Pseudorandom Noise Generators dedicated for Acoustic Measurements

Master Thesis

Xavier Español Espinar

22/07/2011
Para mi madre, el meu pare y mi hermana,
siempre habéis sido mi camino.
**Acknowledgement**

In the first place, I would like to thank my tutor, Vladimír Pavliček, Dr. Ing., for his support and help during the gestation of the project and for all his attention, as well as Vjačeslav Georgiev, Doc. Dr. Ing., for accepting me in the faculty as Erasmus student and allowing me to feel this experience.

I would also like to thank specially to my family. My mother and my father, Mº del Carmen and Miquel, because they have been always by my side, because they have always supported and believed in me, and of course my little sister, Clara, without you the family never would be complete, without you I would never have been complete. You three are the most important. I need to thank to my grandparents, Segundo and Concepcion, and, Miquel and Pilar, you have done everything possible. Finally thank to the rest, but not less important, part of my family.

I want to thanks to everyone I’ve known in Plzen, all of them allow this experience had been so special and so exciting, as the Erasmus world is a world apart. I want to mention to my buddy, Petra, your help at the beginning and your friendship were a nice present. As well as, especially to the Spanish people, all of you have given to me a little part of happiness I will never forget.

Thanks to my friends in Mataró; Emilio, Ire, Carlos, Nurieta, Ruben (Higo), Judit, Dolar and Fede, all the moments we have passed and will pass, are the pieces that build my life. Especial thanks to my cousin; Carlos, and my cousin in law; Yud, you know so special you are for me, not enough words to explain.

I want to give my gratitude to Miona, all the time we were together, all the love we felt and all the moments we shared have been an inspiration for me, and will always be part of me. Thanks for believing in me, all the advices you gave me and everything I have lived with you.

Finally thanks all the friends from school and university and everyone who give me a piece of his time, thanks you.

Gracias de todo corazón. Gràcies de tot cor.
Content

Acknowledgement ............................................................................................................. 3
Introduction ......................................................................................................................... 8
1. Noise ............................................................................................................................... 10
   1.1 White Gaussian Noise (WGN) .................................................................................. 10
      1.1.1 White noise (WN) ............................................................................................ 10
      1.1.2 White Gaussian noise (WGN) .......................................................................... 11
2. Random numbers. Generation ......................................................................................... 13
   2.1 Random Number Generators (RNG) ........................................................................ 13
   2.2 Pseudorandom Number Generators ......................................................................... 13
      2.2.1 Linear Congruential Generator (LCG) ................................................................. 14
      2.2.2 Add with carry (AWC) and Subtract with borrow generators (SWB) ............... 15
      2.2.3 RANLUX Generator ......................................................................................... 16
      2.2.4 Multiply with carry (MWC) ............................................................................... 17
      2.2.5 Xorshift RNG .................................................................................................... 18
      2.2.6 KISS Generator ................................................................................................. 18
3. Normalizing methods ....................................................................................................... 21
   3.1 Box – Muller method ................................................................................................. 21
   3.2 Ziggurat method ....................................................................................................... 21
      3.2.1 Tail algorithm .................................................................................................... 23
4. KISS generator: coding and testing ................................................................................ 25
   4.1 The code .................................................................................................................... 25
      4.2 Diehard battery test ............................................................................................... 29
5. Ziggurat method: implementation and testing ............................................................... 32
   5.1 The implementation of the code ............................................................................... 32
   5.2 Testing: results in Matlab ....................................................................................... 33
      5.2.1 Histograms ........................................................................................................ 33
      5.2.2 First statistics .................................................................................................... 34
      5.2.3 Chi-square test ................................................................................................. 35
      5.2.4 Lillifeors test .................................................................................................... 35
      5.2.5 Kolmogorov-Smirnov ....................................................................................... 36
      5.2.6 The autocorrelation .......................................................................................... 37
      5.2.7 Normplot .......................................................................................................... 38
6. The process: generation and transformation ................................................................. 40
   6.1 Schematic process ..................................................................................................... 40
   6.2 Results in Matlab ..................................................................................................... 41
      6.2.1 Histograms ........................................................................................................ 41
      6.2.2 Statistic .............................................................................................................. 43
6.2.3 Autocorrelation test ................................................................. 43
6.2.4 Normplot test ........................................................................ 44
6.2.5 Matlab Gaussian plot vs. Ziggurat method plot ................. 44
6.3 Conclusion of process ............................................................... 45
7. Testing physical devices ............................................................. 47
  7.1 Statistic .................................................................................. 47
  7.2 Graphical results ................................................................. 48
    7.2.1 WNA-PR-120-PC-R1 ......................................................... 48
    7.2.2 WNA-PR-120-PC-R2 ......................................................... 48
    7.2.3 WNA-R-120-PC-R1 ............................................................ 49
    7.2.4 WNA-R-120-PC-R2 ............................................................ 50
    7.2.5 Wnbb_r_120_mix_r1 ............................................................. 51
    7.2.6 Wnbb_r_120_mix_r2 ............................................................. 51
    7.2.7 Wnnti_r_120_mix_r1 ............................................................. 52
    7.2.8 Wnnti_r_120_mix_r2 ............................................................. 53
  7.3 Summary of results ............................................................... 53
8. Normality Test suite in Matlab ...................................................... 55
9. Final conclusions ........................................................................ 57
10. Annex 1. Diehard exhaustive results ........................................... 59
  10.2 GCD ................................................................................... 59
  10.3 Gorilla .................................................................................. 59
  10.4 Overlapping Permutations ..................................................... 60
  10.5 Ranks of 31x31 and 32x32 matrices ..................................... 60
  10.6 Ranks of 6x8 Matrices .......................................................... 61
  10.7 Monkey Tests on 20-bit Words ............................................. 61
  10.8 Monkey Tests OPSO,OQSO,DNA ......................................... 62
  10.9 Count the 1’s in a Stream of Bytes ....................................... 65
  10.10 Count the 1’s in Specific Bytes ............................................ 65
  10.11 Parking Lot Test .................................................................. 66
  10.12 Minimum Distance Test ..................................................... 67
  10.13 Random Spheres Test ........................................................ 67
  10.14 The Squeeze Test ............................................................... 68
  10.15 Overlapping Sums Test ....................................................... 68
  10.16 Runs Up and Down Test ..................................................... 69
  10.17 The Craps Test ................................................................... 69
11. Annex 2. Ziggurat code ............................................................... 71
  11.1 Ziggurat algorithm in C ....................................................... 71
  11.2 Ziggurat algorithm for Matlab ............................................. 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.3</td>
<td>Ziggurat function modified</td>
<td>75</td>
</tr>
<tr>
<td>12.1</td>
<td>Statistics tests</td>
<td>79</td>
</tr>
<tr>
<td>12.2</td>
<td>Histogram</td>
<td>80</td>
</tr>
<tr>
<td>12.3</td>
<td>Autocorrelation</td>
<td>81</td>
</tr>
<tr>
<td>12.4</td>
<td>Normplot</td>
<td>82</td>
</tr>
<tr>
<td>12.5</td>
<td>Comparison with Matlab function</td>
<td>83</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>85</td>
</tr>
</tbody>
</table>
Introduction
Introduction

This project is part of bigger project focused in frequency characterization for different physical spaces. The main objective of the project is to study the pseudo random noise generation, corresponding to the source necessary to carry out the characterization.

The way to achieve the objective above is studying, designing, testing and implementing software for the noise generator. In order to do this proposals there are some specific objectives it is necessary to work out:

- read up the possibilities and types of acoustic noise generators
- study the algorithms of discrete PRNGs (Pseudo Random Noise Generators)
- design and simulate selected discrete PRNG
- test the designed PRNG
- compare results to professional random noise samples obtained from professional instruments

The first chapter is an introduction in the acoustic noise generation, it talks about basic concepts about the acoustic noise and introduce the mathematical properties important for the work.

The second and the third chapter are an accurate study of the different pseudo random generators. Firstly, the most important uniform pseudo random generators are explained along the time until the one chosen for the project. Secondly, two different methods for normalization are studied and explained.

The next two chapters, the fourth and the fifth, show the implementation and test for the uniform pseudo random generator and the normalizing method separately. The codes used for each purpose are given and explained, as well as the different tests.

The sixth chapter joins the two parts designed above, generator and transformer method. So the final design is passed through different tests in order to verify the goodness of the device.

On the seventh chapter the physical random noise generator devices are tested and compared among them. Finally some conclusions about the project and the comparison between the designed PRNG are given.
Chapter 1
1. Noise

Noise has different meanings depending on the field we are talking about. We could understand the generic noise with the next definitions:

- In electronics: noise (or thermal noise) exists in all circuits and devices as a result of the thermal energy. In electronics circuits, there are random variations in current or voltage caused by the thermal energy. The lower temperature, lower thermal noise.
- In audio: noise in audio recording and transmission is referred to the lower sound (like a hum or whistle) that is often heard in periods should be in silence.
- In telecommunications: noise is the disturbance suffered by a signal while is being transmitted. The noise sometimes depends on the mode of transmission, the means used and the environment in which is transmitted.

We can appreciate differences between the working fields of noise, but it is important to say that in every situation the noise is more or less a random signal (or process) mixed with the main signal (or process).

It is necessary to characterize the random nature of noise in useful parameters, specifically in mathematics parameters. The reason is we must know the properties of this kind of process in order to test it and design devices that can implement this kind of behavior.

1.1 White Gaussian Noise (WGN)

White noise is the kind of noise will be need in our project. It is a random signal with flat power spectral density. In other words, the signal contains equal power within a fixed bandwidth at any center frequency and this is our objective of design.

Anyway, it’s necessary to know that in statistical sense, a time series \( r_t \) is characterized as having weak white noise if \( \{r_t\} \) is a sequence of serially uncorrelated random variables with zero mean and finite variance. Strong white noise also has the quality of being independent and identically distributed, which implies no autocorrelation. In particular, if \( r_t \) is normally distributed with mean zero and standard deviation \( \sigma \), the series is called a Gaussian White Noise.

On the other hand an infinite-bandwidth white noise signal is a purely theoretical construction. The bandwidth of white noise is limited in practice by the mechanism of noise generation, by the transmission medium and by finite observation capabilities. A random signal is considered "white noise" if it is observed to have a flat spectrum over a medium's widest possible bandwidth.

1.1.1 White noise (WN)

A continuous in time random process \( w(t) \) where \( t \in \mathbb{R} \), is a white noise process if, and only if, its mean function and autocorrelation function satisfy the following:

\[
\mu_w = \mathbb{E}(w(k)) = 0
\]

\[
R_{ww}(\Delta) = \mathbb{E}(w(k) \cdot w(k - \Delta)) = \sigma^2 \delta(\Delta)
\]

Thus, it is a zero mean process for all time and has infinite power at zero time shifts since its autocorrelation function is the Dirac delta function.

The above autocorrelation function implies the following power spectral density:
\[ S_{xx}(f) = \text{TF}[R_{ww}(\Delta)] = \text{TF}[\sigma^2 \delta(\Delta)] = \sigma^2, \]

since the Fourier transform (TF) of the delta function is equal to 1.

The reason this kind of noise is named \textit{white} is his power spectral density is the same at all frequencies, analogy to the frequency spectrum of white light. A generalization to random elements on infinite dimensional spaces, such as random fields, is the white noise measure.

1.1.2 \textbf{White Gaussian noise (WGN)}

Sometimes white noise, Gaussian noise and white Gaussian noise is confused. The random process we are searching has both white and Gaussian property.

The Gaussian property is referred to the probability density function as a normal distribution, also known as Gaussian distribution. The Gaussian distribution is one of the most commonly probability distribution that appears in real phenomena.

The Gaussian probability density function is:

\[ f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \]

where parameter \( \mu \) is the \textit{mean} (location of the peak) and \( \sigma^2 \) is the \textit{variance} (the measure of the width of the distribution).

Finally the whole mathematics properties of the signal or process we are going to work with are shown in the next resume table:

<table>
<thead>
<tr>
<th>White process ( \rightarrow ) incorrelation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{In frequency}</td>
</tr>
<tr>
<td>\text{In time}</td>
</tr>
</tbody>
</table>

\textbf{Gaussian process \( \rightarrow \) Normal probability}:

\[ f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]

\textit{Table 1: Properties for WGN}
2. Random numbers. Generation

As it’s seen previously, it is necessary to generate a White Gaussian process. The way to produce it must be generating statistically independent numbers, commonly known as sequences of random numbers.

The way to generate this special numbers is using the random generators. There are two basic types of generators used to produce the random sequences: random number generators (RNG) and pseudorandom number generators (PRNG).

2.1 Random Number Generators (RNG)

This first type of random sequence generator allows truly random numbers, but is highly sensitive to environmental changes. The source for the RNG typically consists of some physical quantity, such as the noise in an electrical circuit, the timing of user processes, or quantum effects in a semiconductor.

The output of an RNG is commonly used feeding PRNG as a seed. The problem is the sequences produced by these generators may be deficient when evaluated by statistical tests. In addition, the production of high-quality random numbers may be too time consuming, making such production undesirable when a large quantity of random numbers is needed. To produce large quantities of random numbers, pseudorandom number generators may be preferable.

2.2 Pseudorandom Number Generators

The second generator type is a pseudorandom number generator (PRNG). A PRNG uses one or more inputs and generates multiple “pseudorandom” numbers. Inputs to PRNGs are called seeds. In contexts in which unpredictability is needed, the seed itself must be random and unpredictable. Hence, by default, a PRNG should obtain its seeds from the outputs of an RNG, as it is said above.

In simulation environments, digital methods for generating random variables are preferred over analog methods. In contrast with the RNGs, digital methods are more desirable due to their robustness, flexibility and speed. Although the resulting number sequences are pseudo random as opposed to truly random, the period can be made sufficiently large such that the sequences never repeat themselves even in the largest practical situations.

If a pseudorandom sequence is properly constructed, each value in the sequence is produced from the previous value via transformations that appear to introduce additional randomness. A series of such transformations can eliminate statistical auto-correlations between input and output. Thus, the outputs of a PRNG may have better statistical properties and be produced faster than an RNG.

In summary, Kahaner, Moler and Nash [1] define five areas in which one should assess a given random number generator and Ripley [2] talk about ideal properties of a good general-purpose PRNG, as follows:
1. **Quality.** Suitable statistical test should be satisfied, so it is a good approximation to a uniform distribution. In addition, samples very close to independent output in a moderate number of dimensions.

2. **Efficiently.** The generator should be quick to run, so minimal storage must be required. In addition the generated sequence must have a long period.

