# Pseudo-random numbers

Anne Fisch
Maxime Rubio
Yannick Verbeelen

*supervised by*
Dr. George Kerchev
Prof. Gabor Wiese
Prof. Ivan Nourdin


Experimental Mathematics 3
University of Luxembourg
Faculty of Science, Technology and Medicine
Academic Year 2020-2021 (Summer Semester)

**Abstract**

The goal of this project is to get to know the notions of pseudo-random numbers as well as the basics of statistical tests and to apply those tests to pseudo-random number generators. The focus of this project will lie on testing a generator invented in 2006 by Alain Schumacher, who also acted as an external expert.

# Contents

# 1 Introduction

## 1.1 Randomness

What exactly is *randomness*? Everybody knows what it is and we get confronted by it everyday but it is actually really hard to define it in a mathematical sense.
For example, taking two sequences of elements from $\{0, 1\}$:

- 1100110011
- 0100101110

Most people would agree that the second-one is more "random" than the first, even tough both sequences have a probability of $\frac{1}{2^{10}}$ of occurring.
This comes from the fact that we can probably predict the next element of the first sequence to be 0 while there is no way of telling what will be the next element of the second one.
Additionally, relying on the predictability of the next element in a sequence, brings a problem of subjectivity since it is also really hard to define the limit where the predictability is obvious enough.
For example, taking the following sequence:

$$28; 448; 298; 149; 2348; 2234; 1117; 17872$$

Some people might recognize that a certain pattern has been used (multiplying by 16, subtracting 150 and dividing by 2, then starting again) and would predict the next number to be $17872 - 150 = 17722$, but this will probably not be the case for everybody.
This is the reason why we often rely on physical events to simulate randomness, for example flipping a coin or rolling a dice. Using coin flipping, it would be possible to randomly obtain our very first sequence by setting $Tails = 1$ and $Head = 0$. It would not be probable but still possible.

However, this method of simulating randomness has quite a lot of disadvantages for areas where huge amounts of random numbers are needed (cryptocurrency, statistics, etc.). For one they are very slow at producing big sequences. Imagine flipping a coin millions of times just to get one random sequence. That is why using computers will probably be the better choice.

However, computers are deterministic as their actions are determined by a prescribed set of instructions, so how could it produce a random sequence? In some instances, an extra piece of hardware can allow a computer to generate random sequences from physical events, such as with a Geiger counter.
This however is again not very fast and also does not allow the reproduction of the same sequence which can in some cases be really practical for example

when testing a program.
This is where pseudo-random number generators step in.

## 1.2 Pseudo-Random Number Generators

*Pseudo-random number generators* (PRNG), as their name suggests, are algorithms that generate sequences that seem random by using mathematical formulas or even precalculated tables.
In most applications using those sequences is good enough and they do not bring the disadvantages of true random number generators (TRNG) already mentioned above. They are more efficient than true random number generators and since they are deterministic, one can easily reproduce the exact same sequences when needed.

A well-known example of a PRNG is the *linear congruential generator* (LCG) which works as follows:
We define the generator by selecting the four integers:

- the modulus $m$; with $m > 0$
- the multiplier $a$; with $0 \leq a < m$
- the increment $c$; with $0 \leq c < m$
- the seed $x_0$; with $0 \leq x_0 < m$

The generated sequence is then defined recursively: $x_{i+1} = (ax_i + c) \ mod(m)$
For example, for $m = 7$; $a = 3$; $c = 5$ and $x_0 = 2$ we get:

$$2; 4; 3; 0; 5; 6; 2; 4; 3; 0$$

Of course, once an integer reappears the sequence is just going to cycle through the same subsequence. The length of this subsequence is called the period, in this case we have a sequence of period 6. Of course, this sequence does not really seem random as the period is really small.
In general, for linear congruential generators the period will always be at most $m$, thus a generator with a small modulus will not do a great job at generating long pseudo-random sequences.

There exists a multitude of other PRNGs like the *Middle-Square Algorithm*, which takes the square of a four-digit number, extracts the middle four digits and continues with the same procedure leaving us with a sequence of numbers between 0 and 9999.
However we will focus on the *AHS number generator* (named after its inventor Alain H. Schumacher), which we will introduce later on, and on

a commonly used linear congruential generator defined by the following parameters: $m = 2^{32}; a = 1664525; c = 1013904223; x_0 = 0$.

The period is only one of the numerous factors one needs to consider before judging or comparing PRNGs. Other factors are for example the distribution of the generated numbers or the independence between elements and subsequences.
For this reason, a lot of tests for PRNGs and their generated sequences exist and need to be applied before choosing which one suits the situation the best.

## 1.3 Testing

There exist two types of tests:

- *Statistical* tests are applied on sequences and do not need any information about how they were generated.
- *Theoretical* tests are applied on the generator itself as they do not need a generated sequence.

In this project, we will set our focus on statistical tests. An important notion is that in a few cases, a sequence generated by a TRNG can seem non-random (for example in section 1.1 where the very first sequence could possibly be obtained by coin flipping). So pseudo-random sequences can in some situations appear more random than real random sequences.
Thus, statistical tests can sometimes deceive us and for that reason it is crucial to understand that the passing of a test only comforts us that a sequence is random but never assures that it really is.
This is why it is important to apply a lot of tests on a sequence before making any assumptions, the more tests it passes, the more confidence we can have that it is random.

### 1.3.1 Statistical Testing

Statistical tests are used to verify if enough evidence is given to reject a certain hypothesis, called the null hypothesis, denoted by $\mathbf{H_0}$. Let us have the null hypothesis be "$\mathbf{H_0}$ : the sequence is random" as an example for comprehension in this section.
For each test a certain statistic, relevant to the randomness, is analysed to determine if $\mathbf{H_0}$ is rejected or not, i.e. to determine if the randomness is rejected. The null hypothesis is never accepted since, as explained before, the passing of a test does not guarantee the randomness but only does not reject it.

Two types of errors can happen: Firstly, we reject $\mathbf{H_0}$ even though the hypothesis is correct (error of the first type) and secondly $\mathbf{H_0}$ is not correct but we do not reject it (error of the second type).
The probability of having a type 1 error is called the level of significance and is most of the time set before the testing and denoted by $\alpha$. In other words, $\alpha$ is the probability of rejecting $\mathbf{H_0}$ even though the hypothesis is true. Commonly used values for $\alpha$ are 0.01, 0.05 and 0.1.

Let us consider the following example: Suppose we roll an unbiased die 20 times and record the values of the die after each roll. The die should land on each value about the same amount of time.
We set "$\mathbf{H_0}$ : the die is not biased towards any value" and $\alpha = 0.01$.
Most of the times the experiment will conclude with $\mathbf{H_0}$ not being rejected, but it would still be possible that one does reject the null hypothesis in some cases. Imagine during the experiment the die lands 9 times on the value 5, which is not impossible. The probability of this happening is:

$$\begin{aligned} p &= \frac{\binom{20}{9} \times 5^{11}}{6^{20}} \\ &\approx 0,0022 \end{aligned}$$

This is smaller than $\alpha$ and so we would reject $\mathbf{H_0}$ even though the die is unbiased, giving us an error of type 1, showing again that failing or passing one test is not sufficient to judge a sequence or a generator.

In most tests, the method will be the same as in this example, we suppose the null hypothesis to be true and then calculate a certain probability which helps us to decide if we should reject $\mathbf{H_0}$ or not.
This probability is called the *p-value* and most of the time, we will reject $\mathbf{H_0}$ if the p-value is smaller than the significance level $\alpha$. How the p-value is computed and what the null hypothesis states always depends on the test that is being regarded. In the following, we will explain the procedure of different tests and apply them on both the AHS generator and the LCG.

# 2 Statistical Tests

## 2.1 Tests for Distribution

In this section, three tests for distribution, so called goodness-of-fit tests, are described. First we start with the Chi-square Test before moving on to the Kolmogorov-Smirnov Test and then finishing with the Cramér-von Mises Test.

