Computer Physics Communications 79 (1994) 111–114
North-Holland

Computer Physics Communications

RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher

F. James

CERN, CH-1211 Geneva 23, Switzerland

Received 10 October 1993

Following some remarks on the quality of pseudorandom number generators commonly used in Monte Carlo calculations in computational physics, we offer a portable Fortran 77 implementation of a high-quality generator called RANLUX (for LUXury RANdom numbers), using the algorithm of Martin Lüscher described in an accompanying article. The implementation allows the user to select different quality or luxury levels, where higher quality requires somewhat longer computing time for the generation. There is a convenient way of initialization (appropriate also for massively parallel Monte Carlo computations) as well as two different methods of restarting from a break point.

PROGRAM SUMMARY

Title of program: RANLUX

Catalogue number: ACPR

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Licensing provisions: none

Computer and operating system: Any system with a standard Fortran 77 compiler Has been tested on Apollo Unix, IBM VM/CMS, Sun UNIX and DEC Vax systems.

Programming language used: Fortran 77 with no extensions

Memory required to execute with typical data: 4342 words

Number of bits in a word: 32 or more

Has the code been vectorised?: Not this implementation, but see ref. [1]. This implementation is immediately appropriate for massively parallel applications.

Number of lines in distributed program, including test data, etc.: 454

Keywords: pseudorandom, random, quality, luxury, chaotic dynamical systems, massively parallel

Nature of physical problem
Any Monte Carlo or other calculation requiring a uniform pseudorandom number generator.

Method of solution
The RCARRY subtract-with-borrow algorithm of Marsaglia and Zaman is improved by the skipping proposed by Lüscher.

Typical running time
Between one and ten times the time required to generate pseudorandom numbers of traditional low quality, depending on the luxury level and the platform.

Unusual features of the program
The user can choose the level of quality (luxury) he wants.

References
[1] M. Lüscher, Comput. Phys. Commun. 79 (1994) 100, this issue.

Correspondence to: F. James, CERN, CH-1211 Geneva 23, Switzerland

SSDI 0010-4655(93)E0110-9

LONG WRITE-UP

1. The quality of pseudorandom number generators

In a review article [1] on the pseudorandom number generators considered to be the best available in 1990, I discussed their various properties. Very little attention was given to the *quality* or *randomness* property in that article, largely because there was no satisfactory measure of quality, and in spite of the considerable amount of research done on random number algorithms, most of the judgment of randomness was based on empirical testing and, even worse, on the unjustified feeling that generators with very long periods probably also had the best distribution.

Fortunately the situation concerning the quality of pseudorandom number generators is now changing rapidly, with a major contribution coming from the work of Martin Lüscher appearing in this journal [2]. To our knowledge, the algorithm proposed by Lüscher and implemented in RANLUX is the first for which there is convincing evidence of high quality.

Among the many random number generation algorithms available to computational physicists, the quality can vary tremendously. Near the bottom of the scale are some generators distributed with PC software, which, as Hamilton [3] says, are "certainly adequate for standard 'PC-type' applications, such as moving the fish around in a screen saver", but they should not be used in Monte Carlo calculations where a reliable quantitative result is required. As Hamilton also points out, personal computers are now powerful enough to perform very serious Monte Carlo calculations and they therefore need much better random number generators.

The question of just how "good" a random number generator has to be for a given application is a difficult one. Typically a given generator is assumed to be good enough until it produces an obviously wrong result. Certain calculations in theoretical physics, such as lattice QCD and Ising model simulations, are known to be sensitive to the quality of the random numbers and regularly give rise to incorrect results directly attributable to the inadequate quality of the random number generator. One of the more spectacular examples of this phenomenon is reported in the recent paper of Ferrenberg et al [4].

In my own laboratory, CERN, large quantities of pseudorandom numbers are consumed in the simulation of high-energy physics events. One of the uses of such simulations is to test the reconstruction programs; for such a purpose, it is probably sufficient that the events be relatively "typical", and screen-saver quality may be good enough. However, these simulated events are also used to calculate detection efficiencies in order to make quantitative corrections to experimental data; the quality required for this application is certainly higher, but it is very hard to determine what the requirements may be.

Consumers of random numbers who have not yet discovered anomalies in their Monte Carlo results often adopt a very optimistic attitude about the quality of the generator they are using. Even Marsaglia himself, the discoverer of the first known systematic defect in multiplicative congruential generators back in 1968, has been quoted more recently as saying: *"A random number generator is much like sex: when it's good it's wonderful, and when it's bad it's still pretty good."* While I don't wish to comment on this attitude toward sex, I do take issue at the part concerning random numbers, since I hold that a random number generator which unknowingly gives you an incorrect result is not only bad, it is catastrophic.

In view of the long and painful history of incorrect Monte Carlo results due to random number generators of insufficiently good quality, I suggest that where high-quality generators are available, consumers would do well to use the best one, even if they are not sure it is needed. To make another analogy, I would say that a random number generator is much like a wine: it is better to choose one which is too good for a particular occasion than one which is not good enough.

