2012

Testing Several Types of Random Number Generator

Liang Li

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

TESTING SEVERAL TYPES OF RANDOM NUMBER GENERATOR

By

LIANG LI

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2012

Liang Li defended this Thesis on Jun 28, 2012.

The members of the supervisory committee were:

Dr. Michael Mascagni
Professor Directing Thesis

Dr. Xin Yuan
Committee Member

Dr. Ashok Srinivasan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the Thesis has been approved in accordance with university requirements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The fast increasing length of random number streams, the application of more powerful cores and emerging various Random Number Generators (RNGs) lead to a revolution from traditional RNGs. The authentic RNGs are mainly based on the physical environments, such as sound waves, light photons, etc. They obtain the advantage of unpredictable, and thus, are random with no doubt. However, in practical applications like Cryptography, lottery, and some academic requirements, etc., reproducibility is a very important attribute. People are searching different ways for generating random number streams to simulate the authentic ones. Because being generated not naturally, they are called Pseudo Random Number Generators (PRNGs). To distinguish the quality of these PRNGs, a level to gauge how similar are they as authentic random numbers is essential. Statistical tests are considered as accurate and efficient tests. Along with the fast development of RNGs and PRNGs, it takes a new theme that how to improve the statistical tests. In this article, we are introducing some practical test suites, including DIEHARD, NIST, TestU01, SPRNG, etc., and trying to elaborate how to use them. Meanwhile, I notice similar characteristics, which lead to the idea that tests in one suite could imitate tests in another suite. Then I've composed a combination file that could use all the testing suites. Due to the high comprehensiveness, TestU01 is used as mother board of this file. Afterward, I'll trace the newest trends of PRNGs, to detect the principle of counter-based PRNG – Random123, and show the testing results from both BigCrush test in TestU01 and combination file. Also, hardware RNGs, which generate authentic random numbers, are also tested to show the quality of statistical tests. Last but not least, parallel tests, whose advantages are more significant, especially when random number streams are getting bigger, are also discussed, and so are software and hardware support.

Keywords: RNG, Statistical test, test suites, NIST, SPRNG, TestU01, Random123, GRANG.

CHAPTER ONE

BACKGROUND

RNGs and PRNGs

Randomness is a common phenomenon. Simply as flipping a coin, randomness works. Although we are living in a random world, it's not easy to trace the random events which happen around us. For example, we know the lottery numbers are within a range from 1 to 60, but don't know which numbers are chosen. People are trying to simulate the randomness by sequences of numbers. Actually, lottery is a general application of random number, however, it's used more widely than this. In areas that people want to simulate random, such as molecular movement on Physics, random sample selection on Statistics, forecasting weather, generating secret keys, etc., huge amount of random numbers are needed. Then, people have to think about how to collect the random numbers. An intuitive idea is based on a physical phenomena where random happens, denote results as numbers and record them. Just imagine toss a coin, and denote head as 1 whereas tail as 0. Some more complex methods are utilized for continuously generating random number sequences, such as entropy distillation process. We could name the non-deterministic sources (i.e., the entropy source), along with some processing function (i.e., the entropy distillation process) as an RNG [1].

Note that RNGs are based on non-deterministic sources, the random numbers they generated should be unpredictable. However, for some applications, we need predictable results. In early 20th century, people recorded a sequence of random numbers into a book. Whenever they need a segment of random numbers, a random page is open and a serial of numbers are chosen. This is the earliest method, which appears to be very naive nowadays. Along with the increasing quality and quantity requirement, people gradually realize that generating random numbers can not follow the pace if only by human powers. Then, by means of computational and physical devices, according to some particular algorithms, endless numbers could be generated. At this point, people don't need to search the book for numbers, but just need to design the algorithms

for the devices that could generate numbers which seem to be random. The composition of a particular device and the algorithm executing on it is a PRNG (Pseudo RNG).

All the algorithms could not start the PRNG without any initial input (or called seeds). If we just input seeds which have particular pattern, the generating result will also be following some particular pattern, which lost the randomness. Simply taking recursive generators (abbr. MRGs) as an example:

$$x_n = (x_{n-1} + \ldots + x_{n-k}) \bmod m$$

If the seeds ($x_{n-1} \ldots x_{n-k}$) are increasing arithmetic series, so are every $m/(x_{n-1} - x_{n-2})$ number sequences. Hence, seeds are required to be randomly chosen. As we mentioned in the beginning of this article, the output of RNG is random, then they could be used as seeds of PRNG. Actually we do this way in applications.

RNG tests

Now we know how to generate random numbers, either by RNGs or by PRNGs. However, they are just kind of simulating randomness. It's very hard for us to determine if they have the attribute as true random numbers, especially for PRNG and when the number sequence is tremendous long. We need some evidences which could persuade ourselves, in another word, before they are putting into applications, we need to do some tests.

In a few cases, true random numbers appear to be non-random in some segments, whereas some pseudo random number sequences appear more random than true random numbers. They are unavoidable. When these cases happen, tester will mislead us to an opposite answer. Hence, a number sequence passes one test only shares us confidence that it is random, but could not guarantee. Generally speaking, a number sequence passes more tests, we can believe more it's random. Meanwhile, the test method is very important. An improper test could not increase any confidence. Due to its strong theoretical and practical support, statistical tests are commonly applied.

The early RNG tests could be tracked back to 18th century. In 1777, Georges-Louis Leclerc, Comte de Buffon proposed a famous problem: Suppose we have a paper with parallel lines on it, each the same width with another neighbor one, and we drop a needle whose length is less than the width between lines, onto the paper (Figure 1). What is the probability that the needle will lie across a line?

He conceived that the probability p could be roughly calculated by equation:

$$p=2l/(\pi t)$$

Where l is the length of needle, and t is the width between two neighbor parallel lines.

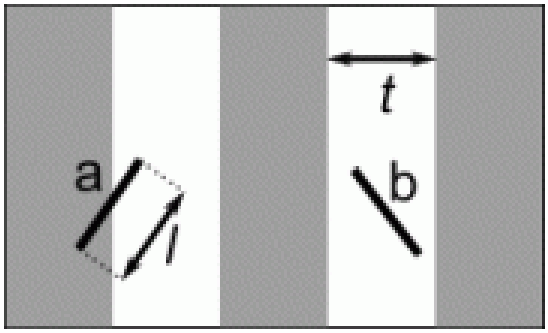

Figure 1

For verifying his idea, he invited the witnesses to toss the needles one by one, and record if the needle cross a line or not. The length of the needles l is set as exactly half of the width t. After tossing 2212 times, he pronounced that the crosses happen for 704 times. 2212 divided by 704 was 3.142, which is very close to π.

