Algorithms used in `ranlux` v3.0

Martin Lüscher May 2001

Introduction

In its original form the `ranlux` generator [1–4] delivers uniformly distributed random numbers in the range

$$x/2^{24}, \qquad x = 0, 1, 2, \ldots, 2^{24} - 1. \tag{1}$$

On computers complying with the IEEE-754 standard, this set of numbers may be represented through single-precision floating point numbers. It is then straightforward to write a program for the generator that implements the underlying algorithm exactly. Since one does not refer to any special properties of the hardware other than those covered by the IEEE-754 standard, a high level of portability is thus achieved.

The generator can also be cast in another mathematically equivalent form that yields uniformly distributed random numbers in the range

$$x/2^{48}, \qquad x = 0, 1, 2, \ldots, 2^{48} - 1. \tag{2}$$

These can be exactly represented by IEEE-754 double-precision numbers, although in this case not all bits of the fractional part are assigned a random value (the 5 least significant bits are set to 0). It is then up to the user to decide whether single- or double-precision random numbers should be produced.

While the `ranlux` generator has provably good statistical properties, it tends to be rather slow. Modern PC processors have extended capabilities that allow several floating point numbers to be processed at once. The version of `ranlux` described in this note takes advantage of some of these features and achieves significantly higher production rates on these machines than previous versions of the program.

Basic algorithm

In the following lines the algorithm underlying the `ranlux` generator is briefly described. For further details and a theoretical discussion of the generator the reader should consult ref. [1].

Let X be the set of integers x in the range $0 \leq x < b$, where b is an integer greater than 1, referred to as the base, that will be specified later. The algorithm generates a sequence $x_0, x_1, x_2, \ldots$ of elements of X recursively, together with a sequence $c_0, c_1, c_2, \ldots$ of "carry bits". The latter take values 0 or 1 and are used internally, i.e. the interesting output of the algorithm are the numbers x_n, or rather x_n/b if one requires random numbers uniformly distributed between 0 and 1.

The recursion involves two fixed lags, r and s, satisfying $r > s \geq 1$. For $n \geq r$ one first computes the difference

$$\Delta_n = x_{n-s} - x_{n-r} - c_{n-1}, \tag{3}$$

and then determines x_n and c_n through

$$\begin{aligned} x_n &= \Delta_n, \qquad & c_n &= 0 \quad \text{if} \quad \Delta_n \geq 0, \\ x_n &= \Delta_n + b, \qquad & c_n &= 1 \quad \text{if} \quad \Delta_n < 0. \end{aligned} \tag{4}$$

To start the recursion, the first r values $x_0, x_1, \ldots, x_{r-1}$ together with an initial carry bit c_{r-1} must be provided. The configurations

$$x_0 = x_1 = \ldots = x_{r-1} = 0, \qquad c_{r-1} = 0, \tag{5}$$

$$x_0 = x_1 = \ldots = x_{r-1} = b - 1, \qquad c_{r-1} = 1, \tag{6}$$

should be avoided, because the algorithm yields uninteresting sequences of numbers in these cases. All other choices of initial values are admissible.

For the `ranlux` generator one sets

$$b = 2^{24}, \qquad r = 24, \qquad s = 10, \tag{7}$$

and uses only a fraction r/p of the numbers generated by the algorithm. As explained in ref. [1], the residual statistical correlations in the resulting sequence of random numbers are exponentially decreasing when p is increased. The published Fortran program for the generator [2] offers several choices of p, referred to as "luxury levels", because the computer time required to generate new random numbers is proportional to p, and large values of p are thus a luxury.

Double-word algorithm

We may now introduce a new sequence of integers through

$$\tilde{x}_n = x_{2n} + x_{2n+1} b, \qquad n = 0, 1, 2, \ldots \tag{8}$$

Since x_{2n} is non-negative and less than b, it is clear that

$$x_{2n} = \tilde{x}_n \bmod b, \qquad x_{2n+1} = (\tilde{x}_n - x_{2n})/b, \tag{9}$$

i.e. there is a one-to-one relation between the old and new sequence of numbers. When written as a binary number, $\tilde{x}_n$ has at most 48 non-zero digits and so may be exactly represented as a double-precision floating point number on any computer complying with the IEEE-754 standard for this data format.

