Quantifying the Impact of Randomness in Simulation Based Studies

By

Sehar Iqbal

2010-NUST-MS-CCS-22

Thesis Supervisor

Dr. Abdul Ghafoor

Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer and Communication Security (MS CCS)

In

Department of Computing (DoC)
School of Electrical Engineering & Computer Science (SEECS)
National University of Sciences & Technology (NUST),
Islamabad, Pakistan
(2014)

Approval

This thesis has been submitted in partial fulfillment of requirements for the Master of Science in Computer and Communication Security (MS CCS) at National University of Sciences & Technology.

It is certified that the contents and form of the thesis entitled "**Quantifying the Impact of Randomness in Simulation Based Studies**" submitted by **Sehar Iqbal** have been found satisfactory for the requirement of the degree.

Advisor:	**Dr. Abdul Ghafoor**
Signature:	________________________
Date:	________________________

Committee Member 1:	**Dr. Hassaan Khaliq Qureshi**
Signature:	________________________
Date:	________________________

Committee Member 2:	**Dr. Adnan Iqbal**
Signature:	________________________
Date:	________________________

Committee Member 3:	**Mr. Mohsan Jameel**
Signature:	________________________
Date:	________________________

Abstract

Many scientific studies in Wireless Sensor Networks rely on simulations and correctness of these simulation results is heavily dependent on random numbers. In current situation, most of researchers generally use random numbers generated through common random number generating APIs of modern programming languages. This research activity describes the comparative analysis of existing random number generators and evaluates their impact of using different types of random numbers in a simulation based study of WSN. In this study, eight different types of random numbers generation algorithms are considered. These random numbers are first evaluated using standard random number testing procedures such as Run Test, Serial Test, and Chi square Test. After that the same random numbers in a Markov chain based probabilistic study of Wireless Sensor Networks are analyzed. Our empirical analysis reveals that there is a correlation between strength of random numbers and accuracy of simulation. Simulation provides correct results if the right Random Number Generator which is strong enough respect to its random properties is chosen. It is also shown that Random Number Generators with similar properties used in the simulation modeling produce similar results.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at National University of Sciences & Technology (NUST) School of Electrical Engineering & Computer Science (SEECS) or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at SEECS NUST or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: Sehar Iqbal

Signature: _______________

Acknowledgement

First of all I thank and praise Allah for giving me persistence to complete my Master degree and then my parents and family, who have always motivated me to be diligent and to believe in my abilities.

This thesis would not have been possible without a kind support of my supervisor **Dr. Abdul Ghafoor** and **Dr. Adnan Iqbal** who has always inspired me with his dedication and enthusiasm to work.

I would like to appreciate and thank my committee members, Dr. Hassaan Khaliq Qureshi and Mr. Mohsan Jameel who have always given me their precious time and guided me through my work. Their critique and constructive comments has always helped me in improving my work.

And finally, special thanks to my friends who have always provided me all kind of encouragement throughout the span of my thesis work.

Sehar Iqbal

Dedication

Dedicated

to

my loving Husband

for supporting me all the way!

Table of Contents

List of Abbreviations

Abbreviation	**Stands for**
RNG	Random Number Generator
PRNG	Pseudo-Random Number Generator
KL Divergence	Kullback-Leibler Divergence
WSN	Wireless Sensor Network
SFMT	SIMD-oriented Fast Mersenne Twister
ISAAC Cipher	Indirection, Shift, Accumulate, Add, and Count Cipher
BSC	Binary Symmetric Channel
BER	Bit Error Rate
FER	Frame Error Rate
RSA	Rivest, Shamir, & Adleman (public key encryption technology)
R Divergence	Resistor Average Divergence

List of Figures

List of Tables

1. Introduction

Employment of Wireless Sensor Networks (WSNs) has increased in many aspects. The vast range of its applications has motivated the researchers around the world to effort into this field. Wireless Sensor Networks are supposed to be in tough environments mostly [1]. Furthermore many network details in WSNs are not yet standardized. Therefore performance evaluation in the real situation is difficult, costly and time consuming. Further repeatability is largely conceded in real environment as many factors have an impact on the experimental results at the same time. Consequently, simulation is essential to study WSNs in detail being the best way to test and analyze applications and protocols.

Risk analysis is vital part for every decision making. Except having an exceptional access to information, we can't accurately predict the future. Simulations are a well-known tool for decision making, performance evaluation and validation of WSN prior its deployment. Simulations generally make use of random numbers for the purpose of evaluation. Use of random number generator (RNG) offers the decision maker with a range of possible outcomes which occurs for every choice of action. It helps them to evaluate the impact of risk for better judgment and decisions. Due to the strong need of random numbers its application in simulation tools can be found in the field of research and development, scientific computing, finance, manufacturing, engineering, oil & gas, transportation, and the environment.

Being the fundamental building blocks for simulation, random numbers are generated in various ways. Sometimes hardware random numbers are used, generated from physical sources such as electronic noise etc. However, hardware random numbers have drawbacks of slow generation, mostly not properly implemented and not repeatable as well. As a result, the majority of random numbers used are pseudo-random generated by pseudo-random number generators. For the fast number generation in simulation work, researchers generally use common APIs and PRNGs of programming languages. Though other methods are also available to produce pseudo-random numbers like OpenSSL, SIMD-oriented Fast Mersenne Twister (SFMT) and Indirection, Shift, Accumulate, Add, and Count (ISAAC) cipher but use of PRNGs provided by programming languages is much more in simulation applications.

Simulation of a system which is dependent on some random event needs a reliable method for random sequence generation. A random sequence can be interpreted as the result of flipping a coin with sides labeled as “0” and “1. The fair coin is the perfect random bit generator, since the “0” and “1” values are randomly and uniformly distributed. Obviously the use of coins is impractical; the hypothetical output of such an idealized generator serves as the benchmark for evaluation of RNGs. The basic criteria to analyze and judge the reliability of random number generators is to test the statistical properties of generated random numbers. The statistical test application is developed so that random numbers can be submitted into it to check the degree of their randomness. It should be noted that the only conclusion we can derive from the result of test applied to a random number is whether it passes or fails this particular test. Nothing can be concluded for other numbers. Still it’s important to analyze the reliability level of random number as it directly affects the simulation results and finally decision making task. Of course if generator passes many tests, this increases our confidence level for it [2].

Large sets of random numbers are obtained from the following sources which are also variable in their reliability level are used in simulation so that the impact of varying natured RNGS are judged and correct evaluations are performed.

- Class rand in C
- Class srand in C++
- Class Rand with Current Time as Seed in Java
- Class Rand with default seed in Java
- Class Random in C#
- Class random in Python
- OpenSSL RSA Big numbers
- SFMT (SIMD-oriented Fast Mersenne Twister)

The numbers generated from above mentioned RNGs are input to the multiple statistical tests and results are recorded. A statistical test algorithm checks a specific null hypothesis (*Ho*) is that *the number being tested is random*. The alternate hypothesis (*Ha*) is that *the number is not random*. For each test a randomness statistic S is calculated and compared with the critical value t to determine the acceptance of null hypothesis. If $S < t$, then null hypothesis is rejected otherwise null hypothesis is accepted. The table given below relates the data to the conclusion using the test application.

True Situation	**Accept Ho**	**Accept Ha**
Data is random (*Ho* is true)	No Error	Type 1 Error
Data is not random (*Ha* is true)	Type 2 Error	No Error

Table 1: Hypothetical Testing of Randomness

If data is random then the decision to reject null hypothesis is called a Type 1 error. If data is non-random, then the decision to accept null hypothesis is called a type 2 error. The probability of type 1 error is called a level of significance (α). α is the probability that sequence is not random when it is really random. Common value of α is 0.01 which is also chosen as t for our test algorithms. The probability of type 2 error is β. β is the probability that sequence is random when it is not. If $P \geq \alpha$ then the null hypothesis is accepted. If $P < \alpha$ then the null hypothesis is rejected [3].

The important issue during a simulation study is that how to check whether a simulation is operating as intended or not? The most common technique to do this is by using a trace. The results of the original trace are compared with the calculations made using different parameters and synthetic traces. Use of a good RNG generates the results which are more close to real results and at the same time pass most of the statistical tests. Using thorough results evaluation proof of concept is justified that the results of test algorithms support the results of simulation. Following points are considered while evaluating the results.

- To identify the properties of random number generators.
- To provide empirical support to Random Number Generators by performing statistical tests.
- To use multiple Random Number Generators. Simulation results produced with only one (type of) generator are not enough for overall evaluation and decision.
- To compare the results of test algorithms performed for each RNG with the simulation using same RNG.

Use of quality random numbers in simulation is more important than any other application as it directly impacts the decision making of system being simulated. Random numbers generated from above mentioned RNGs are used during WSN channel modeling. Two different techniques of 2-Tier models based on events are designed: (1) Employ a Binary Symmetric Channel (BSC) model at the frame-level and stimulates another BSC model at bit-level for bit-errors when a frame is in error, (2) Determine a case that uses a Gilbert model for frame errors and a 3rd order Markov model for bit-errors [4]. BSC, Gilbert and Markov models will be discussed in detail in upcoming chapters.

From the data packets captured at destination points, we first obtain parameters which are required for initial setup of the models and then later we generated synthetic traces of frame and bit errors. Random numbers are directly involved in whole process of synthetic trace generation. As frame and bit error rates are averaged measures that do not provide details for assessment. To evaluate the accuracy of models, we compared source traces and synthetic traces with a Kullback-Leibler information divergence measure [4]. The formula for KL Divergence measure is given as follows:

$$\mathrm{KL}\ (D_1 \,||\, D_2) = \sum_x D_1(x) \log \frac{D_1(x)}{D_2(x)}$$

$$\mathrm{KL}\ (D_2 \,||\, D_1) = \sum_x D_2(x) \log \frac{D_2(x)}{D_1(x)}$$

Where D_1, D_2 represent the burst-length probability mass functions derived from source traces and synthetic traces. As KL divergence is not symmetric and requires D_1 and D_2 to be continuous with respect to each other, we used the resistor-average R divergence measure instead of KL divergence measure.

$$\mathrm{R}\ (D_1 \,||\, D_2) = \frac{\mathrm{KL}(D_1 \,||\, D_2) * \mathrm{KL}(D_2 \,||\, D_1)}{\mathrm{KL}(D_1 \,||\, D_2) + \mathrm{KL}(D_2 \,||\, D_1)}$$

Small values of R represent more similarity between two distributions D_1 and D_2.

1.1 Objectives

Research on the proposed topic is fairly a new area and not much work has been done for its improvement. The objective of research is to present a set of tests for random numbers and determine the impact of random numbers in simulation based applications. The complete set of tests comprises of a mix of both theoretical and empirical. Objective of each test is to focus on some important properties of random numbers. The study of randomness in the simulation applications and to determine the impact of its quality will prove to be a good contribution in research and will be beneficial for all those who want to work further in the same area and domain. Wireless Sensor Network simulator has been developed and impact of numbers generated from various RNGs has been analyzed to provide the performance evaluation of simulation models based on random inputs.

1.2 Motivation

Unfortunately, the random number generators (RNGs) used in practice frequently fail. Since deployed applications provide no security given bad randomness, the attacks that result from RNG failure are spectacular. RNGs have its applications in statistical sampling, computer simulation, cryptography, completely randomized design, and other domains where producing an unpredictable results are required. The need for the true Random and Pseudorandom Numbers arises majorly in cryptographic and network applications. The study of randomness being used in various simulation applications and determining its impact proves to be a good contribution once it is implemented correctly.

1.3 Problem Statement

All cryptographic applications rely on some degree of randomness, which if not fulfilled properly can lead to the breach in data security. Therefore testing and analyzing the output of random number generators is required to have a confidence in them. A comprehensive research for evaluating randomness of RNGs in most common Programming Languages following an implementation of simulation based studies with a use Random number generators to find the impact of true randomness in the simulation results.

1.4 Research Methodology

A research is a systematic study of phenomenon and sources in order to establish facts and reach at conclusions. There are two main approaches for scientific research known as deductive research and inductive research. Deductive research approach works from the more general to the more specific, also known as top-down approach. On contrary inductive research approach works from specific observation to broader generalizations and theories, also known as bottom-up approach [8].

The aim of the current research activity is to describe the problem and then to draw the conclusion by narrowing down the focus. So, I have adopted deductive approach to solve this research problem. The deductive research approach comprises of four major methodologies i.e. a) Theory b) Hypothesis c) Observation and d) Confirmation. First hypothesis is derived from extensive literature review. Then the observations are made to accept or reject the hypothesis. At the end, verification of hypothesis has been done through analysis. I have described these different phases in the chapters mentioned in following table 1. In the first phase problem is identified through the literature survey. In second phase, first I designed the architecture and then implemented it. Finally in third phase I have verified the objectives using the manual analysis methods.

Phases	**Research Methods**	**Outputs**	**Chapter(s)**
1	Literature Survey	Identification of problem	2
2	Design and Implementation	Design and implementation	3,4,5
3	Verification	Verification	6

Table 2: Phases of Deductive Approach

The step by step research methodology is given as follows:

- Obtaining a large set of random numbers through several options like C++, Java, C#, Python, OpenSSL and SFMT
- Implementing set of statistical testing algorithms of random numbers.
- Testing the random numbers using several tests and recording results.
- Analyzing these results to judge randomness.
- Implementing several simulation studies.
- For each simulation study, performing it using all sequences generated in step 1.
- Performing vertical as well as horizontal analysis.
- Analyzing impact of perceived randomness on the simulation in detail.

1.5 Thesis Organization

This thesis consists of seven chapters. Literature review is presented in Chapter 2. In Chapter 3 we focused on the generation of uniformly distributed pseudo random numbers from eight different sources and algorithms. After that Chapter 4 discussed the strategy of testing and test results interpretation. This Chapter also throws light on the implementation of setting up and running the tests. Chapter 5 presents the modeling techniques used for WSN simulation. Chapter 6 describes the results of simulation and presents comparison of these results with results of testing algorithms and finally we concluded in Chapter 7.

2. Related Work

This chapter presents the existing work done for Random Number Generators, their testing and to judge the impacts of RNGs in research domain. It elaborates the efforts done by researchers to establish the test batteries that check reliability level of RNGs. It also emphasizes on the use of Pseudo Random Number Generators with its true implementation at the right place. How the results of system simulation can influence the decision makers is also elaborated in this chapter. Moreover the impact of random attributes is shown.

2.1 Improving WSN simulation and analysis accuracy using 2-Tier Channel Models

[4] This paper proposes and evaluates two classes of Wireless Sensor Network channel models for residual MAC layer bit-errors. In the first class of channel models that is called 1-Tier models, bit-errors are modeled as a standalone error process. Two variations of 1-Tier models are explored: (1) Memoryless Binary Symmetric Channel (BSC) Model and (2) 3rd Order Markov Model. In the second class of channel models that is called 2-Tier models, a higher-level (Tier 1) frame-level model excites a lower-level (Tier 2) bit-error model.

Two variants of 2-Tier models are proposed and validated. In the first step author evaluates a 2-Tier model that employs a BSC model at the frame-level and another BSC model for bit-errors whenever a frame is in error. Later a 2-Tier model is proposed that takes memory into account and uses a Gilbert model for frame errors at Tier 1 and a 3rd order Markov model for bit-errors at Tier 2.

Results and analysis presented in paper were used to demonstrate that 2-Tier model has significantly better results and accuracy than 1-Tier model.

In last section of the paper, it is demonstrated that a bad choice of channel modeling results in an inaccurate simulation results that affects the decision making and other factors related to the network. To provide the proof the concept authors simulated following three metrics:

- Recovery ratio analysis.
- Frame goodput calculation.
- Comparison of number of retransmissions per packet.

<u>Analysis:</u>

This paper shows that WSN channels cannot be characterized using the commonly-used high-order Markov channel model. Instead a 2-Tier model in which a high-level model excites a lower-level model is required. The authors compared the accuracy of 1-Tier and 2-Tier channel models using bit error traces collected from the network. In the first step they analyzed bit error traces collected over a setup (It shall be noted that we used the same setup and source traces in our thesis for providing proof of concept) and later the accuracy of proposed models was validated. Results and analysis presented by Authors were used to demonstrate that 2-Tier model has significantly better accuracy than 1-Tier model.

The paper presents two important findings (1) The role of memory in the frame and bit error processes shall not be ignored (2) 2-Tier Channel model provides the highest accuracy. Moreover the proof of concept was provided to show that 2-Tier Markov Model performs better than other models compared regardless of the nature of application and protocols being simulated.

2.2 High Performance System for Generation and Testing of RNGs

[14] This paper provides many practical applications of random numbers and gives a detailed description regarding the role randomness plays in the cryptographic applications and protocols. Summary of roles is given as follows:

- Cryptographic keys: These keys define the transformation of plaintext into ciphertext in the symmetric and asymmetric ciphers both.
- Initialization vector: IV is used in the symmetric stream and block cipher to produce a unique output. It helps in avoiding laborious work of rekeying under the case of same encryption key even.
- Nonces: Nonce is used for mutual authentication and sharing knowledge of the secret while hiding all other information about the secret [14].
- Cryptographic salt: It is used as an input parameter for the key derivation functions.
- Padding strings: These random strings are used to fill the last block of plaintext in symmetric block cipher [14].

Paper also highlights the process of statistical testing and provides a description of few tests.

- Frequency Test
- Linear Complexity Test
- Serial Test
- Entropy Test
- Runs Test
- Non-overlapping Template Matching Test
- Overlapping Template Matching Test

It is suggested by the author that in order to fulfill the increasing demand of large volume of random numbers that contain properties of good randomness, RNGs with the high throughput should be combined with the high performance test suites. By doing so the time of random number generation task as well as applying tests will be minimized.

It is also discussed in the paper that unfortunately the most popular test suite batteries are not implemented well as these are not focused on high performance and efficiency and they don't support the processing power offered by today's multi-core high end systems. Furthermore are very slow in processing the large volume of random numbers produced by big Random Number Generators. Hence there is a severe need for the test suite that is highly efficient and is compatible with the new technology for processing large numbers.

Analysis:

This paper explains the importance of random number sequences in cryptography. It presents several approaches that are practically possible to generate good random numbers and then testing them as these are intended for cryptographic applications. Author put special emphasis on the importance and requirement of choosing the reliable and suitable random number generator as security flaws in the generator can easily compromise the security of the whole system or it can also change the simulation results effecting the decisions of decision makers.

Author suggests using Pseudo Random Number Generators rather than True Random Number Generators as it provides better affordability, availability, statistical quality and unpredictability in the generated numbers.

Author also focuses on the significance of testing the outcome of random number generators in order to assess the generators reliability and suitability. In the context where applying several batteries of statistical tests on large sequences of RNGs which is mandatory for the system in order to increase the confidence in the selected generator but at the same time this process is a very time taking task, there is a need arises for providing highly efficient statistical tests which are efficient in terms of performance. Author later presents an optimization method that shows how the need for efficiently implemented batteries of statistical tests can be fulfilled by improving the existing and well known statistical test suites to overcome their performance and speed limitations.

2.3 Testing Random Number Generators

[12] This paper discussed a summary of the research done by Daniel Biebighauser during his six weeks research at the Research Experiences for Undergraduates (REU) Summer 2000 program at the University of Minnesota. The research mentored by Prof. Paul Garrett at the University of Minnesota - Twin Cities was initially concentrating towards comparing different random number generators and presenting the results. In the course of reviewing the results Dan became more interested in the actual tests used to compare different random number generators.

This paper begins by introducing random numbers and random number generators, and then explains some empirical tests (tests that are used on a sequence produced by a RNG and don't require knowing exactly how the RNG operates) with an initial application on the chi-square and Kolmogorov-Smirnov (KS) tests and then the theoretical spectral test later. Few other tests compared and implemented are given following:

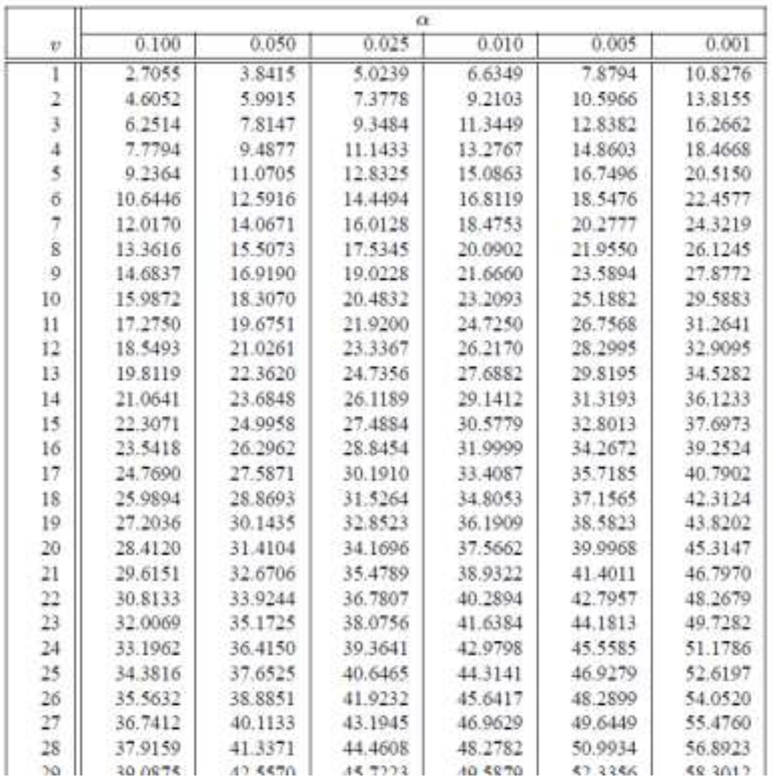
- Equidistribution Test
- Permutation Test
- Coupon Collector's Test
- Run Test
- Poker Test
- Gap Test
- Serial Test

Analysis:

The purpose of this paper is to introduce the reader to the testing of random number generators. The research was initially directed towards comparing random number generators. Later he became more interested in the actual tests used to check and validate different random number generators. He presented a summary of some of the tests used. In doing so the author also discussed some of the definitions of randomness, different ways to generate random numbers, and applications of these random numbers. He presented around a dozen tests, beginning with the foundational chi-square and Kolmogorov-Smirnov tests, then a variety of empirical tests. In a very short time span he presented a spectacular work which is a remarkable addition for the research world. His work and strategies are adapted in my thesis.

2.4 Handbook of Applied Cryptography

[9] This paper presents few techniques to generate the random and pseudorandom bits and integer or float numbers. Stream ciphers, including linear and nonlinear feedback shift registers and the output feedback mode (OFB) of block ciphers are also addressed by the author. A small section presents few tests as well which were designed to measure the quality of a generator supposed to be a random bit generator. The author also presents a method to interpret the results of Randomness Test. If X is a random variable with v degrees of freedom, then for selected α, $P(X > v)$ means the test is passed. Figure 1 presents the criteria to judge any random sequence as Pass or Fail. In better words we can say an acceptance or rejection criteria is presented.