### 2.1.1 The Chi-Squared Test

There are two main types of the Chi-square Test, the test of independence and the goodness-of-fit test. In this section we are interested in the goodness-of-fit test. Generally, this test is based on comparing the number of occurrences of some events with the number of occurrences of the same events, that would be expected under the null hypothesis to be tested.
More precisely, we first take a sample of $n$ observations to be tested and establish a null hypothesis, which claims that the observed data perfectly fits a special distribution (which has to be fixed). Then the sample has to be divided up into $k$ intervals, so that we get for each interval the number of occurrences of the corresponding event. Moreover, the expected numbers of values in each interval are calculated, assuming the null hypothesis is true. Then the chi-square value is given by

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where $k$ is the number of intervals, $E_i$ are the expected and $O_i$ the observed numbers of values in the $i^{th}$ interval.
Now this calculated chi-square value has to be compared with the critical values of the distribution table. For this we need to consider the significance level $\alpha$ and the degree of freedom (denoted by the symbol $df$ or $\nu$) which is given by $k-1$. This can be explained by the fact that if we know how many numbers are in the $k-1$ intervals, the number of values in the $k^{th}$ interval can easily be calculated, knowing the size of the sample. Now we have two ways to determine if these results are statistically significant or not, using the degree of freedom and the significance level.
The first method consists in comparing the chi-square value with the critical values (which can easily be determined by looking at a chi-square distribution table). If the calculated chi-square value is now superior to the critical value, the null hypothesis has to be rejected, if it is inferior, the null hypothesis can not be rejected.
The other method consists in calculating the p-value and comparing this p-value to the significance level $\alpha$. However, since it is beyond our knowledge to calculate the exact value of the p-value, we can simply look at a chi-square distribution table to get an idea in which area the p-value lies. As already mentioned above, if the p-value is strictly inferior to $\alpha$, the null hypothesis has to be rejected, if the p-value is superior to $\alpha$, the null hypothesis can not be rejected.

#### Example

Let us consider a concrete example for better understanding/visualization: Let us consider the experience described in the book "'The Art of Computer

Programming Vol 2"' by Knuth, where two dice are rolled 144 times and the following values for the sum of both are observed:

| i | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $O_i$ | 2 | 6 | 10 | 16 | 18 | 32 | 20 | 13 | 16 | 9 | 2 |

As we consider the sum of the two dice, we see that we have 11 intervals. Since we consider the dice to be unbiased, we suppose that any value (from 1 to 6) has the same probability ($\frac{1}{6}$) to be thrown for the two dice. Let $X$ be the variable which represents the sum of both dice, so we get that:

$$\begin{array}{lll} P(X=2)=\frac{1}{36} & P(X=3)=\frac{2}{36}=\frac{1}{18} & P(X=4)=\frac{3}{36}=\frac{1}{12} \\ P(X=5)=\frac{4}{36}=\frac{1}{9} & P(X=6)=\frac{5}{36} & P(X=7)=\frac{6}{36}=\frac{1}{6} \\ P(X=8)=\frac{5}{36} & P(X=9)=\frac{4}{36}=\frac{1}{9} & P(X=10)=\frac{3}{36}=\frac{1}{12} \\ P(X=11)=\frac{2}{36}=\frac{1}{18} & P(X=12)=\frac{1}{36} & \end{array}$$

This implies that we get the following expected results:

| i | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $E_i$ | 4 | 8 | 12 | 16 | 20 | 24 | 20 | 16 | 12 | 8 | 4 |

So that the chi-square value is the following:

$$\begin{aligned} \chi^2 &= \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \\ &= \frac{4}{4} + \frac{4}{8} + \frac{4}{12} + \frac{0}{16} + \frac{4}{20} + \frac{64}{24} + \frac{0}{20} + \frac{9}{16} + \frac{16}{12} + \frac{1}{8} + \frac{4}{4} \\ &= 1 + \frac{1}{2} + \frac{1}{3} + 0 + \frac{1}{5} + \frac{8}{3} + 0 + \frac{9}{16} + \frac{4}{3} + \frac{1}{8} + 1 \\ &= \frac{1853}{240} \\ &\approx 7,72083 \end{aligned}$$

Since we have 11 intervals, we have $\nu = 11 - 1 = 10$ degree of freedoms and if we take a significance level of $\alpha = 0.05$ we see that the critical value equals 18.31 which is superior to the chi-square value which implies that we can not rejected the hypothesis that the dice are unbiased.
Similarly if we calculate the p-value knowing that $\chi^2 = 7.72$ and $\nu = 10$, we see by looking at the distribution table that the p-value lies between 0.5 and 0.75. Using a p-value calculator , we get that $p = 0.656$ and since $0.656 > 0.05$, we get the same conclusion that we can not reject the null hypothesis. However it is important to emphasize the fact that this does not mean that the dice are not biased, it does just mean that we can not reject the null hypothesis.

### Application of the test

Since we are interested in testing the PRNGs for uniformly distribution on the interval $[0, 1)$, we establish the following null hypothesis:

"$\mathbf{H_0}$ : The numbers are uniformly distributed on the interval $[0, 1)$"
Then we first divide the sequence into 10 intervals, and consider for simplicity reasons the following 10 intervals: $[0, 0.1), [0.1, 0.2), ..., [0.9, 1.0)$.
Then the numbers of data points in each interval has to be counted and the chi-square value is calculated. Since we are talking about uniform distribution, the expected numbers are given by the division $\lfloor \frac{n}{10} \rfloor$ for each interval. Finally we get, for a significance level of $\alpha = 0.05$, a critical value of $z_c = 16.919$. That means that we will reject the null hypothesis if $\chi^2 > z_c$ and otherwise we will not reject it.

### 2.1.2 The Kolmogorov-Smirnov Test

The goal of the Kolmogorov-Smirnov Test is to compare the distribution of a randomly generated sample with the distribution it should theoretically have.
For this test, the null hypothesis is defined as "$\mathbf{H_0}$ : the distribution of the sample is uniform".
Let us say that we have a sample of size $n : \sigma = \{X_1, X_2, \ldots, X_n\}$. In order to quantify the distributions we have two functions:

1. $F(x) = \Pr(X \leq x)$, for a random variable $X$, which is the cumulative distribution function (CDF);
2. $F_n(x) = \dfrac{\#\{X \in \sigma | X \leq x\}}{n}$,called the empirical cumulative distribution function (ECDF).

Since the theoretical distribution, which the generators should produce, is uniform, $F(x) = x \ \ \forall\, x \in [0, 1]$.
To compare these two functions, Andrey Kolmogorov came up with the following statistic, which represents the biggest deviation between them:

$$D_n = \sup_x |F_n(x) - F(x)| \tag{1}$$

The distribution of $D_n$ has been widely studied, at first by Andrey Kolmogorov (see also [5] [7]) who found the limiting form of the distribution function $D_n$:

$$\lim_{n\to\infty} \Pr(\sqrt{n} D_n \leq x) = 1 - 2\sum_{i=1}^{\infty} (-1)^{i-1} e^{-2i^2x^2} = \frac{\sqrt{2\pi}}{x} \sum_{i=1}^{\infty} e^{\frac{-(2i-1)^2\pi^2}{8x^2}} \tag{2}$$

N. V. Smirnov considered Kolmogorov's $D_n$ differently, by having two different statistics $D_n^+$ and $D_n^-$ :

$$D_n^+ = \sup_x (F_n(x) - F(x)) \tag{3}$$

$$D_n^- = \sup_x (F(x) - F_n(x)) \tag{4}$$

Moreover, Smirnov published (see [10]) a table for estimating the value of these statistics. He also found the following formula (see also [4] [11]):

$$\Pr(\sqrt{n}D_n^+ \leq \frac{t}{\sqrt{n}}) = 1 - \frac{t}{n^n} \sum_{t<k\leq n} \binom{n}{k} (k-t)^k (t+n-k)^{n-k-1} \quad (5)$$

Finally, in their article (see [7]), the authors provide an algorithm implemented in C to evaluate precisely and efficiently Kolmogorov's $D_n$ distribution.

