Lab. #1 – Practical Work
Study of pseudo-random number generation

« God does not play dice » A. Einstein

Pedagogical Aims

- Develop 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 of the old technique proposed by John Von Neumann. Implement in C the "middle square technique" on a 'toy' case with 4 digits (Von Neumann, 1944). Start with a seed at N0 = 1234, (representing 0,1234 - 4 decimal digits)

   N0 = 1234   1234 * 1234 = 01522756
   N1 = 5227   5227 * 5227 = 27321529
   N2 = 3215   …

2) Test different seeds, look at some cycles (ex : seed 4100), pseudo-cycles with queues (ex : seed 1301 with around 100 iterataions). Observe convergence towards an absorbing state like 0 (seed 1234) or a different absorbing state : 100 (with 3141 as seed).

3) 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).

4) Check the information on the default random number generator by a "`man 3 rand`". Is Is suitable for scientific applications?

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 – TIPS: 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 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 ok for games, serious game (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
$x_{i+1} = ( a * x_i + c )$ modulo $m$          ([1]) This suppose that you provide (a, c, m)
```

Figure 1: Linear Congruential Generator.

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

With $x_0 = 5$ we have $x_1 = (5 * 5_i + 1)$ modulo 16 = (25 + 1) modulo 16 = 26 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.

If we divide by 16 with obtain normalized numbers in [0..1[

Implement 2 functions: an intRand giving pseudo-random integer as above and a floatRand function giving float numbers (check 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) Implement a basic 4 bits shift register generator (this suppose 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 (& | ^).

[1] *x* = *y* modulo *z* > remaining integer part of the integer division of *y* by *z*. Use ' % ' for C/C++ implementation.

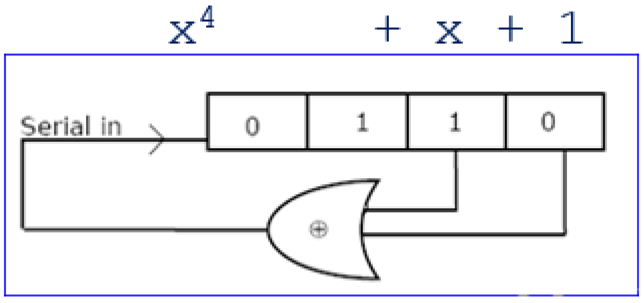

Figure 2: A basic 4 bits right shift register generator with XOR and re-injection.

For fast students who get bored with the previous teasing exercises:

11) 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 2nd 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.

12) Find on the internet scientific libraries implementing up to date random number generators.

13) Find on the internet statistical libraries able to test the quality of pseudo random number sources.

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