	α					
v	0.100	0.050	0.025	0.010	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
3	6.2514	7.8147	9.3484	11.3449	12.8382	16.2662
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.5150
6	10.6446	12.5916	14.4494	16.8119	18.5476	22.4577
7	12.0170	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.9550	26.1245
9	14.6837	16.9190	19.0228	21.6660	23.5894	27.8772
10	15.9872	18.3070	20.4832	23.2093	25.1882	29.5883
11	17.2750	19.6751	21.9200	24.7250	26.7568	31.2641
12	18.5493	21.0261	23.3367	26.2170	28.2995	32.9095
13	19.8119	22.3620	24.7356	27.6882	29.8195	34.5282
14	21.0641	23.6848	26.1189	29.1412	31.3193	36.1233
15	22.3071	24.9958	27.4884	30.5779	32.8013	37.6973
16	23.5418	26.2962	28.8454	31.9999	34.2672	39.2524
17	24.7690	27.5871	30.1910	33.4087	35.7185	40.7902
18	25.9894	28.8693	31.5264	34.8053	37.1565	42.3124
19	27.2036	30.1435	32.8523	36.1909	38.5823	43.8202
20	28.4120	31.4104	34.1696	37.5662	39.9968	45.3147
21	29.6151	32.6706	35.4789	38.9322	41.4011	46.7970
22	30.8133	33.9244	36.7807	40.2894	42.7957	48.2679
23	32.0069	35.1725	38.0756	41.6384	44.1813	49.7282
24	33.1962	36.4150	39.3641	42.9798	45.5585	51.1786
25	34.3816	37.6525	40.6465	44.3141	46.9279	52.6197
26	35.5632	38.8851	41.9232	45.6417	48.2899	54.0520
27	36.7412	40.1133	43.1945	46.9629	49.6449	55.4760
28	37.9159	41.3371	44.4608	48.2782	50.9934	56.8923
29	39.0875	42.5570	45.7223	49.5879	52.3356	58.3012

Figure 1: Criteria to Accept or Reject the Random Sequence

<u>Analysis:</u>

The table given in a paper is of great importance to evaluate the results of a test. According to author the test just helps in detecting the kind of weakness or strength hold by a sequence. It is not possible to provide any kind of proof either mathematical or manual that generator is a true RNG or not. Therefore it is suggested by an author to take a sample output sequence of the generator and apply various statistical tests on that successively.

The discussions and tables provide a proof of concept that every statistical test decides whether a random number possess the certain attribute that a true random number will be likely to exhibit. Few parameters to judge the overall performance of a random number generator are provided which are of special importance for cryptographic applications especially.

2.5 C Library for Empirical Testing of Random Number Generators

[13] This paper presents a software library TestU01 implemented in C language. The software library is a collection of utilities for statistical testing of random number generators (RNGs). A C library provides general implementation of classical test algorithms for RNGs. Important thing to notice is that the test suite for random sequences works over the interval (0, 1). Moreover later the binary sequences were also included. Paper also presents the process of statistical tests given below:

- Runs Test
- Serial Test
- Subsequence Test
- Entropy Test
- Gap Test
- Poker Test

Analysis:

This paper provides us with the general implementation of statistical tests (classical) for RNGs and PRNGs, as well as other several tests which were proposed in the history. The list of tests has been provided above as well. Some of the tools are also provided for performing the thorough analysis of relation in between a specific test and a structure of the points set formulated from the family of a given RNG.

For a given statistical test and a given Random Number Generator, comparison tool determines that how large a sample size should be. It also checks the generator period length before the generator starts to fail a test thoroughly.

A library provides various types of generators implemented in generic form, as well as many specific generators proposed in the literature or found in widely used software. The best feature of library is that it can be applied to all generators which were predefined in the library. Moreover any generator that was user defined and streams of numbers manually input to the library. Library can also take random number files as an input for processing, validating and providing the results about a particular number file.

2.6 Statistical Testing of Random Number Generators

[15] This paper provides a list of statistical test suite sources and illustrates various evaluation methods. It also describes the NIST statistical test algorithms with its application details and presents the guidelines for the interpretation of test results.

- Statistical Test Suite [15]
  1. Frequency Test: Checks how many zeroes or ones in a sequence.
  2. Cumulative Sums Test: Checks how many zeroes or ones at the beginning of a sequence.
  3. Longest Runs of ones Test: Tests the deviation of a distribution of long runs of ones.
  4. Runs Large Test: Tests the number of runs indicating that either the oscillations in a bit stream is too fast or not.
  5. Rank Test: Tests the deviation of rank distribution based on periodicity.
  6. Non-overlapping Test: Tests too many occurrences of non-periodic templates.
  7. Overlapping Test: Tests too many occurrences of m-bit runs of ones.
  8. Entropy Test: Checks the non-uniform distribution of m-length substrings.
  9. Lempel-Ziv Complexity Test: Tests whether a sequence is more compressed than a truly random or not.
  10. Linear Complexity Test: Tests the deviation from a distribution of linear complexity for substrings.

- Proportion of sequences passing a test

$$p \pm 3\sqrt{p(1-p)/n}$$

<u>Analysis:</u>

This paper presents new metrics to explore the randomness of cryptographic RNGs. By doing so it gains confidence that random number generators are acceptable from a statistical point of view. Author explains many numerical experiments conducted to judge NIST statistical tests. He also discusses different evaluation techniques required to judge the randomness.

The problem of analyzing results deduced is also discussed and solutions are also provided. Author puts emphasis on continuous generation of new statistical tests to gather more evidences and confidence that RNG is of high quality in terms of randomness. The acceptable range to validate the Random Sequence Generator is presented by the author that is also used in the thesis.

3. Random Number Generation

3.1 Randomness and Unpredictability

Something that often goes ignored is the fact that all cryptographic as well as simulation applications rely on some degree of randomness, which if not fulfilled properly can lead to wrong decisions and even breach in data security. Therefore analyzing the output of random number generators is required to have a confidence in the algorithm. The algorithms which are most common in use for random number generation must have a good level of randomness and unpredictability. There are two kinds of unpredictability required *forward* as well as *backward*. If the seed of generator is unknown, next output number should be unpredictable even if the previous numbers are known. This quality is called forward unpredictability [3].

It should also not be possible to determine seed of generator from the knowledge of generated numbers in hand. This quality of random numbers is called backward compatibility. Moreover there must not be any correlation between the seed and generated values. It means that each value should appear to be the outcome of an in dependent random event with the probability ½.

To ensure quality of unpredictability, the care must be taken in choosing the seed of a generator. If seed is known the random number generator is completely predictable as the generation algorithm is most of the times available publicly. Furthermore the seed of a generator must also be random and should not be derivable from the random number sequence that it produces [3].

3.2 Random Number Generators (RNGs)

Random Number Generator uses a non-deterministic source (ie the entropy source) with some processing functions to produce randomness in a generator. The use of an entropy distillation process is required to overcome any weakness that can result in the making a number to non-random. The process might use the key strokes or mouse movements or the quantum effects in a semiconductor etc. for adding effects of non-randomness.

Production of high quality random numbers using RNGs is a time taking task that makes such productions undesirable when a large quantity of random numbers are required. Therefore to produce large quantities of random numbers Pseudo Random Number Generators are preferred over RNGs [3].

3.3 Pseudorandom Number Generators (PRNGs)

A PRNG uses one or more inputs (seeds) and generates multiple numbers. To get the unpredictable sequence of numbers the seed must be random and unpredictable. Most of the times RNGs are used to obtain the seed. The whole process to generate pseudo random numbers is deterministic. The deterministic nature of this process leads to the term “Pseudorandom”. As every number of pseudorandom number sequences is reproducible from its seed, only the seed shall be made saved.

Pseudorandom numbers often appears to be more random than random numbers obtained from physical sources if the pseudorandom sequence is properly constructed [3]. The algorithm of PRNG produces each number from the previous number using some transformation which appears to introduce additional randomness. Multiple transformations can eliminate autocorrelation between input and output numbers. Thus the outputs of PRNG may have better statistical properties and be produced faster than RNG [3].

3.4 Using Pseudorandom Number Algorithms

PRNGs are central in applications such as simulations (e.g. of physical systems like Wireless Sensor Networks). Good statistical properties are the basic requirement for the output of a PRNG. In this chapter, we will discuss eight different types of pseudorandom number generation algorithms.

3.4.1 Class rand in C

The algorithm is combination of a Fibonacci sequence (with operation "subtraction plus one, modulo one" and lags of 97 and 33) and an "arithmetic sequence" using Rand

```
//========== Code to produce Pseudorandom numbers using rand in C ========

  // random seeds for the test case
  ij=rand()%(31329);
  kl=rand()%(30082);

  // do the initialization
  rmarin(ij,kl);
  ranmar(temp, 100);
  //Print 10 million random numbers in a file
  for (j=1;j<=100;j++)
  {
        value=static_cast<int>(65536*temp[j]);
        fprintf(file, "%d ", value);
  }
```

3.4.2 Class srand in C++

The algorithm uses time stamp to initialize the seed and produce 10 million random numbers in a file using srand() seed initialization method.

//========== Code to produce Pseudorandom numbers using srand in C ========

```
  //Get value from system clock and place in seconds variable
  time(&seconds);

  //Convert seconds to a unsigned integer and provide as seed
  srand((unsigned int) seconds);

  //Print 10 million random numbers in a file
  for (i = 0; i < 10000000; i++)
  {
      j = (int) (N*rand()/(RAND_MAX+1));
      fprintf(file,"%d",j);
      fprintf(file," ");
  }
```

3.4.3 Class Rand with Current Time as Seed in Java

The algorithm uses a time stamp to initialize the seed and a processing function to produce 10 million random numbers in a file.

//== Code to produce Pseudorandom numbers using timestamp and Processing function in Java ==

```
@Override
protected int next(int nbits)
   {
      long var = seed;
      var ^= (var << 21);
      var ^= (var >>> 35);
      var ^= (var << 4);
      seed = var;
```

```
        var &= ((1L << nbits) - 1);
        return (int) var;
    }

@Override
    public int nextInt()
    {
        return next(16);
    }

Random r = new Randomizer(System.currentTimeMillis());
int size = 10000000; List<Integer> rands = new ArrayList<Integer>(size);

for (int i = 0; i < size; ++i)
       {
            rands.add(r.nextInt());
            buf.append(rands.get(i));
            buf.append(" ");
       }
            byte[] contentInBytes = buf.toString().getBytes();
            fop.write(contentInBytes);
```

3.4.4 Class Rand with default seed in Java

The algorithm uses a default class Random to initialize the seed and write 10 million random numbers in a file.

//========== Code to produce Pseudorandom numbers using Rand in Java ========

```
Random r = new Random();
for(i=0; i<10000000; i++)
     {
        x = r.nextInt(65536);
        buf.append(x);
        buf.append(" ");
     }
byte[] contentInBytes = buf.toString().getBytes();
fop.write(contentInBytes);
```

3.4.5 Class Random in C#

The algorithm uses a default class “Random” of C # to initialize the seed and write 10 million random numbers in a file.

```
//======= Code to produce Pseudorandom numbers using Random class in C # ========

StringBuilder sb = new StringBuilder(10000000);
Random rand = new Random();

for (int i = 0; i < 10000000; i++)
{
        sb.Append(rand.Next(0,65536));
        sb.Append(" ");
}
```

3.4.6 Class random in Python

The algorithm uses a class "random" of Python to initialize the seed and produce 10 million random numbers in a file.

```
//======= Code to produce Pseudorandom numbers using random class in Python ========

for i in range(10000000):
    line = str(random.randint(0, 65535))
    afile.write(line)
    if i < 9999999:
        afile.write(" ")
afile.close()
```

3.4.7 SFMT (SIMD-oriented Fast Mersenne Twister)

SFMT is a Linear Feedbacked Shift Register generator that generates a 128-bit pseudorandom integer at one step. Algorithm to produce pseudorandom numbers is given below:

```
//======= Code to produce 10 million Pseudorandom numbers ========

    final static int BLOCK_SIZE = 100000;
    final static int COUNT = 1000;
    //The SFMT generator has an internal state array of 128-bit integers
    final static int N = MEXP / 128 + 1;
        long hi = ((long) a3 << 32) | (a2 & (-1L >>> 32)),
        lo = ((long) a1 << 32) | (a0 & (-1L >>> 32)),
        outLo = lo << lShift,
        outHi = (hi << lShift) | (lo >>> (64 - lShift));
        int x0 = (int) outLo,
        x1 = (int) (outLo >>> 32),
        x2 = (int) outHi,
        x3 = (int) (outHi >>> 32);
        // 128-bit shift: y = c >>> SR2 * 8:
        final int rShift = SR2 * 8;
        hi = ((long) c[cI + 3] << 32) | (c[cI + 2] & (-1L >>> 32));
        lo = ((long) c[cI + 1] << 32) | (c[cI] & (-1L >>> 32));
        outHi = hi >>> rShift;
        outLo = (lo >>> rShift) | (hi << (64 - rShift));
        int y0 = (int) outLo,
        y1 = (int) (outLo >>> 32),
        y2 = (int) outHi,
        y3 = (int) (outHi >>> 32);
        // rest of forumula:
        r[rI] = a0 ^ x0 ^ ((b[bI] >>> SR1) & MSK1) ^ y0 ^ (d[dI] << SL1);
        r[rI + 1] =
            a1 ^ x1 ^ ((b[bI + 1] >>> SR1) & MSK2) ^ y1 ^ (d[dI + 1] << SL1);
        r[rI + 2] =
            a2 ^ x2 ^ ((b[bI + 2] >>> SR1) & MSK3) ^ y2 ^ (d[dI + 2] << SL1);
        r[rI + 3] =
            a3 ^ x3 ^ ((b[bI + 3] >>> SR1) & MSK4) ^ y3 ^ (d[dI + 3] << SL1);
    // Initializes the internal state array with a 32-bit seed.
    public void setIntSeed (int seed) {
        sfmt[0] = seed;
        for (int i = 1; i < N32; i++) {
            int prev = sfmt[i - 1];
            sfmt[i] = 1812433253 * (prev ^ (prev >>> 30)) + i;
        }
        periodCertification();
        idx = N32;
    }
    // main function
    public static void main(String[] args)
    {
        int[] array32 = new int[BLOCK_SIZE / 4],
        array32_2 = new int[10000],
        ini = {0x1234, 0x5678, 0x9abc, 0xdef0};
        // Create file path to save Random Numbers
        StringBuffer buf = new StringBuffer();
        GregorianCalendar gcalendar=new GregorianCalendar();
        String h=String.valueOf(gcalendar.get(Calendar.HOUR_OF_DAY));
        String m=String.valueOf(gcalendar.get(Calendar.MINUTE));
        String date=String.valueOf(gcalendar.get(Calendar.DATE));
        String mon = String.valueOf(gcalendar.get(Calendar.MONTH)+1);
        String year=String.valueOf(gcalendar.get(Calendar.YEAR));
        String hms="C:\\SFMT-Output\\SFMT_";
        hms+=year+mon+date+"_"+h+m;
        hms+=".txt";
        File file = new File(hms);
        FileOutputStream fop = new FileOutputStream(file);
        if (!file.exists())
        {
                    file.createNewFile();
            }
        if (sfmt.minArraySize() > 10000)
            throw new IllegalStateException("Array size too small!");
        for(int j=0;j<500;j++)
        {
        long l=System.currentTimeMillis();
        int seed=(int)(long)l;
        if(seed<0)
            seed=seed*(-1);
        sfmt.setIntSeed(seed);
        sfmt.fillArray(array32,10000);
        sfmt.fillArray(array32_2,10000);
        sfmt.setIntSeed(seed);
        for (int i = 0; i < 10000; i++)
        {
                short value16bits1 =(short)(array32[i] & 0xFFFF);
                short value16bits2 =(short)(array32[i] >> 16);
                if(value16bits1<0)
                    value16bits1=(short)(value16bits1*(-1));
                if(value16bits2<0)
                    value16bits2=(short)(value16bits2*(-1));
                buf.append(value16bits1);
                buf.append(" ");
                buf.append(value16bits2);
                buf.append(" ");
                if (i % 5000 == 0)
                {
                    fop.flush();
                }
                byte[] contentInBytes = buf.toString().getBytes();
                fop.write(contentInBytes);
                buf.setLength(0);
                int r32 = sfmt.next();
                if (r32 != array32[i])
                throw new RuntimeException(
                    String.format(
                        "mismatch at %d array32:%x next:%x",i,array32[i],r32));
        }
        for (int i = 0; i < 700; i++)
        {
            int r32 = sfmt.next();
            if (r32 != array32_2[i])
                throw new RuntimeException(
                    String.format(
                        "mismatch at %d array32_2:%x next:%x",i,array32_2[i],
                        r32));
        }
      }
      fop.close();
    }
```

3.4.8 OpenSSL RSA Big numbers

The algorithm to generate pseudorandom numbers using OpenSSL big numbers and rand function is given as follows:

//======= Code to produce 10 million Pseudorandom numbers ========

```
#include <openssl/bn.h>
#include <openssl/rand.h>
# include <time.h>
#pragma comment( lib,"ssleay32.lib")
#pragma comment( lib,"libeay32.lib")

int _tmain()
{
          buf=spc_rand(buf,1024);*/
          int i=0,j=0,k=0;
          FILE *file;
        char *rc,*tc;
        static char name[40];
        time_t now = time(0);
        int num[10000];
        strftime(name, sizeof(name), "C:\\OpenSSL-Output\\PRNG_%Y%m%d_%H%M",
        localtime(&now));
        tc=".txt";    rc= strcat ( name,tc );
        file = fopen(rc, "a+");
        unsigned char rnd1[20000];
        unsigned char rnd2[20000];
        unsigned char rnd3[20000];
        unsigned char rnd4[20000];
        RAND_pseudo_bytes(rnd1,sizeof(rnd1));
        RAND_pseudo_bytes(rnd2,sizeof(rnd2));
        RAND_pseudo_bytes(rnd3,sizeof(rnd3));
        RAND_pseudo_bytes(rnd4,sizeof(rnd4));
          for(int count=0;count<50;count++)
          {
          j=0;
          for(i=0;i<sizeof(rnd1)-1;i=i+2)
          {
                    num[j]=rnd1[i]*rnd1[i+1];
                    fprintf(file, "%d", num[j]);
                    fprintf(file, " ");
                    j++;
          }
          j=0;
          for(k=0;k<sizeof(rnd2)-1;k=k+2)
          {
                    num[j]=rnd2[k]*rnd2[k+1];
                    fprintf(file, "%d", num[j]);
                    fprintf(file, " ");
```

```
                j++;
        }
        j=0;
        for(k=0;k<sizeof(rnd3)-1;k=k+2)
        {
                num[j]=rnd3[k]*rnd3[k+1];
                fprintf(file, "%d", num[j]);
                fprintf(file, " ");
                j++;
        }
        j=0;
        for(k=0;k<sizeof(rnd4)-1;k=k+2)
        {
                num[j]=rnd4[k]*rnd4[k+1];
                fprintf(file, "%d", num[j]);
                fprintf(file, " ");
                j++;
        }
        }
        fclose(file);
        printf("Done\n");
        return 0;
}
```

4. Randomness Testing

4.1 Testing Algorithms

The testing algorithms are implemented to test the randomness of sequences being produced by random or pseudorandom number generators. These tests are able to find out different types of non-randomness properties that could exist in a sequence. The names of tests are given below:

- Chi Square Test
- Equidistribution Test
- Entropy (N Gram) Test
- Entropy (Bi Gram) Test
- Entropy (Tri Gram) Test
- Frequency (N Gram) Test
- Frequency (Bi Gram) Test
- Frequency (Tri Gram) Test
- Runs Test
- Subsequence Test
- Permutation Test
- Poker Test
- Serial Test
- Auto Correlation Test
- Pattern Test

For each test, subsections provide a high level description and algorithm of the particular test. The order of the application of a test to random sequence is arbitrary. Testing strategy for better interpretation of results and user's guide for setting up and running the tests is also elaborated in this chapter.

4.1.1 Chi Square Test

The chi-square test determines whether there exist a significant difference between the expected frequencies and the observed frequencies or not. The following algorithm can be applied to the sequence $<U_n>$ having characters as well as digits. It calculates the difference (V) between observed frequencies (Y_s) and expected frequencies (nP_s) of characters or digits in the sequence. So the formula is given as

$$V = \sum_{s=1}^{n} (Y_s - nP_s)^2 / nP_s$$

By expanding $(Y_s-nP_s)^2 = Y_S{}^2 - 2nP_S Y_S + n^2 P_S{}^2$. We arrived at the formula

$$V = \sum_{s=1}^{n} (Y_s^2 / P_s) / n - n$$

Algo:C

```
C1)  [initialize] set sum <- 0, set Y[s]<- 0  for 0≤ s<k , set P[s]<- n/k for 0≤ s<k
C2) [set s zero] set<- 0
C3) [record Ps ,Ys] if s≥k, go to step C6. Otherwise calculate Y[s]
C4) [record sum] set sum <-  sum+ y[s] $^2$/p[s]
C5) [increase s] increase s by 1 and return to step c3
C6) [calculate V] set  V <- sum/n-n
```

4.1.2 Equidistribution Test

When the random integers are generated in [0,d-1] , we need to check whether they come from a uniform distribution, that is, each of the d numbers have equal probability.

Equidistribution test is applied to check whether numbers are uniformly distributed or not. This test is applied on the sequence $<U_n>$ given as

$$<U_n> = U_0 , U_1 , U_2 \ldots\ldots\ldots., U_{n-1}$$

If frequency of each digit is same, it implies that numbers are uniformly distributed. For each integer r, $0 \leq r < d$, count the numbers of times that $r=U_j$ for $0 \leq j < n$.

Algo: E

```
E1) [Initialize] set  count [r]<- 0, for 0≤ r<d , set r<- -1
E2) [set s,sum zero] set s<- 0 , set sum <-0, increase r by 1. if r≥d, go to step E6
E3) [compare values] if r=U_s , increase sum by 1
E4) [increase s] increase s by 1.if s<n, go to step E3
E5) [record count] set count[r] <- sum. Go to step E2
E6) [calculate probability] set p <-n/d
E7) [compare frequencies] if count[r]=p , for 0≤ r<d. Display msg " uniformity is achieved".
```

4.1.3 Serial Test

If the random integers are generated in [0,d-1] range. In the serial test, we generate n pairs of integers in [0,d-1]. Each of the d^2 pairs must be equally likely to occur. Serial test checks the uniformity of pair of successive numbers in an independent manner. To carry out the serial test we count the number of times that the pair $(U_{2j},U_{2j+1}) = (q,r)$ occurs for $0 \leq j < n$.

These counts are made for each pair of integers $0 \leq q,r < d$. chi square test is applied to $k= d^2$ categories with probability $1/d^2$ in each category.

Algo: S

```
S1) [initialize] set q<- -1 , set p <- n/2d$^2$ , set k <- d$^2$ , set y [i] <- 0 for 0≤ i<k , set i<-0
S2) [set r,sum zero] set r<-0,set count<- 0,set j<- 0,increase q by 1. if q ≥ d , go to step S7
S3) [compare pairs] if ( $U_j$ ,$U_{j+1}$ ) =(q,r) , increase count by 1.
S4) [increase j] increase j by 2. if j< n-1 , go to step S3
S5) [record count] set y[i] <- count. Increase i by 1.
S6) [increase r] increase r by 1. if r<d , go to step S3. Otherwise go to step S2.
S7) [apply chi square] calculate chi square coefficient.
```

4.1.4 Permutation Test

If total m integers are generated by RNG, these can be ranked according to their scale; the smallest number is ranked 0, the largest is ranked m-1. So there are m! possible ways in which the ranks can be ordered. For example, if m = 3, then the following orders are possible: (0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,1,0), (2,0,1). This test is repeated n times to check if each possible permutation was equally probable.

Permutation test counts the number of times each ordering of elements appears. To achieve permutation test, input sequence <U> is divided into n groups of t elements, that is

$$(U_{jt} , U_{jt+1} , U_{jt+2} \text{ ....}, U_{jt+t-1}) , \text{ for } 0 \le j < n$$

The elements in each group may have t! possible ordering. The number of times each ordering appears is counted and chi square test is applied with k=t! and with the probability 1/t! for each ordering.

