Analysis of Random Number Generator

Aadish Gupta (2009001)
Sanchit Garg (2009041)

Introduction

Random numbers are widely used for different purposes all over the world. For example in cryptography, casinos, lotteries and many more. Thus the importance of random number generator increases manifold and there is a need for highly reliable and secure Random number generators.

In this project we have tried to analyze and compare different random number generators by performing various tests on them. We have used the NIST test suite which is a world-wide accepted standard for analyzing random number generators [1]. It consists of 15 tests and is an open-source program.

We have also developed a true random number generator which is based on the combination of background noise and mouse movement.

Problem statement

To analyze and compare different random number generators.

Background study

There are two types of random number generators- Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs).
PRNGs are algorithms that use mathematical formulae or simply pre-calculated tables to produce sequences of numbers that appear random. They are efficient as they can produce large no. of random numbers in a very short time but they are predetermined, i.e., the sequence can be regenerated at a later date if the initial seed value is known and are periodic which means that the sequence will eventually repeat itself.
As such there is no true RNG but by definition true random number generator is something which measures some physical phenomena that are expected to be random and introduce it into a computer. They take longer time to produce random numbers but their advantage is that they are non-deterministic and have no periods and the same sequence cannot be reproduced again. They have wide applications in cryptography which needs a highly secure and non-deterministic RNG. True RNGs are also used to generate seeds for pseudo RNGs so as to get a large sequence of numbers in a short time.

There are many different random number generators available. The most common pseudo RNG is the linear congruential random number generator. It is quite easy to implement and is being used by various programming languages like C/C++, Java, Microsoft Visual etc. Another generator used by various languages like python, Ruby, Matlab, Maple, etc is the Mersenne Twister. It has a very long period and is a highly secure random number generator.

Many true random number generators have also been developed. Intel Corp. made a hardware based RNG which sampled thermal noise to generate random numbers[2]. Random.org is another example which generates random numbers based on atmospheric noise and is one of the best true random number generators available [3]. TRNGs based on background noise and audio input have also been developed[4][5][6]. Similarly another true RNG was developed which was based on mouse movement and chaotic hash function[7]. Webcam images were also used to develop cryptographic keys[8].

Methodology

We generated large amount binary data (from 50 MB to 500 MB) from different algorithms to perform tests on them. Most of the tests were done by using 100 MB of data. We passed 100 sequences of 10^6 bits each to the NIST test suite as 7 tests out of the 15 required at least 10^6 bits each. The conclusions from the tests were made in two different ways[9].

The first way was to see how many sequences passed a particular test which was based on the value of alpha - the significance level. We fixed the significance level to 0.01 for all our tests which was the recommended value. If the number of sequences passing the tests is below the minimum threshold then the RNG is said to be non-random. The range of acceptable proportions is determined using the confidence interval defined as,

$$p\pm3\sqrt{p(1-p)/m}$$

where $p=1-\alpha$, and m is the sample size. For 100 sequences and $\alpha=0.01$ the confidence interval is 0.99 ± 0.029849 (i.e., the proportion should lie above 0.9601503). The confidence interval was calculated using a normal distribution as an approximation to the binomial distribution, which is reasonably accurate for large sample sizes (e.g., $n \geq 1000$).

The second method is to examine the p-values corresponding to each test to ensure uniformity. The interval between 0 to 1 is divided into 10 sub intervals, and the p-values that lie in each interval are counted and displayed. Another method to determine uniformity is via an application of Chi-square test on the p-values generated for each test.

We also found out how many sequences are passing all the 15 tests. For this we wrote a program which analyzed the p-values of all the tests corresponding to each sequence.

A 512x512 scatter graph was also plotted for all the RNGs to see if they followed any distribution or had any particular pattern of generating numbers. Python was used to plot the graphs. The x_i coordinate is the first number generated and y coordinate is the immediate x_{i+1} number generated. It can be used to determine if there is any dependency in the output of the random numbers on their previous outputs.