### 2.1.3 Cramér-von Mises Test

Cramér-von Mises Test is similar to the Kolmogorov-Smirnov Test, in the sense that the objective is to analyse the distribution of a random sequence. Thus, we will have the same functions $F$ and $F_n$, representing the cumulative distribution function and the empirical distribution function respectively. Also, we admit the same null hypothesis "$\mathbf{H_0}$ : the distribution of the random sample is uniform".
However, in the case of the Cramér-von Mises Test, the statistic is defined as

$$\omega_n^2 = n \int_{-\infty}^{\infty} (F_n(x) - F(x))^2 \, dF(x)$$

The study of the distribution of $\omega_n^2$ involves resolutions and formulas beyond our knowledge but is explained in the following article [2], which also contains a table of values $x$ such that $\Pr(w_n^2 \leq x) = p$, for different values of $p$ and $n$.

## 2.2 Tests for Independence

Since the pseudo random numbers of a PRNG should appear to be independent and identically distributed, two tests for independence are described in this section. First we start with the Runs Test, before moving on to the Spearman's rank Correlation Test

### 2.2.1 The Runs Test

The Runs Test is a statistical procedure used to check if a sequence of data has been obtained from a random process. To do so the test uses runs of data, which can be defined as follow:
*A run is a sequence of a certain type preceded and followed by occurrences of the alternate type or by no events at all.*
Of course, there are different ways of defining the event being used in the test, as long as it produces a dichotomous sequence of values. In our version we will use the median of the sequence. Other possibilities would have been to use the mean-value of the sequence or to just set a cut-off value ourselves.

(Reminder: If we would rearrange the sequence in an ascending or descending list of numbers, the median would be the middle number of this list. In case of an even amount of numbers in the sequence, the middle pair is added together and divided by two to determine the median.)
We then define a run as being a series of consecutive numbers below or above the median. (If a number equals the median we will include it in the group of superior values.)

Take for example the following sequence:

$$15; 88; 45; 75; 21; 6; 92; 80; 56; 33; 11; 3; 19; 41$$

By rearranging this sequence we can easily recognize that the median is $\frac{33+41}{2} = 37$:

$$\underbrace{3; 6; 11; 15; 19; 21;}_{\text{7 numbers}} \underbrace{33; 41;}_{\text{middle}} \underbrace{45; 56; 75; 80; 88; 92}_{\text{7 numbers}}$$

The next step is to count the number of runs as well as the amount of numbers below and above the median:

$$\underbrace{15;}_{1} \underbrace{88; 45; 75;}_{2} \underbrace{21; 6;}_{3} \underbrace{92; 80; 56}_{4} \underbrace{33; 11; 3; 19;}_{5} \underbrace{41}_{6}$$

Thus we have, runs $r = 6$, numbers above the median $n_1 = 7$, numbers below the median $n_2 = 7$.

Now we need to calculate the test statistic which we will compare to the data expected to be obtained if the null hypothesis is valid. The null hypothesis in this case is:
"$\mathbf{H_0}$ : The occurence of pattern for the two types of the observations is determined by a random process." (The two types of the observations here being either numbers below or above the median.)
It is important to note that generally two different methods are used to calculate the test statistic depending on the size of the sequence.
For smaller samples the critical value can be retrieved from a table by Swed and Eisenhart ([12]). Hence from the table for $n_1 = n_2 = 7$, the upper critical value (Uc) is 3 and the lower critical value (Lc) is 12. Meaning that we would reject the null hypothesis if $r \leq Uc$ or $r \geq Lc$, which isn't the case for our sequence above as $3 < r = 6 < 12$.

Let us now move to the method for bigger samples as this is going to be more practical to test bigger sequences generated by random number generators. For this we will use an approximation technique.

The test-statistic $z$ is calculated by an approximation of the normal distribution and with the following formula:

$$z = \frac{r_O - r_E}{s_R}$$

where $r_O$ is the number of observed runs, $r_E$ is the number of expected runs and $s_R$ is the standard deviation of the number of runs.
We have:

$$r_E = \frac{2n_1n_2}{n_1 + n_2} + 1$$

$$s_R = \sqrt{\frac{(2n_1n_2)(2n_1n_2 - n_1 - n_2)}{(n_1 + n_2)^2(n_1 + n_2 + 1)}}$$

Now the final step is to compare the obtained test-statistic with the critical value for larger samples. For a significance level at 5% the critical value is $z_c = 1,96$. We reject the null hypothesis if $|z| > z_c$. ($-z > z_c$ means there are too few runs and there are too many runs if $z > z_c$.)

### 2.2.2 Spearman's Rank Correlation Coefficient

To answer the question whether there is a relation between two variables or not, a correlation coefficient can be calculated.
The correlation coefficient gives us the direction and the strength of the relationship between two variables. It takes values between $-1$ and 1, where values close to 1 represent a strong positive correlation whereas values close to $-1$ represent a strong negative correlation. Zero correlation however does not imply independence, but it is a necessary condition for independence.
In fact, there are two major methods for correlation analyses. The parametric correlation, also known as Pearson Correlation, determines the strength and the direction of a linear relationship between two variables whereas the non-parametric correlation, also known as Spearman's rank Correlation, measures the strength and the direction of the monotonic relationship.
To be able to talk about the Spearman's rank correlation coefficient, it is important to first understand Pearson's correlation.

#### Pearson's correlation coefficient formula

If we have a sample of size $n$, the Pearson's correlation coefficient is given by:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ are the means of the variables.
By rearranging we get the following formulas:

$$\begin{aligned} r &= \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}\sqrt{n\sum_{i=1}^{n} y_i^2 - (\sum_{i=1}^{n} y_i)^2}} \\ &= \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}\sqrt{\sum_{i=1}^{n} y_i^2 - n\bar{y}^2}} \end{aligned}$$

**Spearman's rank correlation coefficient**

As already mentioned above, the Spearman's rank correlation is the non-parametric version of the Pearson's correlation, it measures the strength and direction of the association between two ranked variables. Most often, this correlation test is used when the requirements of the parametric correlation test (continuous data, linearity, normally distributed variables) are not satisfied. The only requirement is, that the data can be ranked.
Since, monotonic relation instead of linear relation is tested, which is less "restrictive", the Spearman's rank correlation coefficient is able to discover dependencies which the Pearson's correlation coefficient does not recognize. (A positive monotonic relation means that if one variable increases, then the other increases too; a negative monotonic relation means that if one variables increases, the other one decreases)
The method consists in finding ranks $x_i$ and $y_i$ for each pair of $X_i$ and $Y_i$ values and then run Pearson's correlation on these ranks instead of on the raw data.
To do so, we first order the values of the $X$ variable from the smallest to the biggest value and assign to them the integer corresponding to their position (starting with 1). If all the values are different from each other, then the rank $x_i$ of $X_i$ equals the integer of the corresponding position. If multiple values are equal, we say that we have tied ranks, in that case, the rank does not equal the position of the value $X_i$ but we calculate the rank by counting the positions of the multiple values, summing the position integers together and then dividing this sum by how many values are tied.
If for example the positions 4 and 5 correspond to equal values then we sum those positions $4 + 5 = 9$ and divide it through the number of tied ranks $\frac{4+5}{2} = 4.5$. The same procedure is applied to the $Y$ variable. Let's consider an example for a better understanding:

| $X_i$ | 56 | 75 | 45 | 71 | **61** | 64 | 58 | 80 | 76 | **61** |
|---|---|---|---|---|---|---|---|---|---|---|
| Positions $X_i$ | 2 | 8 | 1 | 7 | 4 | 6 | 3 | 10 | 9 | 5 |
| $x_i(ranks)$ | 2 | 8 | 1 | 7 | 4.5 | 6 | 3 | 10 | 9 | 4.5 |
| $Y_i$ | **66** | 70 | **66** | 65 | 56 | **66** | 77 | 67 | 63 | 45 |
| Positions $Y_i$ | 5 | 9 | 6 | 4 | 2 | 7 | 10 | 8 | 3 | 1 |
| $y_i(ranks)$ | 6 | 9 | 6 | 4 | 2 | 6 | 10 | 8 | 3 | 1 |