Algo: P

Given a sequence of elements given

$$<U_j> = (U_1 , U_2 , U_3 \text{ ....}, U_t) , \text{ for } 0 \le j < n$$

We analyze the permutation by computing an integer f such that $0 \leq f < t!$

```
P1) [initialize] set k<-t! , set p<- n/t! , set number[i]<-0 , for 0≤ r<t!
                 set y[i]<- 0 for 0≤ i<t!
P2) [set i zero] set i<- 0
P3) [get element ordering] get number[i]
P4) [set j zero] set j≤ 0
P5) [analyze number] if number[i]= U[j] , increase f by 1.
P6) [increase j] increase j by 1. if j<n , go to step P5.
P7) [record count] set y[i]<- f . increase i by 1. set f<-0 . If i<t! , go to step P3
P8) [apply chi square] calculate chi square coefficient.
```

4.1.5 Frequency Test (N Gram)

Generated random integers are in range [0,d-1] and a frequency test checks whether they come from a uniform distribution or not, that is, each of the d numbers have equal probability.

This test is applied on the sequence $<U_n>$ given as below

$$<U_n> = U_0 , U_1 , U_2 \ldots\ldots\ldots\ldots, U_{n-1}$$

If frequency of each digit is same, it implies that the numbers are uniformly distributed. For each integer r, $0 \leq r < d$, count the numbers of times that $r=U_j$ for $0 \leq j < n$ occurs.

<u>Algo: E</u>

```
E1) [Initialize] set count [r]<- 0, for 0≤ r<d , set r<- -1
E2) [set s,sum zero] set s<- 0 , set sum <-0, increase r by 1. if r≥d, go to step E6
E3) [compare values] if r=U_s , increase sum by 1
```

E4) [increase s] increase s by 1.if $s<n$, go to step E3

E5) [record count] set count[r] <- sum. Go to step E2

E6) [calculate probability] set p <-n/d

E7) [compare frequencies] if count[r]=p , for $0\leq r<d$. Display msg " Frequency test is passed".

4.1.6 Frequency Test (Bi Gram)

The focus of this test is to check uniform occurrence of 2 bit possible combinations in the sequence of U. frequency test measures how much the observed proportion of combinations within 2 bit block match the expected proportion (¼) for following sequence.

$$<U_n> = U_0\,U_1\ ,\ U_2\,U_3\ ,\ U_4U_5\ \ldots\ldots\ldots\ldots,\ U_{n-2}\,U_{n-1}$$

Algo: B

Input sequence is partitioned into N= [n/2] non-overlapping 2 bit blocks $< U_j >$, for $0\leq j<N$.

B1) [initialize] set count [r] <-0 , for $0\leq r<d^2$. Set r <- -1

B2) [increase r] increase r by 1. if $r\geq d^2$, go to step B8.

B3) [get number] get number[r]

B4) [set j, sum zero] set sum<- 0, set j <- 0

B5) [compare values] if number[r] = U[j] , increase sum by 1.

B6) [increase j] increase j by 1. if $j<N$, go to step B5.

B7) [record count] set count [r] <- sum , go to step B2.

B8) [calculate probability] set p<- N/d^2

B9) [compare proportion] if count[r]=p , for $0\leq r<d^2$. Display message "uniformity is achieved in bi gram".

4.1.7 Frequency Test (Tri Gram)

The focus of this test is to check whether 3 bit possible combinations within 3 bit sequence are uniform or not. Frequency test (tri gram) measures how much the observed proportion of 3 bit block match the expected proportion (1/8) .

Sequence for frequency test (tri gram) is given below as

$$\boldsymbol{<U_n> = U_0\,U_1\,U_2\ ,\ U_3\,U_4\,U_5\ ,\ U_6\,U_7\,U_8 \ldots\ldots\ldots\ldots,\ U_{n-3}\ U_{n-2}\,U_{n-1}}$$

<u>Algo: D</u>

Input sequence is partitioned into N= [n/3] non overlapping 3 bit blocks <Uj> for $0 \le j < N$

```
D1) [initialize] set count [r] <- 0 , for $0 \le r < d^3$ . Set  r <-  -1
D2) [increase r] increase r by 1. if $r \ge d^3$, go to step D8.
D3) [get number] get number[r]
D4) [set j, sum zero] set sum<- 0, set j <- 0
D5) [compare values] if number[r] = U[j] , increase sum by 1.
D6) [increase j] increase j by 1. if j<N , go to step D5.
D7) [record count] set count [r] <- sum , go to step D2.
D8) [calculate probability] set p<- N/ $d^3$
D9) [compare proportion] if count[r]=p , for $0 \le r < d^3$. Display message "uniformity achieved in tri gram".
```

4.1.8 Auto correlation Test

A sequence having statistically independent digits is said to be a true random sequence. The purpose of auto correlation test is to check correlation between adjacent digits. Passing auto correlation test implies that an incoming digit cannot be predicted by analyzing prior digit of a sequence.

Auto correlation test is applied on the sequence given below

$$< U_n > = U_0 , U_1 , U_2 \ldots\ldots\ldots\ldots, U_{n-1}$$

The statistical formula used for auto correlation test is given below

$$C = (n \sum U_i U_{i+1} - (\sum U_i)^2) / (n \sum U_i^2 - (\sum U_i)^2)$$

C is the measure of extent to which U_{i+1} depends on U_i

Algo: A

```
A1) [initialize] set U [i] <-0 , for $0\leq i<n-1$ . Set i <- 0 , set sum1<- 0, set sum2 <- 0, set sum3<- 0
A2) [find sum1] set sum1 <- U[i]*U[i+1]+sum1
A3) [find sum2] set sum2 <- U[i]+sum2
A4) [find sum3] set sum3 <- (U[i])2+sum3
A5) [increase i] increase i by 1. if i<n, go to step A2.
A6) [calculate c] set C <-  ( n*sum1 - $sum2^2$ ) / ( n*sum3 - $sum2^2$ )
```

4.1.9 Poker Test

Poker test considers n groups of five successive integers $(U_{5j} , U_{5j+1} , \ldots\ldots\ldots\ldots U_{5j+4})$ for $0\leq j <n$ and observe the number of distinct values in each group so we have five categories

5 = All different

4 = One pair

3 = Two pairs, or three of a kind

2 = Full house, or four of a kind

1 = Five of a kind

In poker test we consider n groups of k successive numbers and count the number of k tuples with r different values. A chi square test is then initiated using the probability

$$p_r = d(d-1)................(d-r+1) \; / \; d^k$$

<u>Algo : K</u>

```
K1) [initialize] set k<- 5 . set M[r]<-0 , for 1≤ r<5 . set P[r]<- 0 for 1≤ i<5 . Set j<- -1 .set d<- k
K2) [find groups] generate n groups of k values
K3) [measure r] observe r from n groups and store in M[r]
K4) [increase j] increase j by 1. if j>k , go to step K9.
K5) [set i] set i<- 0
K6) [find product] set mul <- mul* (d-i)
K7) [increase i] increase i by 1. if i≤ j , go to step K6.
K8) [find expected values] set P[r] <- mul / $d^k$ . go to step K4
K9) [apply chi square] calculate chi square coefficient using P[r] and M[r] for 1≤ r≤ 5.
```

4.1.10 Entropy Test (N Gram)

The entropy and uncertainty are used interchangeably. Entropy is a simple quantitative measure of uncertainty in the data. Entropy is basically a measure of randomness. Suppose that we have alphabets of 28 characters: letters a to z, space and full stop. Using these alphabets, we can write a string of length n. The possible strings are 28^n. We then think about how many strings are legal. The higher proportions of legal strings mean the higher entropy. A language with just letters is much easier to predict so its entropy is low. So a language where all strings are allowed has maximum entropy. It takes $\log_2 n$ bits to code n character alphabets. It is necessary to measure the entropy of a random data. It is the best way to check randomness. The random language has 4.8 value of entropy per character.

Let X be the sequence having values $(X_1, X_2, X_3, \ldots\ldots\ldots, X_n)$ with probabilities $(P_1, P_2, P_3, \ldots\ldots\ldots, P_n)$ such that $P_i \geq 0$, for $1 \leq i \leq n$. The formula for entropy is

$$H(x) = -\sum_{i=1}^{n} p_i \ln (p_i)$$

where P_i is the probability of symbol X_i occurring and log is of base 2.

<u>Algo : Y</u>

```
Y1) [initialize] Set M[i]<-0 and P[i]<- 0, for $1 \leq i \leq n$
Y2) [find occurrence] Calculate M[i], for $1 \leq i \leq n$
Y3) [set i] Set i <- 1
Y4) [calculate $P_i$] Set P[i]<- M[i] / n
Y5) [find sum] Set sum<-  sum + ( P[i]* $\log_2$ P[i] )
Y6) [increase i] Increase i by 1. if $i \leq n$ , go to step Y4.
Y7) [calculate entropy] Set entropy <- sum*(-1)
```

4.1.11 Entropy Test (Bi Gram)

The entropy is a simple quantitative measure of uncertainty in data. Entropy Test (Bi gram) is used to calculate entropy of possible combinations of 2 characters at one time. Everything is same as for Entropy test (n gram). Algorithm to calculate entropy coefficient is also same as for entropy test (n gram). There is just change in input. We input possible combinations of 2 characters and find their entropy. Entropy test is helpful in finding the entropy of 2 characters at a time.

4.1.12 Entropy Test (Tri Gram)

The entropy is a simple quantitative measure of uncertainty in data. Entropy Test (Tri gram) is used to calculate entropy of possible combinations of 3 characters at one time. Everything is same as for Entropy test (n gram). Algorithm to calculate entropy coefficient is also same as for entropy test (n gram). There is just change in input. We input possible combinations of 3 characters and find their entropy. Entropy test is helpful in finding the entropy of 3 characters at a time.

4.1.13 Sub sequence Test

If a random number generator works with three random number variables X,Y,Z , it may consistently invoke the generation of three random numbers at a time. In such applications, it is important that subsequence consisting of every three terms must be random.

The original sequence is given below

$$<U_n> = U_0 \, U_1 \, U_2 \, ... \, U_{n-1}$$

The sequences put for sub sequence test are

$$U_0U_qU_{2q} \, ... , \, U_1U_{q+1}U_{2q+1} \, ... , \, ..., \, U_{q-1}U_{2q-1}U_{3q-1} \, ...$$

In our project, we have taken q=3

Algo : Q

First of all, the limit of integers is find out and set as k. n sequences of three elements are generated to input for subsequence test.

Q1) [initialize] Set i <- 0, Set sub[s]<-0, for $0 \leq s \leq n$, Set s <- 0

Q2) [set j] Set j <- i+1

Q3) [set l] Set l <- j+1

Q4) [find sub sequence] Set sub[s++] = i,j,l

Q5) [increase l] Increase l by 1. if $l \leq max-1$, go to step Q4.

Q6) [increase j] Increase j by 1. if $j \leq max-2$, go to step Q3.

Q7) [increase i] Increase i by 1. if $i \leq max-3$, go to step Q2.

Q8) [count sub sequence] count if sub[s] = U_q

Q9) [apply chi square] calculate chi square coefficient.

4.1.14 Pattern Test

It is the basic property of true random number generators that they produce pattern free random numbers. Pattern helps in predicting the number. It is basic requirement that random number should be unpredictable. If there is found any pattern in random number, it reduces the possibility of number to be random. So Pattern test is one of the major tests to check randomness. Pattern test is very applicable test for testing randomness.

Algo : O

O1) [find patterns] Get all possible patterns from a given number.

O2) [save file] Save all patterns in a file for help in future.

4.1.15 Run Test

In run test, a sequence is tested for “runs up” and “runs down”. It means that sub sequences (segments that are increasing or decreasing) are determined from original sequence. As an example consider the following sequence of 10 digits.

25798435714

By putting a line in between U_j and U_{j+1} whenever $U_j > U_{j+1}$

| 2 5 7 9 | 8 | 4 | 3 5 7 | 1 4 |

which displays "runs up". There is a run of length 4, followed by two runs of length 1, followed by another run of length 3, followed by run of length 2. The purpose of Run test is to check whether the number of Runs of various lengths are as per expected from the random sequence.

<u>Steps :</u>

- Split sequence U into sub sequences observing runs up.
- Calculate number of runs of length i.
- Repeat it n times.
- Compare these with expected distribution of run lengths.
- Apply chi square test.

4.2 The Interpretation of Results

The interpretation of results is performed in two ways.

- The analysis of proportion of sequences that passes a statistical test
- The analysis of distribution of P-values to check uniformity

4.2.1 Proportion of sequences passing a test

Given the results for a particular statistical test, the proportion of sequences that pass is computed. For example, if 1000 sequences were tested (i-e m=1000) and $\alpha=0.01$ (the significance level) and 992 sequences had P-values<0.01 then the proportion is 992/1000=0.992

The range of acceptable proportion is resolute using interval defined as $p \pm 3\sqrt{p(1-p)/m}$, where $p = 1- \alpha$ and m is the sample size. If the proportion lies outside this interval then it is evidence that PRNG is non-random.

4.2.2 Distribution of P-Values

The distribution of P-values is observed to ensure the uniformity of sequence. It is also illustrated using a histogram given in Figure 2 where the interval between 0 and 1 is divided into 10 sub-intervals and the P-value that falls within each sub-interval is counted.

Distribution test on the P-values is found from the statistical tests (i-e a P-value of the P-values) is accomplished by computing following:

$$X^2 = \sum_{i=1}^{n} (F_i - s/10)^2 / s/10$$

where $\mathbf{F_i}$ is the number of P-values in sub interval i and s is the sample size. If P-value ≥ 0.01 then the PRNG is also considered to be uniformly distributed.

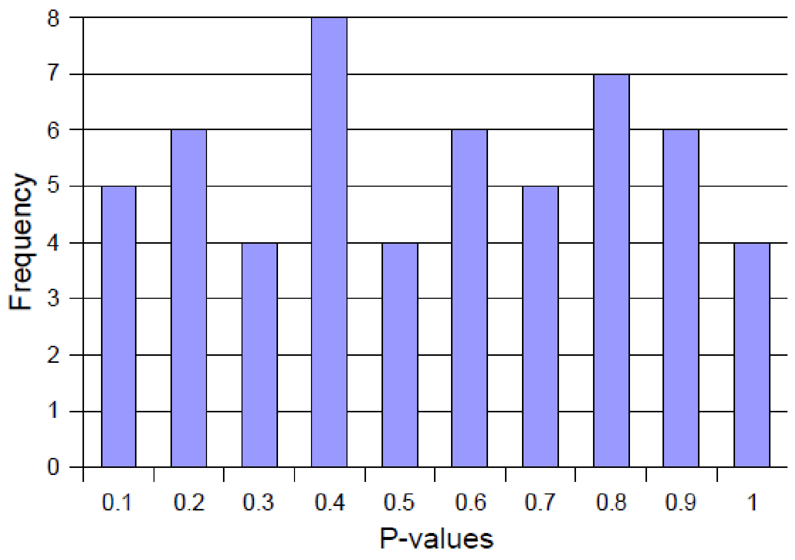


Figure 2: Histogram of P-Values

4.3 Implementation of Tests

Multiple tests are applied to random numbers in order to test the randomness of a generator in the best manner possible. The performance of each test is evaluated by applying a Chi-Square test. When all tests applied to a random sequence, we considered uniformity, correlation and independence properties.

The statistical tests solve the problem of evaluating PRNGs for randomness by identifying RPNG that produce weak or strong random numbers. They tests the level of randomness provided as well as verifies whether the implementation of PRNG is correct or not.

4.3.1 About the application

The code has the following features that make it easy to use in day to day applications.

- **Platform Independency**

  The source code is written in java so the test code is platform independent.

- **Extensibility**

  New statistical tests can easily be added in the existing list.

- **Portability**

  The source code is written in java so the code is portable.

- **Versatility**

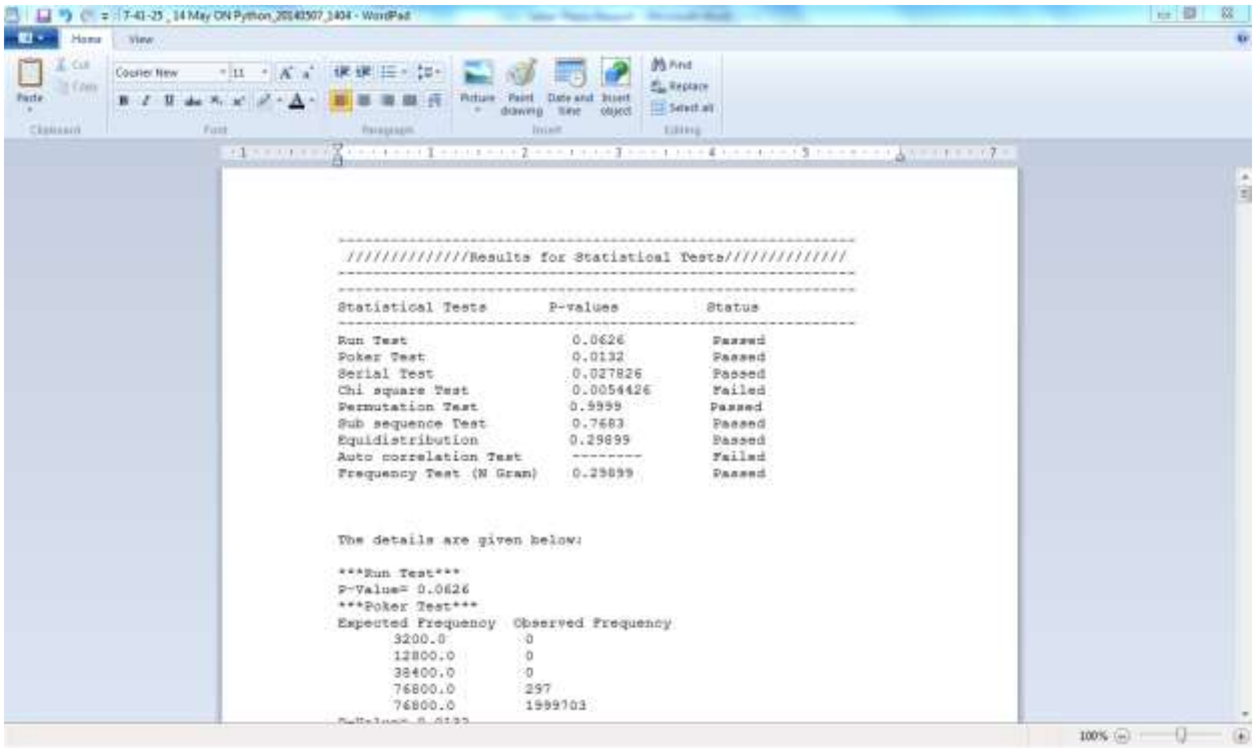
  The test code is useful in performing tests for RNGs and PRNGs both.

- **Efficiency**

  Just Linear time is utilized in testing the random sequences.

4.3.2 Interpretation of results

A simple test report is generated for the interpretation of results. A file with time stamp is generated to maintain logs whenever the statistical test is completed. The report contains the P-values of statistical tests and Pass/Fail status of the tests.



```
-----------------------------------------------------------
 //////////////Results for Statistical Tests//////////////
-----------------------------------------------------------
-----------------------------------------------------------
Statistical Tests        P-values           Status
-----------------------------------------------------------
Run Test                   0.0626            Passed
Poker Test                 0.0132            Passed
Serial Test                0.027826          Passed
Chi square Test            0.0054426         Failed
Permutation Test          0.9999            Passed
Sub sequence Test          0.7683            Passed
Equidistribution          0.29899            Passed
Auto correlation Test      --------          Failed
Frequency Test (N Gram)    0.29899           Passed


The details are given below:

***Run Test***
P-Value= 0.0626
***Poker Test***
Expected Frequency  Observed Frequency
      3200.0          0
      12800.0         0
      38400.0         0
      76800.0         297
      76800.0         1999703
```

Figure 3: Depiction of Final Report

5. WSN Simulation

5.1 Data collection and analysis

I have used the data traces collected by Dr. Adnan and Dr. Ali Khayam. They collected 24 traces in four setups. For each setup 6 traces were collected. In every experiment one sender transmitted data frames to base station and all other senders were inactive. Average frames per traces were approx. 31,000. 10 frames per second were sent and each frame had 20 bytes.

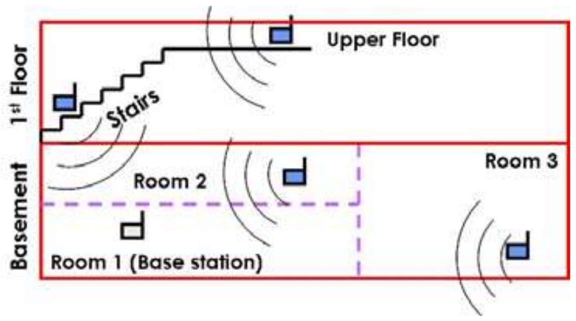


Figure 4: Setup for the trace collection

5.2 Models simulation

I have developed and evaluated two models using random event to provide our proof of concept related to simulation of Wireless Sensor Network. Memoryless as well as Markov models are developed and investigated for the presented channels.

5.2.1 2-Tier Memoryless Model

In 2-Tier model, a higher-level (Tier 1) frame-level model excites a lower-level (Tier 2) bit-error model. In the memoryless class, a BSC model is employed at the frame-level and another BSC model is employed at bit-errors whenever a frame is in error. It means if the tier 1 frame error model expects an error-free frame then there is no need to invoke the bit-error model because in that case it is known that all the bits in the frame are error-free. Whereas if the tier 1 frame error model expects a corrupt frame then the bit-level model at tier 2 is used to produce bits in error in the corrupt frame.

For the 2-Tier memoryless model, we apply BSC models at both tiers. Thus both frame and bit-level models are memoryless. BER and FER parameters for these models are extracted from the source traces discussed earlier.

// Code to generate synthetic traces of Bit Errors and Frame Errors

```
FileWriter ferFile = new
FileWriter("C:\\Source\\"+fileNames[j]+"_FE_"+selectedFolder+"_Mem.txt",false);
FileWriter berFile = new
FileWriter("C:\\Source\\"+fileNames[j]+"_BE_"+selectedFolder+"_Mem.txt",false);
fvalue=lstInputNumbers2.get(j);  //frame error rate
bvalue=lstInputNumbers1.get(j);  //bit error rate
tframes=0;
tbits=0;
l=-1;
a=0;
for (i=0;i<lstInputNumbers.get(j);i++)  //loop will run for total frames of traces
{
  if(l==9999999)
  {
     a++;l=0;
  }
  else l++;
  if(a>=asize)
     a=0;
  if((randInputNumbers.get(a).get(l)<=fvalue)&&(randInputNumbers.get(a).get(l)>=0.0))
  {
     tframes++;
     ferFile.append(one);
     for(k=0;k<160;k++)
     {
       if(l==9999999)
       {
          a++;l=0;
       }
       else
          l++;
       if(a>=asize)
          a=0;
  if((randInputNumbers.get(a).get(l)<=bvalue)&&(randInputNumbers.get(a).get(l)>=0.0))
  {
          tbits++;
          berFile.append(one);
       }
       else
       {
          berFile.append(zero);
       }
     }
```

```
         }
        else
         {
             ferFile.append(zero);
             for(k=0;k<160;k++)
             {
                berFile.append(zero);
             }
         }
       }
       ferFile.flush();
       ferFile.close();
       berFile.flush();
       berFile.close();
       ferror[j]=tframes;
       berror[j]=tbits;
```

5.2.2 2-Tier Markov Model

In 2-Tier model, a higher-level (Tier 1) frame-level model excites a lower-level (Tier 2) bit-error model. A 2-Tier Markov Model is a model takes memory into account and uses a Gilbert model for frame errors at Tier 1 and a 3rd order Markov model for bit-errors at Tier 2 [4]. The parameters required for these models are extracted from source traces discussed in Chapter 2.