The principle of this test is not complex. Let's call the middle point of the needle is M. Suppose the distance between M and the its closest line is x, and the acute angle between needle and parallel lines is φ (Figure 2). Then where a needle is located after random tossing could be denoted by (x , φ).

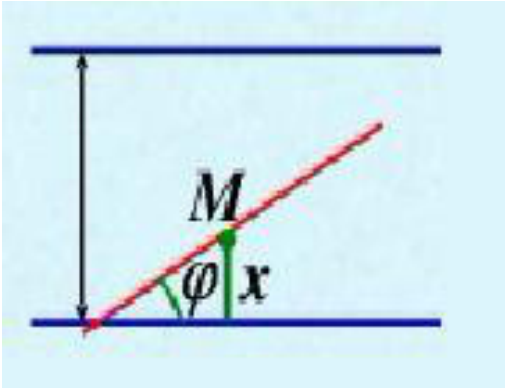

Figure 2

The uniform probability density function of x between 0 and $t/2$ is:

$$\begin{cases} \frac{2}{t} & : \quad 0 \leq x \leq \frac{t}{2} \\ 0 & : \quad \text{elsewhere} \end{cases}$$

The uniform probability density function of φ between 0 and $\pi/2$ is:

$$\begin{cases} \frac{2}{\pi} & : \quad 0 \leq \varphi \leq \frac{\pi}{2} \\ 0 & : \quad \text{elsewhere} \end{cases}$$

Obviously, the x and φ are independent, so the joint probability density function is the product:

$$\begin{cases} \frac{4}{t\pi} & : \quad 0 \leq x \leq \frac{t}{2}, 0 \leq \varphi \leq \frac{\pi}{2} \\ 0 & : \quad \text{elsewhere} \end{cases}$$

The needle crosses a line if:

$$x \leq \frac{l}{2} \sin \varphi$$

To get the probability that the needle will cross a line, we only need to integrating the joint probability density function gives:

$$P = \int_{\varphi=0}^{\frac{\pi}{2}} \int_{x=0}^{\frac{l}{2} \sin \varphi} \frac{4}{t\pi} dx d\varphi = \frac{2l}{t\pi}$$

Like what Buffon did, if we set l to be the half length as t, the probability should be closed to $1/\pi$.

This test was duplicated by different people for many times. A very famous approximation of π, *355/113*, is concluded by Italian mathematician Mario Lazzarini in 1901, by following the same experiment [22]. The length of needle is not limited within the width between two neighbor lines, either. A couple of years ago, people applied this experiment into higher dimensions for particle movement research [23], another level of Monte Carlo simulation.

Originally, this experiment was used to calculate the approximation of π. It's also a very good example to simulate random number streams and testers. Once tossing, record cross as 1 and not cross as 0. Repeat it for *n* times, we get a *n* bit stream. Then we could use the relationship between this stream and the estimator π to estimate if it is random. This is a prototype of statistical tests.

A statistical test provides a mechanism for making quantitative decisions about a process or processes. The intent is to determine whether there is enough evidence to "reject" a conjecture or hypothesis about the process [2]. The conjecture is called the null hypothesis (denoted as H_0). During the experiment, two kind of errors may happen: originally the H_0 is correct, but we reject it(which is called the error of the first type), and originally the H_0 is incorrect, but we accept it(which is called the error of the second type). We are more willing to reduce the second one than the first, so the principle is try to reduce the error of the second type while keeping the error of the first type in an acceptable level (Table 1).

Table 1 H_0 hypothesis

Assumption of H_0	Conclusion	
	Accept H_0	Reject H_0
H_0 is true	No Error	Error of 1st Type
H_0 is false	Error of 2nd Type	No Error

Again, let's take flipping a coin for example. Suppose in an experiment, we are required to flip for 100 times, and record every face as sample. It' estimated that both head and tail may appear for roughly equal times, then we could predefine the H_0 as "this coin is not biased towards heads(or tails)", and set 1% as threshold. In most cases, H_0 will be accepted, however, there still exist a chance, for instance, that the heads appears 90 times or more while tail appears 10 times or less. Then according to probability formula:

$$Prob = \frac{1}{2^{100}} \sum_{i=0}^{10} C(100, i)$$

the result is less that 1%, then H_0 is rejected. In this situation, the error of the first type happens. Similar instance but change H_0 into its opposite proposition -- "this coin is biased towards

heads(or tails)" may leads to be rejected in most cases, and may cause the error of the second type as well.

In different statistical tests, respective estimators are used. We will discuss them in second part of this paper.

CHAPTER TWO

STATISTACAL TESTS

As discussed, a number sequence passes a statistical test cannot guarantee that it is random, but builds more confidence. People usually pack several RNG testers which test in different ways into a battery, then use the test result as a whole to evaluate the randomness. Here I'll cite four random number test suites who are widely applied. Detailed explanations of distinct statistical test methods are provided. I'm also trying to show the connection between different test suites.

Testing theory

In statistical hypothesis testing, *p-value* is a probability. It is used to estimate the difference between the testing sample and other tested samples, when assuming the null hypothesis H_0 is true. The significant level (usually denoted as α) is often 5%, which means the extraordinary any result that is within the most extreme 5% of all possible results under H_0. When p-value is less than α, H_0 is traditionally rejected, and the result is said to be statistically significant. Especially, when *p-value* is less that 1%, it's said to be very statistically significant.

There are multiple methods for calculating *p-value*. In RNG testing, it could be generally separated into two steps. The first step is to choose a statistical estimator for a sample. Then go to the second step, calculate the *p-value* by calling special functions.