We implemented our true random number generator in java. We captured sound using sound API and performed all the tests at different frequencies (From 8000 Hz to 2^5 Hz) [11]. For implementing this the microphone was set to its highest sensitivity and the last bit of each sound sample was stored. All windows options for microphone enhancements were also disabled while capturing the sound. The program was run from 30 minutes to 5 hours depending on the frequency of the sound until a 100 MB file was not obtained.

Similarly we tracked the movement of the mouse [10] and captured the coordinates and stored them in binary form. Since data production using this method is very slow it was not possible to perform all the tests on it. Hence we performed only those tests which required 100 or more bits by passing sequences of 1024 bits.

Results

Below are the results for the total number of sequences passing all the tests of the NIST test suite for different RNGs.

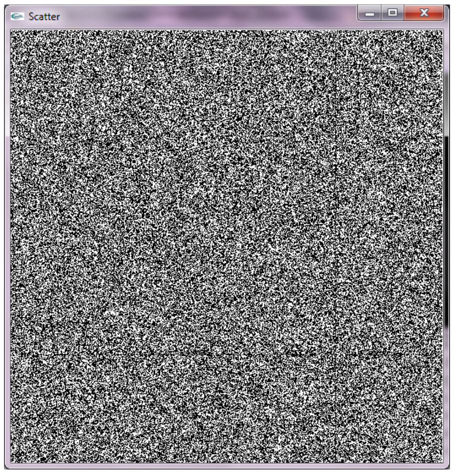
Language/Random Number generator	**%age of sequences passing all tests**
C (Linear congruential generator)	26%
Java (Linear congruential generator) Sample 1	23%
Java -Sample 2	18%
Python (Merenne twister) Sample 1	22%
Python Sample 2	23%
LCG generator combined with different constants	0

Results for true random number generator developed by us at different frequencies

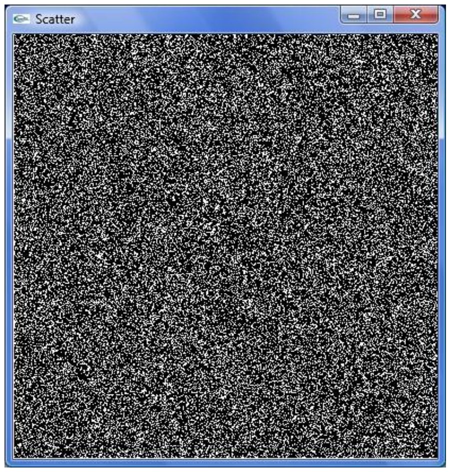
Frequency (Hertz)	**%age sequences passing all tests**
8000	5%
16000	21%
32000	5%
44000	1%
75000	2%
100000	5%
200000	0

All the results above are excluding the random excursions and random excursion variant tests because these tests are not valid for some of the sequences and hence cannot be performed.

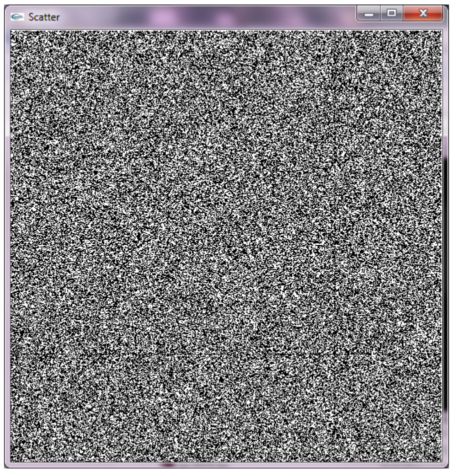
Scatter Plots for various algorithms:



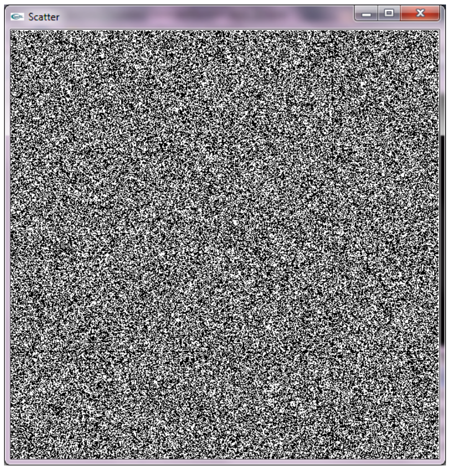
Java



Python

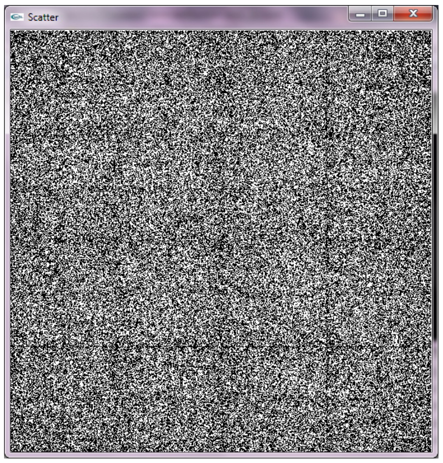


Sound (8000 Hz)

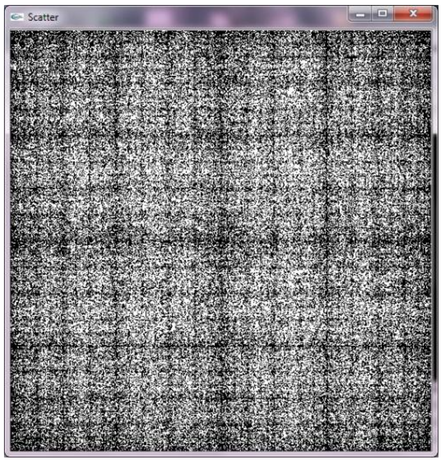


Sound (32000 Hz)

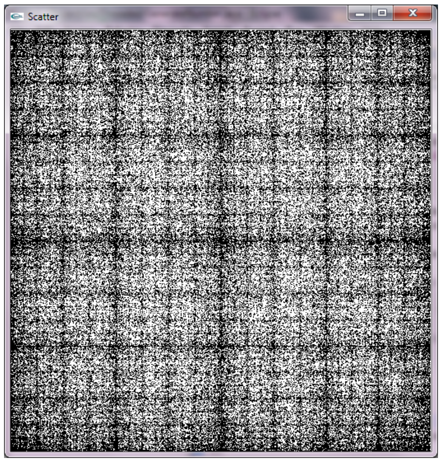
As it can be seen from the above scatter plots there are no visible patterns in any of the figures. But as the frequency was increased some patterns began emerging as shown below.



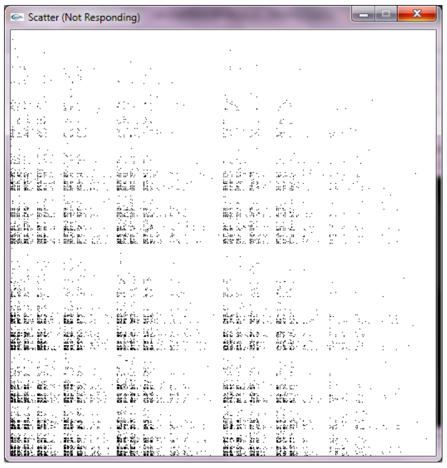
Sound (44000 Hz)



Sound (100000 Hz)



Sound (200000 Hz)