// Code to generate synthetic traces of Bit Errors and Frame Errors

```
       FileWriter ferFile = new
       FileWriter(fileNameFER+"_"+selectedFolder+"_Markov.txt",false);
       FileWriter berFile = new
       FileWriter(fileNameBER+"_"+selectedFolder+"_Markov.txt",false);
       char zero='0',one='1';
       for (int i=0;i<lstInputNumbers.get(fn);i++)
       {
               if(l==9999999)
               {
                  a++;l=0;
               }
               else
                  l++;
               if(a>=asize)
                  a=0;
if((0.0<=randInputNumbers.get(a).get(l))&&(randInputNumbers.get(a).get(l)<=transitionMatrixFE
[frow][0]))
          {
             fstate=fstate.substring(0,1).concat(Integer.toString(0));
             frow=Integer.parseInt(Integer.toHexString(Integer.parseInt(fstate, 2)));
             ferFile.append(zero);
```

```
                for(int k=0;k<160;k++)
                {
                    berFile.append(zero);
                }
            }
else if((transitionMatrixFE[frow][0]< randInputNumbers.get(a).get(l))&&(
randInputNumbers.get(a).get(l)<=1.0))
            {
                fstate=fstate.substring(0,1).concat(Integer.toString(1));
                frow=Integer.parseInt(Integer.toHexString(Integer.parseInt(fstate, 2)));
                ferFile.append(one);
                tframes++;
                for(int k=0;k<160;k++)
                {
                   if(l==9999999)
                   {
                       a++;l=0;
                   }
                   else
                       l++;
                   if(a>=asize)
                       a=0;

if((0.0<=randInputNumbers.get(a).get(l))&&(randInputNumbers.get(a).get(l)<=transitionMatrix[br
ow][0]))
                   {
                     bstate=bstate.substring(1,3).concat(Integer.toString(0));
                     brow=Integer.parseInt(Integer.toHexString(Integer.parseInt(bstate, 2)));
                     berFile.append(zero);
                   }
else if((transitionMatrix[brow][0]< randInputNumbers.get(a).get(l))&&(
randInputNumbers.get(a).get(l)<=1.0))
                     {
                     bstate=bstate.substring(1,3).concat(Integer.toString(1));
                     brow=Integer.parseInt(Integer.toHexString(Integer.parseInt(bstate, 2)));
                     berFile.append(one);
                     tbits++;
                     }
                  }
            }
        }
        ferFile.flush();
        ferFile.close();
        berFile.flush();
        berFile.close();
```

5.3 R Divergence measure calculation

For every synthetic bit error traces generated by implementing different random number generators is saved at a path “C://Source”. R Divergence measure for source and synthetic trace is calculated for both models to find the similarity between the error distributions.

Moreover Divergence measure is also calculated among the synthetic traces generated for each RNG Source to find the similarity level among them. 28 combinations of synthetic trace distributions are found for each of the 24 experiments. Resulted calculations are presented in next chapter whereas the code to calculate the results is given below. Following code can also be found in **KlDivergence.java**.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import jxl.Workbook;
import jxl.read.biff.BiffException;
import jxl.write.Label;
import jxl.write.Number;
import jxl.write.WritableSheet;
import jxl.write.WritableWorkbook;
import jxl.write.WriteException;

public class KlDivergence
{
   public static double calculateDiv(String fileName1,String fileName2)
   {
    String testRF1=null,testRF3=null;
    double cal,klBER1=0.0,klBER2=0.0;
    double berKL;

    try
    {
       BufferedReader brb = new BufferedReader(new FileReader(fileName1+".txt"));
       StringBuilder sbb = new StringBuilder();
       String lineb = brb.readLine();
       while (lineb != null)
       {
           sbb.append(lineb);
           lineb = brb.readLine();
       }
       testRF1=sbb.toString();
       brb.close();

    }
```

```
catch (Exception e)
{
   System.out.println(e.toString());
}
try
{
   BufferedReader brb = new BufferedReader(new FileReader(fileName2+".txt"));
   StringBuilder sbb = new StringBuilder();
   String lineb = brb.readLine();
   while (lineb != null)
   {
       sbb.append(lineb);
       lineb = brb.readLine();
   }
   testRF3=sbb.toString();
   brb.close();
}
catch (Exception e)
{
   System.out.println(e.toString());
}
int ind;
int j,i;
int length2=testRF1.length()*4/7;
int length3=testRF3.length()*4/7;
int ff1[]= new int[length2];
double pf1[]= new double[length2];
int ff3[]= new int[length3];
double pf3[]= new double[length3];
for (i=0; i<length2; i++)
{
  ff1[i]=0;
  pf1[i]=0.0;
}
for (i=0; i<length3; i++)
{
  ff3[i]=0;
  pf3[i]=0.0;
}
int burst;
int maxIndex=0,maxIndex1=0,max2;
burst=0;
for (i=0; i<testRF1.length(); i++)
{
       if(testRF1.charAt(i)=='1')
   {
      ind=testRF1.indexOf("0", i);
      if(ind==-1)
      {
          i=testRF1.length()-1;
      }
```

```
        else
        {
            ff1[ind-i]++;
            i=ind;
        }
        burst++;
    }
}
for (i=0; i<length2; i++)
{
    if(ff1[i]!=0)
    {maxIndex=i;pf1[i]=(double)ff1[i]/burst;}
}
burst=0;
for (i=0; i<testRF3.length(); i++)
{
     if(testRF3.charAt(i)=='1')
    {
        ind=testRF3.indexOf("0", i);
        if(ind==-1)
        {
            i=testRF3.length()-1;
        }
        else
        {
            ff3[ind-i]++;
            i=ind;
        }
        burst++;
    }
}

for (i=0; i<length3; i++)
{
    if(ff3[i]!=0)
    {maxIndex1=i;pf3[i]=(double)ff3[i]/burst;}
}

if(maxIndex<maxIndex1)
    max2=maxIndex;
else
    max2=maxIndex1;

cal=0.0;
for (int s=0;s<max2;s++)
{
    if((pf1[s]!=0.0)&&(pf3[s]!=0.0))
    {
    cal=pf1[s]*Math.log(pf1[s]/pf3[s]);
    klBER1+=cal;
    }
    }
    cal=0.0;
    for (int s=0;s<max2;s++)
    {
       if((pf1[s]!=0.0)&&(pf3[s]!=0.0))
       {
       cal=pf3[s]*Math.log(pf3[s]/pf1[s]);
       klBER2+=cal;
       }
    }
    if((klBER1==0.0)&&(klBER2==0.0))
       berKL=0.0;
    else
       berKL=(klBER1*klBER2)/(klBER1+klBER2);
    if(berKL<0)
       berKL=(-1)*berKL;
    return berKL;
   }

   public static void main(String[] args) throws BiffException, IOException
   {
       int i,m,n,k;
       double klValueMem[][]= new double[28][24];
       double klValueMark[][]= new double[28][24];
       double klDivMem[][]= new double[28][4];
       double klDivMark[][]= new double[28][4];

       String
fileNames[]={"OutRoom_1_5m","OutRoom_3_8m","OutRoom_4_7m","OutRoom_5_8m","OutRo
om_6_5m","OutRoom_7_5m","PhdLab_1_8m","PhdLab_2_7m","PhdLab_3_8m","PhdLab_4_7
m","PhdLab_5_5m","PhdLab_6_6m","Stair_1_5m","Stair_2_5m","Stair_3_5m","Stair_4_5m","St
air_6_5m","Stair_7_4m","Up_Floor_1_12m","Up_Floor_2_12m","Up_Floor_3_12m","Up_Floor_
4_12m","Up_Floor_5_12m","Up_Floor_7_12m"};
String[] file = {"Coutput","C-RNG-Output","Csoutput","Java-RNG-Output","Joutput","OpenSSL-
Output","Python-Output","SFMT-Output"};
       String path="C:\\Source\\";
       for(i=0;i<28;i++)
       {
          for(m=0;m<24;m++)
          {
             klValueMem[i][m]=0.0;
             klValueMark[i][m]=0.0;
             if(m<4)
             {
             klDivMem[i][m]=0.0;
             klDivMark[i][m]=0.0;
             }
          }
       }
       for(k=0;k<24;k++)
       {
        n=0;
        for(i=0;i<8;i++)
        {
            for(m=i+1;m<8;m++)
            {
klValueMem[n][k]=calculateDiv(path+fileNames[k]+"_BE_"+file[i]+"_Mem",path+fileNames[k]+"_
BE_"+file[m]+"_Mem");
klValueMark[n][k]=calculateDiv(path+fileNames[k]+"_BE_"+file[i]+"_Markov",path+fileNames[k]+
"_BE_"+file[m]+"_Markov");
                n++;
            }
        }
        }

        for(i=0;i<28;i++)
        {
        n=0;
        for(m=0;m<24;m+=6)
        {
klDivMem[i][n]=klValueMem[i][m]+klValueMem[i][m+1]+klValueMem[i][m+2]+klValueMem[i][m+3
]+klValueMem[i][m+4]+klValueMem[i][m+5];
klDivMark[i][n]=klValueMark[i][m]+klValueMark[i][m+1]+klValueMark[i][m+2]+klValueMark[i][m+3
]+klValueMark[i][m+4]+klValueMark[i][m+5];
        n++;
        }
        }
        for(i=0;i<28;i++)
        {
        for(m=0;m<4;m++)
        {
        klDivMem[i][m]=klDivMem[i][m]/6.0;
        klDivMark[i][m]=klDivMark[i][m]/6.0;
        }
        }
        try
        {
        WritableWorkbook wworkbook1;
        wworkbook1 = Workbook.createWorkbook(new File("C:\\Memoryless
Model\\klDivergenceMem.xls"));
        WritableSheet wsheet1 = wworkbook1.createSheet("Comparison", 0);
        WritableWorkbook wworkbook2;
        wworkbook2 = Workbook.createWorkbook(new File("C:\\Markov
Model\\klDivergenceMark.xls"));
        WritableSheet wsheet2 = wworkbook2.createSheet("Comparison", 0);
        Label label00 = new Label(1, 1, "Comparison of KL Measures for RNG Sources with
MemoryLess Model");
        wsheet1.addCell(label00);
        Label label0 = new Label(0, 3, "OutRoom");
        wsheet1.addCell(label0);
        Label label1 = new Label(1, 3, "PhdLab");
        wsheet1.addCell(label1);
```