For discrete distributions, due to most of the existed samples follow uniform distribution, Chi-square (χ^2) is most commonly used as estimator in the test suites. Also, normal distribution, Poisson distribution and binary distribution are followed in some existed samples, then respective estimators are adopted. These could be found in any entry-level Statistic books, so it is not necessary to narrate more. For continuous distributions, Kolmogorov-Smirnov test (K-S test) is an advanced option. It tries to determine if two datasets differ significantly.

Before test, it requires the cumulative distribution function of the testing sample, and decides which distribution the sample follows, so it has the advantage of making no assumption about the distribution of data.

Cumulative Distribution Function

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-u^2} du$$

Next, compute the maximum distance between this distribution and the exact tested distribution D (Figure 1), and multiple D with the square root of number of samples. Here absolute distance is taken, so D should be positive.

After obtaining the estimator results, calculating p-value could be executed by calling respective functions.

Complementary Error Function

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} e^{-u^2} du$$

Gamma Function

$$\Gamma(z) = \int_{0}^{\infty} t^{z-1} e^{-t} dt$$

Incomplete Gamma Function

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_{0}^{x} e^{-t} t^{a-1} dt$$

complementary Incomplete Gamma Function (*igamc*)

$$Q(a,x) \equiv 1 - P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_{x}^{\infty} e^{-t} t^{a-1} dt$$

Depending on the values of its parameters *a* and *x*, the incomplete gamma function may be approximated using either a continued fraction development or a series development. The specific functions that are utilized are *igamc* (for the complementary incomplete gamma function) and *lgam* (for the logarithmic gamma function). In last segment of this chapter, how to use the estimator and the special functions to calculate p-value is demonstrated.

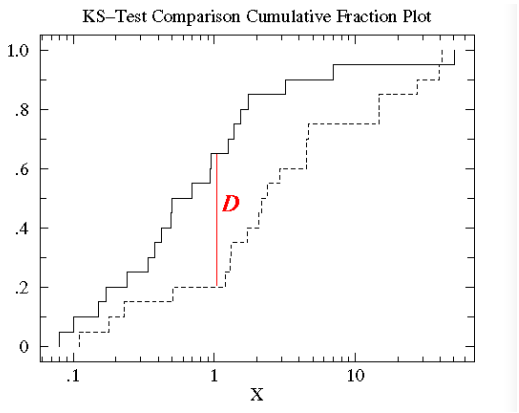

Figure 3

Commonly used test suites

NIST/FIPS

National Institute of Standards and Technology (NIST), through its Information Technology Laboratory (ITL), provides leadership, technical guidance, and coordination of government efforts in the development of standards and guidelines for The Federal Information Processing Standards Publication Series (FIPS PUB Series) [4]. In FIPS PUB 140-2, it pronounces security requirements for cryptographic modules, in which it provides four increasing qualitative levels of security. The standard stipulates that an approved RNG shall be used for generation of cryptographic keys used by an approved security function. For ensuring a module is functioning properly, it is required to pass some particular tests within which statistical tests are contained. For cryptographic applications, ITL at the NIST also prompts a statistical test suite for RNGs and PRNGs.

There are four statistical tests in FIPS statistical test, include Monobit test, Poker test, Runs test and the test for the longest run of 1s in a block. All of them has been included in different test suites, so I'll not repeat explanation here. The NIST Test Suite is a statistical package consisting of 15 tests that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators, including the monobit test, monobit test within a block, the runs test, test for the longest-run-of-ones in a block, the binary matrix rank test, the spectra test, the non-overlapping and overlapping template matching test, the linear complexity test, Maurer's "universal

statistical" test, the serial test, the approximate entropy test, the cumulative sums test, the random excursions test and the random excursions variant test. The tests are designed from various aspects, and verify the randomness from distinct directions, consequently. Next, I'll make a high level description for each test.

Before expanding the narrative, I'd like to introduce something in common. In this suite, the length of testing sequence is usually between 10^3 and 10^7. In order to facilitate the description, I'll only show how a test runs in a relatively shorter length. The tests are applying the same rules to the rest of the sequence. In addition, most of the tests are using standard normal and chi-square (χ^2) as reference distribution. The standard normal distribution is used to compare the value of the test statistic obtained from the RNG with the expected value of the statistic under the assumption of randomness. It has the form:

$$z = \frac{(x - \mu)}{\sigma}$$

where *x* is the sample test value, μ is the expected value and σ^2 is the variance. The χ^2 distribution is used to compare the goodness-of-fit of the observed frequencies of a sample measure to the corresponding expected frequencies of hypothesized distribution. The test statistic is of the form:

$$\chi^2 = \sum \left(\frac{(o_i - e_i)^2}{e_i} \right)$$

where o_i and e_i are the observed and expected frequencies of occurrence of the measure, respectively.

Unless specified, the length of instance sequence is *n*, and the tested stream which is composed by *0*s and *1*s could be denoted as:

$$\varepsilon = \varepsilon_1, \varepsilon_2, \varepsilon_3, \cdots, \varepsilon_n$$

Additional information may be found at http://www.itl.nist.gov/div893/staff/soto/jshome.html.

- Monibit test

This test focusing on the *0*s or *1*s in stream ε. As we could easily imagine, like the game of tolling coin, that both sides should be roughly equal. This test access the closeness of the fraction of *1*s to $\frac{1}{2}$. It's also called Frequency test, and is also employed in FIPS test suite. This is a very basic test. Although there is no requirement for testing sequences, Monobit test is suggested to

be executed before implementing other tests, because if the stream fails this test, it could be expected that there will be a big trouble when executing other tests.

In this test, S_{obs} is used as test statistic:

$$S_{obs} = \frac{\sum_{i=1}^{n} X_i}{\sqrt{n}}$$

where $X_i = 2\varepsilon_i - 1$. The reference distribution is half normal, which means we are using $\frac{|S_{obs}|}{\sqrt{2}}$ instead of $\frac{S_{obs}}{\sqrt{2}}$ when the latter is normal distribution. If the tested stream contains too many *1*s of *0*s, S_{obs} will be forward away from the origin of number axis, which will lead the

$$\textit{P-value}=\textbf{\textit{erfc}}\left(\frac{S_{obs}}{\sqrt{2}}\right)$$

to be out of range (0.01, 0.99), then fail the test.