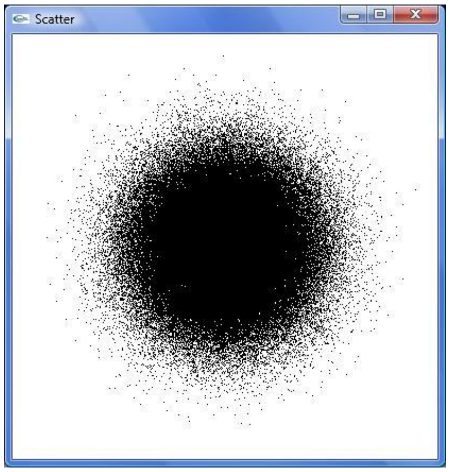


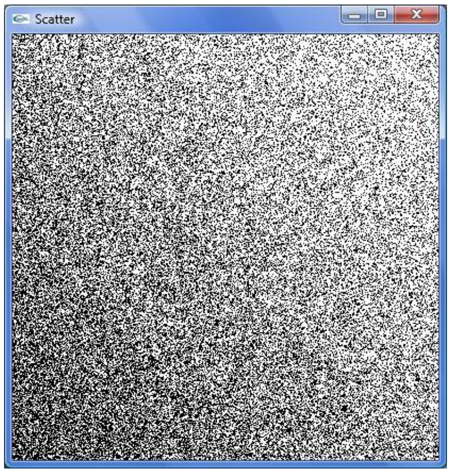
LCG Combined

As the frequency was increased above 44000 Hz the patterns started becoming more and more prominent and the generators starting failing more tests.

We also created a pseudo RNG which was based on the combination of linear congruential generators with different constants. But it failed all the tests and patterns are also visible in its scatter chart.

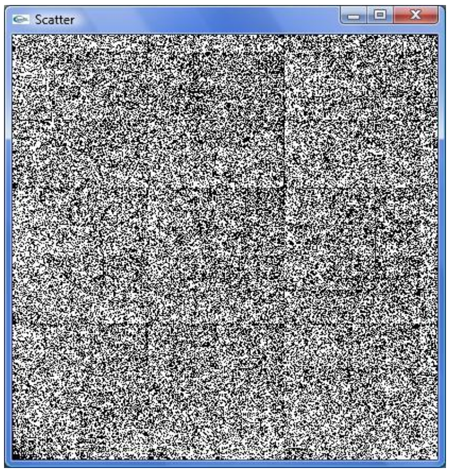
Graphs for python’s Gaussian and gamma variate generators were also plotted which showed the following graphs -





Gaussian distribution

Gamma Variate distribution



Scatter plot generated from mouse movement

Results of the NIST test suite for various RNGs.

The minimum passing threshold has been defined to be 96 sequences out of 100 except for Random excursions and Random excursions Variant test and p values ≥ 0.0001.

STATISTICAL TEST	P-VALUE	PROPORTION
Frequency	0.534146	97/100
Block Frequency	0.153763	100/100
Cumulative Sums	0.897763	97/100
Runs	0.911413	98/100
Longest Run	0.319084	99/100
Rank	0.350485	100/100
FFT	0.051942	99/100
Non Overlapping Template	0.003447	99/100
Overlapping Template	0.971699	99/100
Universal	0.574903	99/100
Approximate Entropy	0.096578	98/100
Random Excursions	0.051391	62/63
Random Excursions Variant	0.689019	62/63
Serial	0.779188	99/100
Linear Complexity	0.202268	100/100

C language (Linear congruential generator)

STATISTICAL	P-VALUE	PROPORTION
Frequency	0.115387	493/500
Block Frequency	0.715679	493/500
Cumulative Sums	0.739918	493/500
Cumulative Sums	0.22248	490/500
Runs	0.41184	494/500
Longest Run	0.357	495/500
Rank	0.809249	490/500
FFT	0.632955	494/500
Non Overlapping Template	0.674543	489/500
Overlapping Template	0.22248	493/500
Universal	0.053969	495/500
Approximate Entropy	0.924076	494/500
Random Excursions	0.295093	286/293
Random Excursions Variant	0.889825	289/293
Serial	0.931185	495/500
Linear Complexity	0.187581	498/500

Minimum passing rate – 488 sequences

Python (Merenne Twister)