If we define an associated sequence of carry bits,

$$\tilde{c}_n = c_{2n+1}, \tag{10}$$

it is straightforward to prove that the recursion

$$\begin{aligned} \tilde{x}_n &= \tilde{\Delta}_n, & \tilde{c}_n &= 0 \quad \text{if} \quad \tilde{\Delta}_n \geq 0, \\ \tilde{x}_n &= \tilde{\Delta}_n + b^2, & \tilde{c}_n &= 1 \quad \text{if} \quad \tilde{\Delta}_n < 0, \end{aligned} \tag{11}$$

holds, where the difference $\tilde{\Delta}_n$ is given by

$$\tilde{\Delta}_n = \tilde{x}_{n-s/2} - \tilde{x}_{n-r/2} - \tilde{c}_{n-1}. \tag{12}$$

Evidently this is the same algorithm as above, with base b^2 and lags $r/2$ and $s/2$. We may, therefore, consider `ranlux` to be a generator that yields random numbers with either 24 or 48 random bits.

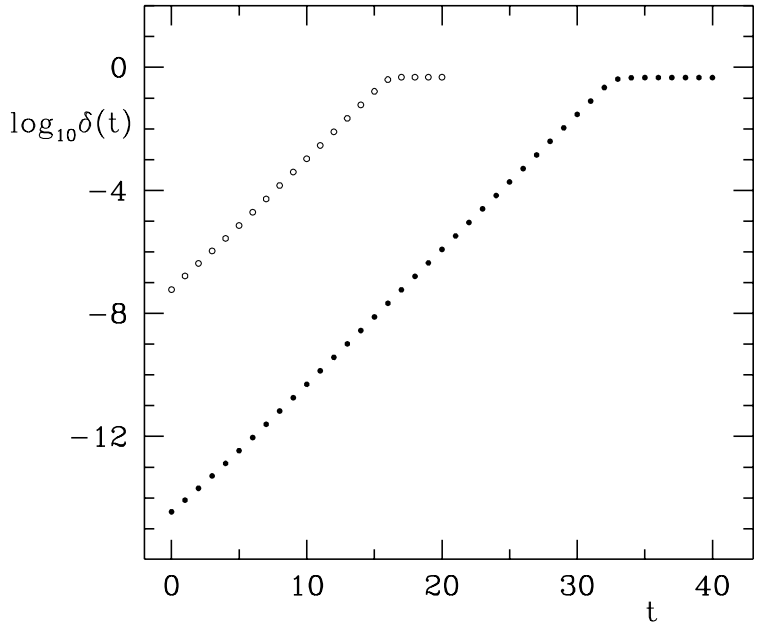

Fig. 1. Average distance $\delta(t)$ between neighbouring trajectories as a function of the evolution time t. Open and full circles refer to the single- and double-word algorithm respectively.

Luxury levels

As discussed in ref. [1] the `ranlux` generator is related to a classical dynamical system that can be proved to be chaotic in a strong sense. In particular, any initial correlations between different states of the generator decrease exponentially as the system evolves. The dynamical system associated with the double-word algorithm belongs to the same category and has the same Liapunov exponent. This can be shown analytically, but is also evident from numerical experiments, where the distance between neighbouring trajectories of the system is measured as a function of time (see Figure 1; the setup is as in sect. 4.1 of ref. [1]).

Vectors of r subsequent elements of the generated sequence of random numbers, separated by $p - r$ discarded elements, may thus be expected to be decorrelated if p is large enough. In the language of the underlying dynamical system, the time separation of these vectors is equal to p/r. So if we choose p to be equal to 218, 404 and 794, for example, a reduction of any initial correlations by approximately 4, 7

Table 1. Luxury levels and merits μ_D of the associated linear congruential generators

level	p	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8
0	218	1.86	1.73	2.75	1.00	0.94	3.77	3.66
1	404	2.54	1.93	0.87	1.34	5.12	1.12	2.26
2	794	1.15	1.63	0.96	1.65	1.07	2.33	0.48

and 14 orders of magnitude is achieved (cf. Figure 1). These particular values of p also fare well in the spectral test, which probes for correlations among D-tuples of vectors over the whole period of the generator. The results of the test are listed in Table 1 together with the recommended assignment of luxury levels †.

All values of p suggested here are even, because odd p's would be inconvenient for the double-word algorithm. In the latter case one keeps the first $r/2$ numbers, discards the next $(p-r)/2$ numbers, uses the following $r/2$ numbers, and so on. This rule matches exactly with the prescription for the basic algorithm.

Integer implementation

Some time ago Hamilton and James [3] pointed out that it is profitable on PC processors to use integer arithmetic for the basic algorithm. The calculated numbers may then be converted to floating point format on output. A potential problem with any such program is that eq. (4) involves a conditional branch that interrupts the processing pipeline half of the time.