- Frequency Test within a Block

In this test, the tested stream is divided into multiple blocks, each of which has length *M*. The rest part, if exist, is discarded. So there will be totally $N = \left\lfloor \frac{n}{M} \right\rfloor$ non-overlapping blocks. If the length of block is equal to the length of stream, it is degenerated to Monobit test. Then the test calculates the proportion of *1*s in each block using the equation:

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)M+j}}{M}$$

After collecting the different proportions, we could compute the χ^2 statistic:

$$\chi^2(obs) = 4M \sum_{i=1}^{N} \left(\pi_i - \frac{1}{2}\right)^2$$

The $\chi^2(obs)$ will be applied into Incomplete Gamma Function with the number of blocks *N*, and calculate the

$$\textit{P-value}=\textbf{\textit{igamc}}\left(\frac{N}{2}, \frac{\chi^2(obs)}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Runs Test

In a stream, if the following bit or bits are the same with the current bit, we call the identical segment a 'Run'. In another word, a 'Run' is a segment of at least two identical bits with both sides are neighbor with a different bit. For example, if

$$\varepsilon = 1110010110$$

then $\varepsilon_1\varepsilon_2\varepsilon_3 = 111$, $\varepsilon_4\varepsilon_5 = 00$, $\varepsilon_8\varepsilon_9 = 11$ are all the three Runs in this stream. This test is caring about the frequency of runs in a stream, and inspecting if the bits in a tested stream is changing too fast or too slow comparing to truly random streams.

Before executing Runs test, we need to guarantee the necessity of going on. If the stream has too many *0*s or *1*s, it will be treated as non-random directly, the *P-value* will set to 0.000 and no more steps are needed. This pretest is less complex than Monobit test, and the result has no reference value to other tests.

First, a threshold value τ is pre-defined as $\tau = \frac{2}{\sqrt{n}}$. Then calculate the proportion of *1*s in the stream:

$$\pi = \frac{\sum_{j=1}^{n} \varepsilon_j}{n}$$

Next we need to check if $\left|\pi - \frac{1}{2}\right| < \tau$. If the inequality doesn't stand, the stream fails the pretest, and fails the Runs test consequently.

If the stream passes the pretest, the total number of runs should be counted. Here we denote it by $V_n(obs)$:

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$$

where $r(k) = 0$ if $\varepsilon_k = \varepsilon_{k+1}$, and $r(k) = 1$ if $\varepsilon_k \neq \varepsilon_{k+1}$. Next we could use the Complementary Error Function to calculate *P-value*:

$$\textit{P-value} = \boldsymbol{erfc}\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Test for the Longest Run of Ones in a Block

This test is mainly focus on the longest runs of *1s* in *M*-bits blocks. For each block with length of *M*, the test will execute more than 10 times, then record the longest runs of *1*s and use χ^2 statistic to evaluate the *P-value*, so the length of the stream should be much bigger than block size. The purpose of this test is to determine whether the length of the longest runs of *1*s within the tested stream is consistent with the length of that would be expected in a truly random stream. Under normal circumstance, the number of *1*s and *0*s are roughly equal. The irregular length of

runs of *1*s always implies the irregular length of runs of *0*s. So executing the same test on runs of *0*s is unnecessary.

The first step is same with the Frequency Test within A Block: dividing the stream into $N = \left\lfloor \frac{n}{M} \right\rfloor$ non-overlapping blocks with length of *M*. The test code has preset 3 length values for test, and provides the minimum lengths of streams when using the values respectively:

$$M = 8 \quad \rightarrow \quad n \geq 128$$

$$M = 128 \quad \rightarrow \quad n \geq 6272$$

$$M = 10{,}000 \quad \rightarrow \quad n \geq 750{,}000$$

The next step is recording the frequencies of the longest runs of *1*s in each block. For clearly illustration, the record is shown as table 2:

Table 2 Record of longest runs of 1s

v_i	*M = 8*	*M = 128*	*M = 10,000*
v_0	≤ 1	≤ 4	≤ 10
v_1	2	5	11
v_2	3	6	12
v_3	≥ 4	7	13
v_4		8	14
v_5		≥ 9	15
v_6			≥ 16
N	16	49	75
K	3	5	6

Where the v_i is the frequency of each length, *K* indicate how many possible frequencies there are, and *N* denotes how many blocks is the stream divided into. After collecting all of these parameters, the test could start calculating the χ^2 statistic:

$$\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

Where the π_i are theoretical probability. They have been preset. People could look up [26] for more information.

Next we could use the Complementary incomplete gamma function to calculate *P-value*:

$$P\text{-}value = \boldsymbol{igamc}\left(\frac{\boldsymbol{K}}{\boldsymbol{2}}, \frac{\chi^2(obs)}{\boldsymbol{2}}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Binary Matrix Rank Test

This test will divide the stream into blocks whose length is determined by the sub-matrices, then calculating the rank of these matrices to evaluating the linear correlation. A truly random number stream should have only a few linear correlations between the sub-sequences. If too many sub-matrices have a small rank, the whole stream is considered to be linear independence, and will fail this test.

First, the size of the sub-matrices should be decided. The test gives *32×32* as default. Users could set the number of rows *M* and number of columns *Q* with preference. However, the approximation E_1, E_2, E_3 ($M \geq 3$) in the following procedure should also be changed.

The tested stream is divided into $N = \left\lfloor \frac{n}{M \times Q} \right\rfloor$ sub-sequences and form *N* sub-matrices. The first line of each matrix is the first *Q* bits set in each sub-sequence, the second line is the second *Q* bits set in each sub-sequence ⋯ Then calculating the rank of every sub-matrix, and record the result:

$$F_M = full\ rank\ (the\ number\ of\ matrices\ with\ rank\ is\ M)$$

$$F_{M\text{-}1} = full\ rank - 1\ (the\ number\ of\ matrices\ with\ rank\ is\ M\text{-}1)$$

$$F_r = N - F_M - F_{M\text{-}1}\ (the\ number\ of\ matrices\ remaining)$$

We don't need to go through into every rank. The rank less than *M-1* could be treated as high correlation and classified into F_r. To every class, there is a fixed approximation. The approximations vary according to the *(M , Q)*. By default, which means *(M , Q) = (32 , 32), (E1, E2, E3) = (0.2888, 0.5776, 0.1336)*. Because the matrices are classified into 3 groups, the result is a χ^2 distributation:

$$\chi^2(obs) = \frac{(F_M - E_1 N)^2}{E_1 N} + \frac{(F_{M-1} - E_2 N)^2}{E_2 N} + \frac{(F_r - E_3 N)^2}{E_3 N}$$