If there are tied ranks in any of the two variables, we apply, as already mentioned, the Person's Correlation Coefficient formula to the ranks. However if there are no tied ranks (i.e $x_i \neq x_j$ for all $i \neq j$), the $x_i$'s and the $y_i$'s both consists of the integers between 1 and $n$, so we get the following formula for the Spearman's rank correlation coefficient:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}$$

where $d_i = x_i - y_i$ is the difference in ranks of the $i^{th}$ element for $i \in \{1, 2, ..., n\}$

### Application of the test

To apply the Spearman's rank correlation test to a PRNG, we first have to divide the sequence of pseudo-random numbers into two, in order to get values for the two variables $X$ and $Y$.
To do so, we define all the even iterations $U_0, U_2, ..., U_{n-2}$ to come from distribution $X$ and all odd iterations $U_1, U_3, ..., U_{n-1}$ to come from distribution $Y$. Now ranks has to be assigned to each $X_i$ based on its value. The smallest $X_i$ will get rank $x_i = 1$, the second smallest will get rank $x_i = 2$ and so on. In the case of tied ranks, so if two or more $X_i$ share the same value, we take the average of their numeric position, just as explained above. The same procedure is applied for the values of $Y$. Then we established the following null hypothesis:
"$\mathbf{H_0}$ : There is no correlation between the variables $X$ and $Y$."
Now we know that if we get $\rho = \pm 1$, a monotonic function would perfectly fit the relation between $X$ and $Y$ and if $\rho = 0$, there is no monotonic function between the variables. To be able to interpret all the other values for $\rho$, (e.g. $\rho = 0.7$ or $\rho = -0.4$), we can use the Fisher Transformation to make $\rho$ comparable to the normal distribution. The Fisher Transformation is defined by:

$$F(r) = \frac{1}{2} \ln \frac{1+r}{1-r} = \tanh^{-1}(r)$$

where $r = \rho$ is the Spearman's rank correlation coefficient. Under the null hypothesis, we will then get:

$$\sqrt{\frac{n-3}{1.06}} F(r) \sim N(0, 1)$$

So if we define the z-score

$$z = \sqrt{\frac{n-3}{1.06}} F(r)$$

which approximately follows the normal distribution under the null hypothesis, we can easily determine the critical values and the p-value. So similarly as for the run test, we get, for a significance level of $\alpha = 0.05$, a critical value of $z_c = 1.96$. So we reject the null hypothesis if $|z| > z_c$

# 3 The AHS random number generator

The AHS generator is a quite new random number generator. It was developed in 2006 by Alain H. SCHUMACHER, after whom it is also named. The main idea of this RNG is to simulate the process of flipping an unbiased coin. To produce a random number sequence of $n$ bits, we have to flip the coin $n$ times, so that each coin flip produces one bit of the sequence. Since flipping an unbiased coin follows a Bernoulli trial with

$$\Pr(\text{"Tail"}) = \Pr(\text{"Head"}) = \frac{1}{2},$$

we see that we will get independent and identically distributed random numbers.
However, as we already pointed out, computers only understand deterministic processes, that is why a lot of different sources of randomness have to be combined in order to perfectly simulate the process of coin flipping. To get a general idea of the algorithm without going into too many details, a so-called Bit fishing table has to be introduced, from where we will get the bits of the sequence. The bit fishing table is a table consisting of equally distributed 0's and 1's. By combining different sources of randomness, the position of the bit is determined, then the bit is extracted from the bit fishing table and added to the sequence of random numbers. For a sequence of $n$ bits, this process is repeated $n$ times.

## 3.1 Transformation of AHS sequence

To apply the five tests explained in section 2, we needed a sequence of numbers uniformly distributed over $(0, 1)$. However, the AHS generator produces 8-bit binary numbers. Once converted in decimal, we get integers between $-128$ and 127 included. We could get numbers over $(0, 1)$ by adding 128 to every number and dividing the result by 256 but we got very poor results when testing such sequence with the Chi-square Test, for example.
Instead of having only 256 different possible numbers, we would like to have $256^2$ numbers. To do so, we combine two different 8-bit binary numbers by concatenation. For example, let's combine 10110101 (181 in decimal) and 11000100 (196 in decimal) to get 1011010111000100 (46532 in decimal).
It allows us to have decimal integers from 0 to $256^2 - 1$ and then divide them by $256^2$.
Two integers, let say 181 and 196 are combined as such : $181 \times 256 + 196 = 46532$. The effect of multiplying the first integer by 256 is to append eight zeros to its binary representation which are then replaced by the binary representation of the second integer when adding both terms.

## 3.2 Testing the AHS-RNG

Since we are interested in testing the AHS generator, we will now apply the statistical tests explained in section 2 to the AHS generator. To do so, we have implemented a Python program (see A.4) which does the computations and got the following results:

data.txt

```
AHS

Test name                   statistic                   p-value
Kolmogorov-Smirnov test     0.0006108642578125378       0.8660095975687303
Chi-square test             1.8015625                   0.9942310385555476
Spearman test               0.0010989930553995848       0.4464164136126185
Runs test                   -1.3798795964816928         0.1676237197407061
Cramér-von Mises test       0.03165801592767239         0.9704088228493296
```

The only thing that has to be done, is the interpretation of those results. For the interpretation of each test, the same significance level $\alpha = 0.05$ is used.

### Kolmogorov-Smirnov test

For this test, the statistical value corresponds to the KS test statistics $D = 0,0006$, and since we have a sequence of 960000 data points we will get the following critical value:

$$z_c = \frac{1.36}{\sqrt{n}} = \frac{1.36}{\sqrt{960000}} \approx 0,001388$$

And since $0.0006 < 0.001388$, we can not reject the null hypothesis which is also confirmed by the p-value of $0.866 > 0.05$. So we conclude that the AHS generator will pass this test.

### Chi-square test

For this test, the statistical value corresponds to the chi-square value, so we see that we will get $\chi^2 \approx 1.802$. As mentioned in the section 2.1.1 the critical value is $z_c = 16.919$ and since $1.802 < 16.919$, the null hypothesis will not be rejected. This is also confirmed by the fact that we get a p-value of $0.994 > 0.05$. So we can conclude that the AHS generator will pass this test.

**Spearman test**

For this test, the statistical value corresponds to the Spearman's rank correlation coefficient $\rho = 0.0011$. As mentioned in section 2.2.2, we will have to apply the Fisher Transformation to make $\rho$ comparable to the normal distribution. So we get:

$$F(\rho) = tanh^{-1}(\rho) \approx 0.0011,$$

which implies that

$$z = \sqrt{\frac{n-3}{1.06}}F(\rho) = \sqrt{\frac{960000-3}{1.06}}0.0011 \approx 1.046$$

And since $z_c = 1.96$, and we have that $1.0646 < 1.96$, we see that we do not reject the null hypothesis, which is again also confirmed by the p-value of $0.446 > 0.05$. So we get that the AHS generator passes this test.

**Runs test**

For this test, the statistical value corresponds to the test-statistic z, so we have that $z = -1.38$. And since we know from section 2.2.1 that we have the same critical value as for the Spearman's rank test, $z_c = 1.96$, we see that $|-1,38| < 1.96$, which implies that we will not reject the null hypothesis. This is again confirmed by the p-value of $0.167 > 0.05$. So we can conclude that the AHS generator passes this test too.