```
	Label label2 = new Label(2, 3, "Stairs");
	wsheet1.addCell(label2);
	Label label3 = new Label(3, 3, "UpFloor");
	wsheet1.addCell(label3);
	n=0;
	for (m=4;m<32;m++)
	{
	Number number0 = new Number(0, m, klDivMem[n][0]);
	wsheet1.addCell(number0);
	Number number1 = new Number(1, m, klDivMem[n][1]);
	wsheet1.addCell(number1);
	Number number2 = new Number(2, m, klDivMem[n][2]);
	wsheet1.addCell(number2);
	Number number3 = new Number(3, m, klDivMem[n++][3]);
	wsheet1.addCell(number3);
	}
	Label label11 = new Label(7, 1, "Comparison of KL Measures for RNG Sources with
Markov Model");
	wsheet2.addCell(label11);
	Label label4 = new Label(0, 3, "OutRoom");
	wsheet2.addCell(label4);
	Label label5 = new Label(1, 3, "PhdLab");
	wsheet2.addCell(label5);
	Label label6 = new Label(2, 3, "Stairs");
	wsheet2.addCell(label6);
	Label label7 = new Label(3, 3, "UpFloor");
	wsheet2.addCell(label7);
	n=0;
	for (m=4;m<32;m++)
	{
	Number number0 = new Number(0, m, klDivMark[n][0]);
	wsheet2.addCell(number0);
	Number number1 = new Number(1, m, klDivMark[n][1]);
	wsheet2.addCell(number1);
	Number number2 = new Number(2, m, klDivMark[n][2]);
	wsheet2.addCell(number2);
	Number number3 = new Number(3, m, klDivMark[n++][3]);
	wsheet2.addCell(number3);
	}
	wworkbook1.write();wworkbook1.close();
	wworkbook2.write();wworkbook2.close();
	}
	catch(IOException ie)
	{	System.out.println(ie.toString());		}
	catch(WriteException we)
	{	System.out.println(we.toString());	}
	System.out.println("Done!!");
	}
}
```

6. Results Evaluation

In this chapter first we evaluate the results produced by the test application (elaborated in Chapter 4) against output of every random number generator. These results provide the background, properties and the level of quality of each random number files we have generated in Chapter 3.

After that we evaluate synthetic traces of bit-errors and frame errors generated from each model simulation. Later we calculate the R Divergence measure of each trace for both models using the formula given in Chapter 1. Hence we get the resultant R values for each setup that is particular to some Random Number Source for both models. For each model eight result sets were produced from 8 RNG sources. R measures show the difference in results of real and synthetic trace outputs. It must be noted that smaller the value of R measure, smaller the difference in results.

Cumulative R Divergence which is calculated with two synthetic trace result sets at a time using outputs of two RNGs is also evaluated later to know the closeness of simulation result sets making use of two different RNGs. Purpose of R Divergence measure calculation among the synthetic traces generated for each RNG Source is to find the similarity level among them. 28 combinations of synthetic trace distributions are found for each of the 24 experiments.

Following factors will be evaluated using the information gathered from results to prove the impact of randomness in simulation based studies.

1. Simulation provides correct results if the right Random Number Generator is chosen.
2. Random Number Generators with similar properties when used in the simulation produce similar results.

6.1 Results of Tests

First of all the random numbers are evaluated by the statistical tests and results in the form of Pass/Fail are recorded for each test. Each random file saves 10 million integers produced by that particular random generator code.

Test Application is used to compile the test results for each random file generated from available eight Random Number Generators. Test results of all Random numbers Generators are given in a table. It is very clear with the results that each RNG lack in some random property. Few are good in one property while others are bad in the same property. So every random number generator is focusing on specifics of their own choice. Results generated for every statistical test is compiled in a table presented following.

Random Number Generators	**Random Files Names**	**Passing Ratio**	**Failure Ratio**
Coutput	CPP_20140126_2313	0.7	0.3
	CPP_20140225_2116		
	CPP_20140225_2121		
C-RNG-Output	C_20140507_1411	0.7333	0.266667
	C_20140507_1412		
	C_20140507_1413		
Csoutput	CS-20140225_2125	0.7333	0.266667
	CS-20140225_2126		
	CS-20140225_2127		
Java-RNG-Output	J_201457_1344	0.8	0.2
	J_2014423_1440		
	J_2014425_1011		
Joutput	Java_2014425_1133	0.8	0.2
	Java_2014428_1024		
	Java_2014514_1836		
OpenSSL	PRNG_20140508_1425	0.8333	0.166667
	PRNG_20140508_1457		
	PRNG_20140508_1458		

Python-Output	Python_20140507_1400	0.8	0.2
	Python_20140507_1402		
	Python_20140507_1404		
SFMT-Output	SFMT_201457_1418	0.9	0.1
	SFMT_201457_1419		
	SFMT_201457_1420		

Table 3: Passing and Failure Ratio of Tests applied to Random Files

Following graph is the evaluation parameter to prove the Factor 1 which states that RNG with high failure ratio when used in simulation will provide the results that differ from the real results in big number. Graph is presenting the failure ratio of numbers produced by RNGs mentioned in Chapter 3. Ratio is calculated from the resultant files created during test application phase.

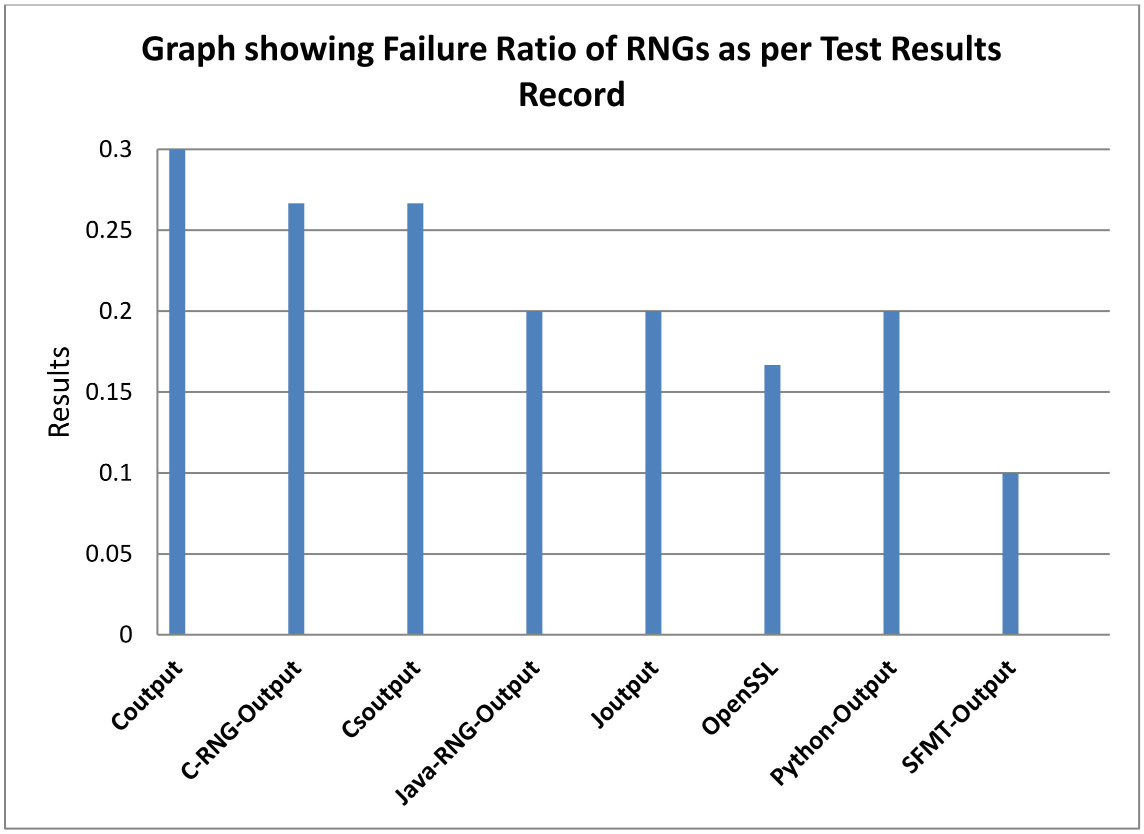


Figure 5: Graph showing Failure Ratio of RNGs as per Test Results Record

Another graph presented below is showing similar and different Random Number Generators on the basis of their Pass/Fail test statistics. This graph is the evaluation parameter to prove the Factor 2 which states that similar Random Number Generators when used in simulation will produce similar results.

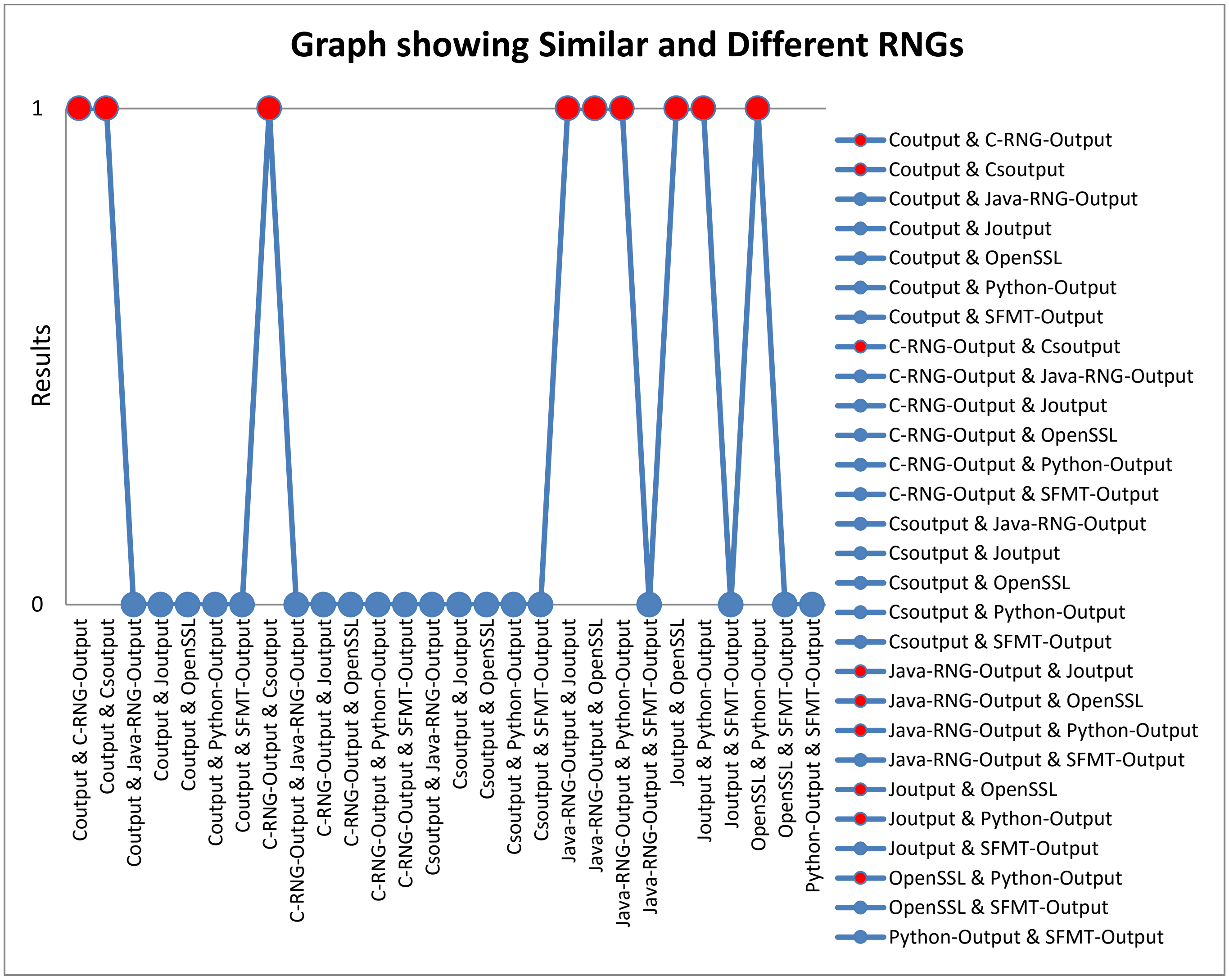


Figure 6: Graph showing Similar and Different RNGs

6.2 2-Tier Memoryless Model

6.2.1 Frame-Error Rate

Table 4 shows comparison of real results and simulated results (generated using Rand PRNG in C) of Frame Error Rate for each setup using 2-Tier Memoryless Model. It must be noted that result is averaged over 6 traces per setup.

Setup Title	Total Frames	Source FER	Memoryless FER
OutRoom	208426	0.01863137	0.01680303
PhdLab	205385	0.29216057	0.304433333
Stair	179132	0.150592548	0.15508186
Up_floor	119763	0.133074983	0.13650929

Table 4: Average FER Comparison with 2-Tier Memoryless Model

6.2.2 Bit-Error Rate

Table 5 presents comparison of real results and simulated results (generated using Rand PRNG in C) of Bit Error Rate for each setup using 2-Tier Memoryless Model. Results of 24 traces are averaged for four setups.

Setup Title	Total Bits	Source BER	Memoryless BER
OutRoom	33348160	0.000483432	0.002400373
PhdLab	32861600	0.013385878	0.291544995
Stair	28661120	0.0054245	0.135821988
Up_floor	19162080	0.006199407	0.129482923

Table 5: Average BER Comparison with 2-Tier Memoryless Model

6.2.3 R Divergence Measure

Table 6 shows average R Divergence measure for each setup with 2-Tier Memoryless Model (generated using Rand PRNG in C) as given following:

Setup	**R Divergence**
OutRoom	0.109410675
PhdLab	0.086685092
Stair	0.101079727
Up_floor	0.3112344

Table 6: Average R Divergence Measure with 2-Tier Memoryless Model

6.3 2-Tier Markov Model

6.3.1 Frame-Error Rate

Table 7 shows comparison in real results and simulated results (generated using Rand PRNG in C) of Frame Error Rate for each setup using 2-Tier Markov Model. It must be noted that result is averaged over 6 traces per setup.

Setup Title	**Total Frames**	**Source FER**	**Markov FER**
OutRoom	208426	0.01863137	0.017451923
PhdLab	205385	0.29216057	0.090240717
Stair	179132	0.150592548	0.13007733
Up_floor	119763	0.133074983	0.138740097

Table 7: Average FER Comparison with 2-Tier Markov Model

6.3.2 Bit-Error Rate

Table 8 presents comparison in real results and simulated results (generated using Rand PRNG in C) of Bit Error Rate for each setup using 2-Tier Markov Model. Results of 24 traces are averaged for four setups.

Setup Title	**Total Bits**	**Source BER**	**Markov BER**
OutRoom	33348160	0.000483432	9.75167E-06
PhdLab	32861600	0.013385878	0.000289392
Stair	28661120	0.0054245	0.00062454
Up_floor	19162080	0.006199407	0.000714957

Table 8: Average BER Comparison with 2-Tier Markov Model

6.3.3 R Divergence Measure

Table 9 shows average R Divergence measure for each setup with 2-Tier Markov Model (generated using Rand PRNG in C) as given below:

Setup	**R Divergence**
OutRoom	0.257466
PhdLab	0.007792
Stair	0.003276
Up_floor	0.004997

Table 9: Average R Divergence Measure with 2-Tier Markov Model

6.4 Cumulative R Divergence Measure Results

6.4.1 2-Tier Memoryless Model

R divergence Measure produced in the output files is combined to see the overall behavior of the simulation results. Average R Divergence Measure using 2-Tier Memoryless model for the four setups is given in the following table. High value of R measure shows the bad simulation results and low values represent good simulation results.

	Coutput	**C-RNG-Output**	**Csoutput**	**Java-RNG-Output**	**Joutput**	**OpenSSL**	**Python-Output**	**SFMT-Output**
OutRoom	0.477235	0.258231	0.201755	0.177566	0.11223	0.146836	0.179476	0.109411
PhdLab	0.216847	0.1349	0.164416	0.113509	0.080948	0.086685	0.087948	0.056419
Stairs	0.492371	0.105607	0.128032	0.10223	0.100534	0.077867	0.10108	0.056342
Up_Floor	0.582273	0.44347	0.370484	0.346066	0.280051	0.267194	0.311234	0.118585

Table 10: Comparison in R Measures using Memoryless Model for given RNGs

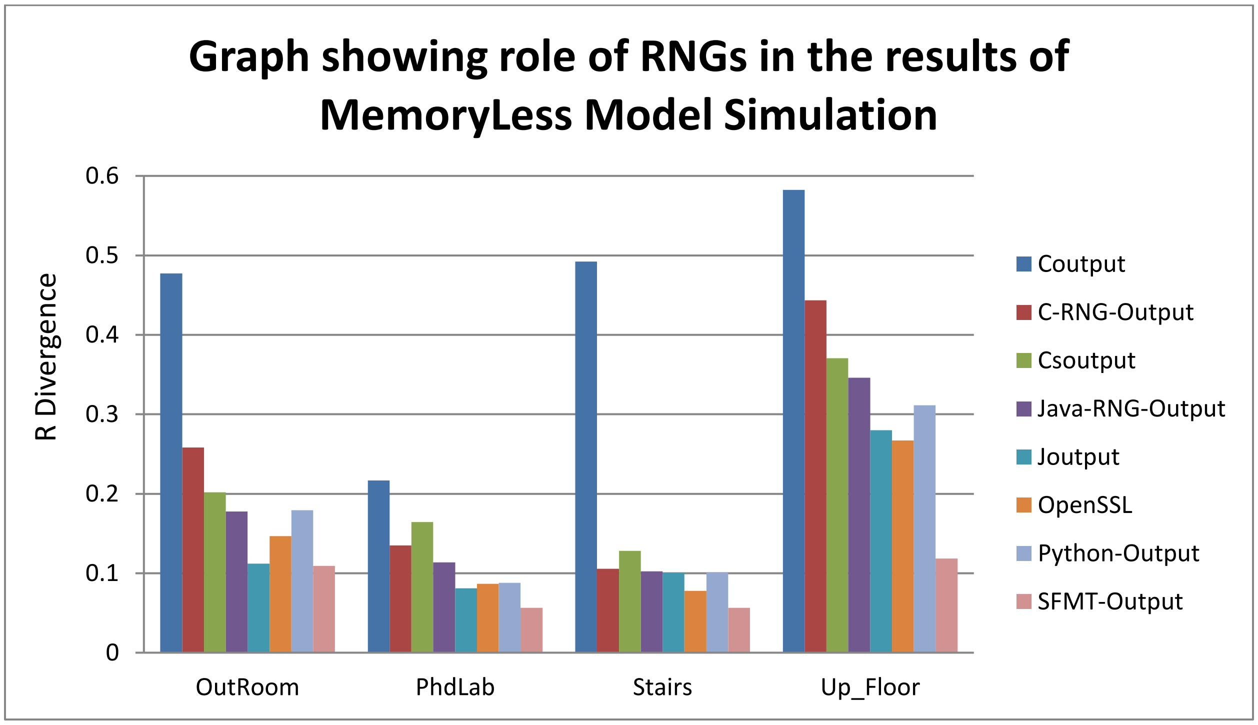


Figure 7: Graph showing role of RNGs in the results of MemoryLess Model Simulation

Analysis:

The graph of above table shows the difference in real trace results and synthetic trace results of Memoryless model simulation. If we analyze the graph we see the RNGs in the following order from highest value to lowest.

Out Room: Coutput - > C-RNG-Output - > Csoutput - > Java-RNG-Output - > Python-Output - > OpenSSL - > Joutput - > SFMT-Output

Phd Lab: Coutput - > Csoutput - > Java-RNG-Output s- > C-RNG-Output - > OpenSSL - > Python-Output - > Joutput - >SFMT-Output

Stairs: Coutput - > Csoutput - > C-RNG-Output - > Java-RNG-Output - > Joutput - > Python-Output - > OpenSSL - > SFMT-Output

Up Floor: Coutput - > C-RNG-Output - > Csoutput - > Java-RNG-Output - > Python-Output - > Joutput -> OpenSSL - > SFMT-Output

According to test results the following order of RNGs as per ratio of failure is given. Order from highest failure ratio to lowest is identified.

RNG Results: Coutput - > C-RNG-Output - > Csoutput - > Java-RNG-Output - > Joutput - > Python-Output - > OpenSSL - > SFMT-Output

As C-RNG-Output and Csoutput are equal, we can interchange them. Similarly Java-RNG-Output, Joutput and Python-Output are equal, we can also interchange these.

If we compare each setup one by one with the RNG results, two values of Out Room are different in the order. In Phd Lab four values are different whereas Stairs is exactly same. Up floor is also exactly same in order.

So overall two setups are completely reflecting the results calculated from the test application whereas results of Out Room and PhdLab are partially same. It proves the Factor 1 that simulation provides correct results if the right Random Number Generator is chosen and simulation differs from real results if weak RNG is used in the simulation.

6.4.2 2-Tier Markov Model

Average R Divergence Measure using 2-Tier Markov model for the four setups is given in the following table. High value of R measure shows the bad simulation results and low values represent good simulation results.

	Coutput	**C-RNG-Output**	**Csoutput**	**Java-RNG-Output**	**Joutput**	**OpenSSL**	**Python-Output**	**SFMT-Output**
OutRoom	0.047915	0.043125	0.046276	0.023656	0.018464	0.022594	0.018713	0.009659
PhdLab	0.007792	0.007892	0.00577	0.007664	0.003666	0.00539	0.005906	0.004817
Stairs	0.003276	0.003388	0.002935	0.004534	0.004198	0.004291	0.005118	0.004306
Up_Floor	0.004997	0.004683	0.008582	0.004644	0.00659	0.005428	0.009623	0.00414

Table 11: Comparison in R Measures using Markov Model for given RNGs

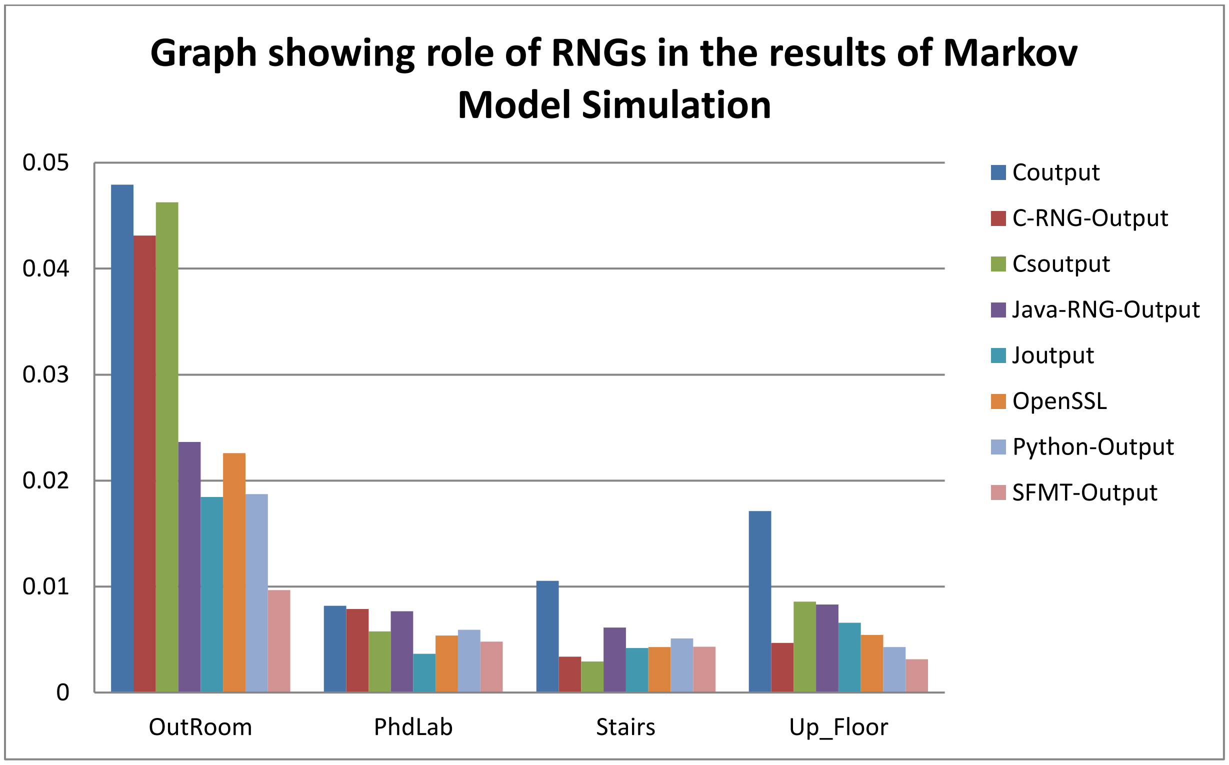


Figure 8: Graph showing role of RNGs in the results of Markov Model Simulation

Analysis:

Above graph is drawn from the table 11 and shows the difference in real traces and synthetic trace results of markov model simulation. If we analyze the graph we see RNGs in the following order from highest value to lowest.

Out Room: Coutput - > Csoutput - > C-RNG-Output - > Java-RNG-Output - > OpenSSL - > Joutput -> Python-Output -> SFMT-Output

Phd Lab: Coutput - > C-RNG-Output - > Java-RNG-Output - > Csoutput - > Python-Output - > OpenSSL - > SFMT-Output - > Joutput

Stairs: Coutput - > Java-RNG-Output - > Python-Output - > Joutput - > OpenSSL - > SFMT-Output - > C-RNG-Output - > Csoutput

Up Floor: Coutput - > Csoutput - > Java-RNG-Output - > Joutput - > OpenSSL - > C-RNG-Output - > Python-Output - > SFMT-Output

According to test results the following order of RNGs as per ratio of failure is identified. Order from highest failure ratio to lowest is identified.

RNG Results: Coutput - > C-RNG-Output - > Csoutput - > Java-RNG-Output - > Joutput - > Python-Output - > OpenSSL - > SFMT-Output

As C-RNG-Output and Csoutput are equal, we can interchange them. Similarly Java-RNG-Output, Joutput and Python-Output are equal, we can also interchange these.

If we compare each setup one by one with the RNG results, one value of Out Room is different in the order. In Phd Lab two values are different. In setup of Stairs one value is out of order. One value is out of order in Up floor.

So overall just few values are not in saline to the base order. It proves the Factor 1 that simulation provides correct results if the right Random Number Generator is chosen and simulation differs from real results if weak RNG is used in the simulation.

6.5 R Divergence Measure Results w.r.t RNGs

6.5.1 2-Tier Memoryless Model

R Diverence measure is used to make comparisons and generate the values given in following table. The table depicts the comparison of Random Source results in combination calculated with 2-Tier Memoryless Model.

RNG SOURCES		**SETUP TITLES**			
RNG Source 1	**RNG Source 2**	**OutRoom**	**PhdLab**	**Stairs**	**Up_floor**
Coutput	C-RNG-Output	0.000214334	0.000889807	0.000652314	0.324821612
Coutput	Csoutput	0.00025825	0.000479126	0.001025599	0.323213666
Coutput	Java-RNG-Output	0.160668197	0.162359394	0.00810264	0.543897932
Coutput	Joutput	0.161222014	0.163302043	0.001056084	0.32537085
Coutput	OpenSSL	0.064397595	0.118612493	0.092809878	0.321987078
Coutput	Python-Output	0.162963522	0.077875547	0.003177214	0.453830562
Coutput	SFMT-Output	0.922788893	0.511641194	0.782129509	0.505058339
C-RNG-Output	Csoutput	5.39716E-05	0.000581962	0.023654875	0.48386394
C-RNG-Output	Java-RNG-Output	0.16094804	0.16187647	0.000723321	0.482759364
C-RNG-Output	Joutput	0.161053449	0.162950451	0.00062044	0.485306759
C-RNG-Output	OpenSSL	0.068262819	0.122335577	0.098576828	0.480814658
C-RNG-Output	Python-Output	0.162245471	0.000746184	0.000609049	0.324588326
C-RNG-Output	SFMT-Output	0.963065683	0.508898778	0.900111076	0.529624362

