# Implementation and Assessment of Pseudo Random Numbers Generators

Manoel C. S. Filho
Computer Department
Instituto Federal de Educação do Tocantins (IFTO)
Palmas, Tocantins, Brazil 77021-090
Email: mcampos@ifto.edu.br
University of Beira Interior (UBI)
Covilhã, Castelo Branco, Portugal 6201-001
Email: d1365@ubi.pt

Pedro R. M. Inácio
Instituto de Telecomunicações
Covilhã, Castelo Branco, Portugal 6201-001
Email: inacio@di.ubi.pt

***Abstract*—Pseudo Random Number Generators (PRNG) are crucial to computing, mainly to games and simulation tools. There are several implementations of PRNG that need to trade-off between accuracy and speed. The current paper presents the implementation and assessment of some C pseudo random number generators. The tests are performed with TestU01 library and the Kolmogorov-Smirnov approach using only open source tools. The developed libraries are provided under open source license and can be used to implement or test others generators.**

## I. Introduction

Random numbers are used in several areas of the knowledge and there are a lot of practical applications to them. In the nature, there are different random events that happens all the time. However, the use of true random numbers to simulate a lot of real world events in computing is not a easy task.

Some implementations like the Random.org [1] and the Quantum Random Bit Generator Service [2] use, for instance, atmospheric data to generate true random numbers. Nevertheless not all computing devices can use Web Service implementations likes these (due lack of Internet connection, for instance) or use internal random sources like keystrokes or mouse movements to obtain random numbers.

So, the implementation of pseudo random number generators are very important, mainly in the simulation area, where applications need to simulate events of a given scenario.

The goodness of fit of generated pseudo random numbers is very important to avoid bias in simulations and obtaining results most close to the simulated scenario.

Accordingly, the current paper implements and assess a series of Pseudo Random Number Generators (PRNG) to measure the quality of the generated numbers. The project uses C language and open source tools in its implementation, creating libraries to generate pseudo numbers in uniform, normal (gaussian) and pareto distributions. It is an open source project available under GNU GPLv3 license at http://github.com/manoelcampos/PseudoRandomNumberGenerators.

The paper is organized as follows. The Section II presents the materials and methods used in the work. The Section III presents the proposal of implementation and assessment of pseudo random numbers generators. The Section IV presents and discusses the results. The Section V presents the conclusions, followed by the acknowledgements.

Scholarship from CAPES - Proc. no 13585/13-4

## II. Materials and Methods

To develop the proposed applications were used the following tools:

- GCC (GNU Compiler Collection) [3] to compile the C applications
- GNU Make 3.81 [4] to automate the compilation process of the project
- gnuplot 4.6 (patchlevel 5) [5] to automate the generation of graphics
- Doxygen 1.8.6 [6] to generate source code documentation

## III. Proposal

At this section the developed project is presented.

### A. Project Structure

The project source code has the following directory structure:

- PseudoRandomNumberGenerators
  - *.c - Each file represents a main program for each defined task
  - lib/ - Directory with the developed libraries containing the functions used by the applications.
  - Makefile - Make file used to compile all the applications source code and the documentation. To compile the applications, only type "make" at a terminal.
  - bin/ - Directory where the compiled applications are stored.
  - Doxyfile.config - Doxygen configuration file, defining the parameters to generate the source documentation. To generate the documentation, only type the command "make doc".
  - doc/ - Directory where the documentation is generated after executing the command "make doc".

- *.gnuplot.sh - gnuplot executable scripts to call the developed applications (to generate random numbers) and from their output, generate the graphics to analyse if the numbers really follow the desired distribution.
- *.eps - Graphics generated from gnuplot scripts as result of Kolmogorov-Smirnov Tests.

## IV. Results

To assessment of the developed PRNG, a battery of tests were performed using TestU01[1] [7] and Kolmogorov-Smirnov Statistical Tests. At this evaluation, some tasks were proposed and each one is presented in the next sub-sections.