STATISTICAL TEST	**P-VALUE**	**PROPORTION**
Frequency	0.289667	100/100
Block Frequency	0.289667	100/100
Cumulative Sums	0.275709	100/100
Cumulative Sums	0.319084	100/100
Runs	0.739918	99/100
Longest Run	0.816537	99/100
Rank	0.23681	99/100
FFT	0.911413	100/100
Non Overlapping Template	0.595549	96/100
Overlapping Template	0.191687	98/100
Universal	0.350485	97/100
Approximate Entropy	0.474986	97/100
Random Excursions	0.035174	56/58
Random Excursions Variant	0.085587	56/58
Linear Complexity	0.816537	100/100

Java

Test	**P-VALUE**	**Proportion**
Frequency	0.181557	99/100
Block Frequency	0.616305	98/100
Cumulative Sums	0.514124	98/100
Runs	0*	79/100*
Longest Run	0.574903	97/100
Rank	0.883171	99/100
FFT	0.779188	100/100
Non Overlapping Template	0.000274	85/100*
Overlapping Template	0.010237	90/100*
Universal	0.437274	98/100
Approximate Entropy	0*	85/100*
Random Excursions	0.551026	65/65
Random Excursions Variant	0.287306	63/65
Serial	0.289667	94/100*
Linear Complexity	0.262249	98/100

Sound (8000 Hz)

STATISTICAL TEST	**P-VALUE**	**PROPORTION**
Frequency	0.834308	98/100
Block Frequency	0.419021	98/100
Cumulative Sums	0.851383	98/100
Runs	0.191687	91/100*
Longest Run	0.739918	97/100
Rank	0.383827	100/100
FFT	0.304126	99/100
Non Overlapping	0.494392	94/100*
Overlapping Template	0.032923	96/100
Universal	0.262249	99/100
Approximate Entropy	0.202268	97/100
Random Excursions	0.723129	60/61
Random Excursions Variant	0.051391	61/61
Serial	0.595549	100/100
Linear Complexity	0.924076	99/100

Sound (16000 Hz)

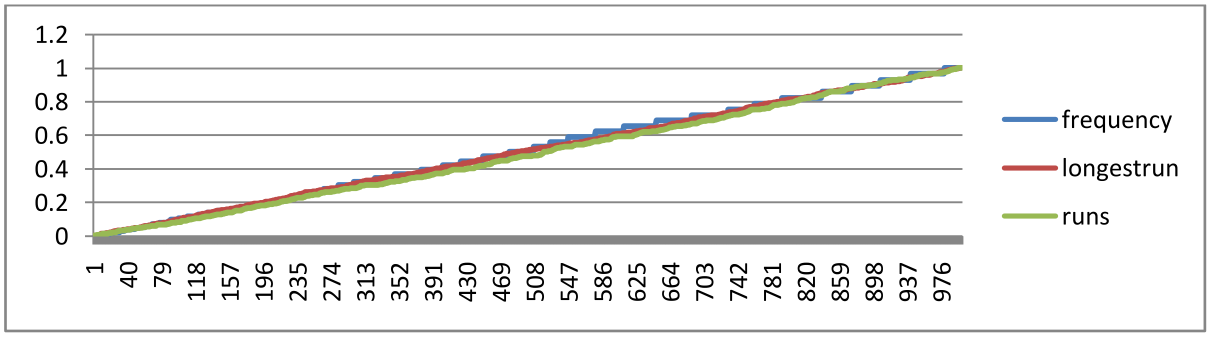
STATISTICAL TEST	**P-VALUE**	**PROPORTION**
Frequency	0.455937	97/100
Block Frequency	0*	83/100*
Cumulative Sums	0.55442	97/100
Cumulative Sums	0.383827	97/100
Runs	0*	10/100*
Longest Run	0*	27/100*
Rank	0.911413	100/100
FFT	0*	41/100*
Non Overlapping Template	0*	11/100*
Overlapping Template	0*	18/100*
Universal	0*	23/100*
Approximate Entropy	0*	16/100*
Random Excursions	0*	43/60*
Random Excursions Variant	0.025193	62/62
Serial	0*	23/100*
Linear Complexity	0.401199	98/100

Sound (100000 Hz)

STATISTICAL TEST	P-VALUE	PROPORTION
Frequency	0*	1206/1500*
Block Frequency	0*	1222/1500*
Cumulative Sums	0*	1205/1500*
Cumulative Sums	0*	1222/1500*
Runs	0*	1332/1500*
Longest Run	0*	1348/1500*

This test was performed on the sequence generated by movement of mouse. 1500 sequences of 1024 bits each were passed. The minimum passing rate needed here was 1473.

