User's guide for `ranlxs` and `ranlxd` v3.2

Martin Lüscher December 2005

The programs described in this guide serve to generate single- and double-precision random numbers. They implement the `ranlux` generator of refs. [1–4] in a form that is particularly well adapted to PC processors. Some details on the underlying algorithms and their coding can be found in the accompanying notes [5].

Machine requirements

For `ranlxs` and `ranlxd` to work correctly, the computer must be able to handle single- and double-precision floating point numbers with mantissa of at least 24 and 48 bits respectively. It is also assumed that the base of such numbers (i.e. the radix) is equal to 2. Any machine complying with the IEEE-754 standard for floating point arithmetic satisfies these conditions, but the standard is not required. It is also assumed that default integers are represented by words of 32 bits or more.

The user does not have to care about these constraints since the initialization programs check that the machine is suitable for the generator that is being initialized. If the tests are not passed, the programs terminate with an error message.

Files

The program and associated header files are

```
ranlxs.c        ranlxs.h
ranlxd.c        ranlxd.h
testlx.c
```

The first two of these are modules containing a set of functions, while `testlx.c` is a main program which allows the user to check that the modules are functioning

correctly on the chosen machine and with the chosen compiler options. If single-precision random numbers are desired one should use the `ranlxs` module, while the double-precision routines are collected in the `ranlxd` module. The modules are independent of each other and can be used individually or simultaneously.

When writing these programs, care has been paid to exclude any misuses or interference effects. The externally accessible functions are

`ranlxs`	`ranlxd`
`rlxs_init`	`rlxd_init`
`rlxs_size`	`rlxd_size`
`rlxs_get`	`rlxd_get`
`rlxs_reset`	`rlxd_reset`

`ranlxs` and `ranlxd` are the main subroutines, while the others are utility programs which provide access to the state of the generators.

Compilation

The programs are written in ISO C and should thus be portable. No special options are required and a command like

```
cc [options] ranlxs.c ranlxd.c testlx.c -o testlx
```

will compile the programs and produce the executable `testlx`. The modules are expected to work correctly, even with the most aggressive optimization options, but it is recommended to check this by running `testlx`. This program performs a number of tests and reports the results to `stdout`. The source code of the program may also serve as an example for the proper usage of the modules.

Random number generation

The functions `ranlxs` and `ranlxd` generate random numbers of type `float` and `double` respectively. The synopsis are

```
#include "ranlxs.h"
void ranlxs(float r[],int n);
```

and

```
#include "ranlxd.h"
void ranlxd(double r[],int n);
```

Both functions generate `n` new random numbers and assign them to the first `n` elements of the array `r`. It is left to the user to ensure that `r` is declared appropriately, i.e. no check on the array bounds is made. While this is not required, it is good practice to choose `n` to be equal to the size of `r`. A typical code then looks like

```
#include "ranlxs.h"
#define LVEC 12
.
.
float rvec[LVEC];
.
.
ranlxs(rvec,LVEC);
```

The random numbers generated by `ranlxs` are uniformly distributed in the range

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

Note that this set includes 0 and excludes 1. All numbers are exactly representable on computers that pass the tests performed by the initialization program.

In the case of `ranlxd` the generated random numbers are uniformly distributed in the extended range

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

They are also exactly representable, but an important detail to keep in mind is that the mantissa of IEEE-754 double-precision floating point numbers have 53 bits, i.e. the 5 least significant bits of the generated numbers are always equal to zero on machines complying with the standard.

Initialization

When `ranlxs` and `ranlxd` are called for the first time, the required initializations are performed automatically with default settings for the parameters. The generators may also be initialized explicitly by calling the functions `rlx*_init` (with `*` being equal to `s` or `d` as appropriate). The synopsis is

```
#include "ranlx*.h"
void rlx*_init(int level,int seed);
```

where the seed is an arbitrary integer in the range $1 \leq \text{seed} < 2^{31}$. Different seeds are guaranteed to result in different sequences of random numbers.

The level has to be equal to $0, 1$ or 2, with only the two higher values being permitted in the case of `ranlxd`. This parameter controls the statistical quality of the random numbers generated. At the lowest level the statistical correlations are already very small †. For most applications, including large scale Monte Carlo simulations, this level should be adequate. Increasing the level by 1 reduces the residual correlations by several orders of magnitude at the cost of doubling the execution time [1,5].

The default initializations are obtained by choosing `seed=1` and the lowest admissible levels (0 in the case of `ranlxs` and 1 for `ranlxd`). The initialization programs set the generators to a definite state and the sequences of random numbers generated by subsequent calls of `ranlxs` and `ranlxd` are thus reproducible.

I/O routines

While the data defining the states of the random number generators are not directly accessible, it is possible to extract the complete information on the current states through the functions `rlxs_get` and `rlxd_get`. The synopsis is

```
#include "ranlx*.h"
void rlx*_get(int state[]);
```

On output the array `state` contains the desired information in an encoded form. The array should be declared to have `n=ranlx*_size()` elements. It may also have more elements, but these are not used.

† Note that the level assignment is different from the one in the published Fortran program for the `ranlux` generator [2]. The levels 0 and 1 roughly correspond to level 3 and 4 there (cf. ref. [5]).

Table 1. Average execution times in μs per random number on a PC with Intel Pentium 4 (1.4 GHz) processor

program	level = 0	level = 1	level = 2
`ranlxs` (ISO C)	0.056	0.090	0.161
`ranlxs` (ISO C & SSE)	0.031	0.050	0.093
`ranlxd` (ISO C)	–	0.173	0.318
`ranlxd` (ISO C & SSE)	–	0.092	0.160

At a later stage in the calling program or in another program, the generators may then be reset to the state defined by the array `state` by invoking the functions `rlxs_reset` and `rlxd_reset`. The synopsis is

```
#include "ranlx*.h"
void rlx*_reset(int state[]);
```

The programs check that the data passed by the argument are sane and exit with an error message if this is not the case.

SSE acceleration

To enable the use of SSE inline assembly code, the macro `SSE` should be defined at compilation time. With the GNU C compiler this can be achieved simply by specifying the option `-DSSE` as in

```
gcc -O2 -DSSE ranlxs.c ranlxd.c testlx.c -o testlx
```

The correct functioning of the binaries `ranlxs.o` and `ranlxd.o` may then be checked by running the test program `testlx`. Other compilers cannot be used at this point unless they understand the GNU inline assembly syntax. Whether the `SSE` macro is set or not has no effect on the functionality of the programs: the generated sequences of random numbers are always exactly the same.

Timing

The computer time required to generate new random numbers depends on the machine, the compiler and the luxury level. Some benchmark results are reported in table 1 for a Linux PC with kernel version 2.4.0, GNU C compiler version 2.95.2 and optimization level `-O2`. These figures can be expected to scale roughly with the frequency of the processor, since the memory latency has a neglible influence on the performance of the programs.

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. Hamiltion and F. James, Acceleration of `ranlux`, Comp. Phys. Comm. 101 (1997) 241

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

[5] M. Lüscher, Algorithms used in `ranlux` v3.0 (May 2001)