Internal Note SUF–PFY/94–01
Stockholm, February 17, 1994

IMPLEMENTATION OF YET ANOTHER UNIFORM RANDOM NUMBER GENERATOR

by

Christian Walck

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

Contents

1 Introduction

For some decades the pseudorandom number generators used in the Particle Physics community were simple mulitplicative congruential generators, the most famuous being **RNDM**. With faster computers and larger simulation programs these generators were inadequate already many years ago. The best replacement during the last years has been the universal generator proposed by G. Marsaglia and coworkers [1, 2] which were implemented within the DELPHI simulation program in 1989 and at the same time we developed a small package for this method [3]. Later on a CERN-library version called **RANMAR** were implemented (entry V113).

We still find this algorithm a very useful and good alternative in most cases. However, a competitive technique has been developed during the last years. This is based on the same type of lagged Fibonacci sequences as is used in the universal generator and was originally proposed by the same authors [4]. An implementations of this method was proposed by F. James [5] and this version was further developed by M. Lüscher [6]. The final CERN-library routine were named **RANLUX** and is just in the process of being released this year as entry V115 (we got hold of a prerelease).

Once again we have developed a small subroutine package of this method according to our own taste. It is made in a similar fashion as the implementation for the universal generator and may replace this if desired. Both generators should behave very well and it is not likely that one would find problems arising from any of them. If, however, one has even the slightest suspection that this may be the case it might be wise to try both.

2 Algorithm

For details we refer to the report by G. Marsaglia, B. Narasimhan, and A. Zaman [4] and the subsequent articles by F. James [5] and M. Lüscher [6]. However, we make a short summary of the algorithm below.

2.1 Subtract-with-borrow Generators

A new type of generators, once again proposed by George Marsaglia and coworkers [4], 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 [2]. The argument is the same namely the theoretical support for the 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 enormous number

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

2.2 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.

2.3 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 [5]. The proposed generator has a period of $(2^{576} - 2^{240})/48 \approx 5.2 \cdot 10^{171}$. If follows closely the recepies 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

$$\begin{aligned}
x_n &= x_{n-r} - 1 \\
x_n &= x_n + x_{n-s} + c \\
x_n &= x_n + 1 \quad \text{and} \quad c = 0 \quad \text{if} \quad x_n < 0.0 \\
c &= 2^{-24} \quad \text{otherwise}
\end{aligned}$$

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

2.4 Luxury generators

Further theoretical details on the generator proposed by F. James [5] may be found in a subsequent work by Martin Lüscher [6]. 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 $p = 389$.

In the description of the RANLUX routine [8] (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 (this is the 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 a problem.

3 Range

As for the universal generator [3] the generator returns uniform pseudorandom numbers between zero and one with the end-points excluded. The floating point numbers returned have a decimal fraction of 24 bits and thus the actual values range from $2^{-24} \approx 6.0 \cdot 10^{-8}$ to $1 - 2^{-24}$ in steps of 2^{-24}. In order to be able to take the logarithm of a random number, *e.g.* when generating random numbers from an exponential distribution, we avoid including the zero in the sequence.

In the CERN implementation RANLUX small numbers with less than 12 significant bis, *i.e.* $< 2^{-12}$, are padded such that 2^{-24} times of the current seeds are added. On top of this exact zeros are put to 2^{-48}. In our implementation, as was done for the universal generator, we have chosen not to do this but to keep the 2^{-24} steps throughout the full range.

4 Period

Our implementation is very similar to the CERN implementation `RANLUX` and thus the period is the same *i.e.*

$$(2^{576} - 2^{240})/48 \approx 5.2 \cdot 10^{171}$$

The luxury skipping of random numbers may affect this, but not necessarily so as was mentioned above. Anyway this is not a real problem due to the enormously long period of the generator.

5 Initialization

The lags r and s ....As was proposed in the original article [4] we use the same algorithm as was used for the universal generator. To initialize the 24 seeds for the sequence two simple random number generators with altogether 4 seeds are used.