Csoutput	Java-RNG-Output	0.160784442	0.161334305	0.001221993	0.323382062
Csoutput	Joutput	0.160636476	0.162391982	0.001033075	0.328250173
Csoutput	OpenSSL	0.068941311	0.117830306	0.096137708	0.339392389
Csoutput	Python-Output	0.16190204	0.000961355	0.001078611	0.323375184
Csoutput	SFMT-Output	0.984892637	0.499143528	0.81623286	0.809559662
Java-RNG-Output	Joutput	0.160394418	0.162043103	0.000519161	0.34875919
Java-RNG-Output	OpenSSL	0.204227397	0.267921092	0.093438814	0.315887996
Java-RNG-Output	Python-Output	0.256566512	0.162121287	0.000814929	0.498924507
Java-RNG-Output	SFMT-Output	0.670121624	0.233478205	0.814245795	0.520889161
Joutput	OpenSSL	0.209062537	0.273010422	0.098268331	0.340861275
Joutput	Python-Output	0.162162976	0.163419781	0.000638312	0.353730246
Joutput	SFMT-Output	0.638385224	0.223349223	0.865583564	0.810368064
OpenSSL	Python-Output	0.207990444	0.121469419	0.091653495	0.316453204
OpenSSL	SFMT-Output	0.615249777	0.333590344	0.758587033	1.066351603
Python-Output	SFMT-Output	0.701015915	0.519894306	0.819546709	0.510185243

Table 12: Comparison in R Measures using Memoryless Model for combinational RNGs

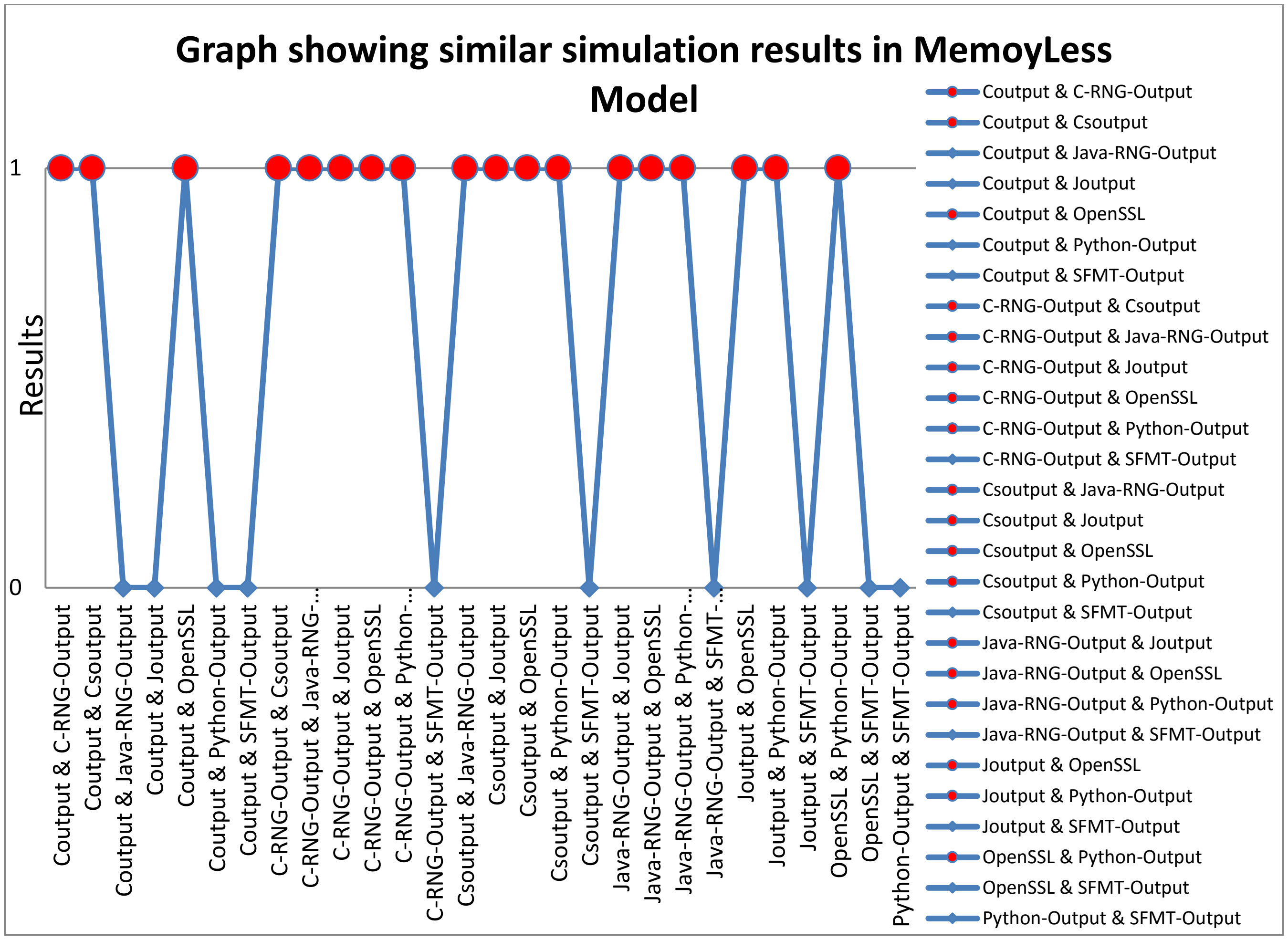


Figure 9: Graph showing similar simulation results in MemoyLess Model

<u>Analysis:</u>

Above graph is drawn from the table 12 that shows the difference in traces for each and every possible combination of RNGs. There are total 28 combinations of traces. With the thorough analysis of graph following combinations are found to be in the same range of R values.

- Coutput & C-RNG-Output
- Coutput & Csoutput
- Coutput & OpenSSL
- C-RNG-Output & Csoutput
- C-RNG-Output & Java-RNG-Output
- C-RNG-Output & Joutput
- C-RNG-Output & OpenSSL

- C-RNG-Output & Python-Output
- Csoutput & Java-RNG-Output
- Csoutput & Joutput
- Csoutput & OpenSSL
- Csoutput & Python-Output
- Java-RNG-Output & Joutput
- Java-RNG-Output & OpenSSL
- Java-RNG-Output & Python-Output
- Joutput & OpenSSL
- Joutput & Python-Output
- OpenSSL & Python-Output

According to test results the following RNGs are found similar as they have the same range of failure ratio.

- Coutput & C-RNG-Output
- Coutput & Csoutput
- C-RNG-Output & Csoutput
- Java-RNG-Output & Joutput
- Java-RNG-Output & OpenSSL
- Java-RNG-Output & Python-Output
- Joutput & OpenSSL
- Joutput & Python-Output
- OpenSSL & Python-Output

Above statistics show that every combination of RNGs which has almost same random qualities and is found similar with the results of test application when applied to simulation produces similar simulation results which can be seen by the R measures that are small in these cases.

It proves the Factor 2 that Random Number Generators with similar properties when used in the simulation produce the similar simulation results.

6.5.2 2-Tier Markov Model

Following table depicts the comparison of Random Source results calculated with 2-Tier Markov Model. R Diverence measure is used for comparison purposes.

RNG SOURCES		**SETUP TITLES**			
RNG Source 1	**RNG Source 2**	**OutRoom**	**PhdLab**	**Stairs**	**Up_floor**
Coutput	C-RNG-Output	0.192877205	0.003849958	0.001864541	0.001985602
Coutput	Csoutput	0.207651921	0.003883554	0.002050279	0.001990353
Coutput	Java-RNG-Output	0.201954425	0.002707976	0.001942049	0.001596568
Coutput	Joutput	0.213776003	0.004014622	0.001347965	0.001730313
Coutput	OpenSSL	0.222020081	0.002717561	0.001667601	0.001951581
Coutput	Python-Output	0.225785854	0.006027323	0.0012275	0.005327181
Coutput	SFMT-Output	0.227313988	0.006446073	0.001055714	0.005879822
C-RNG-Output	Csoutput	0.001227279	0.002230322	0.002037511	0.001811511
C-RNG-Output	Java-RNG-Output	0.001091014	0.001861729	0.002744856	0.002019573
C-RNG-Output	Joutput	0.001230359	0.001861754	0.001616143	0.002484099
C-RNG-Output	OpenSSL	0.001142739	0.001479328	0.00135325	0.019750144
C-RNG-Output	Python-Output	0.001204529	0.002782835	0.004936313	0.002887387
C-RNG-Output	SFMT-Output	0.000787266	0.003375483	0.001838592	0.000826109
Csoutput	Java-RNG-Output	0.00170605	0.001232065	0.002660673	0.002540947
Csoutput	Joutput	0.001754287	0.00222333	0.003054191	0.006460485
Csoutput	OpenSSL	0.001825424	0.001298876	0.0017141	0.001088671
Csoutput	Python-Output	0.001418427	0.003750892	0.002045627	0.001659767

Csoutput	SFMT-Output	0.001637238	0.000935544	0.002491999	0.000986975
Java-RNG-Output	Joutput	0.000998722	0.002233269	0.001500976	0.001602953
Java-RNG-Output	OpenSSL	0.001332982	0.003005983	0.001123086	0.002246656
Java-RNG-Output	Python-Output	0.001116658	0.001769939	0.002079416	0.002353704
Java-RNG-Output	SFMT-Output	0.000725602	0.000768695	0.001620858	0.01716366
Joutput	OpenSSL	0.002170514	0.003264637	0.005255079	0.003373971
Joutput	Python-Output	0.002647433	0.00233293	0.003330898	0.005378822
Joutput	SFMT-Output	0.002745444	0.001570234	0.006188707	0.001264581
OpenSSL	Python-Output	0.001560268	0.00220994	0.001241794	0.002348881
OpenSSL	SFMT-Output	0.002212597	0.002333056	0.001175201	0.001363032
Python-Output	SFMT-Output	0.001769881	0.003549887	0.001152062	0.006701309

Table 13: Comparison in R Measures using Markov Model for combinational RNGs

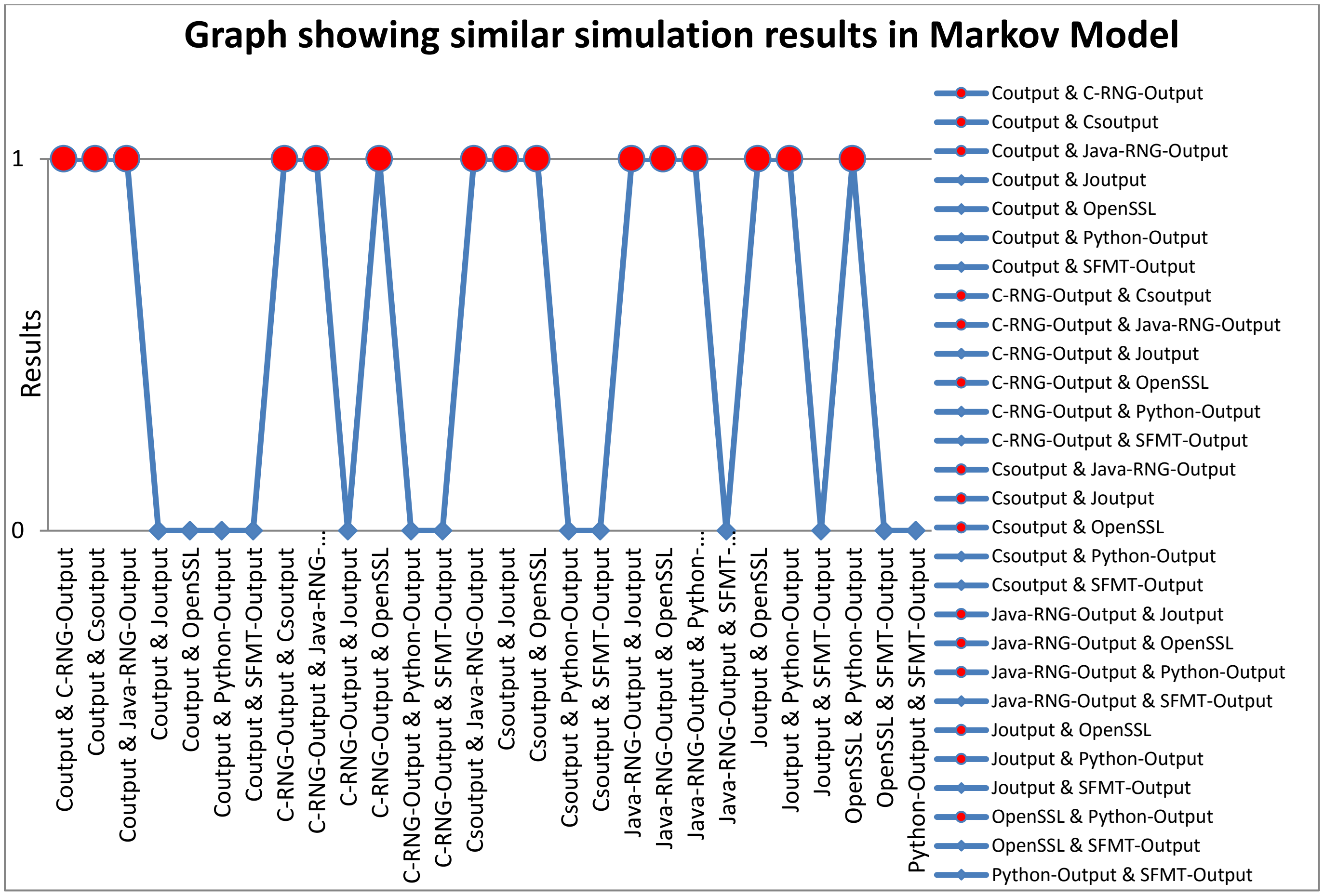


Figure 10: Graph showing similar simulation results in Markov Model

<u>Analysis:</u>

Above graph is drawn from the table 13 and shows the difference in traces for each and every possible combination of RNGs. There are total 28 combinations of traces. With the thorough analysis of graph following combinations of simulation results are found to be in the same range of R values.

- Coutput & C-RNG-Output
- Coutput & Csoutput
- Coutput & Java-RNG-Output
- C-RNG-Output & Csoutput
- C-RNG-Output & Java-RNG-Output
- C-RNG-Output & OpenSSL
- Csoutput & Java-RNG-Output

- Csoutput & Joutput
- Csoutput & OpenSSL
- Java-RNG-Output & Joutput
- Java-RNG-Output & OpenSSL
- Java-RNG-Output & Python-Output
- Joutput & OpenSSL
- Joutput & Python-Output
- OpenSSL & Python-Output

According to test results the following RNGs were found similar as they have the same range of failure ratio.

- Coutput & C-RNG-Output
- Coutput & Csoutput
- C-RNG-Output & Csoutput
- Java-RNG-Output & Joutput
- Java-RNG-Output & OpenSSL
- Java-RNG-Output & Python-Output
- Joutput & OpenSSL
- Joutput & Python-Output
- OpenSSL & Python-Output

Above statistics show that every combination of RNGs which has almost same random qualities and is found similar with the results of test application when applied to simulation produces similar simulation results which can be seen by the R measures that are small in these cases.

It proves the Factor 2 that Random Number Generators with similar properties when used in the simulation produces the similar simulation results.

6.6 Conclusion

Hence we can conclude in lines that to prove the impacts of Randomness in Simulations, following factors are evaluated against the information and results gathered.

1. Simulations provide correct results if the strong RNG with respect to randomness is the choice.
2. Random Number Generators with similar properties when used in the simulation produce the similar simulation results.

7. Conclusion

Thesis can be divided into two main sections. First of all we presented a comprehensive research for evaluating randomness of RNGs in different Programming Languages and cryptographic libraries using Statistical Random Test Algorithms. Later the impact of random numbers on simulation based studies was evaluated in detail. As the use of random numbers generated through APIs of programming languages is more than other options of number generation, the most common PRNGs provided by C, C++, Java, C#, Python, OpenSSL, SFMT are evaluated in two ways. The quality and reliability was first assessed using standard random number testing procedures such as Equidistribution Test, Run Test, Serial Test, and Chi square Test. Later we used the same random numbers in a Markov chain based probabilistic study of Wireless Sensor Networks. For analysis purpose R Divergence measure was used. The results and analysis reveals that choice of random numbers has direct impact over simulation based studies. Use of strong random numbers lead to better and reliable simulations while bad RNG selection will end in bad and unrealistic results that will be vary from real results in a big number. Another concept which is presented in thesis is related to using a RNG same in random properties with some other RNG. The results presented in last chapter are enough to prove that similar RNGs will have same impacts over simulations and almost same results will be generated by applying similar RNGs.

8. References

[1] Havedanloo, Saeed, and Hamid Reza Karimi, “Improving the Performance Metric of Wireless Sensor Networks with Clustering Markov Chain Model and Multilevel Fusion”, Mathematical Problems in Engineering, 2013.

[2] P. Hellekalek, “Good random number generators are (not so) easy to find”, Mathematics and Computers in Simulation, Volume 46, No. 5, 1998, pp. 485-505.

[3] NIST Exploratory Data Analysis:

http://www.itl.nist.gov/div898/handbook/toolaids/pff/E-Handbook.pdf

[4] Adnan Iqbal and Syed Ali Khayam, "Improving WSN Simulation and Analysis Accuracy Using Two-Tier Channel Models," IEEE International Conference on Communications (ICC), May 2008.

[5] U. Maurer, “A Universal Statistical Test for Random Bit Generators,” Journal of Cryptology. Vol. 5, No. 2, 1992, pp. 89-105.

[6] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P., Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, 2nd edition, 1992.

[7] Noble, C. and Sugden, S. J., “Statistical Tests on Random Numbers I”, A consulting report for Jupiters Network Gaming, July 1997.

[8] Inductive and Deductive research Approach:

http://www.drburney.net/INDUCTIVE%20&%20DEDUCTIVE%20RESEARCH%20APPROACH%2006032008.pdf

[9] A. Menezes, et al., Handbook of Applied Cryptography, CRC Press, 1997. http://www.cacr.math.uwaterloo.ca/hac/about/chap5.pdf

[10] W. Press, S. Teukolsky, W. Vetterling, Numerical Recipes in C : The Art of Scientific Computing, 2nd Edition. Cambridge University Press, January 1993.

[11] Donald E.Knuth, “The Art of Computer Programming, Seminumerical Algorithms”,

Volume 2, 3rd edition, 1997.

[12] Daniel P. Biebighauser, “Testing Random Number Generators”, University of Minnesota, REU Summer 2000.

[13] Richard Simard, Pierrel’euyer, A C Library for Empirical Testing of Random Number Generators, Universit ´e de Montr´eal, 1996.

[14] Kinga Marton, A High Performance System for Generation and Testing of Random Number Sequences for Cryptographic Applications, Siemens PSE, 2010.

[15] Juan Soto, Statistical Testing of Random Number Generators, NIST 2010.

[16] http://www.graphpad.com/quickcalcs/randomN1.cfm

[17] http://www.random.org/integers/

[18] http://stattrek.com/Tables/Random.aspx

[19] http://www.dave-reed.com/Nifty/randSeq.html

[20] http://www.random.org/sequences/

Appendix A: Results for sample data

In case of binary file, Results might be given as below:

Statistical Test	P-value
Run Test	0.5782
Permutation Test	0.7608
Poker Test	0.0243
Serial Test	0.1222
Chi square Test	0.0019
Equidistribution Test	0.0312
Frequency (Bi Gram)	0.0221
Frequency (Tri Gram)	0.0111

Table 14: Test results of Sample Binary File

In case of integer file, Results might be given as below:

Statistical Test	P-value
Run Test	0.0626
Frequency Test (N Gram)	0.29899
Poker Test	0.0132

Serial Test	0.027826
Chi square Test	0.005442
Permutation Test	0.9999
Sub sequence Test	0.7683
Equidistribution Test	0.29899

Table 15: Test results of Sample Integer File

Appendix B: Graphical User Interface (GUI)

A simple graphical user interface is developed to test the statistical properties of RNGs. The source code can be found in **randomTestDemo.java**. The interface consists of a single window with four regions. The first region prompts user to get a sequence for testing randomness. For getting a sequence to be tested, he should click the “Get File” button to get a sequence. Then he must select one of the files containing sequences.

The second region consists of check list for the statistical tests. The user selects the set of statistical tests to be executed. Once the user has selected the sequence to be tested, he should click “Apply button” to invoke the statistical tests. This results in displaying the results of selected tests. Results of applied tests are shown in text area of third region. The user can also make and view sample file for testing purposes. Logs of Analysis report is also maintained for analysis purposes. The fourth region of interface shows the P-values and Pass/Fail status of selected tests. The user can later proceed to review the log files that are named according to time in folder “Analysis Reports”.

Another GUI based simulation is provided for two tier Memoryless Model. The source code can be found in **MemorylessModel.java**. The application is divided into three regions. The first region consists of a combo box and a button. User selects the PRNG of his choice by selecting entry from combo box. Later he should click “Execute button” to invoke the routines for producing synthetic traces and producing results that include FER, BER and R Divergence results. The results of real traces are presented in a second region of an application. For all 24 traces total number of frames and bits as well Frame Error Rate, Bit Error Rate calculated using a model are shown in columns of third region.

For two tier Markov Model the application format is implemented on similar lines as for Memoryless Model Simulation. The source code can be found in **MarkovModel.java**. The backend working of application is different that is clearly affected in synthetic traces of frame and bit errors. R Diverence values are also calculated for all 24 traces and average results for four setups are copied to the result sheet generated. It shall be noted that for every random number source different result sheets are generated.

1.Using Test Application

First of all, user enters password to go to the main screen.

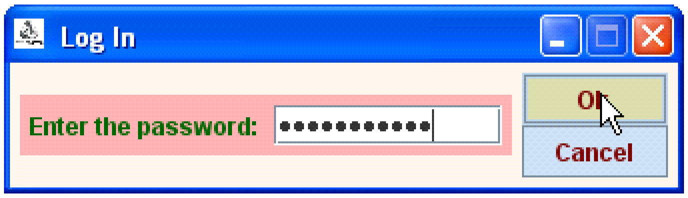


Figure 11: User Authentication Dialogue Box

If the user inputs authenticated password, he sees the main screen of an application.

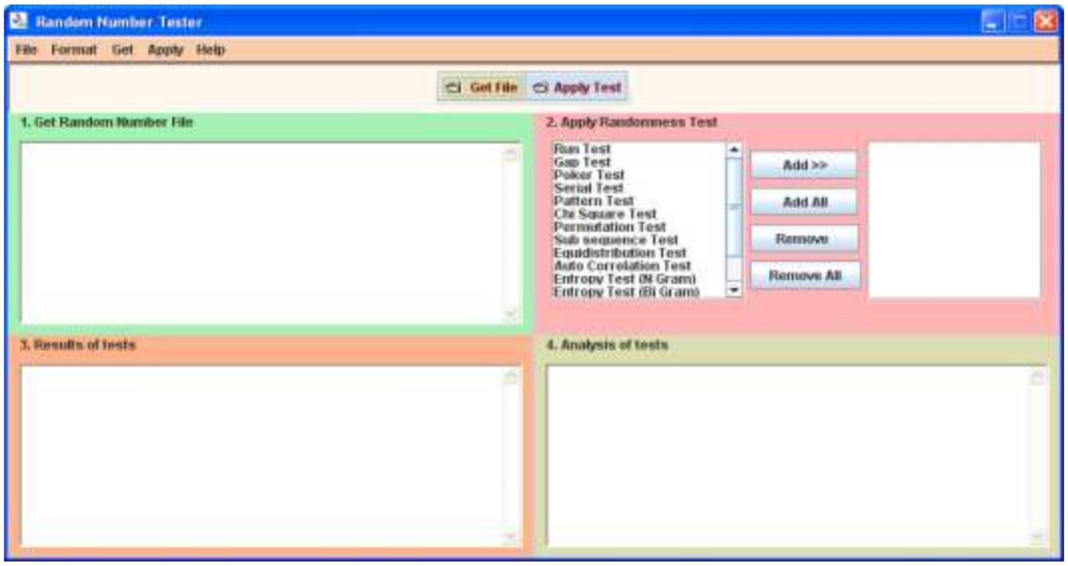
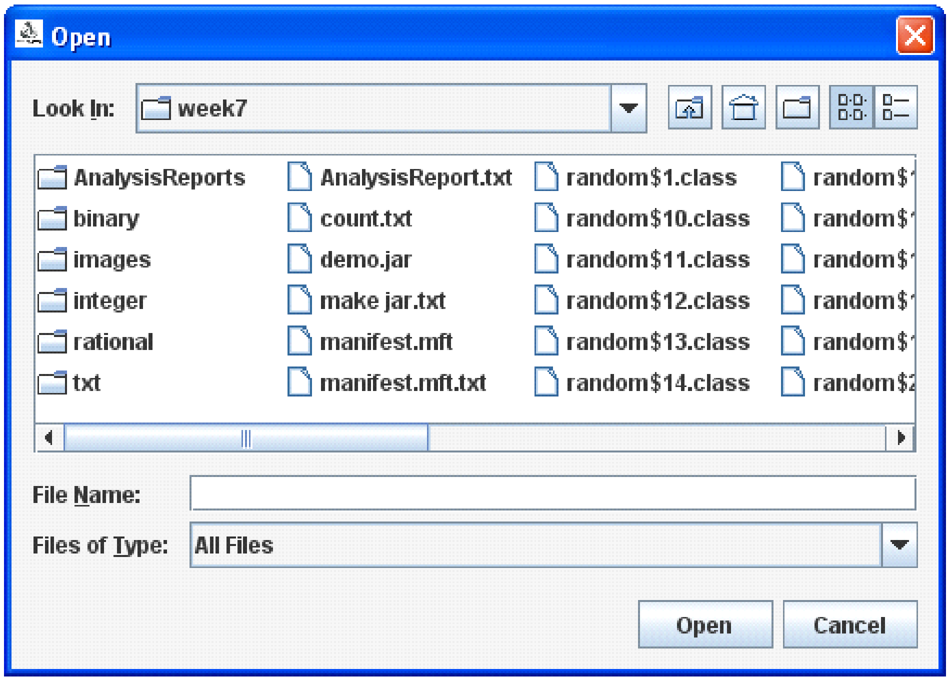


Figure 12: Random Number Testing Main Interface

After authentication, user must get random file to apply test on these. As the user presses "Get File" button, the File Open Dialog Box appears so that the user can get the random file.



Look In: week7

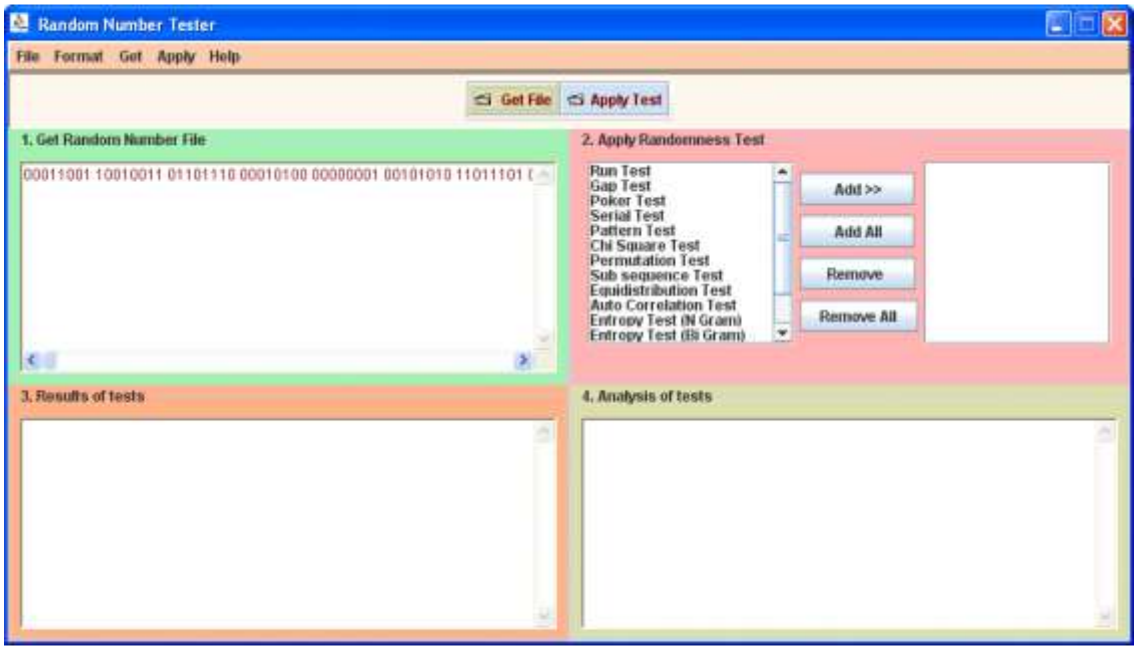
AnalysisReports	AnalysisReport.txt	random$1.class	random$
binary	count.txt	random$10.class	random$
images	demo.jar	random$11.class	random$
integer	make jar.txt	random$12.class	random$
rational	manifest.mft	random$13.class	random$
txt	manifest.mft.txt	random$14.class	random$

File Name:

Files of Type: All Files

Figure 13: Random File Selection Dialogue Box

As the user selects the file, contents of file are appeared in the text box.

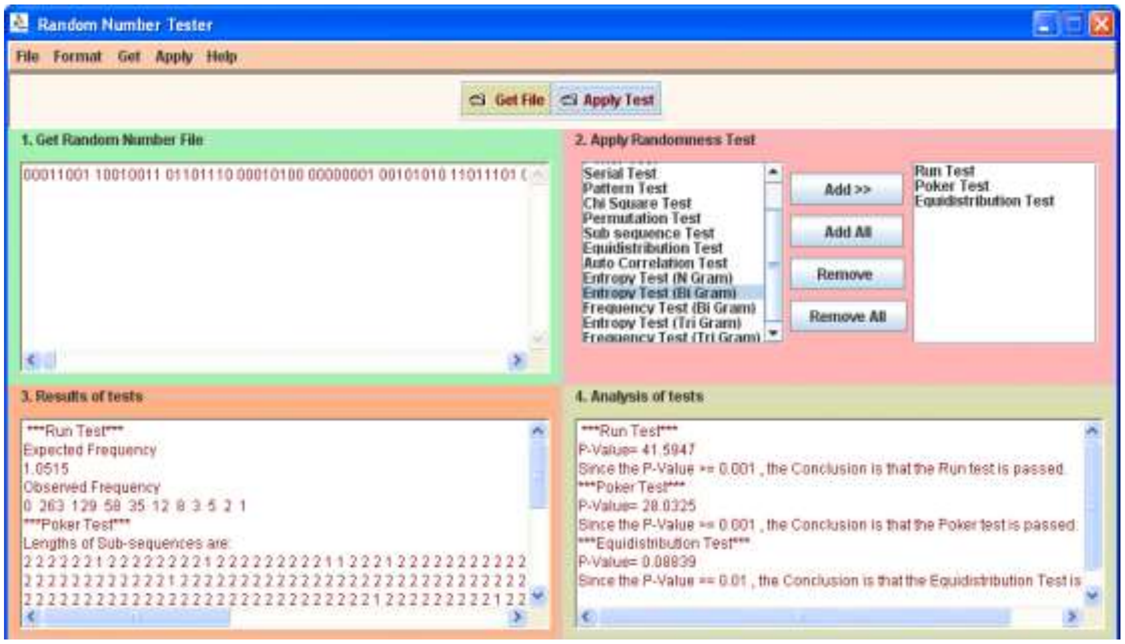