2. The algorithm of Lüscher

As this algorithm is treated very thoroughly in an accompanying paper [2], it is sufficient to describe it here very briefly. It is based on the RCARRY generator [1,5] which has a very long period ($\approx 10^{171}$) but is now known to fail several sensitive tests for randomness. Lüscher's idea was simply to throw away some of the numbers produced by the generator and only deliver a certain fraction to the user. More precisely, the generator delivers twenty-four random numbers to the user, then throws away $p - 24$ numbers before delivering twenty-four more. Lüscher shows quite convincingly that the quality increases continuously as p varies from 24 up to 389, at which point all 24 bits of mantissa are thoroughly chaotic. The RANLUX generator allows the user to choose any value of p from 24 (which corresponds to RCARRY) to 2000, although there is certainly no point in choosing a value larger than 389. Large values of p of course cause the generator to run more slowly, so the user must be able to afford the luxury of high quality. On typical platforms, $p = 389$ runs between five and ten times more slowly than $p = 24$. For many applications, this time is still negligible; in such cases, the user should not deny himself the luxury of demonstrably good random numbers.

3. Subroutine RANLUX, calling sequence and initialization

A vector of N random numbers is generated by the Fortran call:

```
CALL RANLUX (RVEC, N)
```

where the REAL array RVEC must be dimensioned greater or equal to N. If the user does no initialization, a default initialization is performed automatically, and the default luxury level, corresponding to $p = 223$, is chosen. More usually, the user will at least want to choose the luxury (the value of p) himself, and often the initialization as well. Several additional calls are available for these options. In order to simplify the choice of luxury, five standard levels are defined:

- **level 0** ($p = 24$): equivalent to the original RCARRY of Marsaglia and Zaman, very long period, but fails many tests.
- **level 1** ($p = 48$): considerable improvement in quality over level 0, now passes the gap test, but still fails spectral test.
- **level 2** ($p = 97$): passes all known tests, but theoretically still defective.
- **level 3** ($p = 223$): DEFAULT VALUE. Any theoretically possible correlations have a very small chance of being observed.
- **level 4** ($p = 389$): highest possible luxury, all 24 bits of mantissa are chaotic.

Both the initialization and the setting of the luxury level are performed with:

```
CALL RLUXGO (LUX, INT, K1, K2)
```

which initializes the generator from one positive integer INT and sets Luxury Level to LUX if it is an integer between zero and four, or if LUX is greater than 24, it sets p = LUX directly. This allows the user the convenience of choosing a standard luxury level, but also the freedom to choose other values of p if desired. For initialization (as opposed to restarting), K1 and K2 should be set to zero. If INT is set to zero, the default initialization, corresponding to INT = 314159265, is produced. The 32-bit integer INT is used internally to initialize a multiplicative congruential generator which in turn initializes all the 24 seeds inside RANLUX. This means that two values of INT which differ by only one bit will cause totally different initialization of RANLUX, and each of the 2^{31} different values of INT gives rise to an independent sequence which will generate on average about 10^{160} numbers before overlapping any sequence produced by any other value of INT. This generator is therefore appropriate for massively parallel applications.

Two different methods are available for restarting the generator from a given break point. Since the period is very long, the number of different internal states is very large, and restarting requires the specification of a considerable amount of information. This information must be saved at every break point, that is ev-

ery point where the user may want to restart the calculation (for example, in order to repeat a part of the calculation where some exceptional event occurred). The call:

```
CALL RLUXUT (ISVEC)
```

saves in the integer array ISVECT (dimensioned at 25) all the information needed to restart the generator at the current point. Then the call:

```
CALL RLUXIN (ISVEC)
```

restores the generator to the exact state in which it was when the vector ISVEC was saved by RLUXUT.

Since it may not be convenient to save and reload twenty-five integers, another method of restarting is available. With this second method, the state of the generator is saved in only four integers, but the restarting may be very long, since the information saved is simply the initialization information plus the number of random numbers generated since the last initialization, and the restarting skips over all the numbers generated up to the break point. This is still faster than repeating the whole calculation, of course. With this method the information is saved by the call:

```
CALL RLUXAT (LUX,INT,K1,K2)
```

and restored by calling RLUXGO as for initialization, except that K1 and K2 will now be the values returned by RLUXAT instead of zero. In fact the restarting call:

```
CALL RLUXGO (LUX, INT, K1, K2)
```

performs the initialization with LUX and INT and then skips over $K_1 + 10^9 K_2$ internally generated values to bring it into the state it was in when RLUXAT was called giving K1 and K2.

4. Test program

Since this generator is supposed to be portable, the test program serves mainly to verify that the user is getting the "right" numbers from the generator, namely those that I have obtained on all the systems so far tried. The output which should be produced by the test program is contained as comments in the test program itself, so it is easy to verify that it is correct. The test program does test the restarting from break point using both methods, but it does no statistical testing, since these tests require big programs and massive amounts of computing.

References

[1] F. James, A review of pseudorandom number generators, Comput. Phys. Commun. 60 (1990) 329.
[2] M. Lüscher, A portable high-quality random number generator for lattice field theory simulations, Comput. Phys. Commun. 79 (1994) 100, this issue.
[3] K.G. Hamilton, Pseudorandom number generators for personal computers, Comput. Phys. Commun. 75 (1993) 105; 78 (1993) 172.
[4] A. M. Ferrenberg, D. P. Landau and Y.J. Wong, Monte Carlo simulations: hidden errors from "good" random number generators, Phys. Rev. Lett. 69 (1992) 3382.
[5] G. Marsaglia and A. Zaman, A new class of random number generators, Ann. Appl. Prob. 1 (1991) 462.