- a 3-lag Fibonacci generator (3 seeds)
- a multiplicative congruential generator (1 seed)

The three seeds for the 3-lag Fibonacci sequence should be in the range 1 to 178. However, combinations where the three seeds are 1's or 178's (8 combinations) should be avoided. With such start values the generator has a period of 1, 2 or 4. Otherwise the generator have a period of 32044 (in roughly 50 % of all cases) and 16022 or 8011 (each in about 25 % of all cases). The congruential generator has a period of 169 whatever start value, in the range 0 to 168, is used. In order to avoid the unwanted start values we have restricted the values to the ranges 2 to 177 for the 3-lag Fibonacci sequence seeds. This leaves us with $169 \cdot 176^3 = 921350144$ unique possible start combinations.

6 Packing of Seeds

The four initial seeds described above can be fitted into one 32 bit integer. In order to have a one-to-one mapping between the seeds and the integer not leaving any holes we construct:

```
      ISEED = (I-2)*176*176*169 + (J-2)*176*169 + (K-2)*169 + L
   or ISEED = 169 * ( 176 * ( 176*(I-2) + (J-2) ) + (K-2) ) + L
```

In this way `I`, `J` and `K` values from 2 to 177 and `L` values from 0 to 168 maps into `ISEED` values between 0 and 921350143 and vice versa. `ISEED` can be looked upon as a four "digit" number where the base for each digit is varying (176, 176, 176 and 169). If needed we have supplied two routines for packing and unpacking of the four seeds, respectively.

7 Sequences

One of the very nice features of the generator is that the about 920 million different start combinations in the four initial seeds produce *independent* subsequences each with a long period (average length about 10^{162}). This gives many possibilities to allocate unique start values to avoid interference between runs.

For DELPHI simulations we have introduced an option where a laboratory code is used together with the current time in seconds from the beginning of the year to give an integer with the four initial seeds packed. Even in this way one has to be careful when submitting many jobs in parallel at the same time. The danger to submit two identical jobs exactly one year apart is most probably much less.

For the package made for the universal generator we supplied a routine which calculates the number of seconds since January 1st 1989 which may be used as start value for the sequence. This routine will give identical start values approximately 29 years apart. *E.g.* `ISEED=0` was set at 00:00:00 on January 1, 1989 and this will also be the result at 18:35:44 on March 14, 2018 and at 13:11:28 on May 25, 2047 etc. This routine can be used without changes also for this implementation.

8 Restart

The state at each instant of this random number generator is described by altogether 29 variables. These are the 24 seeds, two lags, and three flags (the five last could be packed into one computer word). Earlier pseudorandom number generators such as `RNDM` and `RNDM2` have one and two seeds, respectively, whereas the universal generator needed 100 variables to uniquely describe its state. In the CERN implementations one has chosen to add a counter in order to reduce the number of constants needed for restart purposes. With a counter split into two words (to avoid integer overflow) and the four initial seeds (packed) one may then restart by skipping through all the used random numbers.

In the DELPHI simulation program, where the universal generator is used, the same technique was adopted at first. However, this was soon found to be a big overhead and in order to avoid this problem an option was added to save all 100 constants on a restart file. This is also the suggested method in this package. We supply two routines for this but, of course, any user may include the common `/RNCMMN/` and do whatever is needed with them. In some applications one may *e.g.* want to keep several sequences in parallel in which case a store for 100 constants for each sequence can be maintained in the program. As an example DELPHI has a separate sequence for generation and simulation in order to obtain the same events when changing the conditions in the simulation part.