Next we could use the Complementary incomplete gamma function to calculate *P-value*:

$$P\text{-}value = \boldsymbol{igamc}\left(1, \frac{\chi^2(obs)}{2}\right) = e^{\frac{-\chi^2(obs)}{2}}$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Discrete Fourier Transform Test

This test is also called Spectral test. Discrete Fourier Transform (DFT) is a method always applied in Fourier analysis. It's used to transform one function which is often in time domain into another function which is frequency domain representation. The DFT requires an input function that is discrete. In this test, we will mainly focus on the peak heights in the DFT of the sequence. Through observing whether the number of peaks exceeding the 95% threshold is significantly different than 5%, we could detect the periodic features in the tested stream and evaluate how big the deviation between the tested stream and the assumption of randomness. The test statistic being adopted is *d*: the normalized difference between the observed and the expected number of frequency components that are beyond the 95% threshold.

In the beginning, the stream needs to be substituted to streams with *1*s and *-1*s by function $X_i = 2\varepsilon_i - 1$.The same trick has been used in Monobit test. Then implement DFT to the new stream:

$$f_j = \sum_{k=1}^{n} X_k exp\left(\frac{2\pi_i(k-1)j}{n}\right)$$

Where

$$exp\left(\frac{2\pi_i(k-1)j}{n}\right) = \cos\left(\frac{2\pi kj}{n}\right) + i\sin\left(\frac{2\pi kj}{n}\right)$$

$$j = 0,1,\cdots,n-1, \quad and \quad i \equiv \sqrt{-1}$$

Because of the symmetry of complex-value transform, only the values from *0* to $\left(\frac{n}{2}-1\right)$ are considered, which means we only need to keep $f_0, f_1, \cdots, f_{\lfloor\frac{n}{2}\rfloor}$. For obtaining the peak heights, modulus function will be implemented on the *f* sequence:

$$M_i = modulus(f_i) \qquad for\ i = 0,1,\cdots,\left\lfloor\frac{n}{2}\right\rfloor$$

So far the operation on the sequence has been done. According to the assumption, 95% of the *M* values should be less than:

$$T = \sqrt{(log\frac{1}{1-0.95})n}$$

It's easy to count how many *M*s is less than *T*, and denote it by N_1. And we could also compute the expected theoretical numbers of peaks that are less than T. Notice the symmetry, the expected number $N_0 = \frac{0.95n}{2}$. The test statistic:

$$d = \frac{(N_1 - N_0)}{\sqrt{\frac{0.95 \times 0.05n}{4}}}$$

Next we could use the Complementary Error Function to calculate *P-value*:

$$P\text{-}value = \boldsymbol{erfc}\left(\frac{|d|}{\sqrt{2}}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Non-overlapping Template Matching Test

This test is mainly focus on the repetitive of template (denoted by *B*) within the tested stream. The template is a pre-defined specified non-periodic *m*-bit stream, and the tested stream is divided into *N* blocks (*N* is fixed at 8 in this suite), each with length *M*. The test is running a virtual sliding window. From the start of the tested stream, the window is stuffed by the initial *m* bits. If the stream in the window does not match the template, the window slide one bits position and continue comparing. Once a matching happens, the counter for this block plus one and the window slides *m* bits position directly, which equals to throw away current stream and stuff the following *m* bits in the tested stream inside the window. Suppose there is a block of 10 bits: $\varepsilon = \varepsilon_1\varepsilon_2...\varepsilon_{10} = 1100101101$, and $B = 010$. Obviously, $m = 3$. The bits in initial window will be $\varepsilon_1\varepsilon_2\varepsilon_3 = 110 \neq 010 = B$. So the window slides one position, and the current stream will be $\varepsilon_2\varepsilon_3\varepsilon_4 = 100$. The same case continues until the window contains $\varepsilon_4\varepsilon_5\varepsilon_6 = 010 = B$. Then the counter for this block plus *1*, and the window slide *3* bits position. Now the stream in the window should be $\varepsilon_7\varepsilon_8\varepsilon_9 = 110$. After getting the matching frequencies of each block, the $\chi^2$is used as test statistic.

The theoretical mean μ and variance σ is simple:

$$\mu = \frac{(M - m + 1)}{2^m}$$

$$\sigma = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right)$$

If using the W_i to denote the matching frequency in the i^{th} block, the test statistic:

$$\chi^2(obs) = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$$

Then we could use the Complementary incomplete gamma function to calculate *P-value*:

$$P\text{-}value = \boldsymbol{igamc}\left(\frac{N}{2}, \frac{\chi^2(obs)}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Overlapping Template Matching Test

This test is similar with Non-Overlapping Template Matching Test. Both of them all divide stream into blocks, implement a virtual slide window on tested sub-streams. The difference is this test always slide window by one position even when the matching happens. Besides, this test does count the matching times of each block, however, this number is not used to calculate the χ^2 statistic directly, but accumulate to different frequencies of times. Then execute χ^2 test on these frequencies and evaluate the randomness. This test rejects streams which show too many or too few occurrences of *m*-bit runs of *1*s, but can be modified to detect irregular occurrences of any periodic *B*. For the former application, B is always set to be a sequence of identical *1*s.

Like the previous test, *B* denotes the template to be matched, *m* denotes the length of window and *B*, *M* denotes the length of sub-streams, *N* denotes the number of independent blocks of *n*. In this test suite, the *M* and *N* are preset to be *1032* and *968* respectively. W_i denotes the number of matching in the i^{th} block. After collecting all the W_i, a new set of parameters are added in. v_j is used to denote the frequency of *j* appears in all the W_i. Every single W_i is following the Poisson distribution, and all the v_i are the χ^2 distribution.