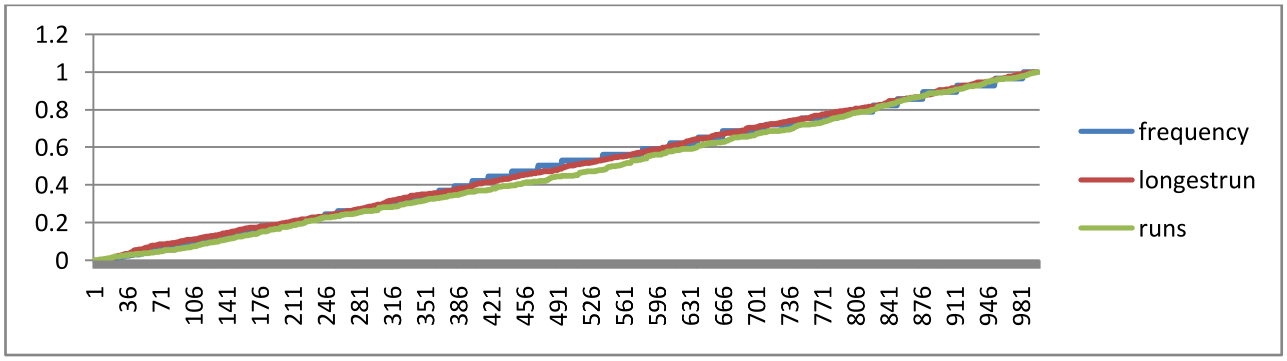
We also plotted graphs for the p-values to check the uniformity by testing 1000 sequences of 2000 bits each for tests that could be performed on 100 bits or more.



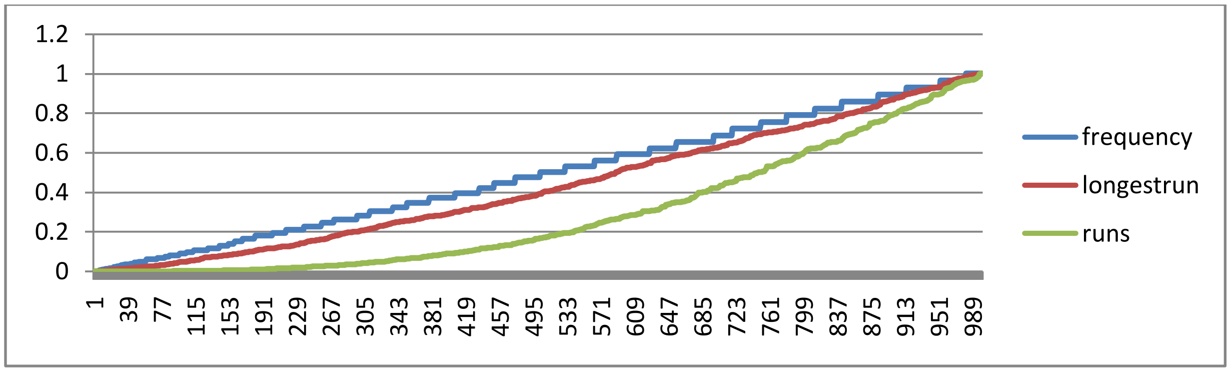
Java

The lines are almost straight depicting a uniform distribution of the p-values.