3. **Repeatability.** The method should be repeatable from a simply specified starting point, to allow an experiment to be reproduced as many times as needed.

4. **Portability.** If the method is implemented on a different system it should produce the same results.

5. **Simplicity.** The generator should be straightforward both to implement and to use.

### 2.2.1 Linear Congruential Generator (LCG)

The following schema is one of the oldest and most studied PRNG’s, introduced by D. H. Lehmer in 1949 [3] and treated later by Knuth [4]. This generator is defined with an elegant recurrence relation:

\[ X_{n+1} = (aX_n + c) \mod m \]

where:

- \( m \), the modulus; \( m > 0 \).
- \( a \), the multiplier; \( 0 \leq a \leq m \).
- \( c \), the increment; \( 0 \leq c \leq m \).
- \( X_0 \), the starting value; \( 0 \leq X_0 \leq m \).

Should be considered some specifications if we are searching for a good generator, thus we have to choice appropriately the parameters above.

Many articles has been written talking about the convenience of using some parameters or others, all of them are reference for this work and must be taken into consideration. Thus, can be read in the article of Park and Miller [5], they proposed firstly the “minimal standard” with modulus \( 2^{31} - 1 \) and multiplier 16807:

\[ X_{n+1} = (16807X_n + c) \mod (2^{31} - 1) \]

Before they evaluate another number for the multiplier \( a \), properties of which were better than the LCG with multiplier 48271:

\[ X_{n+1} = (48271X_n + c) \mod (2^{31} - 1) \]

Usually the increment number \( c \) has the value 0. With this kind of combination of parameters it is possible to generate a sequence with period \( m-1 \), and it is known as full period generator.
The Lehmer’s theory based generators as the explained above can be made to have very long periods, although acceptable, it is inadequate due to the development of much faster processors. In addition, most of the different generators it is possible to obtain have drawbacks. This is the reason it is necessary to contemplate other possibilities, alternately Marsaglia and Zaman [6] proposed a new kind of generators.

On one hand let $b$, $r$, and $s$ be positives integers, where $b$ is called the base and $r > s$ are called lags. The AWC generator is based on the recurrence:

$$x_i = (x_{i-s} + x_{i-r} + c_i) \mod b,$$

$$c_{i+1} = I(x_{i-s} + x_{i-r} + c_i \geq b),$$

where $c_i$ is called the carry, and $I$ is the indicator function, whose value is 1 if its argument is true, and 0 otherwise.

That generator is faster than LCG, since it requires no multiplication, and the modulo operation can be performed by just subtracting $b$ if and only if:

$$x_{i-s} + x_{i-r} + c_i < 0$$

To produce values $\{u_i\}$ whose distribution approximates the $U(0,1)$ distribution, one can use $L \leq r$ successive values of $x_i$ to produce one $u_i$ as follows:

$$u_i = \sum_{j=1}^{L} x_{i-L+j-1} b^{-j}$$

Assuming that $L$ is relatively prime to $M-1$, the sequences $\{u_i\}$ and $\{x_i\}$ have the same periods. If $b$ is small, or if more precision is desired, take a larger $L$. If $b$ is large enough (e.g., a large power of two), one can just take $L = 1$.

On the other hand the SWB is based on the next recurrences, the first one called SWB I:

$$x_i = (x_{i-s} - x_{i-r} - c_i) \mod b,$$

$$c_{i+1} = I(x_{i-s} - x_{i-r} - c_i \geq 0),$$

and the second variant SWB II,

$$x_i = (x_{i-r} - x_{i-s} - c_i) \mod b,$$

$$c_{i+1} = I(x_{i-r} - x_{i-s} - c_i \geq 0).$$

For each of the recurrences above, both the AWC and SWB generators, the maximum possible period is $M-1$, achieved when $M$ is a prime and $b$ is a primitive root modulo $M$, where values of $M$ depends on the variant.
The generator with the particular choice of parameters $b = 2^{24}, r = 24$ and $s = 10$ based in the SWB I generator mentioned above is known by the name RCARRY [7].

In order to start the recursion, the first $r$ values $x_0, x_1, ..., x_{r-1}$ together with the carry bit $c_r$ must be provided. The configurations

$$x_0 = x_1 = \cdots = x_{r-1} = 0, \quad c_r = 0,$$

$$x_0 = x_1 = \cdots = x_{r-1} = b - 1, \quad c_r = 1,$$

should be avoided, because the algorithm yields uninteresting sequences of numbers in these cases. All other choices of initial values are admitted. Using the parameters described above the generator has the tremendously long period of $(2^{24})^2/48 \approx 2^{570}$ or about $5.2 \times 10^{171}$.

Also for the RCARRY generator some deficiencies in empirical tests of randomness were reported in the literature [8].

### 2.2.3 RANLUX Generator

In order to improve the properties of this algorithm and manage to pass the correlation tests, M. Lüscher proposed [9] to discard some of the pseudo random numbers produced by the Marsaglia & Zaman recursion and to use only the remaining ones.

The algorithm begins with a sequence of random numbers $x_0, x_1, ..., x_{r-1}$ generated through the Marsaglia and Zaman recursion, with carry bits $(c_n)_{n>0}$ and proper initial values suggested previously. The difference comes now, instead of using all numbers $x_n$, the $r$ successive elements of the sequence are read, the $p - r$ numbers are discarded, then $r$ numbers are read, and so on.

F. James defined later [10] four levels of rejection (called "luxury levels"), characterized by an integer $p \geq 24$ in which the generator produces 24 pseudo random numbers, then discards the successive $p - 24$ and so on.

Clearly the value $p = 24$ reproduces the original Marsaglia & Zaman recipe where all pseudo random numbers are kept and it is called luxury level zero. The values $p = 48; 97; 223; 389$ define the luxury levels 1; 2; 3; 4 respectively. It has been suggested [9] that level 3 has a good chance of being optimal.
### Table 3: Luxury levels

<table>
<thead>
<tr>
<th>Level</th>
<th>( p )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>Equivalent to original RCARRY</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
<td>Considerable improvement in quality over level 0, now passes the gap test, but still fails spectral test.</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>Passes all known tests, but theoretically still defective.</td>
</tr>
<tr>
<td>3</td>
<td>223</td>
<td>\textit{DEFAULT VALUE}. Any theoretically possible correlations have very small chance of being observed.</td>
</tr>
<tr>
<td>4</td>
<td>389</td>
<td>Highest possible luxury, all 24 bits chaotic.</td>
</tr>
</tbody>
</table>

#### 2.2.4 Multiply with carry (MWC)

The MWC generator was proposed as a modification of the AWC generator. The main advantages of the MWC method are that it invokes simple computer integer arithmetic and leads to very fast generation of sequences of random numbers with immense periods, ranging from around \(2^{60}\) to \(2^{2000000}\).

A MWC sequence is based on arithmetic modulo a base \( b \), usually \( b = 2^{32} \), because arithmetic modulo of that \( b \) is automatic in most computers. However, sometimes a base such as \( b = 2^{32} - 1 \) is used, because arithmetic for modulus \( 2^{32} - 1 \) requires only a simple adjustment from that for \( 2^{32} \), and theory for MWC sequences based on modulus \( 2^{32} \) has some nagging difficulties avoided by using \( b = 2^{32} - 1 \).

In its most common form, MWC generator requires a lag-\( r \), a base \( b \), a multiplier \( a \), and a set of \( r+1 \) random seed values, consisting of \( r \) residues of \( b \),

\[ x_0, x_1, \ldots, x_{r-1}, \]

and an initial carry \( c_{r-1} < a \).

The lag-\( r \) MWC sequence is then a sequence of pairs \( x_n, c_n \) determined by

\[
x_n = (ax_{n-r} + c_{n-1}) \mod b, \]

\[
c_n = \left[ \frac{ax_{n-r} + c_{n-1}}{b} \right], \quad n \geq r,
\]

and the MWC generator output is the sequence of \( x \)’s,

\[ x_r, x_{r+1}, x_{r+2}, \ldots \]

The period of a lag-\( r \) MWC generator is the order of \( b \) in the multiplicative group of numbers modulo \( ab^r - 1 \). It is customary to choose \( a \)’s so that \( p = ab^r - 1 \) is a prime for which the order of \( b \) can be determined. Because \( b = 2^{32} \) cannot be a primitive root of \( p = ab^r - 1 \), there are no MWC generators for base \( 2^{32} \) that have the maximum possible period, one of the difficulties that use of \( b = 2^{32} - 1 \) overcomes.

A theoretical problem with MWC generators, pointed out by Couture and l’Ecuyer (1997) is that the most significant bits are slightly biased; complementary-multiply-with-carry generators do not share this problem. They do not appear to elaborate further as to the extent of the bias. Complementary-multiply-with-carry generators also require slightly more computation time per iteration, so there is a tradeoff to evaluate depending on implementation requirements.
2.2.5 Xorshift RNG

The Xorshift Random Number Generator, otherwise known as SHR3 and developed by Marsaglia [11], produces "medium quality" random numbers: certainly better than the LCG algorithm. And what is powerful is that it does so using low-cost operations: shifts and XORs, with only a single word of state. Thus provides extremely fast and simple RNGs that seem to do very well on tests of randomness.

To give an idea of the power and effectiveness of xorshift operations, here is the essential part of a C procedure that:

```c
x ^= (x << 21);
\n\nx ^= (x >>> 35);
\n\nx ^= (x << 4);
```

In C language, the operator << or >> means shift to the left or to the right, respectively, and the operator ^= implements the XOR function, so the code is extremely simple.

The "magic" values of 21, 35 and 4 have been found to produce good results. With these values, the generator has a full period of $2^{64}-1$. L’Ecuyer & Simard [12] also found that values of 13, 7 and 17 had better results and implements a strong generator.

Longer periods are available, for example, from multiply-with-carry RNGs, but they use integer multiplication and require keeping a (sometimes large) table of the most recently generated values.

However, L’Ecuyer & Simard found that the 32-bit version gives poor results in their statistical tests. It’s also probably not worth using a generator with such a small period unless performance is really tight.

2.2.6 KISS Generator

KISS ('Keep it Simple Stupid') is an efficient pseudo-random number generator specified by G. Marsaglia and A. Zaman in 1993 [13], culminating in a final version in 1999 [14]. Since 1998 Marsaglia has posted a number of variants of KISS (without version numbers), including the last in January 2011.

The reasons the KISS Generator it is chosen to be the main generator in the simulations are because:

- It is proposed by well known and respected authors.
- It has a reasonably long but not excessive (claimed) period.
- It has a compact state.
- The output 'looks random' immediately after initialization.

Other proposals, as it was explained before, often involve large state variables, often calling on simpler RNGs to initialize these large states.

KISS consists of a combination of four sub-generators each with 32 bits of state, of three kinds:

- One linear congruential generator (LCG) modulo $2^{32}$.
- One general binary linear generator Xorshift.
- Two multiply-with-carry (MWC) generators modulo $2^{16}$, with different parameters.
The four generators are updated independently, and their states are combined to form a stream of 32-bit output words. The four state variables are treated as unsigned 32-bit words.

As the C code below shows, it is combined two multiply-with-carry generators in MWC with the 3-shift register SHR3 and the congruential generator CONG, using addition and exclusive-or. The period obtained is about $2^{123}$.

```
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew)
#define SHR3 (jsr^=(jsr<<17), jsr^=(jsr>>13), jsr^=(jsr<<5))
#define CONG (jcong=69069*jcong+1234567)
#define KISS ((MWC^CONG)+SHR3)
```
3. Normalizing methods

The second step, after generating the pseudo random sequence, is to adapt the output obtained from the generator to the requirements. The sequence obtained is statistically uniform and it is necessary to implement white Gaussian noise, so it is necessary to convert the uniform statistical into Gaussian or normal statistical.

3.1 Box–Muller method

The Box–Muller method was introduced by George E. P. Box and Mervin E. Muller in 1958 [15], it allows to generate a pair of standard normally distributed random variables from a source of uniformly distributed variables.

Let \( U_1, U_2 \) be the independent random variables generated previously by a PRNG with uniform density function on the interval \((0,1)\). Consider the random variables:

\[
X_1 = (-2 \log U_1)^{1/2} \cos 2\pi U_2 \\
X_2 = (-2 \log U_1)^{1/2} \sin 2\pi U_2
\]

Then \((X_1, X_2)\) will be a pair of independent random variables from the same normal distribution with mean zero, and unit variance. If it is necessary to generalize the result there is a simple transformation:

\[
Z_1 = \mu + \sigma(-2 \log U_1)^{1/2} \cos 2\pi U_2 \\
Z_2 = \mu + \sigma(-2 \log U_1)^{1/2} \sin 2\pi U_2
\]

this way \(Z_1, Z_2\) are variables have normal distribution \(Z \sim N(\mu, \sigma^2)\), with the mean \(\mu\) and the variance \(\sigma^2\).

3.2 Ziggurat method

The first ziggurat algorithm was introduced by George Marsaglia [16] in the 1960s, after was developed and finally improved with Wai Wan Tsang [17] in 2000.

The algorithm consists of generating a random floating-point value, a random table index, performing one table lookup, one multiply and one comparison. This is considerably faster than the the Box-Muller transform, which require at least a logarithm and a square root. On the other hand, the ziggurat algorithm is more complex to implement and requires precompiled tables, so it is best used when large quantities of random numbers are required.

The Ziggurat method partitions the standard normal density in horizontal blocks of equal area; \(v\), and their right-hand edges are denoted by \(x_k\). The standardization can be omitted, using \(f(x) = e^{-x^2/2}\). All blocks are rectangular boxes, instead the bottom one, which consist in a box joined with the remainder of the density.
All but the bottom of the rectangles can be further divided into two regions: a “subrectangle” bounded on the right by \( x_{i-1} \), which is completely within the PDF, and to the right of that a wedge shaped region, that includes portions both above and below the PDF. The rectangle bounded by \( x_1 \) consists of only a wedge shaped region. Each time a random number is requested, one of the \( n \) sections is randomly (with equal probability) chosen. A uniform sample \( x \) is generated and evaluated to see if it lies within the subrectangle of the chosen section that is completely within the PDF. If so, \( x \) is output as the Gaussian sample. If not, this means that \( x \) lies in the wedge region and an appropriately scaled uniform value is chosen. If the \( x, y \) location is below the PDF in the wedge region, then \( x \) is output. Otherwise \( x \) and \( y \) are discarded and the process starts again from the beginning.

In the case of being in the tail section and \( x > x_{n-1} \), a value from the tail is chosen using a separate procedure. Provided that the tail sampling method is exact, the Ziggurat method as a whole is exact.

