Algorithms used in `ranlxs` and `ranlxd` v3.4

Martin Lüscher May 2019

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, the number of arithmetic operations that need to be performed per delivered random number is fairly large. An efficient implementation of the algorithm, exploiting the capabilities of current processors, is thus required to achieve high production rates.

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

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

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 decimation of the full sequence leads to an exponential suppression of the statistical correlations in the delivered sequence of random numbers. Since the computer time required per delivered random number grows roughly proportionally to p, large values of p are a luxury and the chosen values of p are therefore referred to as "luxury levels" [2].

Double-word algorithm

Given a sequence $x_0, x_1, \ldots$ of numbers generated by the basic algorithm, another sequence may be defined through

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

Since

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

there is a one-to-one relation between the old and the new sequence of numbers. Clearly, the composed numbers $\tilde{x}_n$ may assume any integer value from 0 to $2^{48} - 1$.

If an associated sequence of carry bits

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

is introduced, 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

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

This recursion is the same as the basic one, but with base b^2 and lags $r/2$ and $s/2$. RANLUX may thus be considered a generator of random numbers with either 24 or 48 random bits.

Luxury levels

The basic algorithm is closely related to a classical dynamical system that can be proved to be chaotic in a strong sense [1]. 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

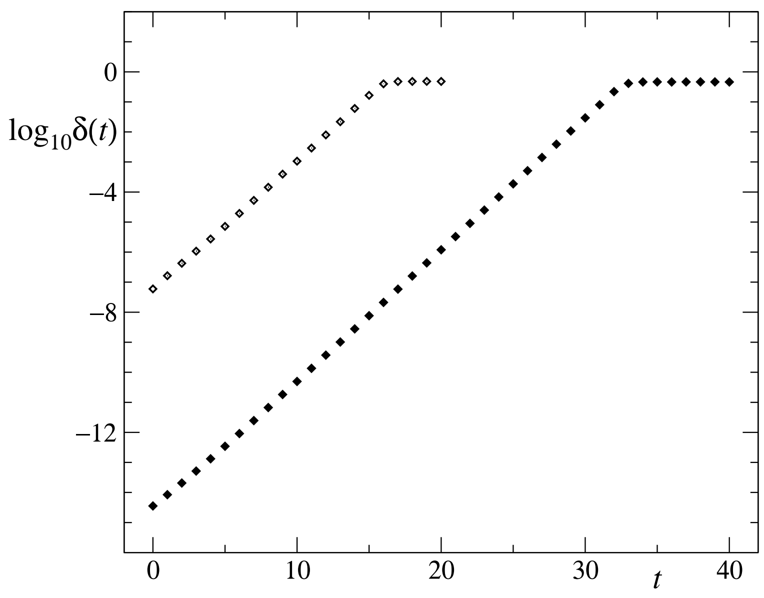

Fig. 1. Average distance $\delta(t)$ of neighbouring trajectories versus the evolution time t. Open and full symbols show the results obtained in the case of the basic and the double-word algorithm. Time is measured in units of full update cycles (i.e. 24 and 12 elementary update steps, respectively).

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 24 subsequent elements of the sequence generated by the basic algorithm, separated by $p-24$ 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 $p/24$. The scheme corresponds to discarding $p/2-12$ numbers from the sequence generated by the double-word algorithm followed 12 used numbers.

If p is set to 218, 404 and 794, for example, a reduction of any initial correlations by approximately 4, 7 and 14 orders of magnitude is achieved (cf. Figure 1). These values of p also fare well in the spectral test [5], which probes for correlations among D-tuples of vectors over the whole period of the generator (see table 1 and refs. [5,1] for the definition of the "merits" μ_D).

The luxury levels in table 1 are the ones chosen in the programs `ranlxs` and `ranlxd` [6] as well as in the `ranlx*` programs provided by the GNU Scientific Library [7], while p can be set to any value in the case of the `C++` standard library [8] by combining the basic generators `ranlux24_base` and `ranlux48_base` with the decimation routine `discard_block_engine`. The `C++` library moreover provides two programs,

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

`ranlux24` and `ranlux48`, with predefined decimation schemes that roughly correspond to level 0 and 2 in table 1.

Integer implementation

The basic and the double-word algorithm can be implemented using either floating-point or integer arithmetic. Since the latter normally consumes less processor cycles, it is profitable to use integer arithmetic internally and to convert the numbers delivered to the calling programs to floating-point format [3]. Moreover, most computers nowadays natively support 64 bit integer arithmetic. The required number of arithmetic operations can thus be halved by using the double-word in place of the basic algorithm.

A further speedup of the generator is achieved by avoiding the branch condition (11) in the code that implements the double-word algorithm. Introducing the constants

```
#define BASE 0x1000000000000L
#define MASK 0xffffffffffffL
```

which are equal to 2^{48} and $2^{48}-1$, the code

```
delta=x[n-5]-x[n-12]-c[n-1];
c[n]=(delta<0);
delta+=BASE;
x[n]=(delta&MASK);
```

performs the update step (11) correctly without generating branch instructions (for simplicity the tilde on $\tilde{x}$, etc., has been omitted). These lines comply with the C99 standard and should give the right results on any machine.

Using x86-64 SIMD instructions

Current AMD and Intel processors support machine-specific SIMD instructions for short vectors of 64 bit integer data, which can be used to increase the speed of the RANLUX generator by a factor 2 to 3. C99 compliance is then lost, but in practice this is of limited importance in view of the omnipresence of these processors.

Both the SSE2 and the more recent AVX2 instruction sets may be used to execute the double-word algorithm in the vector registers of the processors. In order to reduce data dependencies, 4 copies of the generator, with different seeds, may be run in parallel. The highest throughput is then achieved when the states of these copies are loaded to the 16 vector registers and processed, using AVX2 instructions, without intermediate load and store operations.

Programs

A set of C programs implementing the algorithms discussed in these notes is described in ref. [6]. There are two main functions, `ranlxs` and `ranlxd`, that deliver single- and double-precision random numbers. The corresponding initialization programs allow any luxury levels listed in table 1 to be chosen, except for level 0 in the case of `ranlxd` (it would not make much sense to generate double-precision random numbers at this level). The initialization of the generators requires some care and is discussed in the appendix.

The use of x86-64 SIMD instructions may be enabled at compile time. Otherwise the code is C99 compliant and thus portable. In both cases the programs produce exactly the same sequences of random numbers.

Appendix

The initialization of the basic algorithm requires the first 24 numbers $x_0, x_1, \ldots, x_{23}$ and the carry bit c_{23} to be specified. If the latter is set to zero and the initial values are written 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 $(b_n)_{n\geq 0}$ generated recursively through

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

To start the recursion, 31 initial bits must be provided, which may conveniently be taken to be the binary digits of an integer seed in the range from 1 to $2^{31}-1$.

As discussed in sect. 3.2.2 of ref. [5], the period of the bit sequence defined through eq. (13) is equal to $2^{31}-1$. It is then easy to show that different seeds give different initial vectors. Actually, 8 vectors of initial values are required, since the single- and the double-precision programs both run 4 copies of the generator in parallel. In order to guarantee that different seeds lead to pairwise different initializations of these, the 24 bits of selected components x_n of the initial state vectors are reversed according to some pattern that distinguishes the generators.

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

[6] M. Lüscher, User's guide for `ranlxs` and `ranlxd` v3.4 (May 2019)

[7] GNU Scientific library, `https://www.gnu.org/software/gsl/`

[8] `C++` standard library, `https://en.cppreference.com/w/cpp/numeric/random`