In the first step, we could directly get all the W_i by using the same method with the previous test except always sliding the window by one position. Then for each W_i, plus one to v_j if $j = W_i$. The following step is a little complex. For dealing with Poisson distribution, we need to count λ in advance:

$$\lambda = \frac{(M - m + 1)}{2^m} \qquad \eta = \frac{\lambda}{2}$$

If U denotes a random variable with the compound Poisson asymptotic distribution, then for $i \geq 1$ with $\eta = \frac{\lambda}{2}$:

$$\pi_i = P(U = i) = \frac{e^{-\eta}}{2^i} \sum_{l=1}^{i} \binom{i-1}{l-1} \frac{\eta^l}{l!} = \frac{\eta e^{-2\eta}}{2^i} \Phi(i+1, 2, \eta)$$

There are also some requirements for the value of λ and η. However, then are concerning on another concept: degrees of freedom *K*. This is more related to Statistic, so this paper will not involve it. The test predefines *K* to be *5*.

When we found all the parameters we need, the χ^2 could be calculated:

$$\chi^2(obs) = \sum_{i=0}^{5} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

After obtaining the χ^2, the *P-value* could be computed by Complementary incomplete gamma function:

$$\textit{P-value} = \boldsymbol{igamc}\left(\frac{5}{2}, \frac{\chi^2(obs)}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Maurer's "Universal Statistical" Test

This test is designed by Ueli Maurer in 1992. It is working in a unique way. It focuses on the distance between two matching patterns. A stream with relatively short such distances is regarded as that could be significantly compressed without loss of information, thus lack of randomness and fail the test. This test play an important role in Cryptography as the author asserted "it is the correct quality measure for a secret-key source". He also claimed it "measure the actual cryptographic significance of a defect because it is related to the running time of an enemy's optimal key-search strategy".

The tested stream is partitioned into blocks. The block size *L* could be defined by users. In this test, the first *Q* blocks are used as initialization sequence. The rest part could be divided into at most *K* blocks, and the rest bits are discarded (Figure 4).

Next, a table is created for each possible *L*-bit value. Hence, there will be 2^Lentries. For each entry, it only need to record one thing, i.e., the block number (denoted by T_j, where $1 \leq j \leq 2^L$) of the last occurrence of this entry. If an entry never happens before, the number under it should be *0*.

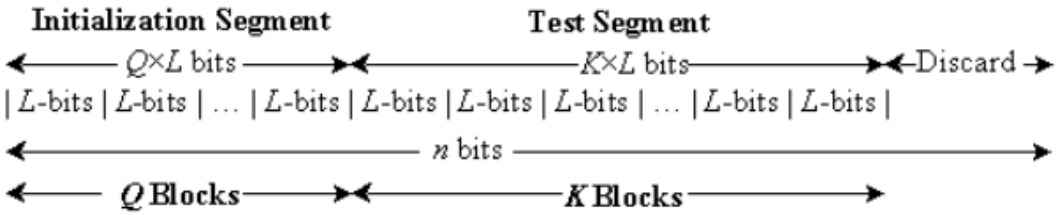

Figure 4

After initialization, the rest *K* blocks are examined one by one. For each block, check the number of blocks since the last occurrence of the same *K*-bit block, replace the value in the table with the number of the current block. Add the calculated distance between re-occurences of the same *L*-bit block to an accumulating log_2 sum of all the differences detected in the K blocks, i.e. $sum = sum + log_2(i - T_j)$, where *i* is the current block number, and T_j is the number under the matching block in the table. The test is using the sum of the log_2 distances between matching blocks as test statistic. It should follow the half-normal distribution. The test statistic could be computed as follow:

$$f_n = \frac{1}{K} \sum_{i=Q+1}^{Q+K} log_2(i - T_j)$$

Thus the *P-value* could be computed by Complementary Error Function:

$$P\text{-}value = \boldsymbol{erfc}\left(\frac{|f_n - expectedValus(L)|}{\sqrt{2}\sigma}\right)$$

Where the *expectedValue(L)* and variance are taken from a table of pre-computed values [27]. Under an assumption of randomness, the sample mean, *expectedValue(L)*, is the theoretical expected value of the computed statistic for the given *L*-bit length. The theoretical standard deviation is given by:

$$\sigma = c\sqrt{\frac{variance(L)}{K}}$$

Where:

$$c = 0.7 - \frac{0.8}{L} + (4 + \frac{32}{L})\frac{K^{\frac{-3}{L}}}{15}$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Linear Complexity Test

This test related to a kind of RNG which is called Linear Feedback Shift Register (LFSR). Before introducing this test, a brief introduction of LFSR is necessary. A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is XOR. Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value (Figure 5). The initial value is called the seed. Once the seed is given, the whole circle is fixed. So it's very important of picking a seed. A bad seed will lead to a short period, thus lose randomness. With each clock tick, certain tapped bits of the LFRS are evaluated by a feedback function. The output of this feedback function is then shifted into the register. The output of the register is the bit that is shifted out. In Figure 5, the feedback algorithm is:

$$f_n = f_{n-24} \oplus f_{n-21} \oplus f_{n-20} \oplus f_{n-10}$$

This test is caring about the length of a LFSR. The length of a LFSR will decide the possible period of the stream. A long LFSR may have a long period, which depends on the seed. However, a short LFSR is destined to be with short period. The purpose of this test is to determine whether or not the stream is complex enough to be considered random.

At first, the stream is partitioned to *N* blocks, each with length *M*. Then implement Berlekamp-Massey algorithm [27] on every block. The B-M algorithm could calculate the lowest degree feedback polynomial of LFSRs to a particular binary stream. Record the lowest degree L_i of the feedback polynomial of the ith block. For each block, calculate a value of T_i:

$$T_i = (-1)^M \times (L_i - \mu) + \frac{2}{9}$$

Where μ is the theoretical mean:

$$\mu = \frac{M}{2} + \frac{(9 + (-1)^{M+1})}{36} - \frac{\left(\frac{M}{3} + \frac{9}{2}\right)}{2^M}$$

The distribution graph of T_i reveals that the most of them are concentrated in the interval (-2.5, 2.5). Next, record the T_i value in $v_0, \cdots, v_6$ as follows:

$\mathrm{T_i} \le -2.5 \quad \mathrm{v_0} = \mathrm{v_0} + 1$ $\quad -2.5 < \mathrm{T_i} \le -1.5 \quad \mathrm{v_1} = \mathrm{v_1} + 1$