```
00011001 10010011 01101110 00010100 00000001 00101010 11011101
```

Run Test
Gap Test
Poker Test
Serial Test
Pattern Test
Chi Square Test
Permutation Test
Sub sequence Test
Equidistribution Test
Auto Correlation Test
Entropy Test (N Gram)
Entropy Test (Bi Gram)

Figure 14: Random Number File content shown in first Text Area

After getting random sequence from a file, the user must select the tests from the list. He can select one or more tests according to his will. After selection of tests, the user must press “Apply” button to test randomness of sequence selected with specific tests.



```
00011001 10010011 01101110 00010100 00000001 00101010 11011101
```

Serial Test
Pattern Test
Chi Square Test
Permutation Test
Sub sequence Test
Equidistribution Test
Auto Correlation Test
Entropy Test (N Gram)
Entropy Test (Bi Gram)
Frequency Test (Bi Gram)
Entropy Test (Tri Gram)
Frequency Test (Tri Gram)

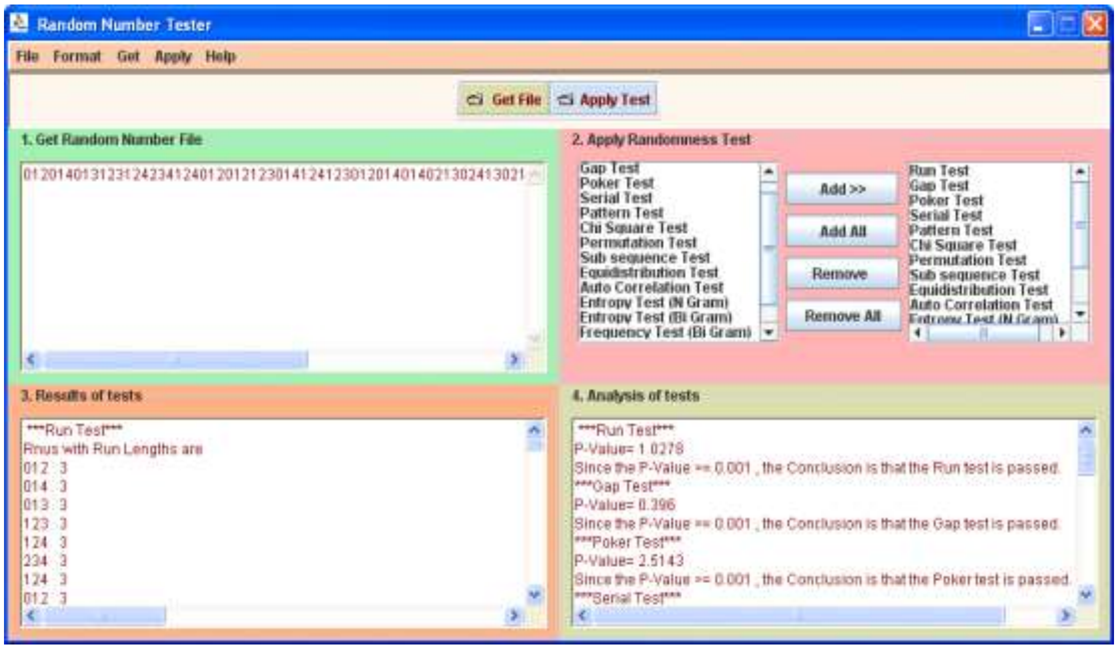
Run Test
Poker Test
Equidistribution Test

```
***Run Test***
Expected Frequency
1.0515
Observed Frequency
0 263 129 58 35 12 8 3 5 2 1
***Poker Test***
Lengths of Sub-sequences are:
2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 2 2
```

```
***Run Test***
P-Value= 41.5947
Since the P-Value >= 0.001 , the Conclusion is that the Run test is passed.
***Poker Test***
P-Value= 28.0325
Since the P-Value >= 0.001 , the Conclusion is that the Poker test is passed.
***Equidistribution Test***
P-Value= 0.08839
Since the P-Value >= 0.01 , the Conclusion is that the Equidistribution Test is
```

Figure 15: Test Results shown after applying Test Algorithms

The user can get different types of sequences like binary, text, decimal and integer. He can also select multiple tests from the list. He can select all tests by pressing button “Add All” button.



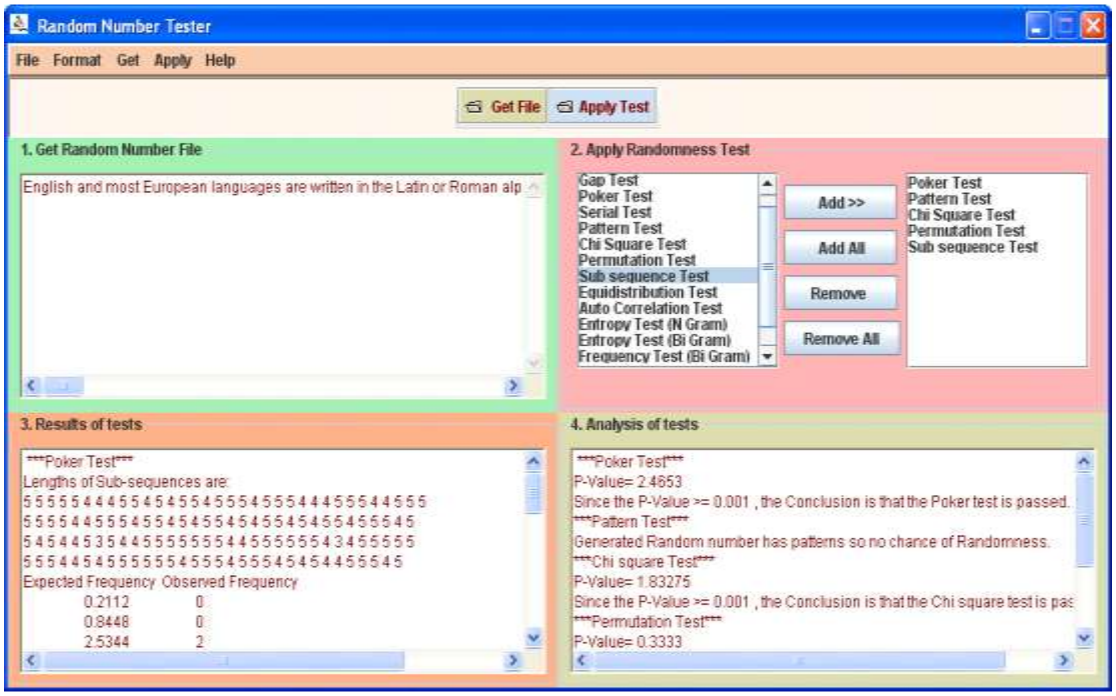
```
0120140131231242341240120121230141241230120140140213024130 21
```

```
***Run Test***
Rnus with Run Lengths are
012  3
014  3
013  3
123  3
124  3
234  3
124  3
012  3
```

```
***Run Test***
P-Value= 1.0278
Since the P-Value >= 0.001 , the Conclusion is that the Run test is passed.
***Gap Test***
P-Value= 0.396
Since the P-Value >= 0.001 , the Conclusion is that the Gap test is passed.
***Poker Test***
P-Value= 2.5143
Since the P-Value >= 0.001 , the Conclusion is that the Poker test is passed.
***Serial Test***
```

Figure 16: Test Algorithm workings with detailed results shown in Text Area 2

When the user applies tests on the sequence, the results of all tests appears in the result and analysis box.



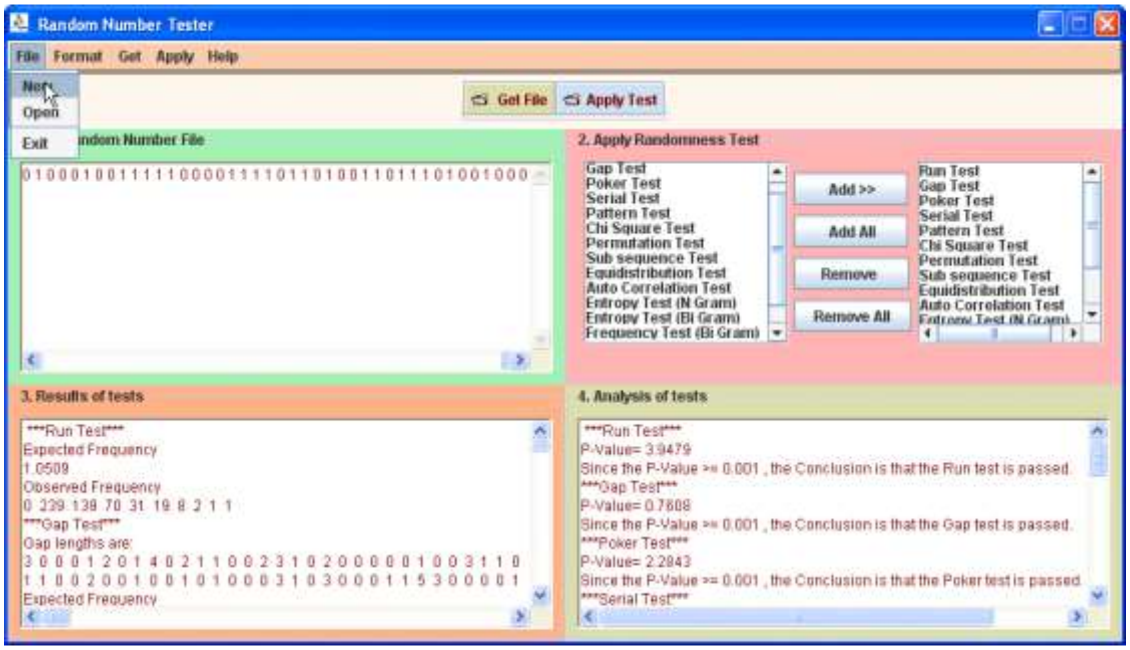
```
English and most European languages are written in the Latin or Roman alp
```

```
***Poker Test***
Lengths of Sub-sequences are:
5 5 5 5 5 4 4 4 5 5 4 5 4 5 5 4 5 5 5 4 5 5 5 4 4 4 5 5 5 4 4 5 5 5
5 5 5 5 4 4 5 5 5 4 5 5 4 5 4 5 5 4 5 4 5 5 4 5 4 5 5 4 5 5 5 4 5
5 4 5 4 4 5 3 5 4 4 5 5 5 5 5 5 5 4 4 5 5 5 5 5 5 4 3 4 5 5 5 5 5
5 5 5 4 4 5 4 5 5 5 5 5 5 4 5 5 5 4 5 5 5 4 5 4 5 4 4 5 5 5 4 5
Expected Frequency  Observed Frequency
          0.2112          0
          0.8448          0
          2.5344          2
```

```
***Poker Test***
P-Value= 2.4653
Since the P-Value >= 0.001 , the Conclusion is that the Poker test is passed.
***Pattern Test***
Generated Random number has patterns so no chance of Randomness.
***Chi square Test***
P-Value= 1.83275
Since the P-Value >= 0.001 , the Conclusion is that the Chi square test is pas
***Permutation Test***
P-Value= 0.3333
```

Figure 17: Status of Test Results in Pass/Fail Format

The user can also open and edit the random sequence. For testing purposes, they can also generate their own sequences.



```
0 1 0 0 0 1 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 0 0
```

```
***Run Test***
Expected Frequency
1.0509
Observed Frequency
0  239  138  70  31  19  8  2  1  1
***Gap Test***
Gap lengths are:
3 0 0 0 1 2 0 1 4 0 2 1 1 0 0 2 3 1 0 2 0 0 0 0 0 1 0 0 3 1 1 0
1 1 0 0 2 0 0 1 0 0 1 0 1 0 0 0 3 1 0 3 0 0 0 1 1 5 3 0 0 0 0 1
Expected Frequency
```

```
***Run Test***
P-Value= 3.9479
Since the P-Value >= 0.001 , the Conclusion is that the Run test is passed.
***Gap Test***
P-Value= 0.7608
Since the P-Value >= 0.001 , the Conclusion is that the Gap test is passed.
***Poker Test***
P-Value= 2.2843
Since the P-Value >= 0.001 , the Conclusion is that the Poker test is passed.
***Serial Test***
```

Figure 18: Using Toolbar for Opening and editing Random Sequences

When the user selects New from Menu, Text File Editor appears to accommodate user for generating and editing random sequences.

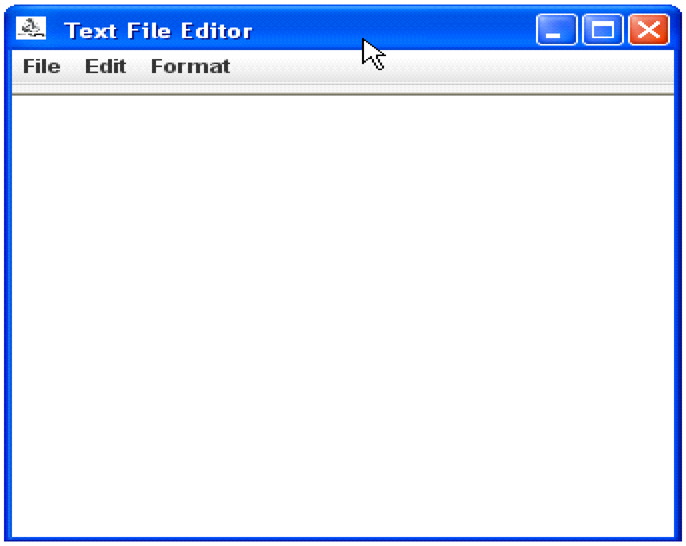
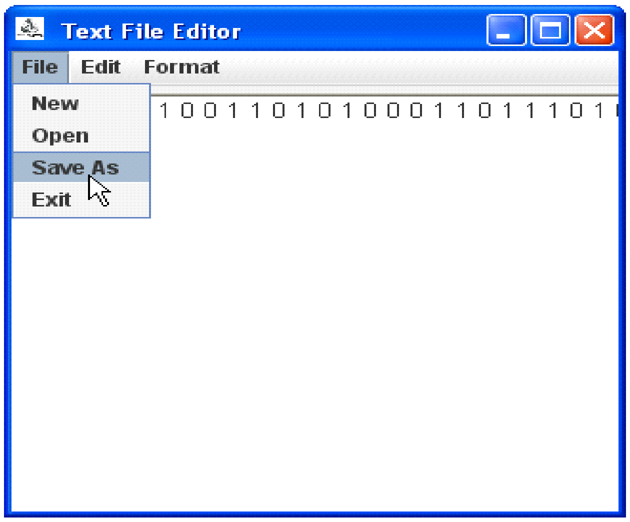


Figure 19: Text File Editor Dialogue Box to edit Random Sequence

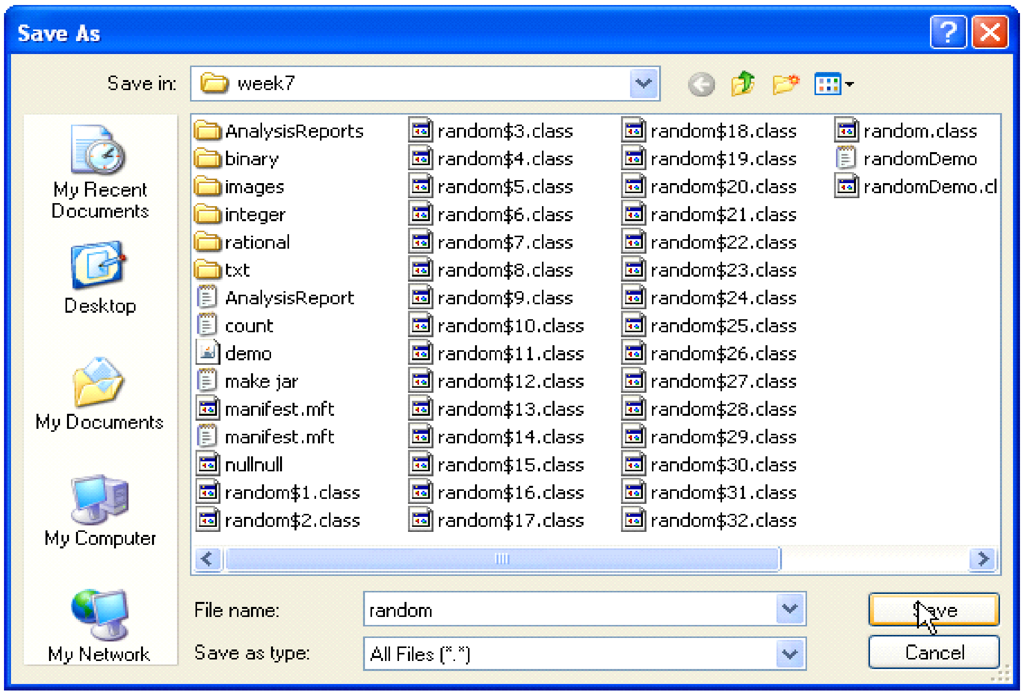
The user can also edit the random sequence and save it. He can also make and Save the sequences at his desired location.



1 0 0 1 1 0 1 0 1 0 0 0 1 1 0 1 1 1 0 1

Figure 20: Save the random number after editing

When the user selects Save As from Menu, Save As Dialog Box appears. The user can save the random sequence file at his desired location.



Save in: week7

AnalysisReports, binary, images, integer, rational, txt, AnalysisReport, count, demo, make jar, manifest.mft, manifest.mft, nullnull, random$1.class, random$2.class, random$3.class, random$4.class, random$5.class, random$6.class, random$7.class, random$8.class, random$9.class, random$10.class, random$11.class, random$12.class, random$13.class, random$14.class, random$15.class, random$16.class, random$17.class, random$18.class, random$19.class, random$20.class, random$21.class, random$22.class, random$23.class, random$24.class, random$25.class, random$26.class, random$27.class, random$28.class, random$29.class, random$30.class, random$31.class, random$32.class, random.class, randomDemo, randomDemo.cl

File name: random

Save as type: All Files (*.*)

Figure 21: File path selection to save the new random sequence

The user can also change the Font Style and Size according to his will.

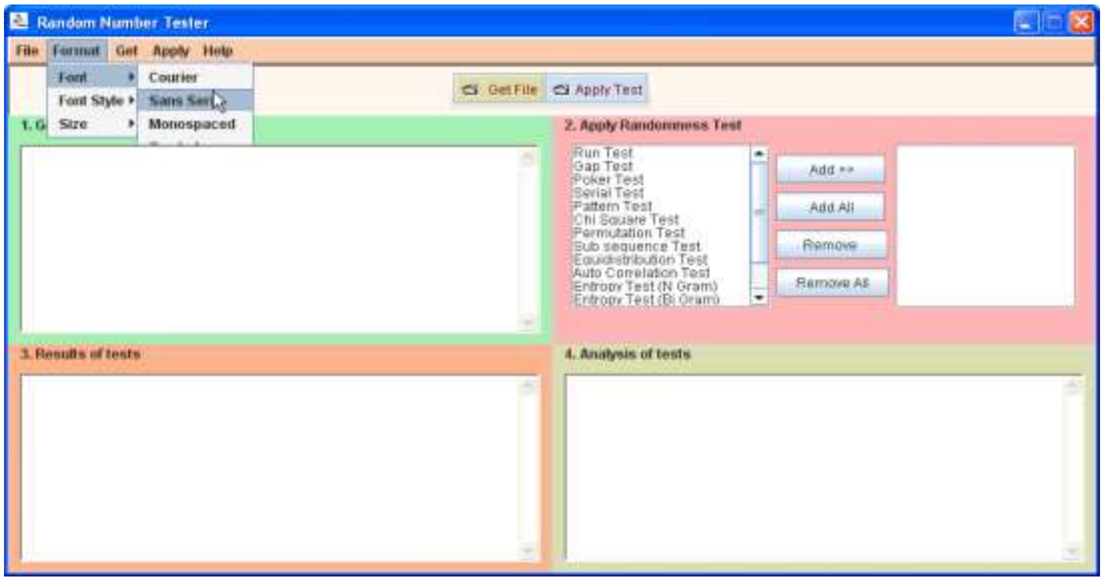
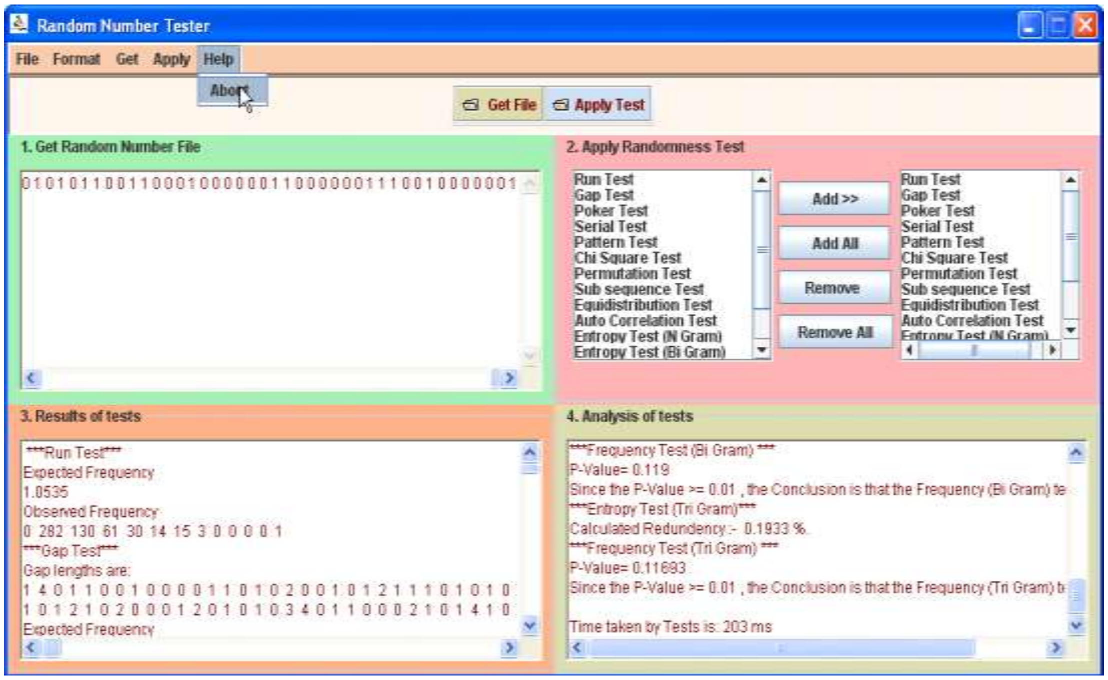


Figure 22: Available Options to change font style and text size

The user can know about details of an application if he wants.



