Lab. #1 – Practical Work
Lodeling & Simulating Randomness with deterministic “pseudo-random” number generators

PEDAGOGICAL AIMS

- Learn how deterministic algorithms can simulate randomness.
- Improve basic Unix skills and C programming ability (when not previously acquired).
- Understand the elementary coding of pseudo-random number generators and discover the importance of repeatability for debugging and for Science > ‘pseudo’ random
- Discover the limits with a tiny nonlinear generator from Von Neuman:
  - extremely short period,
  - cycles and short path to cycles
  - evidence that the quality of the seed/status is fundamental for the quality of the whole generator
- Implement old congruential pseudo-random number generators to see their limits.
- Implement a simple shift register generator.
- Understand pseudo-random number generator shuffling.
- Find the current best generators and testing libraries.

AH AH MOMENTS…

At some time of this lab, you should be surprised or disappointed by the fact that your ‘random number generator’ is giving each time the same numbers… #@!$µ*:!!!

This is designed on purpose, they are **pseudo** random, because programs must be debugged and thus we must able have to follow the same trace each time we run our ‘stochatic program’. **P**RNG are **Pseudo**-random Number Generators. Physicists, computer scientists and specialist of electronics still **fight to maintain the deterministic aspect of our computers.** Otherwise our digital computers are worthless… how do you program if you can’t debug ? The randomness is however observed when we have found a pertinent model of randomness (the generator). In this case the suite of pseudo-random numbers generated will pass the all the known statistical tests of randomness. This is the case since the end of the last millennium. However, we still observer much too frequently cases of bad random number generators. This is why we teach the essence of the known weaknesses before learning that there are some fast and very good PRNGs produced by talented mathematicians.

You should also have noticed that the choice of the initial status impacts the trace but also the quality of the generated numbers. It cannot be left to chance, or to time (or by random choice… ☺)

> From what we previously stated, the code: `srand(time(NULL))` is banned from (serious) scientific programs. However, it can be justified for games or trial games when the programs have been debugged.
>
> On slide 9, you will see how you can run independent replicates to collect statistics. Initialize your PRNG only once, then use sequentially you numbers for all your ‘independent’ experiments. You will practice this in Lab #3 and #4.

Lab questions:

A – Nonlinear generator :

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 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 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?

B – Model usual randomness, coin tossing, dices…

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). Do not use the `srand()` function, keep the basic initialization.

5) Implement a regular dice simulation (6 faces) and test the equidistribution of each face with 10, 100, 1000 and 1000000 experiments (runs).

6) 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. Again, test this code with 10, 100, 1 000 and 1 000 000 experiments (runs).

C – Linear congruential generators

7) 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
$x_{i+1} = ( a * x_i + c )$ modulo $m$          // This supposes that you provide (a, c, m)
```

Figure 1: Linear Congruential Generator.

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

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).

D – miscellaneous

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.

E – Shift register generators with prime polynomial (Tausworthe generatoirs) this is an 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 (& | ^).

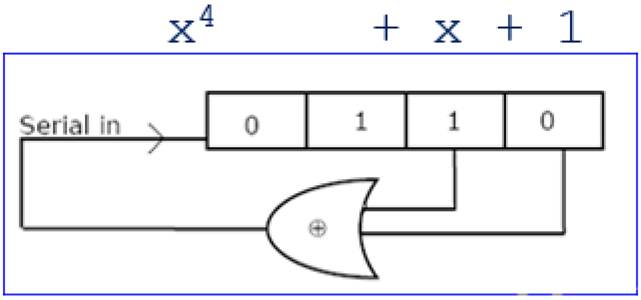

Figure 2: A basic 4 bits right shift register generator with XOR and re-injection.