$-1.5 < T_\mathrm{i} \le -0.5 \quad \mathrm{v_2} = \mathrm{v_2} + 1$ $\quad -0.5 < T_\mathrm{i} \le 0.5 \quad \mathrm{v_4} = \mathrm{v_4} + 1$

$0.5 < T_\mathrm{i} \le 1.5 \quad \mathrm{v_5} = \mathrm{v_5} + 1$ $\quad 1.5 < T_\mathrm{i} \le 2.5 \quad \mathrm{v_6} = \mathrm{v_6} + 1$

$$\mathrm{T_i} > 2.5 \quad \mathrm{v_7} = \mathrm{v_7} + 1$$

The theoretical probabilities $\pi_0, \pi_1, \cdots, \pi_6$ of these intervals have been pre-calculated and given below:

$$\pi_0 = 0.010417, \ \pi_1 = 0.3125, \ \pi_2 = 0.125, \ \pi_3 = 0.5$$

$$\pi_4 = 0.25, \ \pi_6 = 0.0625, \ \pi_7 = 0.20833$$

After collecting all the parameters, the χ^2 statistic could be calculated:

$$\chi^2(obs) = \sum_{i=0}^{6} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

And then the *P-value* could be computed by Complementary incomplete gamma function:

$$P\text{-}value = \boldsymbol{igamc}\left(3, \frac{\chi^2(obs)}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

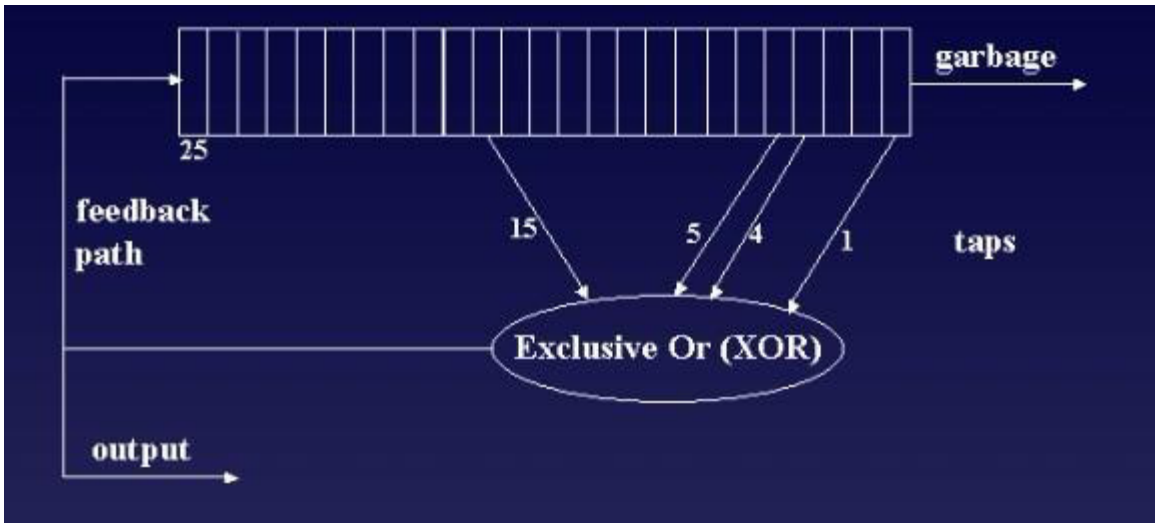

Figure 5

- Serial Test

The focus of this test is the frequency of every *m*-bit permutation stream in the tested stream. In a truly random stream, all the 2^m possible patterns should have roughly equivalent occurrences. This test is checking if one pattern of *m*-bit stream appears approximately the same as other patterns in a tested stream. When *m = 1*, this test degenerates to Monobit test.

The length value *m* could be chosen by user. The test will not only check all the possible *m*-bit overlapping permutation streams, but also check *m-1*-bit and *m-2*-bit if $m \geq 2$. When executing checking a *m*-bit stream, the tested stream is required to add the initial *m-1* bits to the tail. The *m-2* or *m-3* initial bits will add to the tail when checking a *m-1*-bit or *m-2*-bit stream respectively. The test statistic will show how well the observed frequencies of *m*-bit patterns match the expected frequencies with the same length.

For each *m*-bit pattern $i_1 \cdots i_m$, the number of matching through the tested stream will be recorded in $\nu_{i_1 \cdots i_m}$. Similarly, the appearance of each *m-1*-bit pattern $i_1 \cdots i_{m-1}$ and *m-2*-bit pattern $i_1 \cdots i_{m-2}$ will be recorded in $\nu_{i_1 \cdots i_{m-1}}$ and $\nu_{i_1 \cdots i_{m-2}}$ respectively.

The test will count all the appearances of these stream, and then compute:

$$\psi_m^2 = \frac{2^m}{n} \sum_{i_1 \cdots i_m} \left(\nu_{i_1 \cdots i_m} - \frac{n}{2^m} \right)^2 = \frac{2^m}{n} \sum_{i_1 \cdots i_m} {\nu_{i_1 \cdots i_m}}^2 - n$$

$$\psi_{m-1}^2 = \frac{2^{m-1}}{n} \sum_{i_1 \cdots i_{m-1}} \left(\nu_{i_1 \cdots i_{m-1}} - \frac{n}{2^{m-1}}\right)^2 = \frac{2^{m-1}}{n} \sum_{i_1 \cdots i_{m-1}} {\nu_{i_1 \cdots i_{m-1}}}^2 - n$$

$$\psi_{m-2}^2 = \frac{2^{m-2}}{n} \sum_{i_1 \cdots i_{m-2}} \left(\nu_{i_1 \cdots i_{m-2}} - \frac{n}{2^{m-2}}\right)^2 = \frac{2^{m-2}}{n} \sum_{i_1 \cdots i_{m-2}} {\nu_{i_1 \cdots i_{m-2}}}^2 - n$$

The test statistic needs the difference and square of difference between these accumulations:

$$\nabla\psi_m^2 = \psi_m^2 - \psi_{m-1}^2$$

$$\nabla^2\psi_m^2 = \psi_m^2 - 2\psi_{m-1}^2 + \psi_{m-2}^2$$

Both of the upper test statistics could be used to evaluate *P-value* by Complementary incomplete gamma function:

P-value1 = ***igamc*** $(2^{m-2}, \nabla\psi_m^2)$ and *P-value2* = ***igamc*** $(2^{m-3}, \nabla^2\psi_m^2)$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Approximate Entropy Test

Similar with the Series Test, this test is also interested in the frequency of all possible *m*-bit permutation streams, but it evaluates the randomness in a different way. Given a length value *m*, in addition to the counting of all the *m*-bit patterns, the test also counts all the possible *m+1*-bit permutation streams. The difference between these two situations is compared with the difference in truly random number sequences and decide whether the tested stream is random or not.