**Cramér-von Mises test**

For this test, the statistical value corresponds to the Cramér-von Mieses statistics, so we have $\omega_n^2 = 0.0316$. And since we get a p-value of 0.97, we see that $0.97 > 0.05$. So we can conclude that the AHS generator also passes this test

## 3.3 Testing the LCG

As already mentioned in the section 1.2, we will compare the AHS generator to the LCG with the following parameters:

- a = 1664525
- c = 1013904223
- m = $2^{32}$
- $x_0 = 0$

That is why we generated a sequence of 1920000 numbers of this LCG and applied the same tests to this sequence as we applied to the sequence of the AHS generator and we got the following results:

data.txt

```
LCG

Test name                   statistic                   p-value
Kolmogorov-Smirnov test     0.0010912617830869387       0.020641421075878887
Chi-square test             14.902028176026992          0.09366248811839077
Spearman test               -0.00062095935292248        0.5429137952913983
Runs test                   0.3846596172236302          0.7004896180591527
Cramér-von Mises test       0.4851985395651715          0.043442056008061725
```

If we do the interpretations in the same way as we did it for the AHS generator, with a significance level of $\alpha = 0.05$ we get:

### Kolmogorov-Smirnov test

Here we see that $D = 0,001$, and since we have a sequence of 1920000 data points we will get the following critical value:

$$z_c = \frac{1.36}{\sqrt{n}} = \frac{1.36}{\sqrt{1920000}} \approx 0,000981$$

And since $0.001 > 0.000981$, we will reject the null hypothesis which is also confirmed by the p-value of $0.021 < 0.05$. So we conclude that the LCG fails this test.

### Chi-square test

For this test we see that $\chi^2 \approx 14.9$. And since $z_c = 16.919$ and $14.9 < 16.919$, the null hypothesis will not be rejected. This is also confirmed by the fact that we get a p-value of $0.09 > 0.05$. So we can conclude that the LCG will pass this test.

### Spearman test

For this test, we get $\rho = -0.00062$. Calculating the Fisher Transformation, we get:

$$F(\rho) = tanh^{-1}(\rho) \approx -0.00062,$$

which implies that

$$z = \sqrt{\frac{n-3}{1.06}} F(\rho) = \sqrt{\frac{1920000-3}{1.06}}(-0.00062) \approx -0.834$$

And since $z_c = 1.96$, and we have that $|-0.834| < 1.96$, we see that we do not reject the null hypothesis, which is again also confirmed by the p-value of $0.543 > 0.05$. So we get that the LCG passes this test.

#### Runs test

For this test, we have that $z = 0.385$. And since $z_c = 1.96$, we see that $0.385 < 1.96$, which implies that we will not reject the null hypothesis. This is again confirmed by the p-value of $0.70 > 0.05$. So we can conclude that the LCG passes this test too.

#### Cramér-von Mises test

Here we see that $\omega_n^2 = 0.485$ and we get a p-value of 0.043 and since $0.043 < 0.05$, we can conclude that the LCG does not pass this test.

### 3.4 Comparing the AHS generator to the LCG

As we can see from the two previous sections, the AHS generator passes all 5 tests whereas the LCG only passes 3 of the 5 tests. The LCG fails the Kolmogorov-Smirnov and the Cramer-von Mises test, which are both tests for uniform distribution. Another test for uniform distribution is the chi-square test. For this test we see that the test statistic is very close to the critical value and that the p-value is therefore close to zero, so even if the LCG passes this test, it does not pass it as good as the AHS generator does (which has a p-value of almost 1 for this test). However, the tests for independence are passed by the two generators without any problems. So we can conclude that the AHS generator seems to be a better generator in terms of uniform distribution than the LCG, while we can not really compare the respective independence of their elements, since we have only done two tests for independence which were both passed by the two generators.

## 3.5 Test Suites

As a "bad" PRNG can mess up the analysis of a research, it is important for the researcher to know if the PRNG is suited for his work.
Of course, the best option for this is to run tests on the PRNG. However, not every researcher might be advanced enough in this area to run the right tests or to make the right interpretations out of it. Especially as so many tests exist, it is impossible to find a PRNG that does not fail any test. So, the aim is to use PRNGs that do not fail any "reasonable" tests.
But again, not every researcher is going to have the knowledge of which tests will fall into that category.
This is why one solution is to trust experts to put together a battery of tests, that a "good'" PRNG should not fail. The very first battery of statistical tests for uniform RNGs was proposed by Donald KNUTH in the first edition of his book 'The Art of Computer Programming' in 1968. In 1995, George MARSAGLIA programmed the DIEHARD tests which have been widely used in the testing of PRNGs for years. However, it is only constituted of about 15 tests and it is not extensible as it does not allow new tests to be added. To fix theses flaws Pierre L'ECUYER and Richard SIMARD programmed the software library TestU01 in 2007.

### 3.5.1 TestU01

TestU01 is a software library providing a multitude of utilities for statistical testing of uniform RNGs. It provides many implementations of different PRNGs and a large variety of statistical tests.
However, the most interesting part for us here, is that it also includes different test suites:

- Small Crush (consisting of 10 tests)
- Crush (96 tests)
- Big Crush (160 tests)

The usual procedure is to run Small Crush on the PRNG and if there are no failures, one of the bigger suites is applied. This is also what we did for the AHS-generator. We applied the test to the generator by taking 32-bit binary numbers since this is what is advised. To be more precise, we defined a generator that loops through a sequence of 32-bit binary numbers generated by the AHS. (This is of course not ideal compared to taking a long enough sequence to run the test suite on, however for our purpose this will be sufficient, as the test results show.) For the transformation to 32-bit binary integers, we used the same principle as in 3.1.

Here is the summary (see appendix A.5 for the full breakdown) of the results:

data.txt

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        AHS
 Number of statistics:  15
 Total CPU time:   00:00:09.66

 All tests were passed
```

As all tests are passed, the next step would be to run Crush on the generator. However for this, the sequence should ideally be constituted of $2^{32}$ elements. Of course, if we generate such a long sequence by looping through a smaller sequence, the test results for uniform distribution will not be great.
Because of some technical difficulties and lack of time, we will not be doing this in this report, but this would be the next step if this research would be expanded.

We did also run the Small Crush battery on the LCG to verify if the intuition we got from our previous tests about uniform distribution for the LCG was right. Here is the summary of the results:

data.txt

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        LCG
 Number of statistics:  15
 Total CPU time:   00:00:09.24
 The following tests gave p-values outside [0.001, 1]:
 eps  means a value < 1.0e-300:
 eps1 means a value < 1.0e-15:

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  3  Gap                               eps
  4  SimpPoker                         eps
  5  CouponCollector                   eps
  7  WeightDistrib                     eps
  8  MatrixRank                        eps
  9  HammingIndep                      eps
 10  RandomWalk1 H                     eps
```

```
 10  RandomWalk1 M                    eps
 10  RandomWalk1 J                   4.0e-5
 10  RandomWalk1 R                    eps
 10  RandomWalk1 C                    eps
-----------------------------------------------
All other tests were passed
```

---

As we can see, the LCG fails on 12 of the 15 calculated statistics, reinforcing that the sequence generated by the LCG does not fare well in tests for uniform distribution.

# 4 Conclusion

In conclusion, we can say that the AHS generator seems to be quite a good generator, since it has passed all the tests to which we have applied it. Especially in terms of uniform distribution, this generator seems to be very strong. Also compared to the LCG, the AHS generator seems to be a lot better. Another interesting step would be to compare the AHS generator with a well-known and very good generator, which could unfortunately not be done in this project because of lack of time.

## References


[1] Mohamad Bujang and Fatin Sapri. An application of the runs test to test for randomness of observations obtained from a clinical survey in an ordered population. *The Malaysian Journal of Medical Sciences*, 25(4):146–151, 2018.

[2] Sandor Csorgo and Julian J. Faraway. The exact and asymptotic distributions of cramer-von mises statistics. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):221–234, 1996.

[3] James E Gentle. *Random number generation and Monte Carlo methods.* Statistics and computing 448149. Springer, New York, 2nd ed.. edition, 2003.

[4] Donald Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition.* 3rd edition. edition, 1997.

[5] A. KOLMOGOROV. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 4:83–91, 1933.

[6] Liang Li. Testing several types of random number generator. 2012.

[7] G. Marsaglia, W.W. Tsang, and J. Wang. Evaluating kolmogorov's distribution. *Journal of Statistical Software*, 8:1–4, 2003.

[8] B. D. Mccullough. A review of testu01. *Journal of Applied Econometrics*, 21(5):677–682, 2006.

[9] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, 2001.

[10] N. Smirnov. Table for estimating the goodness of fit of empirical distributions. *The Annals of Mathematical Statistics*, 19(2):279–281, 1948.

[11] N. V. Smirnov. Approximate laws of distribution of random variables from empirical data. *Uspekhi Mat. Nauk*, (10):179–206, 1944.

