

# Lab. #1 – Practical Work

## Study of pseudo-random number generation

### PEDAGOGICAL AIMS

- Develop and improve basic Unix skills and C programming ability (if not previously acquired)
- Understand the elementary coding of pseudo-random number generators and the importance of generator quality.
- Implement old congruential pseudo-random number generators
- Implement a shift register generator.
- Understand pseudo-random number generator shuffling.
- Find the current best generators and testing libraries.

- 1) Test one of the first non-linear technique proposed by John Von Neumann. Implement in C the “middle square technique” with a small ‘toy’ case with 4 digits (Von Neumann, 1944). Start with a seed at  $N_0 = 1234$ , (representing 0,1234 - 4 decimal digits).

$N_0 = 1234$      $1234 * 1234 = 01522756$   
 $N_1 = 5227$      $5227 * 5227 = 27321529$   
 $N_2 = 3215$     ...

- 2) Test different seeds, look at some cycles (ex : seed 4100), pseudo-cycles with queues (ex : seed 1324 or 1301 after a some iterations). Observe convergence towards an absorbing state like 0 (seed 1234) or a different absorbing state: 100 (with 3141 as seed).
- 3) Check the information on the default random number generator by a “`man 3 rand`”. Is it suitable for scientific applications?
- 4) Implement a coin tossing simulation using the classical `rand` number generator given by `stdlib` and test the equidistribution (heads or tails) with 10, 100 and 1000 experiments (runs).
- 5) Implement a regular dice simulation (6 faces) and test the equidistribution of each face with 100, 1000 experiments (runs). After this test, implement a 10 faces dice simulation and test the equidistribution in 10 bins with a smart test (not a sequence of if – elseif). Advice: use a 10 int array and C coding possibilities to store in this array the frequency of appearance of each face. Test this code with 10, 100, 1 000 and 1 000 000 experiments (runs).
- 6) Implement and test linear congruential generators (LCGs). They are fast and have been intensively used in the past, but they have structural statistical flaws. **They have to be absolutely avoided for scientific programs**, but they are repeatable and ok for games and

even serious games (a growing part of simulation programs) and many testing (like code checking) and additionally they are fast.

```
Initialize the seed  $x_0$   
Draw the next number with the following equation  

$$x_{i+1} = (a * x_i + c) \text{ modulo } m$$
 // This supposes that you provide (a, c, m)
```

*Figure 1: Linear Congruential Generator.*

Test for instance:  $x_{i+1} = (5 * x_i + 1) \text{ modulo } 16$

With  $x_0 = 5$  we have  $x_1 = (5 * 5 + 1) \text{ modulo } 16 = (25 + 1) \text{ modulo } 16 = 26 \text{ modulo } 16 = 10$

Implement this generator and check that the 32 first pseudo-random numbers are:

10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5.

Implement a function named `intRand` able to give this trace. The main program should call `intRand()` as a regular call of `rand()`.

- 7) To obtain floating numbers we can divide by 16, to obtain normalized numbers in  $[0..1[$

Implement a function named `floatRand` giving the float pseudo-random numbers corresponding to the suite of `intRand()`. You should call the `int` function and then check your float results with the 32 first numbers below:

0.625, 0.1875, 0, 0.0625, 0.375, 0.9375, 0.75, 0.8125, 0.125, 0.6875, 0.5, 0.5625,  
0.875, 0.4375, 0.25, 0.3125, 0.625, 0.1875, 0, 0.0625, 0.375, 0.9375, 0.75, 0.8125,  
0.125, 0.6875, 0.5, 0.5625, 0.875, 0.4375, 0.25, 0.3125...

- 8) Test different seeds and (a, c, m) tuples.
- 9) Thanks to Wikipedia, find smart tuples and seeds to optimize 32 bits LCGs. (see Linear Congruential Generator on the Internet).
- 10) Find on the internet scientific libraries implementing up to date random number generators.
- 11) Find on the internet statistical libraries able to test the quality of pseudo random number sources.

### Additional questions for fast students with developing skills:

Implement a basic 4 bits shift register generator (this supposes bit coding in C, with XOR masks (^) and left shift (<<)). Use the generator shown in the lecture.

The characteristic polynomial for this generator below is  $x^4 + x + 1$  (remark: characteristic polynomials are given with the maximum power equal to the number of bits – the powers of 2 below 'n bits' (in this case  $2^1$  and  $2^0$  are showing the bits which will be used in the 4 bits registers to compute the XOR that will be injected in the left position after a right shift). You can implement this with bit fields with simple masks on bytes (unsigned char). Look on the internet if you need to learn or revise C binary operators ( & | ^ ).

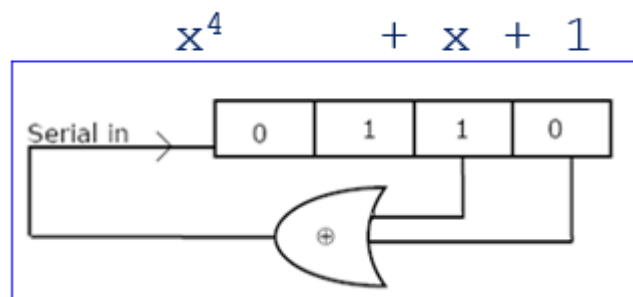


Figure 2: A basic 4 bits right shift register generator with XOR and re-injection.

## Anciens étudiants de L2

Etudier le logiciel TestU01 de Pierre l'Ecuyer, puis faire un rapport présentant comment installer cet outil et comment s'en servir sur le générateur donné en exemple ci-après. Faire un test avec variables d'initialisations proposées, puis placer z et c à 0 et renouveler le test.

<http://simul.iro.umontreal.ca/testu01/tu01.html>

Le générateur KISS (Keep It Simple Stupid) a été proposé par G. Marsaglia (une version légèrement plus ancienne s'appelle KISS99 dans un article de Pierre l'Ecuyer). Ce générateur remarquablement court et rapide combine 3 générateurs simples différents et a une « courte » période d'environ  $2^{120}$ .

```
/* Proposed Seeds */
static unsigned int    x = 123456789,
                       y = 362436000,
                       z = 521288629,
                       c = 7654321;

unsigned int KISS()
{
    unsigned long long t, a = 698769069ULL;

    x = 69069 * x + 12345;
    y ^= (y << 13); y ^= (y >> 17); y ^= (y << 5);

    /* y must never be set to zero! */
    t = a * z + c;
    c = (t >> 32);

    /* Also avoid setting z = c = 0 */

    return x + y + (z = t);
}
```

13) Random number generators shuffling: if GX and GY are 2 generators, the shuffling principle is given below:

An initialization function, `initShuffledGen()`, is used and uses the first generator GX. This generator is used to fill an array of random numbers (for instance N=100)

For i = 1 to 100 do Array[ i ] = a number drawn using GX End For

Another function is proposed for the end-user: `suffledGen()`. This function returns a random number (integer or float depending on your implementation choice)

Shuffled random function

Begin

With the 2<sup>nd</sup> generator GY draw an index between 0 and the max size of you have retained for your global array of random numbers filled by the previous initialization.

The random number at present Array [index] have been produced by GX  
It will be given to the user.  
RN = Array [index]

Before returning this value to the user of this function we have to refill the array with GX

Array [index] = a new random number drawn by GX

Return RN

End

*Figure 3: Shuffling of 2 generators.*

14) Look at quasi-random number generation and also at true random number services.