```
0 1 0 1 0 1 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1
```

```
***Run Test***
Expected Frequency
1.0535
Observed Frequency
0  282  130  61  30  14  15  3  0  0  0  0  1
***Gap Test***
Gap lengths are:
1  4  0  1  1  0  0  1  0  0  0  0  1  1  0  1  0  2  0  0  1  0  1  2  1  1  1  0  1  0  1  0
1  0  1  2  1  0  2  0  0  0  1  2  0  1  0  1  0  3  4  0  1  1  0  0  0  2  1  0  1  4  1  0
Expected Frequency
```

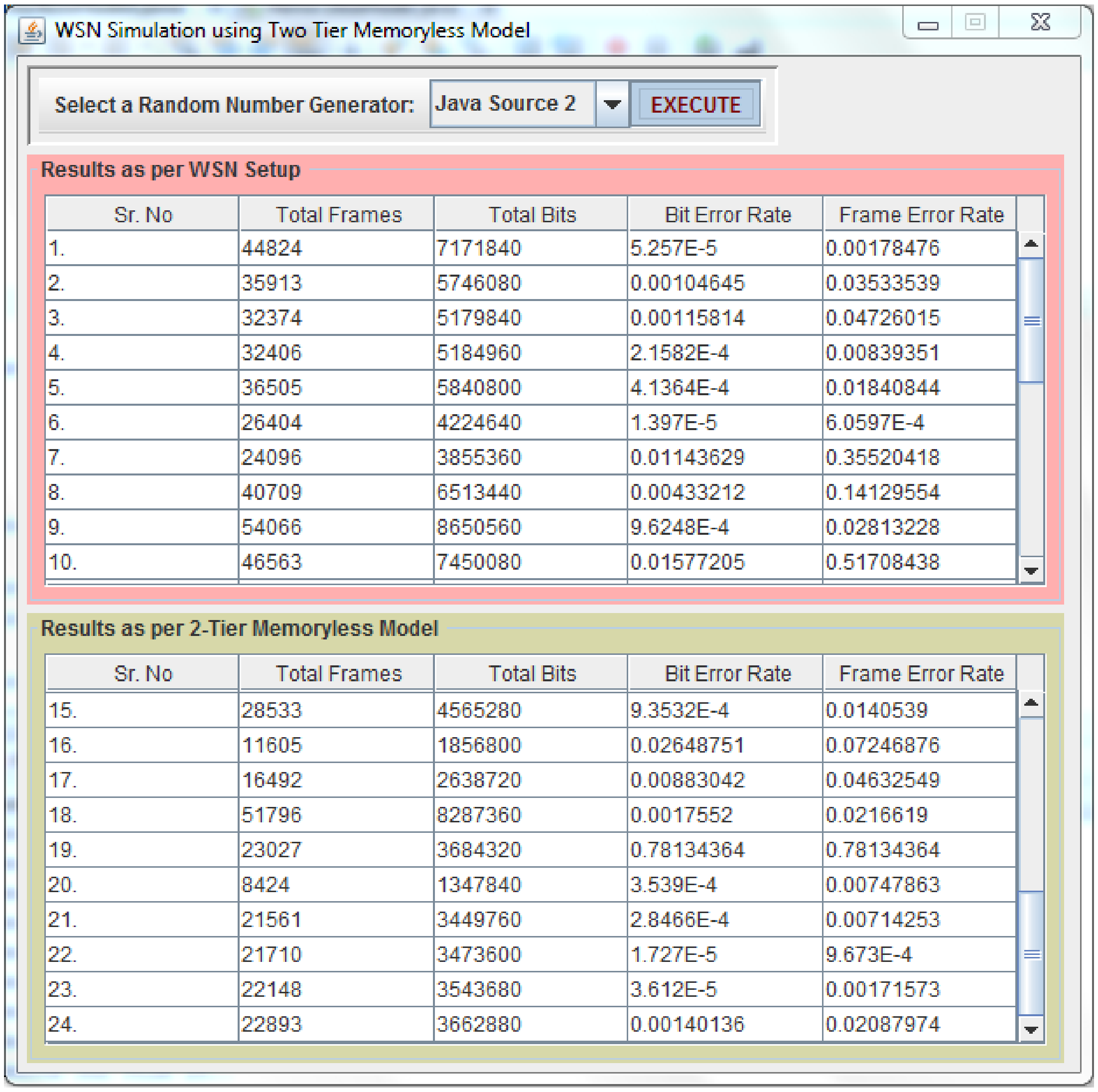
```
***Frequency Test (Bi Gram) ***
P-Value= 0.119
Since the P-Value >= 0.01 , the Conclusion is that the Frequency (Bi Gram) te
***Entropy Test (Tri Gram)***
Calculated Redundency :-  0.1933 %
***Frequency Test (Tri Gram) ***
P-Value= 0.11693
Since the P-Value >= 0.01 , the Conclusion is that the Frequency (Tri Gram) b

Time taken by Tests is: 203 ms
```

Figure 23: About Dialogue Box of an Application

2.Using Memoryless Model Simulation Application

All the options for random number sources are given in combo box. User selects the Random number source and then press “Execute” button to generate the Synthetic traces and R Divergence calculation results.


WSN Simulation using Two Tier Memoryless Model
Select a Random Number Generator:
Java Source 2
EXECUTE

Results as per WSN Setup

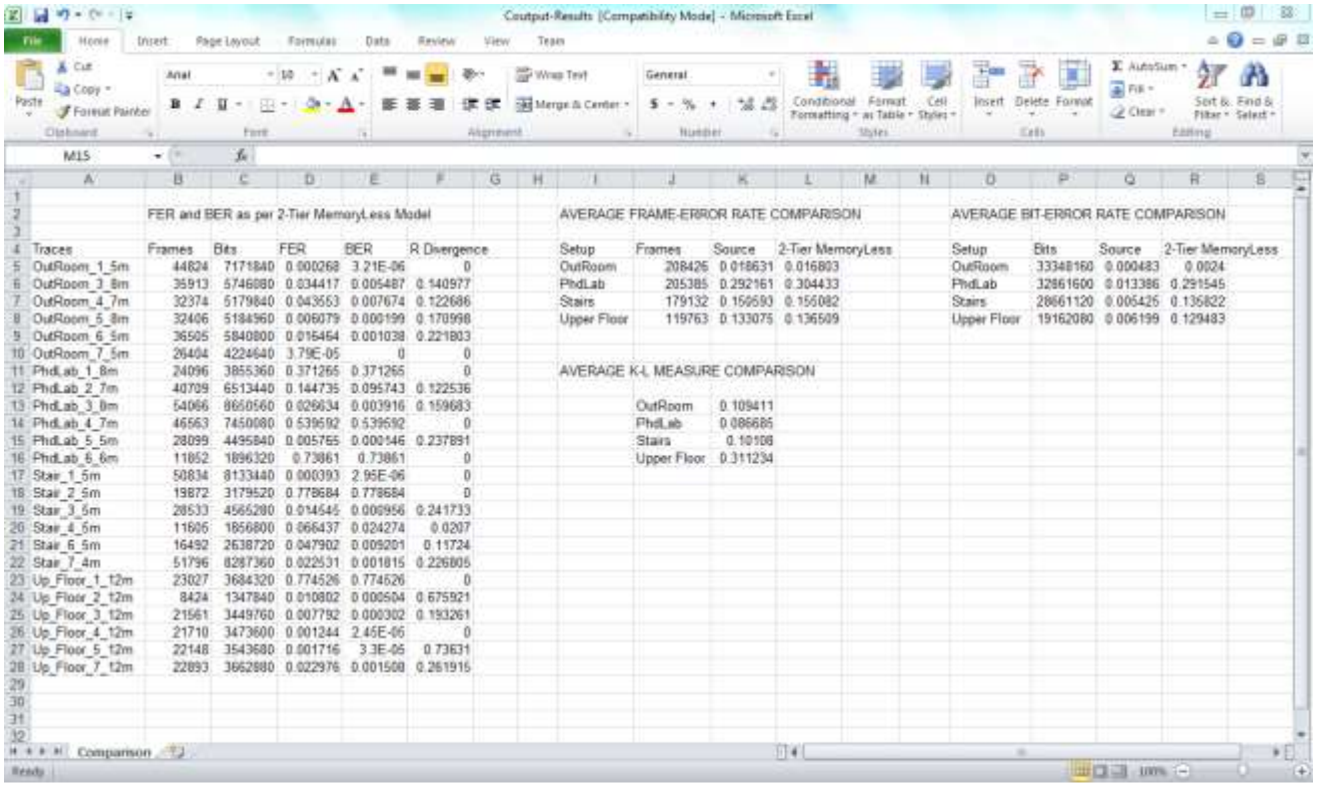
Sr. No	Total Frames	Total Bits	Bit Error Rate	Frame Error Rate
1.	44824	7171840	5.257E-5	0.00178476
2.	35913	5746080	0.00104645	0.03533539
3.	32374	5179840	0.00115814	0.04726015
4.	32406	5184960	2.1582E-4	0.00839351
5.	36505	5840800	4.1364E-4	0.01840844
6.	26404	4224640	1.397E-5	6.0597E-4
7.	24096	3855360	0.01143629	0.35520418
8.	40709	6513440	0.00433212	0.14129554
9.	54066	8650560	9.6248E-4	0.02813228
10.	46563	7450080	0.01577205	0.51708438

Results as per 2-Tier Memoryless Model

Sr. No	Total Frames	Total Bits	Bit Error Rate	Frame Error Rate
15.	28533	4565280	9.3532E-4	0.0140539
16.	11605	1856800	0.02648751	0.07246876
17.	16492	2638720	0.00883042	0.04632549
18.	51796	8287360	0.0017552	0.0216619
19.	23027	3684320	0.78134364	0.78134364
20.	8424	1347840	3.539E-4	0.00747863
21.	21561	3449760	2.8466E-4	0.00714253
22.	21710	3473600	1.727E-5	9.673E-4
23.	22148	3543680	3.612E-5	0.00171573
24.	22893	3662880	0.00140136	0.02087974

Figure 24: WSN Simulation Application using 2-Tier Memoryless Model

In a result, for every selected Random Number Source the following Result sheet implementing Memoryless model is produced that prints FER, BER and R Divergence calculations for all 24 traces.



FER and BER as per 2-Tier MemoryLess Model

Traces	Frames	Bits	FER	BER	R Divergence
OutRoom_1_5m	44824	7171840	0.000268	3.21E-06	0
OutRoom_3_8m	35913	5746080	0.034417	0.005487	0.140977
OutRoom_4_7m	32374	5179840	0.043553	0.007674	0.122686
OutRoom_5_8m	32406	5184960	0.006079	0.000199	0.170998
OutRoom_6_5m	36505	5840800	0.015464	0.001038	0.221803
OutRoom_7_5m	26404	4224640	3.79E-05	0	0
PhdLab_1_8m	24096	3855360	0.371265	0.371265	0
PhdLab_2_7m	40709	6513440	0.144735	0.095743	0.122536
PhdLab_3_8m	54066	8650560	0.026634	0.003916	0.159683
PhdLab_4_7m	46563	7450080	0.539692	0.539692	0
PhdLab_5_5m	28099	4495840	0.005765	0.000546	0.237891
PhdLab_6_6m	11852	1896320	0.73861	0.73861	0
Stair_1_5m	50834	8133440	0.000393	2.95E-06	0
Stair_2_5m	19872	3179520	0.778684	0.778684	0
Stair_3_5m	28533	4565280	0.014545	0.000956	0.241733
Stair_4_5m	11606	1856800	0.066437	0.024274	0.0207
Stair_6_5m	16492	2638720	0.047902	0.009201	0.11724
Stair_7_4m	51796	8287360	0.022531	0.001815	0.226805
Up_Floor_1_12m	23027	3684320	0.774526	0.774526	0
Up_Floor_2_12m	8424	1347840	0.010802	0.000504	0.675921
Up_Floor_3_12m	21561	3449760	0.007792	0.000302	0.193261
Up_Floor_4_12m	21710	3473600	0.001244	2.45E-06	0
Up_Floor_5_12m	22148	3543680	0.001716	3.3E-05	0.73631
Up_Floor_7_12m	22893	3662880	0.022976	0.001508	0.261915

AVERAGE FRAME-ERROR RATE COMPARISON

Setup	Frames	Source	2-Tier MemoryLess
OutRoom	208426	0.018631	0.016803
PhdLab	205385	0.292161	0.304433
Stairs	179132	0.159593	0.155082
Upper Floor	119763	0.133075	0.136509

AVERAGE BIT-ERROR RATE COMPARISON

Setup	Bits	Source	2-Tier MemoryLess
OutRoom	33348160	0.000483	0.0024
PhdLab	32861600	0.013386	0.291545
Stairs	28661120	0.005425	0.135822
Upper Floor	19162080	0.006199	0.129483

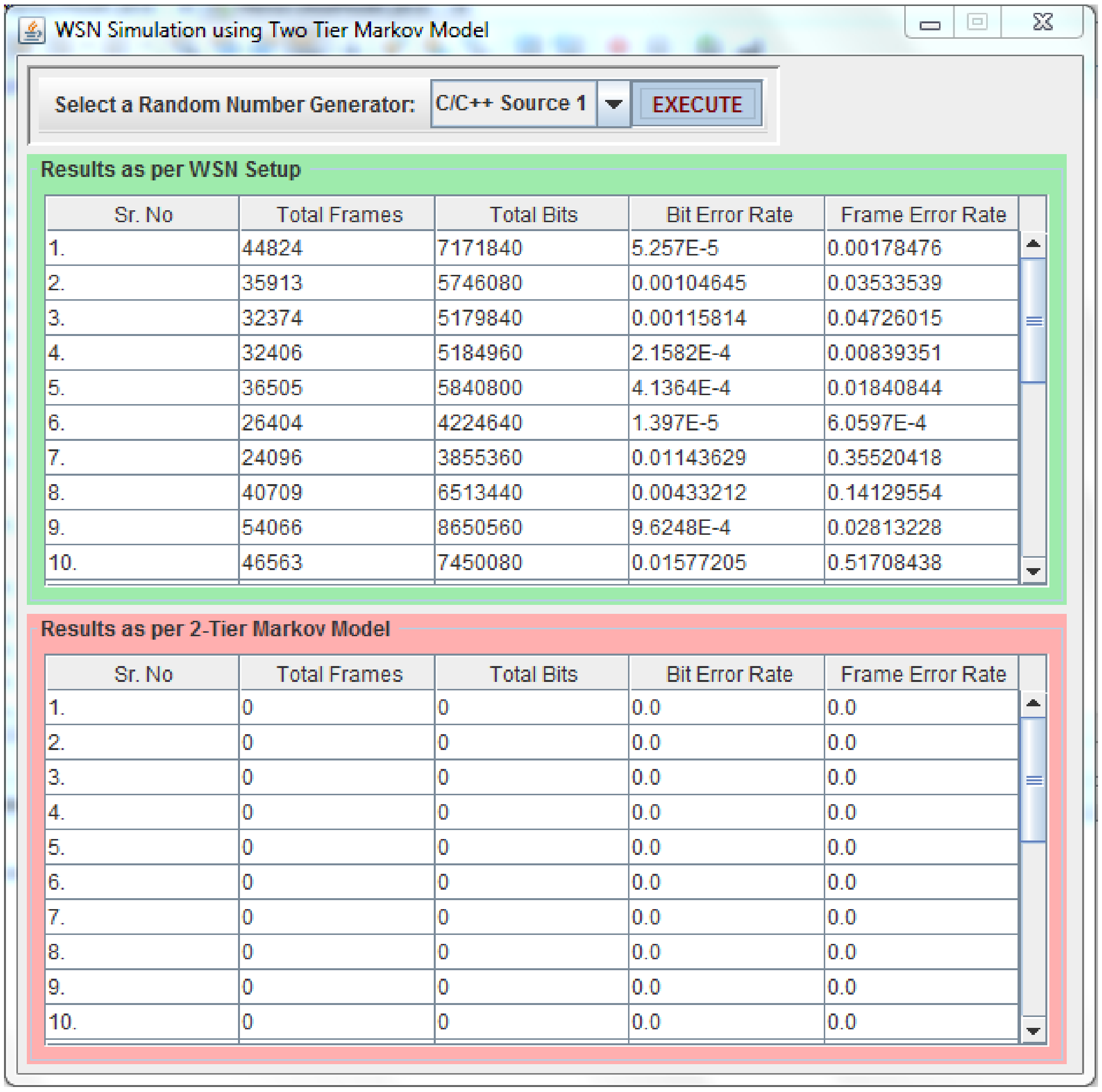
AVERAGE K-L MEASURE COMPARISON

OutRoom	0.109411
PhdLab	0.086686
Stairs	0.10108
Upper Floor	0.311234

Figure 25: FER, BER and R Divergence Measures in Result Sheet of Memoryless Model

3.<u>Using Markov Model Simulation Application</u>

The options for random number sources are presented to user in a combo box. User selects the Random number source and then press “Execute” button to generate the Synthetic traces and R Divergence calculation results.


WSN Simulation using Two Tier Markov Model
Select a Random Number Generator:
C/C++ Source 1
EXECUTE

Results as per WSN Setup

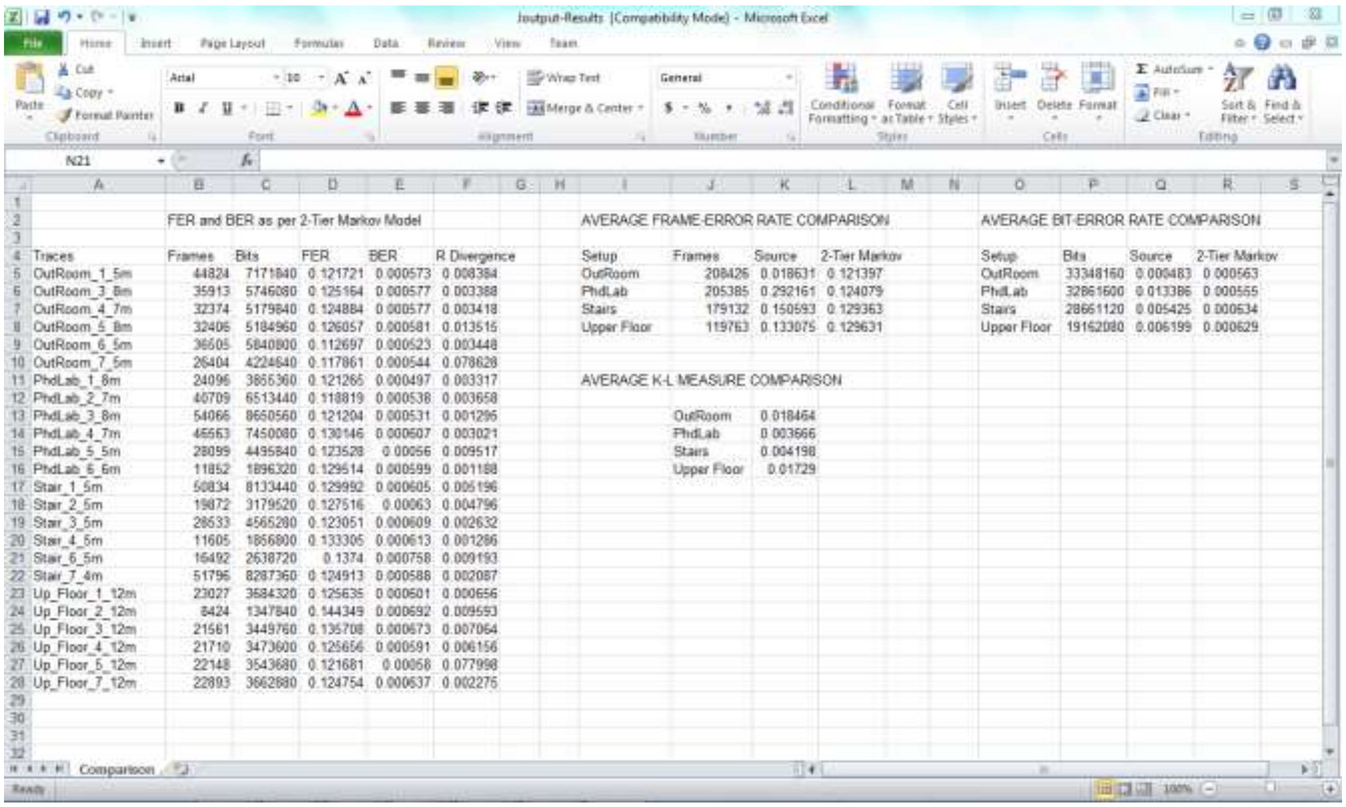
Sr. No	Total Frames	Total Bits	Bit Error Rate	Frame Error Rate
1.	44824	7171840	5.257E-5	0.00178476
2.	35913	5746080	0.00104645	0.03533539
3.	32374	5179840	0.00115814	0.04726015
4.	32406	5184960	2.1582E-4	0.00839351
5.	36505	5840800	4.1364E-4	0.01840844
6.	26404	4224640	1.397E-5	6.0597E-4
7.	24096	3855360	0.01143629	0.35520418
8.	40709	6513440	0.00433212	0.14129554
9.	54066	8650560	9.6248E-4	0.02813228
10.	46563	7450080	0.01577205	0.51708438

Results as per 2-Tier Markov Model

Sr. No	Total Frames	Total Bits	Bit Error Rate	Frame Error Rate
1.	0	0	0.0	0.0
2.	0	0	0.0	0.0
3.	0	0	0.0	0.0
4.	0	0	0.0	0.0
5.	0	0	0.0	0.0
6.	0	0	0.0	0.0
7.	0	0	0.0	0.0
8.	0	0	0.0	0.0
9.	0	0	0.0	0.0
10.	0	0	0.0	0.0

Figure 26: WSN Simulation Application using 2-Tier Markov Model

For every selected Random Number Source the following Result sheet implementing Markov model is created that prints FER, BER and R Divergence calculations for all 24 traces.



FER and BER as per 2-Tier Markov Model

Traces	Frames	Bits	FER	BER	R Divergence
OutRoom_1_5m	44824	7171840	0.121721	0.000573	0.008384
OutRoom_3_8m	35913	5746080	0.125164	0.000577	0.003388
OutRoom_4_7m	32374	5179840	0.124884	0.000577	0.003418
OutRoom_5_8m	32406	5184960	0.126057	0.000581	0.013515
OutRoom_6_5m	36505	5840800	0.112697	0.000523	0.003448
OutRoom_7_5m	26404	4224640	0.117861	0.000544	0.078628
PhdLab_1_8m	24096	3855360	0.121265	0.000497	0.003317
PhdLab_2_7m	40709	6513440	0.118819	0.000538	0.003658
PhdLab_3_8m	54066	8650560	0.121204	0.000531	0.001296
PhdLab_4_7m	46563	7450080	0.130146	0.000607	0.003021
PhdLab_5_5m	28099	4495840	0.123528	0.00056	0.009517
PhdLab_6_6m	11852	1896320	0.129514	0.000599	0.001188
Stair_1_5m	50834	8133440	0.129992	0.000605	0.005196
Stair_2_5m	19872	3179520	0.127516	0.00063	0.004796
Stair_3_5m	28533	4565280	0.123051	0.000609	0.002632
Stair_4_5m	11605	1856800	0.133305	0.000613	0.001286
Stair_6_5m	16492	2638720	0.1374	0.000758	0.009193
Stair_7_4m	51796	8287360	0.124913	0.000588	0.002087
Up_Floor_1_12m	23027	3684320	0.125635	0.000601	0.000656
Up_Floor_2_12m	8424	1347840	0.144349	0.000692	0.009593
Up_Floor_3_12m	21561	3449760	0.135708	0.000673	0.007064
Up_Floor_4_12m	21710	3473600	0.125656	0.000591	0.006156
Up_Floor_5_12m	22148	3543680	0.121681	0.00058	0.077998
Up_Floor_7_12m	22893	3662880	0.124754	0.000637	0.002275

AVERAGE FRAME-ERROR RATE COMPARISON

Setup	Frames	Source	2-Tier Markov
OutRoom	208426	0.018631	0.121397
PhdLab	205385	0.292161	0.124079
Stairs	179132	0.150593	0.129363
Upper Floor	119763	0.133075	0.129631

AVERAGE BIT-ERROR RATE COMPARISON

Setup	Bits	Source	2-Tier Markov
OutRoom	33348160	0.000483	0.000563
PhdLab	32861600	0.013386	0.000555
Stairs	28661120	0.005425	0.000634
Upper Floor	19162080	0.006199	0.000629

AVERAGE K-L MEASURE COMPARISON

OutRoom	0.018464
PhdLab	0.003666
Stairs	0.004198
Upper Floor	0.01729

Figure 27: FER, BER and R Divergence Measures in Result Sheet of Markov Model