[12] Frieda S. Swed and C. Eisenhart. Tables for testing randomness of grouping in a sequence of alternatives. *The Annals of Mathematical Statistics*, 14(1):66–87, 1943.

# Appendices

## A Code

In this section, the descriptions and the code can be found of the programs which we implemented by ourselves.

### A.1 Chi-square Program

Since we are interested in testing the PRNGs for uniformly distribution on the interval $[0, 1)$, we have written a program in Python which does exactly this test:
First of all, the user is asked to enter the name of the file of the sequence he wants to test. Then the real process begins, for simplicity reasons we consider the following 10 intervals: $[0, 0.1), [0.1, 0.2), ..., [0.9, 1.0)$. Then the program loops though the n data points of the sample and counts the number of data points in each interval, which is done in the function called `subdivisions`. Next the function `chi_square_val` is called, where the chi-square value is calculated. And finally the calculated chi-square value is compared to the critical values to determine whether the null hypothesis has to be rejected or accepted. This is done using the function `significance test`, where we took the critical values from a chi-square distribution table using the degree of freedom $\nu = 9$ (since we have $k = 10$ intervals).

```
def chi_square_val(dataset, n):
    """ Null hypothesis: Numbers are distributed uniformly on
    [0, 1)
        dataset = dictionnary with 10 intervals
        n = number of datapoints
        output: chi-squared value
        """
    chi_sq_value = 0
    e_val = n/10
    for o_val in dataset:
        chi_sq_value += pow((e_val - dataset[o_val]), 2)/e_val
    return chi_sq_value

def subdivisions(file):
    """ output: dictionnary with 10 intervals containing the
    number of datapoints of each subdivision"""
    subdiv = {"1":  0,
              "2":  0,
              "3":  0,
              "4":  0,
              "5":  0,
              "6":  0,
              "7":  0,
              "8":  0,
              "9":  0,
```

```
                "10": 0
            }

    file.seek(0)
    data_points = file.readlines()

    for n in data_points:
        n = float(n)
        if n < 0.1:
            subdiv["1"] += 1
        elif n < 0.2:
            subdiv["2"] += 1
        elif n < 0.3:
            subdiv["3"] += 1
        elif n < 0.4:
            subdiv["4"] += 1
        elif n < 0.5:
            subdiv["5"] += 1
        elif n < 0.6:
            subdiv["6"] += 1
        elif n < 0.7:
            subdiv["7"] += 1
        elif n < 0.8:
            subdiv["8"] += 1
        elif n < 0.9:
            subdiv["9"] += 1
        elif n < 1.0:
            subdiv["10"] += 1
    return subdiv

def significiance_test(chi_sq_val, alpha = 0.05):
    """since we have 10 intervals, df= 10-1=9; critical values
    are taken from a distribution table """
    critical_val = 16.919
    if chi_sq_val > critical_val:
        return "REJECT"
    else:
        return "DO NOT REJECT"

filename = input("Enter the filename of the sequence you want
    to work on: ")
alpha = 0.05
print('Significance level: 0.05')
f = open(filename + '.txt', 'r')
s = 0
for line in f:
    s += 1
print(s)
t = chi_square_val(subdivisions(f), s)
print("chi squared value: ", t)
print(significiance_test(t, alpha))
```

## A.2 Kolmogorov-Smirnov Program

We applied the test in two ways, both are implemented in a Python file (ref). First, the user is ask to input the size $n$ of the sample and the probability $p$ he wants to work with. Then the program produces the graph of $F(x)$ and $F_n(x)$ together with an acceptance band at length $d_\alpha(n)$ of $F(x)$. $d_\alpha(n)$ is simply determined by dividing the corresponding critical value by $\sqrt{n}$. To illustrate why, here is an example: $\Pr(K_{10}^+ \leq 1.1658) = 0.95$ which means, in this case, that $\sqrt{10}\sup(F_{10}(x) - F(x)) \leq 1.1658$ (to succeed the test) $\Rightarrow \sup(F_{10}(x) - F(x)) \leq \frac{1.1658}{\sqrt{10}}$. Thus $F_{10}(x)$ has to be maximum at $\frac{1.1658}{\sqrt{10}}$ from $F(x)$. If the graph of $F_n(x)$ gets out of the acceptance band, then we reject the null hypothesis and the test failed.
Finally, the program computes the statistics $K_n^+$ and $K_n^-$ and compare them to the critical value computed by the program if $n > 32$ or given by the user otherwise.

```
import random, math
import numpy as np
import matplotlib.pyplot as plt
#from scipy.stats import kstest

filename = input("Input the name of the textfile you want to
    work on: ")
n = eval(input("Input the number of observations you want to
    work with: "))
alpha = eval(input("Among these values {0.01, 0.05, 0.25, 0.5,
    0.75, 0.95, 0.99}, pick up the level of significance (alpha
    ) you want to work with: "))

def build_sample():
    """Build an array of size n with pseudorandom numbers from
    the file filename."""
    with open(filename, 'r') as of:
        i = 0
        seq = []
        for line in of:
            seq.append(eval(line))
            i += 1
    sample = [random.randint(0, i) for j in range (n)]
    for j in range (n):
        sample[j] = seq[sample[j]]
    return sample

def F(x):
    """The theoretical distribution function : F(x) =
    Probability(X <= x)"""
    if x <= 1:
        return x
    else:
        return 1

```

```
30 def Fn(x):
31     """Fn(x) = (number of elements in the sample which are <= x
       ) / n
32     It's the empirical distribution function."""
33     count = 0
34     for a in sample:
35         if a <= x:
36             count += 1
37     return count/n
38 
39 def get_distribution_of_K():
40     """if n <= 35: Asks the user to input the corresponding
       value for K+ and K- from the table.
41     else: compute that value."""
42     if n < 31:
43         d = eval(input("Input the corresponding value for K+
       and K- from the table: "))
44     else:
45         g = lambda x: x - 1/(6*math.sqrt(n))
46         if alpha == 0.01:
47             yp = 0.07089
48         if alpha == 0.05:
49             yp = 0.1601
50         if alpha == 0.25:
51             yp = 0.3793
52         if alpha == 0.5:
53             yp = 0.5887
54         if alpha == 0.75:
55             yp = 0.8326
56         if alpha == 0.95:
57             yp = 1.2239
58         if alpha == 0.99:
59             yp = 1.5174
60         d = g(yp)
61     return d
62 
63 
64 ### 1) Builiding the sample to work on ###
65 sample = build_sample()
66 sample.sort()
67 
68 ### 2) Graph to compare distributions ###
69 max_dev = get_distribution_of_K()/math.sqrt(n) # It's the
       maximum deviation between distributions we will tolerate
       according to alpha
70 x = np.linspace(0,1.2, 100)
71 y1 = [F(a) for a in x]
72 y2 = [Fn(a) for a in x]
73 y3 = [F(a) - max_dev for a in x]
74 y4 = [F(a) + max_dev for a in x]
75 g1, = plt.plot(x, y1, linewidth=1.2)
76 g2, = plt.plot(x, y2, linewidth=1.2)
77 g3, = plt.plot(x, y3, 'k--', linewidth=1.2)
78 g4, = plt.plot(x, y4, 'k--', linewidth=1.2)
```

```
plt.legend([g1, g2, g3], [r"Theoretical distribution: $F(x)$",
    r"Empirical distribution: $F_n(x)$", r"Acceptance band: $F(
    x) \pm d_\alpha(n)$"])
plt.xlabel(r'$x$')
plt.ylabel(r"$y$")
plt.savefig(filename+"-KS-test.pdf")

### 3) We compute K+ and K- statistics ###
list_plus = [(j/n) - F(sample[j-1]) for j in range(1, n+1, 1)]
list_minus = [F(sample[j-1]) - (j-1)/n for j in range(1, n+1,
    1)]
K_plus = math.sqrt(n)*max(list_plus)
K_minus = math.sqrt(n)*max(list_minus)
print("K+ = ", K_plus)
print("K- = ", K_minus)
K = get_distribution_of_K()
if K_plus > K or K_minus > K:
    print("The test failed.")
else:
    print("The test succeeded.")
```