### A. Task 1

At this task, a simple Linear Congruential Generator (LCG) is implemented. The application task1mylcg.c outputs a sequence of generated numbers. The files task1mylcg_testu01.c and task1native_rand_testu01.c test the implemented generator and the C native generator, respectively. The tests are made using the TestU01 library.

These last two programs accept a command line parameter to define what TestU01 battery will be applied to the generator. Calling the applications without parameters will show the available options.

The developed functions are in the file lib/lcglib.c. The values of the parameters seed, C and M were defined in the function lgc_initialize, where C and M are coprimes. The used values are $seed = 10; C = 12345; M = 2^{32}$

- Q1: What is the lenght of the period of the PRNG implemented within the scope of this task?
  The period length is $2^{32} = 4294967296$ because this value is used in a mod operation. So, all M values (remainders) will be returned before the sequence starting to repeat.

- Q2: Does the lenght of the period depend of the condition $gcd(M, C) = 1$?
  Yes, it does. Because M and C must be coprime, in other words, the greater common divisor of them must be 1. So, assuming that M and C are natural numbers, C cannot be divided by M. This ensure that all reminders in the set $[0, M[$ will be returned, obtaining a pseudo random numbers sequence.

- Q3: How many tests did the implemented generator pass?
  The generator did not pass any test in Small Crush battery due all p-values were outside of the interval $[0.001, 0.9990]$. This represents that the null hypothesis at every statistical test was refused, in other words, the hypothesis being tested are not confirmed and the RNG did not pass the tests.

  At the Crush battery, the generator did not pass in 136 tests, almost all tests. Crush battery performs more tests than Small Crush and the generator passed in some tests performed at this new battery.

- Q4: (Optional) How many tests does the C native PRNG pass?
  The native C PRNG did not pass in 9 tests in Small Crush and 97 tests in Crush batteries.

### B. Task 2

At this task, a better LCG is implemented using 32 generators instead of only one. The program task2.c outputs a sequence of generated numbers and task2testu01.c tests the generator using the TestU01 library. This second program accept a command line parameter to define what TestU01 battery will be applied to the generator. Calling the application without parameters will show the available options.

The developed functions are in the file lib/betterlcglib.c (realize that the functions have the same name in the lib/lcglib.c library). The values of the parameters seed (10) and M ($2^{32}$) were defined in the function lgc_initialize, where C and M are coprimes. Now, the C variable is a array of 33 elements, initialized by prime numbers, manual and randomly chosen. This ensure they will be coprime with M. The selected C values are presented in the Listing 1.

Listing 1. Task 2: Better LCG C array

```
static unsigned int C[GENERATORS] = {
    1607,  61, 1019, 523,  907, 887,  431, 12821,
    769, 9173, 223, 7127, 5939, 919, 131, 23, 911,
    5189, 83, 13001,98713, 7229, 967, 1277,  877,
    719, 277, 8929, 16033, 733, 3833, 383, 28657
};
```

A optimized version of this lcg_rand function is shown in the Listing 2.

Listing 2. Task 2: Implementing a better LCG for PRNG

```
unsigned int lcg_rand(){
    unsigned int x, prn = 0;
    int i=GENERATORS-1;
    //Uses right bit shift and AND operation
    //to check what bits of last is equals 1
    //(x is the last value in the array xn)
    for(x=(xn[i]+=C[i]), i=0; x>0; i++, x>>=1)
        if(x & 1)
            prn ^= (xn[i]+=C[i]);

    return prn;
}
```

- Q5: What is the lenght of the period of this generator?
  The period length is $2^{32}$, the original value defined to $M$. This does not change because all the 33 generators use the module operation with the same M value.

- Q6.: How many tests did the implemented generator pass?
  The better LCG did not only pass in the sknuth_MaxOft Anderson-Darling test at Small Crush battery. So, is clear

[1]Only the TestU01 Small Crush and Crush batteries were analysed because the Big Crush battery did not converge. This battery runs for a long time and does not finish.

the improvement in this new version of the LCG. In the previous implementation (Task 1), the generator did not pass any test in Small Crush battery. At Crush Batteries, the generator did not pass in 46 tests.