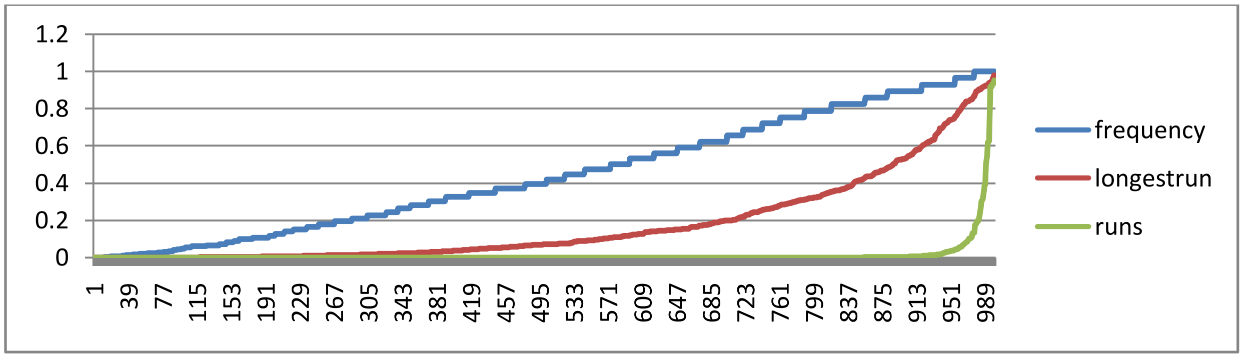
Similar graphs were plotted for out true random number generator-



Sound (16000 Hz)



Sound (44000 Hz)



Sound (100000 Hz)

As the frequency of sound was increased the graphs transformed more into curves than from straight lines and the uniformity of the p-values started decreasing.

Conclusion

According to the NIST test suite its statistically impossible to compare random number generators if they pass all the tests. Based on the tests a RNG can declared either random or non – random. Hence we tried to see if we could differentiate them by finding the minimum number of sequences passing all the tests. But since the results are almost similar nothing concrete can be said about it.

Even in the scattering graph no pattern or distribution was seen for the RNGs that passed all the tests. Hence it is not possible to decide which is a better random number generator.

The pseudo random number generator which we developed based on the combination of linear congruential generators with different constants did not work out as expected and failed all the tests and showed clear patterns in the scatter chart.

We created a random number generator based on natural phenomenon without using any mathematical formulation or applying any algorithm on the data generated. Our random number generator is not as powerful as a pseudo random number generator but still it has a good amount of randomness.

The RNG with sound frequency of 16000 Hz failed only two tests out of 15 that too with a small margin. No particular pattern or distribution was seen in its scattering graph as in the case with RNGs with higher frequencies. This algorithm generates 16000 bytes per second which is non-deterministic, irreproducible, and has no period and is almost at par with the NIST standards.

Future Work

- We will try to capture images using webcam and use those images to store the last bit of each pixel and combine it with the above program to make the data even more random.
- Since the generator is failing some of the tests we will try to improve the program so that is passes all the tests at significance level of 0.01 which is the recommended value by NIST.
- Some processing can be done on the data generated by mouse to make it at par with the NIST standards and hence can be used for generating short cryptographic keys.

Individual Contribution

Both of us have equally worked on all parts of the project and its really not possible to differentiate who has done what.

References

[1] http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[2] http://www.cryptography.com/public/pdf/IntelRNG.pdf

[3] www.random.org

[4] Nur Azman Abu and Zulkiflee Muslim, “Random Room Noise for Cryptographic Key”.

[5] Qing Zhou, Xiaofeng Liao, Kwok-wo Wong, Yue Hu, Di Xiao, True random number generator based on mouse movement and chaotic hash function.

[6] Nur Azman Abu and Shahrin Sahib, Random Ambience Key Generation Live on Demand

[7] Roger Morrison, Design of a True Random Number Generator Using Audio Input

[8] Wong Siaw Lang, Nur Azman Abu and Shahrin Sahib, “CryptographicKey from Webcam Image”

[9] Juan Soto, Statistical Testing of Random Number Generators, NIST

[10] http://download.oracle.com/javase/tutorial/uiswing/events/mousemotionlistener.html

[11]http://www.developer.com/java/other/article.php/2105421/Java-Sound-Capturing-Microphone-Data-into-an-Audio-File.htm