The values of \( x_i \) \((i = 1, 2, \ldots, n)\) are calculated prior to execution, or on program startup, and are determined by equating the area of each of the rectangles with that of the base region. If this area is \( v \), the equations are as follows:

\[
v = x_i [\Phi(x_{i-1}) - \Phi(x_i)] = r \Phi(r) + \int_r^\infty \Phi(x)dx
\]

The value of \( r \) can be determined numerically, and can then be used to calculate the values of \( x_i \). When \( n = 256 \) the probability of choosing a rectangular region is 99%. Thus, the algorithm in pseudo code is:

```
1: loop
2: \( i \leftarrow 1 + \lfloor n/4 \rfloor \), \{Usually \( n \) is a binary power: can be done by bitwise mask.\}
3: \( i \leftarrow x_i u_2 \) \{\( u_1 \) and \( u_2 \) Uniform random number.\}
4: if \( |x| < x_{i-1} \) then
5: \hspace{1em} return \( x \) \{Point completely within rectangle.\}
6: else if \( i = n \) then \{Note that \( \Phi(x_{i-1}) \) and \( \Phi(x_i) \) are table look-ups.\}
7: \hspace{1em} \( y \leftarrow (\Phi(x_{i-1}) - \Phi(x_i)) \cdot u \) \{Generate random vertical position.\}
8: \hspace{1em} if \( y < (\Phi(x_{i-1}) - \Phi(x_i)) \) then \{Test position against PDF.\}
9: \hspace{2em} return \( x \)
10: end if
11: else
12: \hspace{1em} return \( |x| > r \) from the tail. \{Specific algorithm\}
13: end if
14: end loop
```
3.2.1 Tail algorithm

The newest Marsaglia Tail Method requires only one loop and fewer operations than the original method, although it requires two logarithms per iteration rather than just one:

1: repeat
2: \( a \leftarrow U_1, b \leftarrow U_2 \)
3: \( x \leftarrow -\frac{1}{r} \ln|a|, y \leftarrow -\ln b \)
4: until \( 2y > x^2 \)
5: return \( a > 0? r + x : -(r + x) \)
4. KISS generator: coding and testing

The code of the PRNG has been programmed in C code, because it allows an efficient, simple and fast performance. The main part of the code has been inspired from the code provided by George Marsaglia [18] through a mail explaining different kind of generators.

4.1 The code

The C code is the next one:

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include <windows.h>

#define UL unsigned long
#define znew ((z=36969*(z&65535)+(z>>16)))
#define wnew ((w=18000*(w&65535)+(w>>16)))
#define MWC ((znew<<16)+wnew)
#define SHR3 (jsr^=(jsr<<17), jsr^=(jsr>>13), jsr^=(jsr<<5))
#define CONG (jcong=69069*jcong+1234567)
#define KISS ((MWC^CONG)+SHR3)
#define UNI (KISS*2.328306e-10)

/* Variables globales */
static UL z=362436069, w=521288629, jsr=123456789, jcong=380116160;
static FILE *fich1, *fich2;
static long int Max=0;

void seeds(){
    char seed[10];
    int op=0;
    printf("Do you want to introduce the seed? (y/n)\n");
    op=getch();
    switch (op){
        case 'y':
            printf("Please introduce de seeds between %d and %.0f and press ENTER\n",0,pow(2,32));
```
for (int i=0; i<4; i++)
{
    printf("\nSeed %d: ", i);
    fgets (seed, 256, stdin);
    if (i==0) z=atoi(seed);
    if (i==1) w=atoi(seed);
    if (i==2) jsr=atoi(seed);
    if (i==3) jcong=atoi(seed);
}
break;
case 'n':
    printf("Thanks. Default mode\n");
    break;
default :
    printf("Incorrect option. Default mode\n");
    break;
}

printf("\nDo you want to introduce the sequence length? (y/n)\n");
op=getch();
switch (op){
case 'y':
    printf("Please introduce de sequence length and then press ENTER\n");
    printf("\nLength: ");
    scanf("%ld", &Max);
    break;
case 'n':
    printf("Thanks. Default mode has length: %d\n", 3e6);
    Max=3e6;
    break;
default :
    printf("Incorrect option. Default mode\n");
    Max=3e6;
    break;
}
void menu()
{

  char name[20];
  char name1[20];
  char seed[10];
  int op=0;
  int ok=0;

  printf("\n\n|-------- KISS GENERATOR --------|\n");
  printf("\nChoose the option: \n");
  printf("1. Default output file (archivo)\n");
  printf("2. Choose the output file\n");

  do{
    op=getch();
    if(op == '1'){
      printf("\n-- Op 1. Default\t\n");
      fich1 = fopen("archivo.norm","w+");
      fich2 = fopen("archivo.dieh","w+b");
      ok=1;
    }else{
      if(op == '2'){
        printf("\n-- Op 2. Enter the output name: ");
        scanf("%s",&name);
        strcpy(name1,name);
        fich1 = fopen(strcat(name,".norm"),"w+");
        fich2 = fopen(strcat(name1,".dieh"),"w+b");
        ok=1;
      }else{
        printf("-- Wrong option. Try again\t\n");
        Sleep(2000);
        printf("\n\t\n");
        fflush(stdout);
        ok=0;
      }
    }
  }while (!ok);

  printf("\n\n");
}
main()
{
    long int num=0;

    //ask for seeds
    seeds();

    //print the main menu
    menu();
    num=ceil(0.03*Max);

    //generation of numbers
    for(long int i=0;i<Max;i++){
        if(((i%num+1)/num){
            printf("-");
        }
        fprintf(fich1,"%lu\n",KISS);

        if(((i%10+1)/10){
            fprintf(fich2,"%lx\n",KISS);
        }else{
            fprintf(fich2,"%lx",KISS);
        }
    }
    fclose(fich1);
    fclose(fich2);
    printf("\nGeneration finished. Dekuji\n");
    getch();
}

Notice that it is used the KISS generator, but additionally four generators are needed to be implemented:

- **znew & wnew**: Both are Multiply-with-carry generators. The low order 16 bits are random looking, and with a period determined by the entire construct. The most significant bits of the register are not very random.
- **MWC**: It is a combination of the two previous; the low 16 bits of znew are shifted left before being added to the whole register wnew, presumably to cover up this “nonrandomness”.
- **SHR3**: is a 3-shift-register generator with period $2^{32} - 1$. Seems to pass all except those related to the binary rank test.
- **CONG**: is a congruential generator with the widely used 69069 multiplier. It has period $2^{32}$. The leading half of its 32 bits seem to pass tests, but bits in the last half are too regular.
- **KISS**: this design is apparent with the use of simple generators with known flaws to cover each others’ deficiencies.
• **UNI**: same as **KISS** but uniformly distributed between 0 and 1.

The random numbers are generated and saved into a specific file:

• The file name can be chosen by the user at the beginning, or the default name **archivo** will be the output file.
• The file must have an appropriate format in order to be tested after by the Diehard Battery Test. It means the numbers must have hexadecimal representation, must be sorted without spaces between each one and just ten numbers per line.

```
archivo.dieh
```

2ddccfe0 2c3a35a8 7e6ee31a a73a60ce bf9847a7 e03d2a6d 797a2c20 9ae5fba6 db5ffbd5 341dc464
ba4c087960b84752e552a41fe7e1eb3f2487f1a8ca7d6431b27bf2e54f467812189fd2f840c5e5d2
cf8f8398ec32f9283a07e7f695216c689b27ed57e030dc29cd86ed9644c7d9b76906d8b371787ce4
46e18ab7a9c87a388151f8c1839d886ae417479b4c52d1b5c55d1019da5e42450a0c8f2a10fa151
b6d794932826af4089a058c9b2b02abb432a65bf79a641da9fd37068949824a980bb395383f803
dd116dd529faef4d5e05cc18a83071dcf850a51128702ce73569f539ce5da9f9950f5c36771c7e8
...

• There will be three different output files with extension .dieh, .norm and .uni. The first one is specifically formatted for the Diehard Test as above, and the second print the decimal numbers, one per line. The extension .uni is the file with distribution Uniform (0,1).

```
archivo.norm
```

| 769445856 | 0.168345 |
| 2121196314 | 0.263749 |
| 3214428071 | 0.668381 |
| 2038049824 | 0.269416 |
| 3680500693 | 0.701982 |
| ... | ... |

```
archivo.uni
```

4.2 Diehard battery test

Firstly it is necessary to test the numbers generated, as it is recommended along the literature the Diehard Test is a good test suite, although there is newer battery test; **Dieharder**, but the older one is enough for the objective of the project. **Diehard battery** was programmed and proposed by George Marsaglia and it contains a total of 17 different tests.

Most of the tests in **Diehard** return a p-value, which should be uniform on [0,1) if the input file contains truly independent random bits. Those p-values are obtained by p=F(X), where F is normal distribution.
But that assumed $F$ is often just an approximation, for which the fit will likely be worst in the tails. Thus, occasional p-values near 0 or 1, such as .0012 or .9983 can be obtained. When a bit stream really FAILS BIG, there are p’s of 0 or 1 to six or more places.

The exhaustive results of the **Diehard battery test** are detailed in the Annex 1. The following table shows a summary of the different test:

<table>
<thead>
<tr>
<th>Test n°</th>
<th>Name</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Birthday Spacing</td>
<td>0.274</td>
<td>PASSED</td>
</tr>
<tr>
<td>2</td>
<td>GCD</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>Gorilla</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>Overlapping Permutations</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>Ranks of 31x31 matrices</td>
<td>0.175</td>
<td>PASSED</td>
</tr>
<tr>
<td></td>
<td>Ranks of 32x32 matrices</td>
<td>0.669</td>
<td>PASSED</td>
</tr>
<tr>
<td>6</td>
<td>Ranks of 6x8 matrices</td>
<td>0.762</td>
<td>PASSED</td>
</tr>
<tr>
<td>7</td>
<td>Bit-stream</td>
<td>0.867</td>
<td>PASSED</td>
</tr>
<tr>
<td>8</td>
<td>Monkey test: OPSO</td>
<td>0.747</td>
<td>PASSED</td>
</tr>
<tr>
<td></td>
<td>Monkey test: OQSO</td>
<td>0.876</td>
<td>PASSED</td>
</tr>
<tr>
<td></td>
<td>Monkey test: DNA</td>
<td>0.978</td>
<td>FAILED</td>
</tr>
<tr>
<td>9</td>
<td>1’s in Stream of Bytes</td>
<td>0.147</td>
<td>PASSED</td>
</tr>
<tr>
<td>10</td>
<td>1’s in Specific Bytes</td>
<td>0.604</td>
<td>PASSED</td>
</tr>
<tr>
<td>11</td>
<td>Parking lot</td>
<td>0.000</td>
<td>FAILED</td>
</tr>
<tr>
<td>12</td>
<td>Minimum distance</td>
<td>0.222</td>
<td>PASSED</td>
</tr>
<tr>
<td>13</td>
<td>Random Spheres</td>
<td>0.848</td>
<td>PASSED</td>
</tr>
<tr>
<td>14</td>
<td>The Squeeze</td>
<td>1.000</td>
<td>PASSED</td>
</tr>
<tr>
<td>15</td>
<td>Overlapping Sums</td>
<td>0.566</td>
<td>PASSED</td>
</tr>
<tr>
<td>16</td>
<td>Runs Up &amp; Down</td>
<td>0.653</td>
<td>PASSED</td>
</tr>
<tr>
<td>17</td>
<td>Craps: nº of wins</td>
<td>0.695</td>
<td>PASSED</td>
</tr>
<tr>
<td></td>
<td>Craps: throws/game</td>
<td>0.986</td>
<td>FAILED</td>
</tr>
</tbody>
</table>

* Not enough samples for the test

The generator pass 14 tests with good p-values, and fails 3 of them. It is necessary to point out the threshold is near of two of the failed test, so probably in other performances would pass these tests. Note that there are 3 tests needed higher sample length; anyway it is enough with the results obtained.

Finally, the summary shows the generator pass most of the test proposed in the battery test, these results mean the generator proposed has the expected behavior and it is proper for the application.
5. Ziggurat method: implementation and testing

5.1 The implementation of the code

In order to implement the Ziggurat algorithm it is necessary to implement some tables with constants that allow calculating the “boxes” where the uniform random numbers may lay. These tables do not take so much memory and it is really useful to store it in order to speed up the process. A choice of 256 boxes it is enough for the application.

Let $v$ be the area of each box and let $f(x)$ be the (unscaled) Gaussian density. The procedure to generate the right edges of the rectangles at $0 = x_0 < x_1 < x_2 < \cdots < x_{255}$, in order to find the equal-area boxes, is shown in pseudo code:

```pseudo
for (i = 254; i > 0; i--)
    $x_i = f^{-1}(v/x_{i+1} + f(x_{i+1}))$;
```

Then the problem is to find the value of $x_{255}$ that solves the next equation:

$$ v = x_{255} \cdot f(x_{255}) + \int_{x_{255}}^{\infty} f(x) \cdot dx $$

Experimentally it is known that for $f(x) = e^{-x^2/2}$ the choice $x_{255} = 3,654,152,885,361,008$ solves the equation above. Thus, the other $x$’s may be found with the first algorithm.

The common area of the rectangles and the base turns out to be $v = 0.00492867323399$, making the efficiency of the rejection procedure 99.33%.

Finally the optimal process pass through forming the next tables:

- An auxiliary table of integers $k_i = [2^{32}(x_{i-1}/x_i)]$.
- Rather than storing a table of $x$’s, store $w_i = 0.5^{32} \cdot x_i$.
- Then the fast part of the generating procedure is: generate $j$.
- Form $i$ from the last 8 bits of $j$. If $j < k_i$, return $x = j \cdot w_i$. (Special values $k_0 = [2^{32} \cdot x_{255} \cdot f(x_{255})/v]$ and $w_0 = 0.5^{32} \cdot v/f(x_{255})$ provide for the fast part when $i = 0$).
- A third table $f_i = f(x_i)$.

The essential part of the generating procedure in C looks like this, assuming the variables have been declared, static tables $k[256], w[256]$ and $f[256]$ are setup, and KISS, UNI are inline generators of unsigned longs or uniform $(0,1)$. The result is a remarkably simple generating procedure, using a 32-bit integer generator such as KISS, described below:

```c
for (;;) {
    j = KISS; i = (j&255);
    x = j \cdot w_i; if (j < k_i) return x;
    if (i == 0) return x from the tail.
    if (UNI*|f_{i-1} - f_i| < |f(x) - f_i|) return x;
}
```
The infinite for is executed until one of the three return's provides the required x (better than 99% of the time from the first return).

The whole C code for the program and the Matlab algorithm for the ziggurat method are in the Annex 2.

5.2 Testing: results in Matlab

The source used for this experience was the sequence provided by the random integer generator function in Matlab: randi(). The specific call for this experience was:

\[ \text{randi}(2^{32}-1,1,1e5); \]

This function \( r = \text{randi}(\text{IMAX},M,N) \) returns an M-by-N matrix containing pseudorandom integer values drawn from the discrete uniform distribution on 1:IMAX.

Why is it used the function randi instead of generator designed and explained in sections above? The reason is the objective of this experience is testing the proper working of the implemented Matlab code. Later the generator designed will be tested properly applying the Ziggurat method implemented.

5.2.1 Histograms

The next plots show the histogram for the Uniform distributed sequence of random integers and the result after applying the Ziggurat method with the Matlab algorithm:

![Graph showing histograms](image-url)
Then the result of applying the algorithm is transforming the Uniform distribution into a Normal distribution. The conversion takes place in 10 seconds approximately, and it performs a good approximation of the Gaussian function.

The plot below shows the final result:

![Output Histogram: Normal random numbers distribution](image)

*Fig. 4: Histogram for output sequence in Ziggurat method*

The histogram function has called:

```plaintext
hist(seq,1000);
```

where variate *seq* contains the final sequence once the method has been applied.

### 5.2.2 First statistics

In order to testing the algorithm the next table shows the mean and the standard deviation for some experiences of applying the ziggurat method in Matlab:

<table>
<thead>
<tr>
<th>Experience n°</th>
<th>Mean (µ)</th>
<th>St. Deviation (σ)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-6.13E-03</td>
<td>9.99E-01</td>
<td>8.69E-02</td>
</tr>
<tr>
<td>2</td>
<td>1.26E-03</td>
<td>9.93E-01</td>
<td>8.19E-02</td>
</tr>
<tr>
<td>3</td>
<td>5.89E-04</td>
<td>9.98E-01</td>
<td>7.74E-02</td>
</tr>
<tr>
<td>4</td>
<td>-5.31E-03</td>
<td>9.95E-01</td>
<td>7.66E-02</td>
</tr>
<tr>
<td>5</td>
<td>-5.03E-03</td>
<td>9.99E-01</td>
<td>7.70E-02</td>
</tr>
<tr>
<td>6</td>
<td>1.28E-03</td>
<td>9.95E-01</td>
<td>7.59E-02</td>
</tr>
<tr>
<td>7</td>
<td>-5.28E-04</td>
<td>9.96E-01</td>
<td>7.57E-02</td>
</tr>
<tr>
<td>8</td>
<td>-1.02E-03</td>
<td>9.97E-01</td>
<td>7.60E-02</td>
</tr>
<tr>
<td>9</td>
<td>-2.60E-03</td>
<td>9.98E-01</td>
<td>7.81E-02</td>
</tr>
<tr>
<td>10</td>
<td>5.05E-03</td>
<td>9.99E-01</td>
<td>7.54E-02</td>
</tr>
<tr>
<td>Mean</td>
<td>-1.25E-03</td>
<td>9.97E-01</td>
<td>7.81E-02</td>
</tr>
</tbody>
</table>

*Table 5: Statistics results for output sequence from Ziggurat method*
The table shows that the Gaussian density has the expected values; it means the standard deviation near to the unit ($\sigma = 1$) and null mean ($\mu = 0$).

5.2.3 Chi-square test

This test is performed by grouping the data into bins, calculating the observed and expected counts for those bins, and computing the chi-square test statistic $\sum ((\text{obs} - \text{exp})^2 / \text{exp})$, where $\text{obs}$ is the observed counts and $\text{exp}$ is the expected counts. This test statistic has an approximate chi-square distribution when the counts are sufficiently large.

In order to test if the density gives a truly Normal data distribution, the Chi-square goodness-of-fit test:

\[ [h, p] = \text{chi2gof}(s); \]

is the first “function test” we are going to use from Matlab.

After 10 different calls to the Ziggurat generator and the test, the results are shown in the table below:

<table>
<thead>
<tr>
<th>Experience nº</th>
<th>p-value</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.17E-01</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>1.11E-02</td>
<td>KO</td>
</tr>
<tr>
<td>3</td>
<td>5.69E-02</td>
<td>OK</td>
</tr>
<tr>
<td>4</td>
<td>6.84E-02</td>
<td>OK</td>
</tr>
<tr>
<td>5</td>
<td>3.89E-02</td>
<td>KO</td>
</tr>
<tr>
<td>6</td>
<td>9.58E-02</td>
<td>OK</td>
</tr>
<tr>
<td>7</td>
<td>2.54E-02</td>
<td>KO</td>
</tr>
<tr>
<td>8</td>
<td>7.75E-01</td>
<td>OK</td>
</tr>
<tr>
<td>9</td>
<td>3.78E-01</td>
<td>OK</td>
</tr>
<tr>
<td>10</td>
<td>2.58E-01</td>
<td>OK</td>
</tr>
</tbody>
</table>

*Table 6: Results of Chi-square test*

The values for the ten experiences above are more than half successful. So the method performs a good approximation for a Gaussian distribution.

5.2.4 Lillifeors test

The next statistical test used is the Lilliefors test. Used when the normal distribution is not specified, does not specify the expected value and the variance.

The test proceeds as follows:

1. First estimate the population mean and population variance based on the data.

2. Then find the maximum discrepancy between the empirical distribution function and the cumulative distribution function (CDF) of the normal distribution with the estimated mean and estimated variance. Just as in the Kolmogorov–Smirnov test, this will be the test statistic.

3. Finally, it is confronted the question of whether the maximum discrepancy is large enough to be statistically significant, thus requiring rejection of the null hypothesis.
By default in Matlab the next call performs the test described above:

\[ [h,p]=lillietest(s); \]

After 10 different calls to the Ziggurat generator and the test, the results are shown in the table below:

<table>
<thead>
<tr>
<th>Experience nº</th>
<th>p-value</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5000</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>0.1523</td>
<td>OK</td>
</tr>
<tr>
<td>3</td>
<td>0.1147</td>
<td>OK</td>
</tr>
<tr>
<td>4</td>
<td>0.2069</td>
<td>OK</td>
</tr>
<tr>
<td>5</td>
<td>0.0418</td>
<td>KO</td>
</tr>
<tr>
<td>6</td>
<td>0.5000</td>
<td>OK</td>
</tr>
<tr>
<td>7</td>
<td>0.1070</td>
<td>OK</td>
</tr>
<tr>
<td>8</td>
<td>0.2197</td>
<td>OK</td>
</tr>
<tr>
<td>9</td>
<td>0.0809</td>
<td>OK</td>
</tr>
<tr>
<td>10</td>
<td>0.0547</td>
<td>OK</td>
</tr>
</tbody>
</table>

*Table 7: Results for Lilliefors test*

The values for the ten experiences above are all successful except one. Although it is necessary to say that the threshold is in 5%, so the one which doesn’t pass the test is pretty close.

### 5.2.5 Kolmogorov-Smirnov

The Kolmogorov-Smirnov test (K–S test) is a nonparametric test for the equality of continuous, one-dimensional probability distributions that can be used to compare a sample with a reference probability distribution (one-sample K–S test). The Kolmogorov–Smirnov test can be modified to serve as a goodness of fit test. In the special case of testing for normality of the distribution, samples are standardized and compared with a standard normal distribution.

The call it is necessary to run the KS test in Matlab is:

\[ [h,p]=kstest(s); \]

After 10 different calls to the Ziggurat generator and the test, the results are shown in the table below:

<table>
<thead>
<tr>
<th>Experience nº</th>
<th>p-value</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8129</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>0.6938</td>
<td>OK</td>
</tr>
<tr>
<td>3</td>
<td>0.2262</td>
<td>OK</td>
</tr>
<tr>
<td>4</td>
<td>0.1506</td>
<td>OK</td>
</tr>
<tr>
<td>5</td>
<td>0.8747</td>
<td>OK</td>
</tr>
<tr>
<td>6</td>
<td>0.4722</td>
<td>OK</td>
</tr>
<tr>
<td>7</td>
<td>0.2304</td>
<td>OK</td>
</tr>
<tr>
<td>8</td>
<td>0.5120</td>
<td>OK</td>
</tr>
<tr>
<td>9</td>
<td>0.7900</td>
<td>OK</td>
</tr>
<tr>
<td>10</td>
<td>0.0762</td>
<td>OK</td>
</tr>
</tbody>
</table>

*Table 8: Results for KS test*
5.2.6 The autocorrelation

The autocorrelation of the sequence generated has the objective to give us information of possible unexpected periods in the output. Thus, the expected situation doesn’t allow periods in the sequence, so the next image shows it:

![Autocorrelation output sequence. Samples:100000](image)

*Fig. 5: Autocorrelation for output sequence from Ziggurat method*

The theory says if there is not cycles along the sequence, there shouldn’t be values along the result of the correlation except in the origin. The autocorrelation plot is the expected; there is one high value just in the origin and zero along the *Lag* axe.
5.2.7 **Normplot**

The purpose of a normal probability plot is to graphically assess whether the data in X could come from a normal distribution. If the data are normal the plot will be linear. Other distribution types will introduce curvature in the plot.

The call in order to displays the normal probability plot is:

\[
\text{normplot}(s); \\
\]

The image result is shown in the table below:

<table>
<thead>
<tr>
<th>Data</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>0.001</td>
</tr>
<tr>
<td>-4</td>
<td>0.003</td>
</tr>
<tr>
<td>-3</td>
<td>0.010</td>
</tr>
<tr>
<td>-2</td>
<td>0.020</td>
</tr>
<tr>
<td>-1</td>
<td>0.050</td>
</tr>
<tr>
<td>0</td>
<td>0.250</td>
</tr>
<tr>
<td>1</td>
<td>0.500</td>
</tr>
<tr>
<td>2</td>
<td>0.750</td>
</tr>
<tr>
<td>3</td>
<td>0.900</td>
</tr>
<tr>
<td>4</td>
<td>0.950</td>
</tr>
<tr>
<td>5</td>
<td>0.980</td>
</tr>
<tr>
<td>6</td>
<td>0.990</td>
</tr>
<tr>
<td>7</td>
<td>0.997</td>
</tr>
<tr>
<td>8</td>
<td>0.999</td>
</tr>
</tbody>
</table>

A red line shows the normal probability curve, just in the right top and in the left bottom it can be seen. The blue crosses are the samples from the tested source, corresponding to the sequence from the Ziggurat function.

The curve resultant, the blue one, fits to the red curve, so that means the plot is linear and therefore the source has a normal probability.

*Fig. 6: Normplot for output sequence from Ziggurat method*
6. The process: generation and transformation

On one hand, the pseudo random generator has been tested, and it is known good statistical pseudo random numbers are generated. Thus, the first step is generating the output, these random numbers uniformly distributed.

On the other hand, it is necessary to convert the sequence through Ziggurat function programmed with Matlab. Once the input data is provided, the function transforms it into a sequence with Gaussian probability density instead of Uniform probability density.

Finally is necessary to make some test that tells whether the whole device chain is working properly and quantify the goodness of it. What follows is a summary of an exhaustive analysis done and provided in Annex 3.

6.1 Schematic process

The next block diagram shows the schema correspondent to the process it is followed by the whole device:

Fig. 7: Schematic process
6.2 Results in Matlab

6.2.1 Histograms

The Image 1 shows the histogram for $10^5$ input:

![Input Histogram: Uniform random numbers distribution](image)

*Fig. 8: Input samples histogram. Nº of samples: $1e5$.*

The Image 2 shows the output histogram corresponding to the input above:

![Output Histogram: Normal random numbers distribution](image)

*Fig. 9: Output samples histogram. Nº of samples: $1e5$*
The *Image 3* shows the histogram for $10^6$ input:

![Input Histogram: Uniform random numbers distribution](image3)

*Fig. 10: Input samples histogram. Nº of samples: 1e6*

The *Image 4* shows the output histogram corresponding to the input above:

![Output Histogram: Normal random numbers distribution](image4)

*Fig. 11: Output samples histogram. Nº of samples: 1e6*
The next tests show the behavior of the generator and the transformation method together. In this section the statistical test have been shown together in the same table of results and the graphical test will help to view how the final probability distribution fit into the expected Gaussian distribution.

### 6.2.2 Statistic

These are recommended normality test, explained in the section above, and give first view whether the sequence comes from standard Normal.

<table>
<thead>
<tr>
<th>Nº samples</th>
<th>Test</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>Mean</td>
<td>-0.0011286</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>Standard deviation</td>
<td>0.99726</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>Chi-square</td>
<td>0.26026</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>Lillie</td>
<td>0.34594</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>KS</td>
<td>0.68972</td>
<td>Ok</td>
</tr>
<tr>
<td>$10^6$</td>
<td>Mean</td>
<td>0.00027465</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>Standard deviation</td>
<td>0.99723</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>Chi-square</td>
<td>7.3011e-006</td>
<td>KO</td>
</tr>
<tr>
<td></td>
<td>Lillie</td>
<td>0.001</td>
<td>KO</td>
</tr>
<tr>
<td></td>
<td>KS</td>
<td>0.17114</td>
<td>Ok</td>
</tr>
</tbody>
</table>

*Table 9: Final output statistics*

### 6.2.3 Autocorrelation test

The image resulting after applying the autocorrelation shows whether the sequence provided, both the input from the generator and the output from the Ziggurat method, has a period.

In both autocorrelations the most important value it is found in the zero Lag axe. In fact, that means there is no period into the sequence, since if it was any period, other points along the Lag axe would have values different to zero.
6.2.4 Normplot test

The purpose of a normal probability plot is to graphically assess whether the data in X could come from a normal distribution. If the data are normal the plot will be linear. Other distribution types will introduce curvature in the plot. Normplot uses midpoint probability plotting positions.

![Normplot final output](image)

*Fig. 13: Normplot final output*

The line resultant, the blue one, fits to the linear function, so that means the source has a normal probability.

6.2.5 Matlab Gaussian plot vs. Ziggurat method plot

The next image shows the profile of the histogram function from two different sources. The black one is the tested output, generated from the Ziggurat function implemented in Matlab, the blue one comes from the sequence generated by the *randn()* function from Matlab. The *randn()* function output gives directly a sequence with Gaussian density distribution.
Both results have similar form, although they are not equal. However, the behavior is the expected, fit into a Gaussian density form with zero mean (corresponding to number 50 in the axe) and the unit as the standard deviation. One can observe both profiles are not completely equal, but this is considered as good behavior, since the sequences are pseudo-random and it is highly improbable they fit completely each other.

6.3 Conclusion of process

Taking into consideration the different results obtained from the tests made with Matlab one can say the process implemented meets the objectives proposed.

With the implemented generator in C code and the Ziggurat transform in Matlab, it is possible to achieve a sequence of $10^6$ samples in a few seconds with a good behavior.

It is true the sequence with $10^5$ samples has better statistical results than longer sequence, but this behavior commonly happened with the Chi-square and Lillie test, however the KZ test is passed mainly in both situations.

The reason can be found in the way Matlab performs the test; Lilliefors test need a table from Monte Carlo results, but Matlab interpolate the results, as one can read in the help explanation:

“LILLIETEST uses a table of critical values computed using Monte Carlo simulation for sample sizes less than 1000 and significance levels between 0.001 and 0.50. The table is larger and more accurate than the table introduced by Lilliefors. Critical values for a test are computed by interpolating into the table, using an analytic approximation when extrapolating for larger sample sizes.”

Anyway, the other graphical and statistical tests are good, so the results and the output could be good for implementation in some future device. Maybe some improvement could be increasing the number of squares in Ziggurat algorithm (255 in the current implementation).
7. Testing physical devices

In this section it is shown the test of normality done with the sequences obtained from different sources and devices in the laboratory at University.

In fact, there are three different devices commonly used in different projects at University. These three devices are: white noise provided by a software (WNA), white noise provided by a University own device (WNNTI) and white noise provided by commercial devise (WNBB).

The two different realizations of the noise from each source have been recorded in order to obtain better and more objective results. It is important to mention that two different methods has been used to record the sequences: using directly a PC (with the Pc sound card) and a professional mixer table.

The same methodology is used in this section to test sequences as in sections before, a serial of statistical tests and some other graphical one’s.

7.1 Statistic

<table>
<thead>
<tr>
<th>Device</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Chi-square</th>
<th>Lillie</th>
<th>KS</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNA-PR-120-PC-R1</td>
<td>-4.8728E-04</td>
<td>0,114</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
<tr>
<td>WNA-PR-120-PC-R2</td>
<td>-4.6576E-04</td>
<td>0,114</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
<tr>
<td>WNA-R-120-PC-R1</td>
<td>-6.9743E-04</td>
<td>0,131</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
<tr>
<td>WNA-R-120-PC-R2</td>
<td>-6.4869E-04</td>
<td>0,130</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
<tr>
<td>WNBB-R-120-MIX-R1</td>
<td>2.87E-06</td>
<td>0,120</td>
<td>0.666</td>
<td>0,306</td>
<td>0</td>
</tr>
<tr>
<td>WNBB-R-120-MIX-R2</td>
<td>-3.51E-06</td>
<td>0,120</td>
<td>0.506</td>
<td>0,5</td>
<td>0</td>
</tr>
<tr>
<td>WNNTI-R-120-MIX-R1</td>
<td>7.99E-06</td>
<td>0,214</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
<tr>
<td>WNNTI-R-120-MIX-R2</td>
<td>-1.94E-06</td>
<td>0,214</td>
<td>0</td>
<td>0,001</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 10: Statistics for physical devices

Firstly, it is necessary to explain the nomenclature of the source in the table:

Secondly, the table shows a common result in all devices; no one pass the KS test. The reason is the KS test is one of the best testing normality, but is made for testing a standard normal distribution ( N(0,1) ), but all devices have no standard distribution.

Finally, there is only one device that passes some test: the WNBB. The possible problem for WNA is the implementation in the software, maybe not good implementation for the pseudo-random generator or the transform algorithm, on the other hand, the WNNTI is an unknown “black box”, so the problems are unpredictable.

