposed by G. Marsaglia and coworkers [3, 5]. There are several reasons for doing this although there are working versions within the CERN-library (RANMAR and RANLUX):

- Although there are some arguments against having a function with a dummy argument as in the good old RNDM-days we do find it much more convenient and efficient to supply such a function also for the new generators. We supply, however, also a subroutine filling a vector useful in some connections. In most application it is far from trivial to make efficient use of such a subroutine. It easily creates bookkeeping problems when restarting a sequences and most algorithms is most easily programmed using one random number at a time. *E.g.* in the common reject-accept (or hit-miss) techniques one does not know in advance how many random numbers will be consumed. When applicable, however, making many random numbers in one subroutine call is faster.

- The CERN implementations adds an overhead using counters split into two computer words. This is intended for restarting a sequence by looping all random numbers used to a certain point. This unnecessarily slows the routines down and such restarts are not efficient. We propose to accept the presence of many seeds all of which have to be saved for a fast restart (indeed the CERN implementation RANLUX includes such an options too).

- We also allow ourselves not to have a lot of safety code in the basic generator. In initialization routines such checks are present but, although less important today than before, we try to keep the generator simple and fast.

- We do not use Fortran entry-points but rather build up a small subroutine package. Which is more elegant may be discussed but at least previously the syntax for entry-points were not machine independent. This also implies that the often called basic function may be small and efficient. The drawback is the introduction of a common-block.

- We also give, as an option, routines working internally with integer arithmetics. This is possible on all 32-bit computers and produces identical sequences as the floating point versions (which are used in the CERN implementations of RANMAR and RANLUX). On many computers this is much more efficient than the floating point versions. On recent computers, however, the floating point version is often faster.

References

[1] Ch. Walck. *Random Number Generation.* USIP Report 87-15, University of Stockholm, December 1987.

[2] F. James. *A Review of Pseudorandom Number Generators.* Computer Physics Communications **60** (1990) 329–344.

[3] George Marsaglia, Arif Zaman and Wai Wan Tsang. *Toward a Universal Random Number Generator.* Statistics & Probability Letters **9** (1990) 35–39.

[4] Pierre L'Ecuyer. *Efficient and Portable Combined Random Number Generators.* Comm. ACM **31** (1988) 742.

[5] George Marsaglia, B. Narasimhan and Arif Zaman. *A Random Number Generator for PC's.* Computer Physics Communications **60** (1990) 345–349.

[6] George Marsaglia and Arif Zaman. *Toward a Universal Random Number Generator.* FSU-SCRI-87-50, Supercomputer Computations Research Institute and Department of Statistics, The Florida State University, Tallahassee, Florida, US, September 18, 1987.

[7] I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-Nissila. *Influence of Implementation on the Properties of Pseudorandom Number Generators with a Carry Bit.* Helsinki University preprint HU-TFT-93-33.

[8] Martin Lüscher. *A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations.* DESY 93–133, September 1993; Computer Physics Communications **79** (1994) 100–110.

[9] Fred James. *Latest Development in Random Numbers.* CERN Computer Newsletter 213 (1993) 8–10.

[10] Ch. Walck. *Implementation of a New Uniform Random Number Generator (including benchmark tests).* Internal Note SUF–PFY/89–01, Particle Physics Group, Fysikum, Stockholm University, December 21, 1989.

[11] Ch. Walck. *Implementation of Yet Another New Uniform Random Number Generator.* Internal Note SUF–PFY/94–01, Particle Physics Group, Fysikum, Stockholm University, February 17, 1994.