It is possible to avoid this completely in the following way. First introduce the constants

```
#define BASE 0x1000000
#define MASK 0xffffff
```

that are equal to 2^{24} and $2^{24}-1$ respectively. The code

```
delta=x[n-s]-x[n-r]-c[n-1];
c[n]=(delta<0);
```

† Note that the level assignment is different from the one in the published Fortran program [2]. The levels 0 and 1 roughly correspond to level 3 and 4 there.

```
delta+=BASE;
x[n]=(delta&MASK);
```

then implements the recursion (4) correctly without generating branch instructions. Note that these lines are ISO C compliant and should thus produce the right result on any modern computer.

Using SSE registers

The recent Intel Pentium processors have an additional set of registers for vectors of 4 single-precision floating point numbers. There is a corresponding set of instructions (referred to as Streaming SIMD Extension) that allows them to be manipulated in various ways. In particular, the registers can be added and multiplied.

Efficient use of these capabilities can be made when 4 copies of the `ranlux` generator, with different initial data, are run in parallel. Here too it is possible to avoid branches in the program without increasing the number of code lines significantly.

Programs that use the SSE registers are necessarily system and compiler specific. Portability is thus lost, but this deficit is to be balanced against the fact that speed-up factors of 2 or so can be achieved in this way with little effort.

Programs

In ref. [5] a set of C programs is described that implement the algorithms discussed above. There are two main subroutines, `ranlxs` and `ranlxd`, that deliver single- and double-precision random numbers respectively. In the first case the user can choose any of the luxury levels listed in Table 1, with 0 being the default level. For the double-precision routine only the two higher levels are admitted in order to guarantee that also the lower bits are decorrelated (it would otherwise not make much sense to generate double-precision random numbers). The initialization of the generators requires some care and is discussed in appendix A.

Whether use is to be made of the SSE registers or not can be controlled at compile time through the macro `SSE`. If `SSE` is not defined, the code that is being compiled is ISO C compliant. Portability is thus preserved. Otherwise the code contains SSE instructions, but will produce exactly the same random numbers on machines that support these (cf. ref. [5]).

Appendix A

To initialize the basic algorithm the first 24 numbers $x_0, x_1, \ldots, x_{23}$ and the carry bit c_{23} need to be specified. If we set the latter to zero for simplicity and write the initial values in binary form, the total number of bits that must be provided is 576. In the version of `ranlxs` and `ranlxd` discussed here, the bits are taken from a random sequence $(b_n)_{n\geq 0}$ that is generated recursively through

$$b_n = (b_{n-13} + b_{n-31}) \bmod 2. \tag{13}$$

To start the recursion 31 initial bits are required which may conveniently be taken to be the binary digits of an integer seed between 1 and $2^{31} - 1$.

As discussed in sect. 3.2.2 of ref. [6], the period of the bit sequence generated through eq. (13) is equal to $2^{31} - 1$. It is then easy to show that different seeds give different initial vectors. We actually need 4 vectors of initial values since the programs run 4 copies of the generator in parallel. To ensure that all these are initialized differently, the reversed bits are used for the initial values x_{4l+i} of the generator number i.

Internally the double-precision program `ranlxd` runs the basic algorithm too and combines pairs of successive numbers on output according to eq. (8). In this case the initialization program reverses the bits of all initial values x_k except for the values x_{4l+i}. So even if the single- and double precision programs are used in the same main program, all initial vectors for the basic algorithm are guaranteed to be different.

References

[1] M. Lüscher, A portable high-quality random number generator for lattice field theory simulations, Comp. Phys. Comm. 79 (1994) 100

[2] F. James, `ranlux`: a Fortran implementation of the high-quality pseudo-random number generator of Lüscher, Comp. Phys. Comm. 79 (1994) 111 [E: *ibid.* 97 (1996) 357]

[3] K. G. Hamilton and F. James, Acceleration of `ranlux`, Comp. Phys. Comm. 101 (1997) 241

[4] K. G. Hamilton, Assembler `ranlux` for PCs, Comp. Phys. Comm. 101 (1997) 249

[5] M. Lüscher, User's guide for `ranlxs` and `ranlxd` v3.0 (May 2001)

[6] D. E. Knuth, Semi-Numerical Algorithms, *in*: The Art of Computer Programming, vol. 2, 2nd ed. (Addison-Wesley, Reading MA, 1981)