Now it is necessary complete the results, tests and conclusions with the graphical results in the next section.
7.2 Graphical results

7.2.1 WNA-PR-120-PC-R1

7.2.2 WNA-PR-120-PC-R2
The same behavior in both realizations, there is a problem with samples in 0. The histogram and the comparison graphics show there are a high peak of samples equal to 0, and that is not good for the proposal.

There is another big problem, the pseudo random generator have a important correlation, so that means the period is extremely short. Thus, this device is not a good source of WGN.

### 7.2.3 WNA-R-120-PC-R1
The software allows switching between random and pseudorandom, although the methods are not specified. Anyway, there is the same problem with samples equal to 0, there are too much, in fact, there is an important peak.

The behavior of the period has changed radically; there is not period here, so that is a good improvement. Anyway, the results show poor results, so it is recommended to use other devices or be aware the deficiencies.
7.2.5 Wnbb_r_120_mix_r1

7.2.6 Wnbb_r_120_mix_r2
The second device shows much better results than the first one. The histogram and the comparison graphics are near to the output from Matlab’s software, and the other graphical test shows the expected behavior.

Except the KS test, this device can be perfectly used in projects and one can be sure the source of WGN is the properly source.

7.2.7 \textit{Wanti_r_120_mix_r1}
This is the last tested device, and it has a middle quality. The histogram seems to be good, but then the comparison shows that the Gaussian profile is not so close to a good profile. The autocorrelation is the expected, without appreciable period, and the Normplot shows there is deviation from the linear, so the source is not totally standard Normal.

7.3 Summary of results

There is a common problem in all devices; the standard deviation differs from the standard normal in one magnitude order. However, is only a problem because the test are expecting the standard normal N (0,1), this must be taken into consideration.

The results show there are different quality categories. The recommended device in practical situations is the commercial device, corresponding to the results of WNBB. The software device (WNA) is not recommended in professional projects, because the output is extremely poor in comparison of other sources. The last one (WNNTI) has a middle quality and can be used in semi-professional situations or in academic laboratories, or other applications.
Normality test suite
8. Normality Test suite in Matlab

In order to speed up the process of testing the sequences generated it has been convenient to program a tests suite in Matlab that allows to test it easily.

The test suite is simple and interactive; it is necessary to introduce the sample size you want to test and the name of file, without the extension .norm or .uni, and then load it.

Once the files are loaded, it depends on the length of the sequence it takes some seconds, and then the RUN TEST button is enable.

By default all tests are selected, both numerical and graphical, but one can chose the preferred test. Then is time to RUN the test. Once the tests are done, the numerical results appear in a table and the graphical tests are shown separately. Using a popup menu the user can choose the graphical test want to see.

Additionally the current image corresponding to graphical test can be saved in the current path with the chosen name.

The interface looks like:

This Normality Suite gives the test used during the project, divided in Numerical test; Mean, Standard deviation, Chi-square test, Lilliefors test and Kolmogorov-Smirnov, and the Graphical test; Histogram, Autocorrelation, Normplot and Comparison.

Additionally there is a graphic, maybe not a test, but gives the frequency spectrum of the sequence. This allows seeing how the output works in frequency, ideally has a flat form.
Final conclusions
9. Final conclusions

The initially proposed targets now must be checked in order to extract proper conclusions and assess whether have been accomplished.

First, it was necessary to design and program a code that allows obtaining the proper sequence. That sequence had to have special features in order to become a good random sequence.

The necessary information was collected and the KISS random numbers generator was chosen to be the main generator. It is true that there are latest generators, like Twister-Marsenne, used by default in Matlab software, but it was not necessary such a long period and features. Additionally, the KISS generator code has been optimized in C language and has a high performance.

The generator pass the Diehard battery test, commonly used in many projects as the reference test battery, and the results are entirely satisfactory. In short, the generator obtained meets the expectations.

Second, the output sequence generated by the KISS has a statistically uniform distribution, therefore was necessary to transform the sequence into statistically Gaussian distribution.

The procedure was the same as followed in the first objective, collecting the necessary information about normalization methods, choosing the most appropriate and implementing and testing the code. In that case, the method has been coded in Matlab software, in order to run the test easily. The implementation is in that case optimized for Matlab software, using the power of vectors instead of loops, thus the algorithm is fast and compact.

The Ziggurat method was chosen and has been tested using commonly statistical and graphical normality tests, all of them with Matlab software. In order to have a useful tool, during the project and in the future, a uniformity test suit has been implemented. The results from the tests show a correct behavior for the transformation method.

It is necessary to note there could be an important improvement in this point, the number of boxes in the Ziggurat method. Maybe this is the reason why some tests have not been totally passed. The most reasonable step would be to increase the number from 255 to 512 boxes. It means change some values in the algorithm and adequate it properly.

Third, the devices available from the University had to be tested and compared. The sequences generated have been tested, from different sources and different recording methods. It is true the Mixer is a better recording method, but there are not major differences between them. The important difference is the quality of the output, each one has different. As it is said in the corresponding section, each one must be used in the proper situation.

Finally, it is necessary to compare the results of the designed white noise generator and the physical devices. The designed is near to the highest device in the laboratory, so it could be interesting to implement it and be sure will be near to high performance devices.

In summary, the main objectives have been achieved, and additional material, the test suite has been created. Now this project allows to someone have enough material to perform a physical implementation of a White Noise Generator device dedicated to Acoustic Measurements.
Annexes
10. Annex 1. Diehard exhaustive results

10.1 Birthday Spacing

Choose m birthdays in a "year" of n days. List the spaces between the birthdays. Let j be the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean \( m^3/4n \). Experience shows n must be quite large, say \( n \geq 2^{18} \), for comparing the results to the Poisson distribution with that mean. This test uses \( n = 2^{24} \) and \( m = 2^{10} \), so that the underlying distribution for j is taken to be Poisson with \( \lambda = 2^{30}/2^{26} = 16 \). A sample of 200 j's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from 32-bit integers in the specified file. The file is closed and reopened, and then bits 2-25 of the same integers are used to provide birthdays, and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST.

<table>
<thead>
<tr>
<th>Bits used</th>
<th>Mean</th>
<th>Chisqr</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 24</td>
<td>15.63</td>
<td>28.2222</td>
<td>0.957593</td>
</tr>
<tr>
<td>2 to 25</td>
<td>15.94</td>
<td>10.6530</td>
<td>0.125974</td>
</tr>
<tr>
<td>3 to 26</td>
<td>15.99</td>
<td>11.8487</td>
<td>0.190787</td>
</tr>
<tr>
<td>4 to 27</td>
<td>15.61</td>
<td>11.2263</td>
<td>0.155438</td>
</tr>
<tr>
<td>5 to 28</td>
<td>15.94</td>
<td>8.4581</td>
<td>0.044254</td>
</tr>
<tr>
<td>6 to 29</td>
<td>15.70</td>
<td>23.8143</td>
<td>0.875430</td>
</tr>
<tr>
<td>7 to 30</td>
<td>15.58</td>
<td>19.8569</td>
<td>0.718385</td>
</tr>
<tr>
<td>8 to 31</td>
<td>15.93</td>
<td>10.0852</td>
<td>0.099999</td>
</tr>
<tr>
<td>9 to 32</td>
<td>15.82</td>
<td>21.1346</td>
<td>0.779671</td>
</tr>
</tbody>
</table>

Chi-square degrees of freedom: 17. P-value for KS-test on those 9 p-values: **0.274469**

10.2 GCD

Let the (32-bit) RNG produce two successive integers u, v. Use Euclides algorithm to find the gcd, say x, of u and v. Let k be the number of steps needed to get x. Then k is approximately binomial with \( p=0.376 \) and \( n=50 \), while the distribution of x is very close to \( \Pr(x = i) = c/i^2 \), with \( c = 6/\pi^2 \). The gcd test uses ten million such pairs u, v to see if the resulting frequencies of k's and x's are consistent with the above distributions. Congruential RNG's---even those with prime modulus---fail this test for the distribution of k, the number of steps, and often for the distribution of gcd values x as well.

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
</table>

Not enough random numbers for this test. Minimum is 20000000. The test is skipped.

10.3 Gorilla

It concerns strings formed from specified bits in 32-bit integers from the RNG. We specify the bit position to be studied, from 0 to 31, say bit 3. Then we generate 67,108,889 \( (2^{26} + 25) \) numbers from the generator and form a string of \( 2^{26} + 25 \) bits by taking bit 3 from each of those numbers. In that string of \( 2^{26} + 25 \) bits we count the number of 26-bit segments that do not appear. That count should be approximately normal with mean 24687971 and std. deviation 4170. This leads to a normal z-score and hence to a p-value. The test is applied for each bit position 0 (leftmost) to 31.

59
10.4 Overlapping Permutations

It looks at a sequence of ten million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. Thus the 5th, 6th, 7th, ... numbers each provide a state. As many thousands of state transitions are observed, accumulative counts are made of the number of occurrences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test that the 120 cell counts came from the specified (asymptotic) distribution with the specified means and 120x120 covariance.

10.5 Ranks of 31x31 and 32x32 matrices

The leftmost 31 bits of 31 random integers from the test sequence are used to form a 31x31 binary matrix over the field \{0,1\}. The rank is determined. That rank can be from 0 to 31, but ranks<28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 31,30,28 and <=28. (The 31x31 choice is based on the unjustified popularity of the proposed "industry standard" generator \(x(n) = 16807 \times x(n-1) \mod 2^{31}-1\), not a very good one).

### Results

Rank test for binary matrices (31x31) for KISS:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Observed</th>
<th>Expected</th>
<th>((O - E)^2/E)</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r \leq 28)</td>
<td>219</td>
<td>211.4</td>
<td>0.272</td>
<td>0.272</td>
</tr>
<tr>
<td>(r = 29)</td>
<td>5184</td>
<td>5134</td>
<td>0.487</td>
<td>0.759</td>
</tr>
<tr>
<td>(r = 30)</td>
<td>23084</td>
<td>23103.0</td>
<td>0.016</td>
<td>0.774</td>
</tr>
<tr>
<td>(r = 31)</td>
<td>11513</td>
<td>11551.5</td>
<td>0.128</td>
<td>0.903</td>
</tr>
</tbody>
</table>

Chi-square = 0.903 with df = 3; p-value = 0.175

Rank test for binary matrices (32x32) for KISS

<table>
<thead>
<tr>
<th>Rank</th>
<th>Observed</th>
<th>Expected</th>
<th>((O - E)^2/E)</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r \leq 29)</td>
<td>221</td>
<td>211.4</td>
<td>0.434</td>
<td>0.434</td>
</tr>
<tr>
<td>(r = 30)</td>
<td>5233</td>
<td>5134.0</td>
<td>1.909</td>
<td>2.343</td>
</tr>
<tr>
<td>(r = 31)</td>
<td>23106</td>
<td>23103.0</td>
<td>0.000</td>
<td>2.343</td>
</tr>
<tr>
<td>(r = 32)</td>
<td>11440</td>
<td>11551.5</td>
<td>1.077</td>
<td>3.420</td>
</tr>
</tbody>
</table>

Chi-square = 3.420 with df = 3; p-value = 0.669
10.6 Ranks of 6x8 Matrices

From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks <=4, 5 and 6.

<table>
<thead>
<tr>
<th>Bits</th>
<th>P value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 8</td>
<td>1.00000</td>
</tr>
<tr>
<td>2 to 9</td>
<td>1.00000</td>
</tr>
<tr>
<td>3 to 10</td>
<td>1.00000</td>
</tr>
<tr>
<td>4 to 11</td>
<td>1.00000</td>
</tr>
<tr>
<td>5 to 12</td>
<td>1.00000</td>
</tr>
<tr>
<td>6 to 13</td>
<td>1.00000</td>
</tr>
<tr>
<td>7 to 14</td>
<td>1.00000</td>
</tr>
<tr>
<td>8 to 15</td>
<td>0.99998</td>
</tr>
<tr>
<td>9 to 16</td>
<td>0.99939</td>
</tr>
<tr>
<td>10 to 17</td>
<td>0.89212</td>
</tr>
<tr>
<td>11 to 18</td>
<td>0.70379</td>
</tr>
<tr>
<td>12 to 19</td>
<td>0.70656</td>
</tr>
<tr>
<td>13 to 20</td>
<td>0.11962</td>
</tr>
<tr>
<td>14 to 21</td>
<td>0.84272</td>
</tr>
<tr>
<td>15 to 22</td>
<td>0.22832</td>
</tr>
<tr>
<td>16 to 23</td>
<td>0.75638</td>
</tr>
<tr>
<td>17 to 24</td>
<td>0.08370</td>
</tr>
<tr>
<td>18 to 25</td>
<td>0.25157</td>
</tr>
<tr>
<td>19 to 26</td>
<td>0.64053</td>
</tr>
<tr>
<td>20 to 27</td>
<td>0.73316</td>
</tr>
<tr>
<td>21 to 28</td>
<td>0.92168</td>
</tr>
<tr>
<td>22 to 29</td>
<td>0.87055</td>
</tr>
<tr>
<td>23 to 30</td>
<td>0.72411</td>
</tr>
<tr>
<td>24 to 31</td>
<td>0.81413</td>
</tr>
<tr>
<td>25 to 32</td>
<td>0.99513</td>
</tr>
</tbody>
</table>

10.7 Monkey Tests on 20-bit Words

The file under test is viewed as a stream of bits. Call them b1,b2,... . Consider an alphabet with two "letters", 0 and 1 and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is b1b2...b20, the second is b2b3...b21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^21 overlapping 20-letter words. There are 2^20 possible 20 letter words. For a truly random string of 2^21+19 bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus (j-141909)/428 should be a standard normal variate (z score) that leads to a uniform [0,1) p value. The test is repeated twenty times.
Expected behavior: 20 bits/word, 2097152 words 20 bit-streams. Nº missing words should average 141909.33 with sigma=428.00

<table>
<thead>
<tr>
<th>Bit-stream</th>
<th>Nº missing words</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>143239</td>
<td>3.11</td>
<td>0.999054</td>
</tr>
<tr>
<td>2</td>
<td>142978</td>
<td>2.50</td>
<td>0.993736</td>
</tr>
<tr>
<td>3</td>
<td>142061</td>
<td>0.35</td>
<td>0.638469</td>
</tr>
<tr>
<td>4</td>
<td>142794</td>
<td>2.07</td>
<td>0.980632</td>
</tr>
<tr>
<td>5</td>
<td>142972</td>
<td>2.48</td>
<td>0.993484</td>
</tr>
<tr>
<td>6</td>
<td>143422</td>
<td>3.53</td>
<td>0.999796</td>
</tr>
<tr>
<td>7</td>
<td>142922</td>
<td>2.37</td>
<td>0.991011</td>
</tr>
<tr>
<td>8</td>
<td>142301</td>
<td>0.92</td>
<td>0.819935</td>
</tr>
<tr>
<td>9</td>
<td>142226</td>
<td>0.74</td>
<td>0.770315</td>
</tr>
<tr>
<td>10</td>
<td>142376</td>
<td>1.09</td>
<td>0.862221</td>
</tr>
<tr>
<td>11</td>
<td>142236</td>
<td>0.76</td>
<td>0.777342</td>
</tr>
<tr>
<td>12</td>
<td>142787</td>
<td>2.05</td>
<td>0.979849</td>
</tr>
<tr>
<td>13</td>
<td>142428</td>
<td>1.21</td>
<td>0.887214</td>
</tr>
<tr>
<td>14</td>
<td>142358</td>
<td>1.05</td>
<td>0.852749</td>
</tr>
<tr>
<td>15</td>
<td>143664</td>
<td>4.10</td>
<td>0.999979</td>
</tr>
<tr>
<td>16</td>
<td>142610</td>
<td>1.64</td>
<td>0.949193</td>
</tr>
<tr>
<td>17</td>
<td>142747</td>
<td>1.96</td>
<td>0.974836</td>
</tr>
<tr>
<td>18</td>
<td>143379</td>
<td>3.43</td>
<td>0.999702</td>
</tr>
<tr>
<td>19</td>
<td>142421</td>
<td>1.20</td>
<td>0.884052</td>
</tr>
<tr>
<td>20</td>
<td>142791</td>
<td>2.06</td>
<td>0.980300</td>
</tr>
</tbody>
</table>

10.8 Monkey Tests OPSO,OQSO,DNA

The OPSO (Overlapping-Pairs-Sparse-Occupancy) test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32-bit integer in the sequence to be tested. OPSO generates $2^{21}$ (overlapping) 2-letter words (from $2^{21}+1$ "keystrokes") and counts the number of missing words---that is, 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, sigma 290. Thus (missingwrd\(s\)-141909)/290 should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on.

<table>
<thead>
<tr>
<th>Bit used</th>
<th>Nº missing words</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 to 32</td>
<td>143783</td>
<td>6.4609</td>
<td>1.000000</td>
</tr>
<tr>
<td>22 to 31</td>
<td>143309</td>
<td>4.8264</td>
<td>0.999999</td>
</tr>
<tr>
<td>21 to 30</td>
<td>142171</td>
<td>0.9023</td>
<td>0.816554</td>
</tr>
<tr>
<td>20 to 29</td>
<td>142561</td>
<td>2.2471</td>
<td>0.987684</td>
</tr>
<tr>
<td>19 to 28</td>
<td>142296</td>
<td>1.3333</td>
<td>0.908791</td>
</tr>
</tbody>
</table>