9 Routines

The basic routines written is one initialization routine `RNINIT` and two generators `RNSWB` and `RNSWBV`. The function `RNSWB` gives one random number at a time while the subroutine `RNSWBV` fills a vector with random numbers. We have also written some utilities to pack/unpack seeds, write/read seeds to file and save/recall seeds in memory. In addition one routine using the current time to get a start value for the generator is supplied. Some routines are common to the implementation of the universal generator [3]. The routines supplied in this package are (those common to the package for the universal generator is marked with an asterisk):

- RNSWB: the basic generator. `R=RNSWB(DUM)` returns a pseudorandom number in the range zero to one (end-points excluded). Beware of the risks with the dummy argument in connection to compiler optimization. Always use different arguments within an expression as well as an index-dependent argument in a loop environment.
- RNSWBV: `CALL RNSWBV ( RN, N )` fills the vector `RN` with `N` uniform pseudorandom numbers.
- RNAWC: an add-with-carry version. `R = RNAWC(DUM)` returns a pseudorandom number in the range zero to one (end-points excluded).
- RNAWCV: `CALL RNAWCV ( RN, N )` fills the vector `RN` with `N` uniform pseudorandom numbers.
- RNISWB: `CALL RNISWB ( ISEED )` initializes the sequence. `ISEED` is a packed word with the initial four seeds.
- RNJSWB: `CALL RNJSWB ( ISEED )` initializes the sequence using the same algorithm as is used in `RANLUX`. Used for comparative studies between the two implementations.
- RNWRTE: `CALL RNWRTE ( LUN, LBL1, LBL2 )` saves all 100 constants on the file assigned to unit `LUN` giving the entry the two labels `LBL1` and `LBL2`.
- RNRDIN: `CALL RNRDIN ( LUN, LBL1, LBL2 )` reads all 100 constants from the file assigned to unit `LUN` for the entry with labels `LBL1` and `LBL2`.
- RNCLCK*: `CALL RNCLCK ( ISEED )` calculates the number of seconds since January 1, 1989 to be used as start value for the packed seed word.
- RNDATE*: `CALL RNDATE ( JY, JM, JD, IH, IM, IS )` gives the current system time and is used by `RNCLCK` for the clock start.
- RNSVE: `CALL RNSVE ( ISEQ )` saves the seeds for sequence number `ISEQ`.
- RNRCA: `CALL RNRCA ( ISEQ )` recalls the seeds for sequence number `ISEQ`.
- RNPACK*: `CALL RNPACK ( I, J, K, L, ISEED )` pack the four initial seeds `I, J, K` and `L` into one integer word in `ISEED`. This routine is common to the package for the universal generator.
- RNUNPK*: `CALL RNUNPK ( ISEED, I, J, K, L )` unpacks the four initial seeds `I, J, K` and `L` from the packed integer `ISEED`.

The routines are listed in Appendix A.

10 Options

10.1 Integer and floating point versions

We have added an option RNINT[1] which, if selected, gives a version of the package where integer arithmetics is used internally until the very end of the routine. The default version uses floating point arithmetics throughout. The result, however, is identical whether or not this option is selected since even in the floating point version all seeds are exact 24 bits fractions. In the integer version this fact is used and the final number is multiplied by 2^{-24} only at the end.

10.2 Clock start for different computers

All routines but RNDATE are written in machine independent standard FORTRAN 77. In RNDATE we need the system date and time which necessarily becomes machine dependent. For this purpose this routine includes options for many computers ALT (Alliant), APOLLO, CDC (Control Data Corporation), CONVEX, CRAY, DECS (DECstation), DGE (Data General), GOULD, HP (Hewlett Packard), IBM, IRS (IBM RISC Systems), MPW (MacIntosh), NORD (Norsk Data), SUN, and VAX (from Digital Equipment Corporation). For Cray an additional tag US selects american standard for date, with the order of date and month reversed, if needed. For computers where we have no code with system calls, or if it is desired for any other reason, an option DATIMH selects a version of the routine using this routine. Within the Particle Physics community this may be recommended in order to put machine dependence outside this package and thus being less vulnerable for future changes. However, unfortunately we still have to take care of the difference in DATIMH calls on computers having 32 bit or 64 bit word length and therefore some machine dependence always remain. Besides DATIMH is written in an old-fashioned way packing the system time into Hollerith constants. This is a unnecessary detour in achieving these quantities. If the routines for clock start of the generator poses any problem on some installation this option should be disregarded or if possible an alternative RNDATE routine returning the system date and time written.