## A.3 Linear Congruent Generator Program

This program will generate a sequence of numbers between $[0, 1)$ by using the method of a linear congruent generator. First a variable called `LIMIT` is determined which represents the length of the sequence to be generated. Then the user is asked to enter the file name to store the sequence and to enter the different parameters of the LCG (seed, multiplier, increment and modulus). Then the algorithm opens the file and stores the generated numbers, obtained by the formula of the LCG: $X_n = (aX_{n-1} + c)mod(m)$, in that file.

```
LIMIT = 10**6

filename = input("Enter a filename to store the sequence ")
seed = int(eval(input("Input the seed ")))
a = int(eval(input("Input the multiplier ")))
c = int(eval(input("Input the increment ")))
m = int(eval(input("Input the modulus ")))

of = open(filename +".txt", 'w')
n = 1
of.write(str(seed/m))
of.write("\n")
x0 = seed
while n <= LIMIT:
    x1 = (a*x0+c)%m
    of.write(str(x1/m))
    of.write("\n")
    x0 = x1
    n += 1
of.close()
```

## A.4 Testing Program

This program will apply the following 5 tests to a sequence of any generator which generates numbers between $[0, 1)$:
KS-test, Chi-square test, Spearman's rank test, Runs test, Cramér-von Mieses test
First the user is asked to enter the name of the generator and to enter the name of the file containing the sequence to test. Then the program applies the five tests to this sequence and writes down the test statistic and the p-value for each tests in a new text file, which is done in the function called `gen_outputs`. Finally this text file is generated as an output.

```
from numpy.lib.npyio import genfromtxt
from scipy import stats
import numpy as np
from statsmodels.sandbox.stats.runs import runstest_1samp

def gen_output(title, stat, pvalue):
    of.write(title)
    n = 40-len(title)
    for i in range(n):
        of.write(' ')
    sstat = str(stat)
    of.write(sstat)
    n = 40-len(sstat)
    for i in range(n):
        of.write(' ')
    spavlue = str(pvalue)
    of.write(spavlue)
    n = 40-len(spavlue)
    for i in range(n):
        of.write(' ')
    of.write('\n')

gen_name = input("Input the name of the generator you are
    testing : ")
filename = input("Input the name of the file containing the
    sequence to test : ")
seq = genfromtxt(filename)

of = open('output-'+gen_name+'-test.txt', 'w')
of.write(gen_name+"\n \n")
of.write("Test name")
for i in range(31):
    of.write(" ")
of.write("statistic")
for i in range(40-len("statistic")):
    of.write(" ")
of.write("p-value")
for i in range(33):
    of.write(" ")
of.write("\n")

```

```
40
41 ### TESTING ###
42
43 # KS-test
44 stat, pvalue = stats.kstest(seq, 'uniform')
45 gen_output("Kolmogorov-Smirnov test", stat, pvalue)
46
47 # Chi-square test
48 freq_obs = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
49 for a in seq:
50     freq_obs[int(a*10)] += 1
51 stat, pvalue = stats.chisquare(freq_obs)
52 gen_output("Chi-square test", stat, pvalue)
53
54 # Spearman test
55 l1 = []
56 l2 = []
57 ltot = list(seq)
58 for i in range(0, len(ltot)-1, 2):
59     l1.append(ltot[i])
60     l2.append(ltot[i+1])
61 stat, pvalue = stats.spearmanr(l1, l2)
62 gen_output("Spearman test", stat, pvalue)
63
64 # Runs test
65 stat, pvalue = runstest_1samp(seq, cutoff='median', correction=
       False)
66 gen_output('Runs test', stat, pvalue)
67
68 # Cramér-von Mises test
69 res = stats.cramervonmises(seq, 'uniform')
70 gen_output('Cramér-von Mises test', res.statistic, res.pvalue)
71
72
73 of.close()
```

## A.5 TestU01-AHS-generator

The Small Crush battery applied to the AHS generator yielded the following:

data.txt

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                Starting SmallCrush
                Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx


HOST = gauss, Linux

AHS
```

```
smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1


      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051


-------------------------------------------------
Total expected number = NTotal observed number                :      28
p-value of test                       :    0.46


-----------------------------------------------
CPU time used                    :  00:00:01.60

Generator state:



Test sknuth_Collision calling smultin_Multinomial

HOST = gauss, Linux

AHS


smultin_Multinomial test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,   d = 65536,   t =  2,
       Sparse =   TRUE

       GenerCell = smultin_GenerCellSerial
       Number of cells = d^t =         4294967296
       Expected number per cell =  1 /  858.99346
       EColl = n^2 / 2k =  2910.383046
       Hashing =   TRUE

       Collision test,    Mu =      2909.2534,    Sigma =    53.8957

-----------------------------------------------
Test Results for Collisions

Expected number of collisions = Mu    :     2909.25
Observed number of collisions         :     2932
p-value of test                       :    0.34

-----------------------------
Total number of cells containing j balls

  j =  0                              :       4289970228
  j =  1                              :          4994137
```

```
  j =  2                              :             2930
  j =  3                              :                1
  j =  4                              :                0
  j =  5                              :                0

-----------------------------------------------
CPU time used                    :  00:00:01.78

Generator state:



HOST = gauss, Linux

AHS


sknuth_Gap test:
-----------------------------------------------
   N =  1,  n = 200000,  r = 22,   Alpha =        0,   Beta  = 0.00390625


-----------------------------------------------
Number of degrees of freedom          : 1114
Chi-square statistic                  : 1024.51
p-value of test                       :    0.97

-----------------------------------------------
CPU time used                    :  00:00:00.58

Generator state:



HOST = gauss, Linux

AHS


sknuth_SimpPoker test:
-----------------------------------------------
   N =  1,  n = 400000,  r = 24,   d =   64,   k =   64


-----------------------------------------------
Number of degrees of freedom          :   19
Chi-square statistic                  :   17.14
p-value of test                       :    0.58

-----------------------------------------------
CPU time used                    :  00:00:00.57
```

```
Generator state:



HOST = gauss, Linux

AHS


sknuth_CouponCollector test:
-----------------------------------------------
   N =  1,  n = 500000,  r = 26,   d =   16


-----------------------------------------------
Number of degrees of freedom          :   44
Chi-square statistic                  :   41.61
p-value of test                       :    0.57

-----------------------------------------------
CPU time used                    :  00:00:00.54

Generator state:



HOST = gauss, Linux

AHS


sknuth_MaxOft test:
-----------------------------------------------
   N =  1,  n = 2000000,  r =  0,   d = 100000,   t =  6

      Number of categories = 100000
      Expected number per category  = 20.00


-----------------------------------------------
Number of degrees of freedom          : 99999
Chi-square statistic                  : 1.01e+5
p-value of test                       :    0.07


-----------------------------------------------
Anderson-Darling statistic            :    0.98
p-value of test                       :    0.02


-----------------------------------------------
CPU time used                    :  00:00:01.05
```

```
Generator state:



HOST = gauss, Linux

AHS


svaria_WeightDistrib test:
-----------------------------------------------
   N =  1,  n = 200000,  r = 27,  k = 256,  Alpha =      0,  Beta =  0.125


-----------------------------------------------
Number of degrees of freedom          :   41
Chi-square statistic                  :   40.33
p-value of test                       :    0.50

-----------------------------------------------
CPU time used                    :  00:00:00.48

Generator state:



HOST = gauss, Linux

AHS


smarsa_MatrixRank test:
-----------------------------------------------
   N =  1,  n = 20000,  r = 20,    s = 10,    L = 60,    k = 60


-----------------------------------------------
Number of degrees of freedom          :    3
Chi-square statistic                  :    0.32
p-value of test                       :    0.96

-----------------------------------------------
CPU time used                    :  00:00:00.64

Generator state:



HOST = gauss, Linux
```

```
AHS


sstring_HammingIndep test:
-----------------------------------------------
   N =  1,  n = 500000,  r = 20,   s = 10,   L = 300,   d = 0