The test OQSO (Overlapping-Quadruples-Sparse-Occupancy) is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of $2^{21}$ four-letter words, $(2^{21}+3 \"keystrokes\")$, is again 141909, with sigma = 295. The mean is based on theory; sigma comes from extensive simulation.

### Results

<table>
<thead>
<tr>
<th>Bit used</th>
<th>Nº missing words</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 to 32</td>
<td>144672</td>
<td>9.3650</td>
<td>1.000000</td>
</tr>
<tr>
<td>27 to 31</td>
<td>143284</td>
<td>4.6599</td>
<td>0.999998</td>
</tr>
<tr>
<td>26 to 30</td>
<td>142596</td>
<td>2.3277</td>
<td>0.990036</td>
</tr>
<tr>
<td>25 to 29</td>
<td>141844</td>
<td>-0.2215</td>
<td>0.412368</td>
</tr>
<tr>
<td>24 to 28</td>
<td>141935</td>
<td>0.0870</td>
<td>0.534671</td>
</tr>
<tr>
<td>23 to 27</td>
<td>142286</td>
<td>1.2768</td>
<td>0.899172</td>
</tr>
<tr>
<td>22 to 26</td>
<td>142526</td>
<td>2.0904</td>
<td>0.981709</td>
</tr>
<tr>
<td>21 to 25</td>
<td>142290</td>
<td>1.2904</td>
<td>0.901545</td>
</tr>
<tr>
<td>20 to 24</td>
<td>141899</td>
<td>-0.0350</td>
<td>0.486033</td>
</tr>
<tr>
<td>19 to 23</td>
<td>142028</td>
<td>0.4023</td>
<td>0.656258</td>
</tr>
<tr>
<td>18 to 22</td>
<td>142032</td>
<td>0.4158</td>
<td>0.661233</td>
</tr>
<tr>
<td>17 to 21</td>
<td>142164</td>
<td>0.8633</td>
<td>0.806011</td>
</tr>
<tr>
<td>16 to 20</td>
<td>141660</td>
<td>-0.8452</td>
<td>0.199003</td>
</tr>
<tr>
<td>15 to 19</td>
<td>141682</td>
<td>-0.7706</td>
<td>0.220469</td>
</tr>
<tr>
<td>14 to 18</td>
<td>141844</td>
<td>-0.2215</td>
<td>0.412368</td>
</tr>
<tr>
<td>13 to 17</td>
<td>142091</td>
<td>0.6158</td>
<td>0.730997</td>
</tr>
<tr>
<td>12 to 16</td>
<td>142146</td>
<td>0.8023</td>
<td>0.788802</td>
</tr>
<tr>
<td>11 to 15</td>
<td>141695</td>
<td>-0.7265</td>
<td>0.233753</td>
</tr>
<tr>
<td>10 to 14</td>
<td>141600</td>
<td>-1.0486</td>
<td>0.147187</td>
</tr>
</tbody>
</table>
The DNA test considers an alphabet of 4 letters: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are $2^{20}$ possible words, and the mean number of missing words from a string of $2^{21}$ (overlapping) 10-letter words ($2^{21}$+9 "keystrokes") is 141909. The standard deviation $\sigma=339$ was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation.)

<table>
<thead>
<tr>
<th>Bit used</th>
<th>Nº missing words</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 to 32</td>
<td>143831</td>
<td>5.6686</td>
<td>1.00000</td>
</tr>
<tr>
<td>30 to 31</td>
<td>142639</td>
<td>2.1524</td>
<td>0.984318</td>
</tr>
<tr>
<td>29 to 30</td>
<td>142916</td>
<td>2.9695</td>
<td>0.998509</td>
</tr>
<tr>
<td>28 to 29</td>
<td>142748</td>
<td>2.4740</td>
<td>0.993319</td>
</tr>
<tr>
<td>27 to 28</td>
<td>141512</td>
<td>-1.1721</td>
<td>0.120586</td>
</tr>
<tr>
<td>26 to 27</td>
<td>142491</td>
<td>1.7158</td>
<td>0.956904</td>
</tr>
<tr>
<td>25 to 26</td>
<td>142467</td>
<td>1.6450</td>
<td>0.950020</td>
</tr>
<tr>
<td>24 to 25</td>
<td>142087</td>
<td>0.5241</td>
<td>0.699896</td>
</tr>
<tr>
<td>23 to 24</td>
<td>141809</td>
<td>-0.2960</td>
<td>0.383631</td>
</tr>
<tr>
<td>22 to 23</td>
<td>141987</td>
<td>0.2291</td>
<td>0.590610</td>
</tr>
<tr>
<td>21 to 22</td>
<td>141750</td>
<td>-0.4700</td>
<td>0.319178</td>
</tr>
<tr>
<td>20 to 21</td>
<td>142013</td>
<td>0.3058</td>
<td>0.620126</td>
</tr>
<tr>
<td>19 to 20</td>
<td>141955</td>
<td>0.1347</td>
<td>0.553583</td>
</tr>
<tr>
<td>18 to 19</td>
<td>141995</td>
<td>0.2527</td>
<td>0.599755</td>
</tr>
<tr>
<td>17 to 18</td>
<td>141404</td>
<td>-1.4906</td>
<td>0.068027</td>
</tr>
<tr>
<td>16 to 17</td>
<td>142307</td>
<td>1.1731</td>
<td>0.879616</td>
</tr>
<tr>
<td>15 to 16</td>
<td>142386</td>
<td>1.4061</td>
<td>0.920154</td>
</tr>
<tr>
<td>14 to 15</td>
<td>142323</td>
<td>1.2203</td>
<td>0.888818</td>
</tr>
<tr>
<td>13 to 14</td>
<td>142642</td>
<td>2.1613</td>
<td>0.984663</td>
</tr>
<tr>
<td>12 to 13</td>
<td>141604</td>
<td>-0.9007</td>
<td>0.183880</td>
</tr>
<tr>
<td>11 to 12</td>
<td>142307</td>
<td>1.1731</td>
<td>0.879616</td>
</tr>
<tr>
<td>10 to 11</td>
<td>142093</td>
<td>0.5418</td>
<td>0.706022</td>
</tr>
<tr>
<td>9 to 10</td>
<td>142416</td>
<td>1.4946</td>
<td>0.932491</td>
</tr>
<tr>
<td>8 to 9</td>
<td>142563</td>
<td>1.9282</td>
<td>0.973087</td>
</tr>
<tr>
<td>7 to 8</td>
<td>144175</td>
<td>6.6834</td>
<td>1.000000</td>
</tr>
<tr>
<td>6 to 7</td>
<td>143925</td>
<td>5.9459</td>
<td>1.000000</td>
</tr>
<tr>
<td>5 to 6</td>
<td>143684</td>
<td>5.2350</td>
<td>1.000000</td>
</tr>
<tr>
<td>4 to 5</td>
<td>142814</td>
<td>2.6686</td>
<td>0.996192</td>
</tr>
<tr>
<td>3 to 4</td>
<td>142418</td>
<td>1.5005</td>
<td>0.933258</td>
</tr>
</tbody>
</table>

The DNA test considers an alphabet of 4 letters: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are $2^{20}$ possible words, and the mean number of missing words from a string of $2^{21}$ (overlapping) 10-letter words ($2^{21}$+9 "keystrokes") is 141909. The standard deviation $\sigma=339$ was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation.)
10.9 Count the 1's in a Stream of Bytes

Consider the file under test as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's with probabilities 1, 8, 28, 56, 70, 56, 28, 8, 1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0, 1, or 2 yields A, 3 yields B, 4 yields C, 5 yields D and 6, 7 or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37, 56, 70, 56, 37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test: Q_5-Q_4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

Results

Degrees of freedom: 5^4-5^3=2500; sample size: 2560000

<table>
<thead>
<tr>
<th>Chisquare</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2425.90</td>
<td>-1.048</td>
<td>0.147326</td>
</tr>
</tbody>
</table>

10.10 Count the 1’s in Specific Bytes

Consider the file under test as a stream of 32-bit integers. From each integer, a specific byte is chosen, say the left-most: bits 1 to 8. Each byte can contain from 0 to 8 1's, with probability 1, 8, 28, 56, 70, 56, 28, 8, 1 over 256. Now let the specified bytes from successive integers provide a string of (overlapping) 5-letter words, each "letter" taking values A, B, C, D, E. The letters are determined by the number of 1's, in that byte: 0, 1, or 2 --> A, 3 --> B, 4 --> C, 5 --> D, and 6, 7 or 8 --> E. Thus we have a monkey at a typewriter hitting five keys with various probabilities: 37, 56, 70, 56, 37 over 256. There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test: Q_5-Q_4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

Results

Degrees of freedom: 5^4-5^3=2500; sample size: 256000

<table>
<thead>
<tr>
<th>Bits used</th>
<th>Chisquare</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 8</td>
<td>2579.12</td>
<td>1.119</td>
<td>0.868399</td>
</tr>
<tr>
<td>2 to 9</td>
<td>2508.49</td>
<td>0.120</td>
<td>0.547763</td>
</tr>
<tr>
<td>3 to 10</td>
<td>2733.62</td>
<td>3.304</td>
<td>0.999523</td>
</tr>
<tr>
<td>4 to 11</td>
<td>2666.30</td>
<td>2.352</td>
<td>0.990659</td>
</tr>
<tr>
<td>5 to 12</td>
<td>2773.79</td>
<td>3.872</td>
<td>0.999946</td>
</tr>
<tr>
<td>6 to 13</td>
<td>2697.14</td>
<td>2.788</td>
<td>0.997349</td>
</tr>
<tr>
<td>7 to 14</td>
<td>2645.37</td>
<td>2.056</td>
<td>0.980101</td>
</tr>
</tbody>
</table>
In a square of side 100, randomly "park" a car—a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. If we plot n: the number of attempts, versus k: the number successfully parked, we get a curve that should be similar to those provided by a perfect random numbergenerator. A simple characterization of the random experiment is used: k, the number of cars successfully parked after n=12,000 attempts. Simulation shows that k should average 3523 with 
and be approximate to normally distributed. Thus (k-3523)/21.9 should serve as a standard normal variable, which, converted to a p uniform in [0,1), provides input to a KSTEST based on a sample of 10.

Results

Below 10 tests of 12000 tries, the average nº of successes should be 3523.0 with $\sigma = 21.9$.

Square side=100, avg. no. parked=3432.80 sample std.=17.94
10.12 Minimum Distance Test

It does this ten times: choose \( n=8000 \) random points in a square of side 10000. Find \( d \), the minimum distance between the \((n^2-n)/2\) pairs of points. If the points are truly independent uniform, then \( d^2 \), the square of the minimum distance should be (very close to) exponentially distributed with mean 0.995. Thus \( 1-\exp(-d^2/0.995) \) should provide a p-value and a KSTEST on the resulting 10 values serves as a test of uniformity for those samples of 8000 random points in a square.

Results

<table>
<thead>
<tr>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5238</td>
</tr>
<tr>
<td>0.3919</td>
</tr>
<tr>
<td>0.4563</td>
</tr>
<tr>
<td>0.4010</td>
</tr>
<tr>
<td>0.0864</td>
</tr>
<tr>
<td>0.1813</td>
</tr>
<tr>
<td>0.9167</td>
</tr>
<tr>
<td>0.6559</td>
</tr>
<tr>
<td>0.0083</td>
</tr>
<tr>
<td>0.0955</td>
</tr>
</tbody>
</table>

The KS test for those 10 p-values: **0.222272**

10.13 Random Spheres Test

Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean \( 120\pi/3 \). Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive simulation). The 3DSPHERES test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of \( 1-\exp(-r^3/30.) \), then a KSTEST is done on the 20 p-values.

Results

<table>
<thead>
<tr>
<th>Sample nº</th>
<th>( r^3 )</th>
<th>Equivalent uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36.990</td>
<td>0.708586</td>
</tr>
<tr>
<td>2</td>
<td>15.940</td>
<td>0.412173</td>
</tr>
<tr>
<td>3</td>
<td>16.474</td>
<td>0.422542</td>
</tr>
<tr>
<td>4</td>
<td>62.992</td>
<td>0.877510</td>
</tr>
<tr>
<td>5</td>
<td>21.332</td>
<td>0.508672</td>
</tr>
<tr>
<td>6</td>
<td>114.982</td>
<td>0.978350</td>
</tr>
<tr>
<td>7</td>
<td>100.249</td>
<td>0.964621</td>
</tr>
<tr>
<td>8</td>
<td>76.449</td>
<td>0.921787</td>
</tr>
<tr>
<td>9</td>
<td>53.225</td>
<td>0.830377</td>
</tr>
<tr>
<td>10</td>
<td>45.623</td>
<td>0.781457</td>
</tr>
<tr>
<td>11</td>
<td>32.332</td>
<td>0.639634</td>
</tr>
<tr>
<td>12</td>
<td>4.447</td>
<td>0.137772</td>
</tr>
<tr>
<td>13</td>
<td>33.399</td>
<td>0.671526</td>
</tr>
<tr>
<td>14</td>
<td>14.651</td>
<td>0.386366</td>
</tr>
</tbody>
</table>
Random integers are floated to get uniforms on [0,1). Starting with $k=2^{31}=2147483647$, the test finds $j$, the number of iterations necessary to reduce $k$ to 1, using the reduction $k=\text{ceiling}(k*U)$, with $U$ provided by floating integers from the file being tested. Such $j$'s are found 100,000 times, then counts for the number of times $j$ was $\leq 6, 7, ..., 47, >48$ are used to provide a chi-square test for cell frequencies.

Results

Table of standardized frequency counts $(\text{obs-exp})^2/\text{exp}$ for $j=(1,..,6), 7, ..., 47, (48,...) :

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq 6$</td>
<td>25.4</td>
<td>13.0</td>
<td>11.8</td>
<td>12.6</td>
<td>33.1</td>
<td>34.1</td>
<td>32.9</td>
<td>28.2</td>
<td>22.5</td>
<td>16.7</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>-4.2</td>
<td>35.6</td>
<td>-9.8</td>
<td>-11.7</td>
<td>-12.5</td>
<td>-12.1</td>
<td>-12.6</td>
<td>-10.0</td>
<td>-9.3</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>-7.5</td>
<td>-5.8</td>
<td>10.5</td>
<td>-9.5</td>
<td>-5.3</td>
<td>-5.3</td>
<td>-3.4</td>
<td>-1.6</td>
<td>-2.3</td>
<td>-0.8</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>$\geq 48$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0.4</td>
<td>1.3</td>
<td>-0.8</td>
<td>-0.5</td>
<td>0.2</td>
<td>-0.8</td>
<td>-0.7</td>
<td>-1.3</td>
<td>1.0</td>
<td>-1.1</td>
</tr>
</tbody>
</table>

Chi-square with 42 degrees of freedom: 8778.301863, z-score=953.208681, p-value=1.000000

10.15 Overlapping Sums Test

Integers are floated to get a sequence $U(1), U(2), ...$ of uniform [0,1) variables. Then overlapping sums, $S(1)=U(1)+...+U(100)$, $S2=U(2)+...+U(101)$,... are formed. The $S$'s are virtually normal with a certain covariance matrix. A linear transformation of the $S$'s converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST.

Results
<table>
<thead>
<tr>
<th>Test n°</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.036921</td>
</tr>
<tr>
<td>2</td>
<td>0.510911</td>
</tr>
<tr>
<td>3</td>
<td>0.746414</td>
</tr>
<tr>
<td>4</td>
<td>0.943120</td>
</tr>
<tr>
<td>5</td>
<td>0.276076</td>
</tr>
<tr>
<td>6</td>
<td>0.838749</td>
</tr>
<tr>
<td>7</td>
<td>0.351691</td>
</tr>
<tr>
<td>8</td>
<td>0.991275</td>
</tr>
<tr>
<td>9</td>
<td>0.219279</td>
</tr>
<tr>
<td>10</td>
<td>0.799875</td>
</tr>
</tbody>
</table>

P-value for 10 KS tests on 100 sums: **0.566157**

### 10.16 Runs Up and Down Test

An up-run of length \( n \) has \( x_1 < \ldots < x_n \) and \( x_n > x_{n+1} \), while a down-run of length \( n \) has \( x_1 > \ldots > x_n \) and \( x_n < x_{n+1} \). The value that ends a run is not part of the run that follows, so a long sequence of numbers contains independent runs, up or down, of length \( k \) with probability \( 2k/(k+1)! \). This test generates values until 100,000 up-runs and 100,000 down-runs are found, then it tests to see if the frequencies of lengths for 100,000 of each type are consistent with underlying theory. It also tests to see if the number of values needed to get 100,000 of each type is consistent with theory.

**Results**

Test for lengths of runs up and runs down, 100,000 each for KISS:

<table>
<thead>
<tr>
<th>Length</th>
<th>Expected</th>
<th>Up Runs</th>
<th>((O - E)^2/E)</th>
<th>Down Runs</th>
<th>((O - E)^2/E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>66666.67</td>
<td>66742</td>
<td>0.09</td>
<td>66099</td>
<td>4.83</td>
</tr>
<tr>
<td>3</td>
<td>25000.00</td>
<td>24815</td>
<td>1.37</td>
<td>25533</td>
<td>11.36</td>
</tr>
<tr>
<td>4</td>
<td>66666.67</td>
<td>6769</td>
<td>1.57</td>
<td>6685</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>1388.89</td>
<td>1408</td>
<td>0.26</td>
<td>1370</td>
<td>0.26</td>
</tr>
<tr>
<td>6</td>
<td>238.10</td>
<td>216</td>
<td>2.05</td>
<td>277</td>
<td>6.36</td>
</tr>
<tr>
<td>7</td>
<td>34.72</td>
<td>41</td>
<td>1.14</td>
<td>31</td>
<td>0.40</td>
</tr>
<tr>
<td>8</td>
<td>4.96</td>
<td>8</td>
<td>1.86</td>
<td>4</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Number of rngs required: 688744, p-value: **0.653**

### 10.17 The Craps Test

It plays 200,000 games of craps, counts the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean \( 200000p \) and variance \( 200000p(1-p) \), and \( p=244/495 \). Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts. Each
32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result.

Results

<table>
<thead>
<tr>
<th>Nº of wins:</th>
<th>Observed</th>
<th>Expected</th>
<th>z-score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>98700</td>
<td>98585.9</td>
<td>0.511</td>
<td>0.69515</td>
</tr>
</tbody>
</table>

Analysis of Throws-per-Game:

<table>
<thead>
<tr>
<th>Throws</th>
<th>Observed</th>
<th>Expected</th>
<th>Chi-square</th>
<th>Sum of $(O - E)^2 / E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67016</td>
<td>66666.7</td>
<td>1.831</td>
<td>1.831</td>
</tr>
<tr>
<td>2</td>
<td>37496</td>
<td>37654.3</td>
<td>0.666</td>
<td>2.496</td>
</tr>
<tr>
<td>3</td>
<td>26693</td>
<td>26954.7</td>
<td>2.541</td>
<td>5.038</td>
</tr>
<tr>
<td>4</td>
<td>19306</td>
<td>19313.5</td>
<td>0.003</td>
<td>5.041</td>
</tr>
<tr>
<td>5</td>
<td>13772</td>
<td>13851.4</td>
<td>0.455</td>
<td>5.496</td>
</tr>
<tr>
<td>6</td>
<td>10211</td>
<td>9943.5</td>
<td>7.194</td>
<td>12.690</td>
</tr>
<tr>
<td>7</td>
<td>7054</td>
<td>7145.0</td>
<td>1.160</td>
<td>13.849</td>
</tr>
<tr>
<td>8</td>
<td>5065</td>
<td>5139.1</td>
<td>1.068</td>
<td>14.917</td>
</tr>
<tr>
<td>9</td>
<td>3671</td>
<td>3699.9</td>
<td>0.225</td>
<td>15.142</td>
</tr>
<tr>
<td>10</td>
<td>2684</td>
<td>2666.3</td>
<td>0.118</td>
<td>15.260</td>
</tr>
<tr>
<td>11</td>
<td>1955</td>
<td>1923.3</td>
<td>0.522</td>
<td>15.781</td>
</tr>
<tr>
<td>12</td>
<td>1337</td>
<td>1388.7</td>
<td>1.928</td>
<td>17.709</td>
</tr>
<tr>
<td>13</td>
<td>1022</td>
<td>1003.7</td>
<td>0.333</td>
<td>18.042</td>
</tr>
<tr>
<td>14</td>
<td>770</td>
<td>726.1</td>
<td>2.649</td>
<td>20.691</td>
</tr>
<tr>
<td>15</td>
<td>535</td>
<td>525.8</td>
<td>0.160</td>
<td>20.851</td>
</tr>
<tr>
<td>16</td>
<td>352</td>
<td>381.2</td>
<td>2.229</td>
<td>23.080</td>
</tr>
<tr>
<td>17</td>
<td>292</td>
<td>276.5</td>
<td>0.864</td>
<td>23.945</td>
</tr>
<tr>
<td>18</td>
<td>200</td>
<td>200.8</td>
<td>0.003</td>
<td>23.948</td>
</tr>
<tr>
<td>19</td>
<td>127</td>
<td>146.0</td>
<td>2.469</td>
<td>26.417</td>
</tr>
<tr>
<td>20</td>
<td>137</td>
<td>106.2</td>
<td>8.922</td>
<td>35.339</td>
</tr>
<tr>
<td>21</td>
<td>305</td>
<td>287.1</td>
<td>1.114</td>
<td>36.454</td>
</tr>
</tbody>
</table>

Chi-square = 36.45 for 20 degrees of freedom, p= 0.98640

SUMMARY of Crap test:

p-value for no. of wins: 0.695152

p-value for throws/game: 0.986404

In this annex can be found different kind of programs that allows implementing the Ziggurat method. Some codes are in the original C, partially provided by the original papers where the method is explained, and there are also Matlab codifications for the same C codes.

11.1 Ziggurat algorithm in C

The code shown below corresponds to the first implementation of Ziggurat method and it corresponds to a C codification:

```c
#define KISS /*KISS generator*/
#define UNI /*KISS generator with Uniform distribution (0,1)*/
#define RNOR (hz=KISS, iz=hz&255, (abs(hz)<kn[iz])? hz*wn[iz] : nfix())

static unsigned long iz, jz, jsr = 123456789, kn[256];
static long hz; static float wn[256], fn[256];
float nfix(void) {
    const float r = 3.654153f; static float x, y;
    for(;;){
        x=hz*wn[iz];
        if(iz==0){
            do{
                x=-log(UNI)*0.2904764;
                y=-log(UNI);
            } while(y+y<x*x);
            return (hz>0)? r+x : -r-x;
        }
        if( fn[iz]+UNI*(fn[iz-1]-fn[iz]) < exp(-.5*x*x) ) return x;
        hz=SHR3; iz=hz&255;if(abs(hz)<kn[iz]) return (hz*wn[iz]);
    }
}
```

As explained in section Normalizing with Ziggurat method, in order to implement a fast code it is necessary some tables. The code below corresponds to the generation of the tables:

```c
/*-------This procedure sets the seed and creates the tables-------*/
void zigset() {
    const double m1 = 2147483648.0;
    double dn=3.6541528853610088, tn=dn, vn=0.00492867323399, q;
    int i;
    q=vn/exp(-.5*dn*dn);
    kn[0]=(dn/q)*m1; kn[1]=0;
    wn[0]=q/m1; wn[255]=dn/m1;
```
fn[0]=1.; fn[255]=exp(-.5*dn*dn);
for(i=254;i>=1;i--) {
    dn=sqrt(-2.*log(vn/dn+exp(-.5*dn*dn)));
    kn[i+1]=(dn/tn)*m1; tn=dn;
    fn[i]=exp(-.5*dn*dn); wn[i]=dn/m1;
}

11.2 Ziggurat algorithm for Matlab

The next code is the implementation in Matlab for the previous C code, but it has some differences. The Matlab code allows the user to load the whole sequence of random numbers with Uniform distribution and convert it into Normal distribution just in one call, avoiding the loops and working with vectors in place.

function [seq, m, st]=zigurat3
    tic, fin=0;
    max=10;

    %------ Loading data ------
    hz = randi(2^32,1,max); % Loading the random numbers
    iz = bitand(hz,255)+1; % Add 1 'cause the index in Matlab can’t be 0
    [kn, wn, fn] = tables; % Loading tables from the function tables

    %------ Data +/- ------
    u = (-1).^randi(2,1,max); % Making the whole Normal distribution
    hz=hz.*u;

    %---- First condition ----
    indx = abs(hz)<kn(iz); % First condition in the original algorithm
    seq = indx.*(hz.*wn(iz)); % Saving data passing the condition above

    %---- Data in the tail ----
    indxTail=not(indx).*(iz==1); % Index the data lying in the tail
    t_iz1=sum(indxTail); % How many numbers into the tail
%--- Tail dealing ---------------------------------------------
if(sum(indxTail))
    r = 3.654153; % r is the x_255 value
    q=zeros(1,t_iz1);
    x=zeros(1,t_iz1);

    while(~fin)
        x=x.*q + not(q).*(-log(rand(1,t_iz1))/r);
        y=q.*x + not(q).*(-log(rand(1,t_iz1)));
        q=q + not(q).*(y+y >= x.*x);
        if(q) fin=1; end
    end

    seq(logical(indxTail))= u(find(indxTail)).*(x+r); % Adding data from tail
end
%--- Ending of dealing ------------------------------------------

%--- Wedge dealing -----------------------------------------------
indxW = iz.*xor(indxTail,not(indx)); % Index the wedge data
i_posW = indxW(logical(indxW)); % Positions to check
t_iz2=length(i_posW);
    x=hz(find(indxW)).*wn(i_posW);

    f0=fn(i_posW-1);
    f1=fn(i_posW);

    result = (f0 + (f0-f1).*rand(1,t_iz2)) < exp(-.5*x.*x); %Condition in wedge

if(sum(result)) % Numbers on the wedge
    resOK=result.*find(indxW);
    resOK=resOK(logical(resOK));
    x=result.*x; x=x(logical(x));
    seq(resOK)=x;
end

ind=logical(indx).*not(logical(seq)); %Index containing not rejected data
if(sum(not(result))){ % Numbers rejected
    posKO=not(result).*find(indxW);
    posKO=posKO(logical(posKO));
    seq(posKO)=0;
}

%--- End of dealing --------------------------------------------
clear kn wn fn indx indxTail indxW i_posW hz u

%------ Final output ------
seq = seq(or(ind,logical(seq))); % Taking out the rejected data
figure; hist(seq,1000); % Plotting the final Histogram
title('Output Histogram: Normal random numbers distribution');
m=mean(seq);st=std(seq);
toc

The code shown below corresponds to the function tables, responsible to generate the constant arrays kn, wn and fn used in the main program for the improvement of it.

function [kn wn fn] = tables

%--------This procedure creates the tables--------

m1 = 2^32;
DN = 3.6541528853610088;
TN = DN;
VN = 0.00492867323399;
KN = zeros(1,256);
WN = zeros(1,256);
FN = zeros(1,256);
Q = VN/exp(-.5*DN*DN);
KN(1)=(DN/Q)*M1; KN(2)=0;
WN(1)=Q/M1; WN(256)=DN/M1;
FN(1)=1.; FN(256)=exp(-.5*DN*DN);

for I=255:-1:2
    DN=sqrt(-2.*log(VN/DN+exp(-.5*DN*DN)));
    KN(I+1)=(DN/TN)*M1; TN=DN;
    FN(I)=exp(-.5*DN*DN); WN(I)=DN/M1;
end
end
11.3 Ziggurat function modified

The first Ziggurat code must be adapted in order to load the sequence generated from the designed pseudo random generator previously.

The changes come from the beginning part of the code:

- It is possible to introduce the name of the file where the data will be loaded (varargin{1}).
- It is possible to indicate the number of samples for the algorithm when the file to load is given.
- Otherwise the default length (1e5) will be loaded, and the random function from Matlab will be used.

The final code modified is:

```matlab
function [seq m st hz]=zigurat4(varargin)
tic,fin=0;

%------ Loading data ------
switch nargin
    case 1
        % Load data from the output file from KISS generator
        max=1e6;
        nameK=strcat(varargin{1},'.norm');
        nameU=strcat(varargin{1},'.uni');

        tabla=textscan(fopen(nameK,'r'),'%u');
        h = tabla{1}'; h = h(1:max);
        hz = double(h);

        tabla=textscan(fopen(nameU,'r'),'%f');
        ranu = tabla{1}'; ranu = ranu(1:max);
        disp(['Data loaded from: ',nameK]);

        clear tabla nameK nameU h ran
    case 2
        % Load data from the output file from KISS generator and the size
        max=varargin{2};
        nameK=strcat(varargin{1},'.norm');
        nameU=strcat(varargin{1},'.uni');

        tabla=textscan(fopen(nameK,'r'),'%u');
        h = tabla{1}'; h = h(1:max);
        hz=double(h);
```


tabla1=textscan(fopen(nameU,'r'),'%f');
ranu = tabla1{1}'; ranu = ranu(1:max);

disp(['Data loaded from: ',nameK]);
disp(['Number of samples ',num2str(varargin{2})]);

clear tabla nameK nameU h ran

otherwise    % Default option: generator from Matlab and size 1e5
    max=1e5;
    hz = randi(2^32,1,max);
    ranu = rand(1,max);
    disp('Default');
end

figure,hist(hz(1:max),1000); title('Input Histogram: Uniform random numbers distribution');

iz = bitand(hz,255)+1;  % Add 1 'cause the index in Matlab can’t be 0
[kn wn fn] = tables;  % Loading tables from the function tables

%-------- Data +/- --------

u = 2*ranu-1;      % Making the whole Normal distribution
u = u./abs(u);
hz=hz.*u;

%---- First condition ----

indx = abs(hz)<kn(iz);  % First condition in the original algorithm
seq = indx.*(hz.*wn(iz));  % Saving data passing the condition above

%---- Data in the tail ----

indxTail=not(indx).*((iz==1));  % Index the data lying in the tail
t_iz1=sum(indxTail);  % How many numbers into the tail

%---- Tail dealing --------------------------------------

if(sum(indxTail))
i=1;
    r = 3.654153;  % r is the \text{x}_{255} value

q=zeros(1,t_iz1);
x=zeros(1,t_iz1);

while(~fin)
    x = q.*x + not(q).*(-log(ranu(i:t_iz1+(i-1)))/r);
    y = q.*x + not(q).*(-log(ranu(i:t_iz1+(i-1))));
    q=q + not(q).*(y+y >= x.*x);
    i=ranu(ceil(sum(x)))*20;
    if(q) fin=1; end
end

seq(logical(indxTail))= u(find(indxTail)).*(x+r); % Adding data from tail
end
%--- Ending of dealing -----------------------------------------------

%--- Wedge dealing -----------------------------------------------------

indxW = iz.*xor(indxTail,not(indx)); % Index the wedge data
i_posW = indxW(logical(indxW)); % Positions to check
t_iz2=length(i_posW);
x=hz(find(indxW)).*wn(i_posW);

f0=fn(i_posW-1);
f1=fn(i_posW);

if(t_iz2 ~= 0)
    result = (f0 + (f0-f1).*ranu(1:t_iz2)) < exp(-.5*x.*x); %Condition in wedge
    if(sum(result)) % Numbers on the wedge
        resOK=result.*find(indxW);
        resOK=resOK(logical(resOK));
        x=result.*x; x=x(logical(x));
        seq(resOK)=x;
    end
    if(sum(not(result))) % Numbers rejected

posKO=not(result).*find(indxW);
posKO=posKO(logical(posKO));
seq(posKO)=0;
end
end

%--- End of dealing --------------------------------------------

clear kn wn fn indx indxTail indxW i_posW u

%------ Final output ------
seq = seq(logical(seq));        % Taking out the rejected data
figure; hist(seq,100);          % Plotting the final Histogram
title('Output Histogram: Normal random numbers distribution');
m=mean(seq);st=std(seq);
disp(['Mean: ',num2str(m)]);
disp(['Std. deviation: ',num2str(st)]);
toc
12. Annex 3. Specific tests for final output

12.1 Statistics tests

It is necessary to pass some mathematical tests in order to know how statistically behaves the different sequences.

Firstly the mean and the standard deviation gives first order information, necessary condition to be sure the sequence has the proper form.

Secondly the sequences are subjected to three kind of test: Chi-square, Lillie and Kolmogorov-Smirnov (KS) tests. Both tests are focused on determine the goodness of fit into the standard normal distribution.

The tables below show how the pseudo random sequence generated is near to a perfect Gaussian behavior:

<table>
<thead>
<tr>
<th>File</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Chi-square</th>
<th>Lillie</th>
<th>KS</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2.norm</td>
<td>-0,000203</td>
<td>0,99683</td>
<td>0,47376</td>
<td>0,12419</td>
<td>0,36721</td>
</tr>
<tr>
<td>a3.norm</td>
<td>-0,0025877</td>
<td>0,99969</td>
<td>0,037265</td>
<td>0,0015069</td>
<td>0,41041</td>
</tr>
<tr>
<td>a4.norm</td>
<td>-0,0027778</td>
<td>1,00300</td>
<td>0,53528</td>
<td>0,5</td>
<td>0,54344</td>
</tr>
<tr>
<td>a5.norm</td>
<td>-0,00073</td>
<td>0,99739</td>
<td>0,51599</td>
<td>0,16863</td>
<td>0,53765</td>
</tr>
<tr>
<td>a6.norm</td>
<td>0,0012055</td>
<td>0,99853</td>
<td>0,17043</td>
<td>0,070717</td>
<td>0,38712</td>
</tr>
<tr>
<td>a7.norm</td>
<td>0,0040027</td>
<td>0,99549</td>
<td>0,94258</td>
<td>0,5</td>
<td>0,25115</td>
</tr>
</tbody>
</table>

Nº of samples: 1e5

The table above shows how the experiences with $10^5$ samples pass all tests in each situation, instead of the second sequence which doesn’t pass neither the Chi-square test nor the Lillie test. However, all the different sequences pass the KS test. It means these sequences have the expected behavior.

<table>
<thead>
<tr>
<th>File</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Chi-square</th>
<th>Lillie</th>
<th>KS</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2.norm</td>
<td>0,00052078</td>
<td>0,99707</td>
<td>2,23E-03</td>
<td>0,001</td>
<td>0,16671</td>
</tr>
<tr>
<td>a3.norm</td>
<td>0,0012384</td>
<td>0,9976</td>
<td>3,28E-02</td>
<td>0,001</td>
<td>0,047747</td>
</tr>
<tr>
<td>a4.norm</td>
<td>-7,02E-05</td>
<td>0,99799</td>
<td>1,40E-07</td>
<td>0,001</td>
<td>0,056397</td>
</tr>
<tr>
<td>a5.norm</td>
<td>-0,00076294</td>
<td>0,99848</td>
<td>3,11E-07</td>
<td>0,001</td>
<td>0,084762</td>
</tr>
<tr>
<td>a6.norm</td>
<td>-0,0020237</td>
<td>0,99855</td>
<td>7,06E-06</td>
<td>0,0017588</td>
<td>0,0015384</td>
</tr>
<tr>
<td>a7.norm</td>
<td>0,0010268</td>
<td>0,99723</td>
<td>7,47E-07</td>
<td>0,001</td>
<td>0,005664</td>
</tr>
</tbody>
</table>

Nº of samples: 1e6

The second table shows the experiences with $10^6$ samples. The two first columns have expected values for a standard Normal, but the tests Chi-square and Lillie fails in all experiences. However, the KS test is passed in more than half the cases. So it is something inconclusive, it is necessary wait for more tests.
12.2 Histogram

Experiences with $10^5$ samples

Fig 1. Histogram a2.morn file

Fig 2. Histogram a3.norm file

Fig 3. Histogram a4.morn file

Fig 4. Histogram a5.norm file

Fig 5. Histogram a6.morn file

Fig 6. Histogram a7.norm file

All the histogram has a good appearance to be fitted into the standard Gaussian density. The histograms for $10^6$ samples are not attached because are similar to the six before.
12.3 Autocorrelation

Experiences with $10^5$ samples

All the autocorrelation for the different experiences have only one important value in the origin, so that means there is no period. The same happens with the autocorrelation when $10^6$ samples are used.
12.4 Normplot

Experiences with $10^5$ samples

Fig 13. Normplot A2.morm file

Fig 14. Normplot A3.norm file

Fig 15. Normplot A4.morm file

Fig 16. Normplot A5.norm file

Fig 17. Normplot A6.morm file

Fig 18. Normplot A7.norm file
Experiences with $10^6$ samples

All the graphics fit into a linear function, that means the sequences comes from a Normal source, so the numbers have standard Normal density. In that case are included two images for the experiences with $10^6$ samples.

12.5 Comparison with Matlab function
In some experiences both densities are clearly similar, other times the mean are displaced from one another, and in other experiences the peaks and the tail are maladjusted. Anyway, the comparison shows the expected behavior between the densities.
References