11 Timing

Although the speed of the random number generator normally is not a big problem in High Energy Physics simulations it is customary to try to achieve as fast as possible random number generators. Indeed, a considerable time is still spent in generating random numbers especially in small Monte Carlo calculations.

The tests were primarily done in order to check that the routines work properly on different computers, especially since we had to enter some machine dependence in the clock start utility routine. Our previous implementation of the universal generator [3] became "‘by accident" a small humble benchmark test. We do not intend to make the same effort this time but rather to test a few computers and make comparisons to the CERN implementation RANLUX.

Of course, the larger value of p that is used the slower the routine will be. Below we show some timing results obtained on a VAX 4000-500 (node VANA) and a DEC 3000 model 500 (node VANZ). We have chosen the p-values coresponding to the five different luxury levels supported in RANLUX and in each case we give numbers for a call requesting one random number (using the function in case of RNSWB) as well as the asymptotic value requesting many random numbers (using RANLUX and RNSWBV). For our implementation we also give the values for the default floating-point (f) as well as the optional integer version (i).

[1]The program is maintained in the framework of the source code management program PATCHY common within the Particle Physics community. In this system options may be entered with so called self-tags selecting different program versions.

Node	routine	$p = 24$		$p = 48$		$p = 97$		$p = 223$		$p = 389$	
VANA	RANLUX	3.72	1.19	4.49	1.96	6.08	3.46	10.2	7.43	15.2	12.5
	RNSWB f	1.86	1.11	2.62	1.92	4.16	3.48	8.19	7.58	13.6	12.9
	RNSWB i	1.85	1.08	2.64	1.81	4.11	3.30	7.91	7.15	13.0	12.2
VANZ	RANLUX	1.56	0.71	2.06	1.21	3.09	2.21	5.68	4.80	9.10	8.21
	RNSWB f	0.71	0.60	1.20	1.15	2.21	2.26	4.83	5.12	8.30	8.79
	RNSWB i	1.00	0.64	1.48	1.12	2.42	2.10	4.84	4.60	8.06	7.88

Our function `RNSWB` is thus faster than `RANLUX` and sometimes the integer version is faster than the floating point version for our routines. This may be compared to the universal generator `RNUNIF` which takes 1.62 (0.47) μs per call for the function call and 1.01 (0.39) μs per random number asymptotically for the subroutine on VANA (VANZ) using the floating-point version. For the DEC 3000 model 500 the new algorithm is thus a factor of ten slower than the universal generator using the default value 223 for p and even without any skipping of numbers from the sequence, *i.e.* $p = 24$, it is considerably slower.

12 Summary

An implementation of the random number generator originally proposed by G. Marsaglia, B. Narasimhan and A. Zaman [4] as proposed by F. James and M. Lüscher [5, 6] has been presented in the form of a FORTRAN subroutine package.

This generator has excellent properties when tested for "randomness". It has an extremely long period of approximately $5.2 \cdot 10^{171}$ and nice features for starting at different points in the sequence. It is along with the universal generator [1, 2] on of the best uniform pseudorandom number generator existing on the market for the moment. The algorithm is easily coded in high level languages for most if not all existing computers. A minor drawback is that the number of seeds needed in order to restart at a certain point in the sequence are about 30 (24 seeds, 2 lags, the carry bit, p, n).

The package includes, besides routines for generating one random number at a time or a vector filled with random numbers some extra utility routines. The latter include options for starting or restarting the generator as well as maintaining several independent sequences in parallel.

The package is written in standard FORTRAN and is intended to easily be installed on any computer. The sequence of random numbers generated is identical on all computers.

References

[1] 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.

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

[3] Ch. Walck. *Implementation of a New Uniform Random Number Generator (including benchmark tests).* Unofficial report SUF–PFY/89–01, Stockholm, December 21, 1989.

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

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

[6] Martin Lüscher. *A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations.* DESY 93–133, Spetember 1993; to be published in Computer Physics Communications.

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