Counters with expected numbers >= 10
-----------------------------------------------
Number of degrees of freedom          : 2209
Chi-square statistic                  : 2204.25
p-value of test                       :    0.52

-----------------------------------------------
CPU time used                    :  00:00:00.67

Generator state:




HOST = gauss, Linux

AHS


swalk_RandomWalk1 test:
-----------------------------------------------
   N =  1,  n = 1000000,  r =  0,   s = 30,   L0 =  150,   L1 =  150



-----------------------------------------------
Test on the values of the Statistic H

Number of degrees of freedom          :   52
ChiSquare statistic                   :   48.79
p-value of test                       :    0.60


-----------------------------------------------
Test on the values of the Statistic M

Number of degrees of freedom          :   52
ChiSquare statistic                   :   53.87
p-value of test                       :    0.40


-----------------------------------------------
Test on the values of the Statistic J

Number of degrees of freedom          :   75
```

```
ChiSquare statistic                   :   64.53
p-value of test                       :    0.80


-----------------------------------------------
Test on the values of the Statistic R

Number of degrees of freedom          :   44
ChiSquare statistic                   :   43.01
p-value of test                       :    0.51


-----------------------------------------------
Test on the values of the Statistic C

Number of degrees of freedom          :   26
ChiSquare statistic                   :   18.58
p-value of test                       :    0.85


-----------------------------------------------
CPU time used                    :  00:00:01.73

Generator state:




========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        AHS
 Number of statistics:  15
 Total CPU time:   00:00:09.66

 All tests were passed
```

## A.6 TestU01-LCG-generator

The Small Crush battery applied to the sequence generated by the LCG generator yielded the following:

data.txt

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                Starting SmallCrush
                Version: TestU01 1.2.3
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx


HOST = gauss, Linux

LCG


smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,    d = 1073741824,    t = 2,    p = 1


      Number of cells = d^t = 1152921504606846976
      Lambda = Poisson mean =      27.1051


-------------------------------------------------
Total expected number = NTotal observed number                : 4989070
p-value of test                       :   eps

-----------------------------------------------
CPU time used                    :  00:00:01.37

Generator state:




Test sknuth_Collision calling smultin_Multinomial

HOST = gauss, Linux

LCG


smultin_Multinomial test:
-----------------------------------------------
   N =  1,  n = 5000000,  r =  0,   d = 65536,   t =  2,
       Sparse =   TRUE

       GenerCell = smultin_GenerCellSerial
       Number of cells = d^t =         4294967296
       Expected number per cell =  1 /  858.99346
       EColl = n^2 / 2k =  2910.383046
       Hashing =   TRUE

       Collision test,    Mu =      2909.2534,    Sigma =    53.8957

-----------------------------------------------
Test Results for Collisions

Expected number of collisions = Mu    :     2909.25
Observed number of collisions         :        0
```

```
p-value of test                       : 1 - eps1
----------------------------
Total number of cells containing j balls

  j =  0                              :       4289967296
  j =  1                              :          5000000
  j =  2                              :                0
  j =  3                              :                0
  j =  4                              :                0
  j =  5                              :                0

------------------------------------------------
CPU time used                    :  00:00:01.83

Generator state:



HOST = gauss, Linux

LCG


sknuth_Gap test:
------------------------------------------------
   N =  1,  n = 200000,  r = 22,   Alpha =        0,   Beta  = 0.00390625


------------------------------------------------
Number of degrees of freedom          : 1114
Chi-square statistic                  : 4.10e+7
p-value of test                       :   eps
------------------------------------------------
CPU time used                    :  00:00:00.51

Generator state:



HOST = gauss, Linux

LCG


sknuth_SimpPoker test:
------------------------------------------------
   N =  1,  n = 400000,  r = 24,   d =   64,   k =   64


------------------------------------------------
Number of degrees of freedom          :   19
Chi-square statistic                  : 1.10e+7
```

```
p-value of test                       :   eps
-----------------------------------------------
CPU time used                    :  00:00:00.53

Generator state:



HOST = gauss, Linux

LCG


sknuth_CouponCollector test:
-----------------------------------------------
   N =  1,  n = 500000,  r = 26,   d =   16


-----------------------------------------------
Number of degrees of freedom          :   44
Chi-square statistic                  : 3.40e+7
p-value of test                       :   eps
-----------------------------------------------
CPU time used                    :  00:00:00.33

Generator state:



HOST = gauss, Linux

LCG


sknuth_MaxOft test:
-----------------------------------------------
   N =  1,  n = 2000000,  r =  0,   d = 100000,   t =  6

      Number of categories = 100000
      Expected number per category  = 20.00


-----------------------------------------------
Number of degrees of freedom          : 99999
Chi-square statistic                  :99374.70
p-value of test                       :    0.92


-----------------------------------------------
Anderson-Darling statistic            :    0.39
p-value of test                       :    0.61
```

```
-----------------------------------------------
CPU time used                   :  00:00:01.04

Generator state:



HOST = gauss, Linux

LCG


svaria_WeightDistrib test:
-----------------------------------------------
   N =  1,  n = 200000,  r = 27,  k = 256,  Alpha =      0,  Beta =  0.125


-----------------------------------------------
Number of degrees of freedom          :   41
Chi-square statistic                  : 2.46e+6
p-value of test                       :   eps
-----------------------------------------------
CPU time used                   :  00:00:00.44

Generator state:



HOST = gauss, Linux

LCG


smarsa_MatrixRank test:
-----------------------------------------------
   N =  1,  n = 20000,  r = 20,    s = 10,    L = 60,    k = 60


-----------------------------------------------
Number of degrees of freedom          :    3
Chi-square statistic                  : 3.76e+6
p-value of test                       :   eps
-----------------------------------------------
CPU time used                   :  00:00:00.62

Generator state:



HOST = gauss, Linux
```

```
LCG


sstring_HammingIndep test:
-----------------------------------------------
   N =  1,  n = 500000,  r = 20,   s = 10,   L = 300,   d = 0



Counters with expected numbers >= 10
-----------------------------------------------
Number of degrees of freedom          : 2209
Chi-square statistic                  : 6.05e+5
p-value of test                       :   eps
-----------------------------------------------
CPU time used                    :  00:00:00.77

Generator state:




HOST = gauss, Linux

LCG


swalk_RandomWalk1 test:
-----------------------------------------------
   N =  1,  n = 1000000,  r =  0,   s = 30,   L0 =  150,   L1 =  150



-----------------------------------------------
Test on the values of the Statistic H

Number of degrees of freedom          :   52
ChiSquare statistic                   : 3070.62
p-value of test                       :   eps

-----------------------------------------------
Test on the values of the Statistic M

Number of degrees of freedom          :   52
ChiSquare statistic                   :  858.37
p-value of test                       :   eps

-----------------------------------------------
Test on the values of the Statistic J

Number of degrees of freedom          :   75
ChiSquare statistic                   :  133.32
p-value of test                       :  4.0e-5
```

```
-----------------------------------------------
Test on the values of the Statistic R

Number of degrees of freedom          :   44
ChiSquare statistic                   :  184.71
p-value of test                       :   eps

-----------------------------------------------
Test on the values of the Statistic C

Number of degrees of freedom          :   26
ChiSquare statistic                   :  169.33
p-value of test                       :   eps

-----------------------------------------------
CPU time used                    :  00:00:01.74

Generator state:
```

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        LCG
 Number of statistics:  15
 Total CPU time:   00:00:09.24
 The following tests gave p-values outside [0.001, 0.9990]:
 eps  means a value < 1.0e-300:
 eps1 means a value < 1.0e-15:

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  2  Collision                       1 - eps1
  3  Gap                               eps
  4  SimpPoker                         eps
  5  CouponCollector                   eps
  7  WeightDistrib                     eps
  8  MatrixRank                        eps
  9  HammingIndep                      eps
 10  RandomWalk1 H                     eps
 10  RandomWalk1 M                     eps
 10  RandomWalk1 J                    4.0e-5
 10  RandomWalk1 R                     eps
 10  RandomWalk1 C                     eps
 ----------------------------------------------
 All other tests were passed
```