### C. Task 3

The program task3boxmuller.c implements a Gaussian Random Number Generator (GRNG) based on the Box-Muller method. The application runs without parameters and outputs the generated random numbers in the first column. The next column values are used to generate the graphs to the Kolmogorov-Smirnov Test and represent, for instance, the Cumulative Distribution Function (CDF) of the random variable. The program task3pollar_rejection.c implements the GRNG based on the Pollar Rejection method and works like the first one. In the case of these two programs, there are respective *.gnuplot.sh files that are gnuplot executable scripts to run the previous C programs and generate the graphs for the Kolmogorov-Smirnov Test. The resulting graphs are saved in eps files in the current directory.

The Box-Muller generator uses the function of Listing 3 to generate a random number.

Listing 3. Box-Muller Gaussian Pseudo Random Number Generator

```
double boxmuller(){
        #define RAND() (rand()/((double) RAND_MAX))
        static unsigned short cont = 3;
        static double a, b;
        if(cont > 2){
                a = sqrt(-2 * log(RAND()));
                b = 2 * M_PI * (RAND());
                cont = 1;
        }

        return (cont++ == 1 ? a*sin(b) : a*cos(b));
}
```

The Pollar Rejection generator uses the function of Listing 4 to generate a random number.

Listing 4. Pollar Rejection Gaussian Pseudo Random Number Generator

```
double pollar_rejection(){
        #define RAND()(2*(rand()/(1.0*RAND_MAX))-1)
        static unsigned short cont = 3;
        static double v1, v2, d, f;

        if(cont > 2){
                do {
                        v1 = RAND();  v2 = RAND();
                        d = v1*v1 + v2*v2;
                } while(d <= 0 || d >= 1);
                cont = 0;
                f = sqrt(-2 * log(d)/d);
        }

        return (++cont == 1 ? f*v1 : f*v2 );
}
```

- Q7: What were the algorithms that you chose to implement?
  The chosen algorithms were the Box-Muller and Pollar Rejection (the first two presented in the [8]). They were chosen due their simplicity to implement and speed to generate the pseudo random numbers.
- Q8: Where the algorithms exact or approximate?
  Exact algorithms generate perfect random numbers for the desired distribution (in the case, the gaussian distribution). The Box-Muller and Pollar Rejection methods are exact methods [8].

The Figures 1 and 2 present the results of the Kolmogorov-Smirnov Statistical Tests to Box-Muller and Pollar Rejection GPRNG, respectively. The maximum distance of the curve for the generated numbers CDF and the standard normal (gaussian) distribution CDF is presented in the graphs. How can be seen in the mentioned figures, the maximum distance were 0.027054 and 0.036330 for the Box-Muller and Pollar Rejection implementation, respectively. How these values are lower than the standard critical value 0.05, we can considerate that both generators really follow the gaussian (normal) distribution.

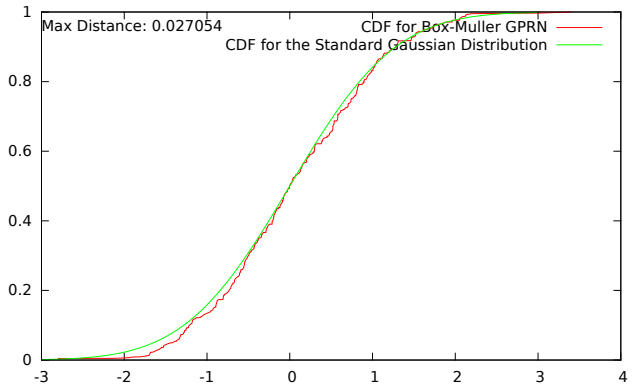


Fig. 1. Box-Muller Kolmogorov-Smirnov Test

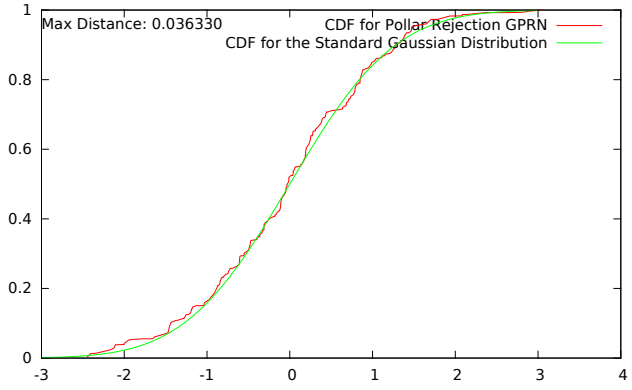


Fig. 2. Pollar Rejection Kolmogorov-Smirnov Test

### D. Task 4

The program task4pareto.c implements a Pareto Random Number Generator. The program works like the applications of the previous tasks. The file task4pareto.gnuplot.sh is the respective gnuplot script to generate the graph (pareto.eps) for the Kolmogorov-Smirnov Test.

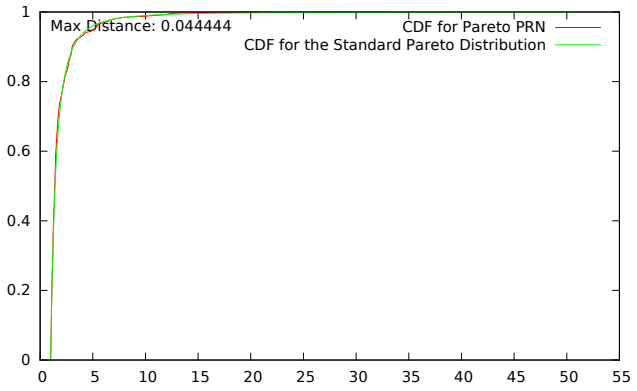


Fig. 3. Pareto Kolmogorov-Smirnov Test

The Pareto generator uses the function of Listing 5 to generate a random number.

Listing 5. Pareto Pseudo Random Number Generator

```
double pareto(){
    #define RAND() ( rand()/(double)RAND_MAX )
    double alpha=2.2, m = 1.0;
    return m/(pow(RAND(), 1.0/alpha));
}
```

The Figure 3 presents the results of the Kolmogorov-Smirnov Statistical Tests. The maximum distance of the curve for the generated numbers CDF and the standard pareto distribution CDF is presented in the graph. How can be seen in the mentioned figure, the maximum distance was 0.036330, representing that the generator really follows the pareto distribution.

## V. Conclusion

At this paper, a series of pseudo random number generators were implemented and assessed. The project provided libraries to generate pseudo random numbers following the uniform, normal and pareto distributions and provides a set of libraries to use this generators in others applications. It used automated ways to assess the good of fitness of the implemented generators, with a full documented source code. All the implemented generators proved to follow the desired distribution.

## Acknowledgement

The authors would like to thank CAPES Brazilian agency for the scholarship to support this work.

## References

[1] Random.org. (2014) True Random Number Service. [Online]. Available: http://www.random.org
[2] Radomir Stevanović. (2014) Quantum Random Bit Generator Service. [Online]. Available: http://random.irb.hr
[3] GCC. (2014) GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org
[4] G. N. U. Make. (2014) GNU Make. [Online]. Available: http://www.gnu.org/software/make/
[5] Gnuplot. (2014) gnuplot homepage. [Online]. Available: http://www.gnuplot.info
[6] Doxygen. (2014) Doxygen: Main Page. [Online]. Available: www.doxygen.org
[7] P. L'Ecuyer and R. Simard, "TestU01," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 22–es, Aug. 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1268776.1268777
[8] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor, "Gaussian random number generators," *ACM Computing Surveys*, vol. 39, no. 4, pp. 11–es, Nov. 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1287620.1287622