Here the same denotations with the former test are used. Counting the occurrences is following the same way, so more repeat is not necessary. Suppose the collecting of all the $\nu_{i_1 \cdots i_m}$ and $\nu_{i_1 \cdots i_{m+1}}$ has been done. Compute:

$$C_{i_1 \cdots i_m}^m = \frac{\nu_{i_1 \cdots i_m}}{n} \quad \text{and} \quad C_{i_1 \cdots i_{m+1}}^{m+1} = \frac{\nu_{i_1 \cdots i_{m+1}}}{n}$$

for all the $i_1 \cdots i_m$ and $i_1 \cdots i_{m+1}$. This is the proportion of matching times of each sub-stream in the overlapping tested stream. Then compute:

$$\varphi^{(m)} = \sum_{i=0}^{2^m-1} \pi_i \, log_2 \, \pi_i \quad \text{and} \quad \varphi^{(m+1)} = \sum_{i=0}^{2^{m+1}-1} \pi_i \, log_2 \, \pi_i$$

Where $\pi_i = C_{i_1 \cdots i_m}^m$ and $\pi_i = C_{i_1 \cdots i_{m+1}}^{m+1}$ respectively. The difference between these two values could be used as test statistic:

$$\chi^2 = 2n\left(\log_2\left(\varphi^{(m+1)} - \varphi^{(m)}\right)\right)$$

And then the *P-value* could be computed by Complementary incomplete gamma function:

$$P\text{-}value = \boldsymbol{igamc}\left(2^{m-1}, \frac{\chi^2}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Cumulative Sums Test

In a random number stream, the number of *1*s and *0*s should be roughly equal to each other. This is a basic consensus and has been applied into tests by different way. This test reaffirms this rule. It provides two scanning orders and use a parameter *mode* to control: *mode = 0* means scan forward and *mode = 1* for backward. Without loss of generality, suppose *mode = 0* in following introduction. It turns the tested stream into bits of *1*s and *-1*s by a trick which has been played for several times: the new stream is $X = X_1X_2\cdots X_n$ where $X_i = 2\varepsilon_i - 1$, then accumulate them together. The function of the test is like a man who is walking from origin point. When processing the i^{th} bit, it equals to let the man move one step to left if $X_i = -1$, or right if $X_i = 1$. The distance after each step, S_i, should be recorded. After processing all the bits, it will compare all the S_i and get $z = max_{1\le k\le n}|S_k|$, which means the maximum distance from origin point to the man when he was walking. Obviously, z will follow normal distribution. Hence the *P-value* could be calculated by:

$$P\text{-}Value = 1 - \sum_{k=\frac{\left(-\frac{n}{2}+1\right)}{4}}^{\frac{\left(\frac{n}{2}-1\right)}{4}}\left[\Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k-1)z}{\sqrt{n}}\right)\right] + \sum_{k=\frac{\left(-\frac{n}{2}-3\right)}{4}}^{\frac{\left(\frac{n}{2}-1\right)}{4}}\left[\Phi\left(\frac{(4k+3)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)\right]$$

Where Φ is the standard normal cumulative probability distribution function:

$$\Phi(z) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z} e^{\frac{-u^2}{2}}\,du$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails.

- Random Excursions Test

This test is interested in the same thing with the previous test: the walking distance from the origin point. However, this one is in a more complex manner. It more focuses on the number of circles having exactly *K* visits in a cumulative sum random walk. A 'circle' means the walking man leaves the origin point and walk back again. The purpose of this test is to determine if the

number of visits to a particular state within a cycle deviates from what one could expect for a random sequence. This test contain a series of eight tests, each conclude a *P-value* separately. The same parameters with the previous test will follow the same denotations. The initial steps are also normalizing tested stream ε to *X*, and record the distance S_i after each step, then form the set $S = \{S_1, S_2, \cdots, S_n\}$.

Next the set *S* should be installed in head and tail with a *0* separately and form S', $S' = \{0, S_1, S_2, \cdots, S_n, 0\}$. If using *J* to denote the circles hidden in S', $J =$ the total number of 0s in S' - 1. In the test suite, if a long bit stream with $J < 500$, it will be treated less of randomness and the test doesn't continue any more.

Afterward, for each cycle and for each non-zero state value *x* having values in set $\{-4, -3, -2, -1, 1, 2, 3, 4\}$, compute the frequency of each *x* within each cycle. Then for each of the eight states of *x*, compute $\nu_k(x)$ = the total number of cycles in which state *x* occurs exactly *k* times among all cycles, for $k = 0, 1, \cdots, 5$. All the frequencies greater than 5 are stored in $\nu_5(x)$. Note that $\sum_{k=0}^{5} \nu_k(x) = j$. For each of the eight states *x*, $\nu_0(x), \nu_1(x), \cdots, \nu_5(x)$ could be used to compute test statistic χ^2:

$$\chi^2(obs) = \sum_{k=0}^{5} \frac{\left(\nu_k(x) - J\pi_k(x)\right)^2}{J\pi_k(x)}$$

Where $\pi_k(x)$ is the probability that the state *x* occur *k* times in a truly random distribution. For each state of *x*, the *P-value* could be computed by complementary incomplete gamma function:

$$P\text{-}value = \boldsymbol{igamc}\left(\frac{5}{2}, \frac{\chi^2(obs)}{2}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise the stream fails the test. In this test, it will generate eight *P-values*. If some of them are within (0.01, 0.99) yet some are not, further streams should be examined to determine whether or not this behavior is typical of the generator.

- Random Excursions Variant Test

This test is interested in the same thing with the previous test: the walking distance from the origin point. However, it's concerning on the occurrence of distance after each move. The purpose of this test is to detect deviations from the expected number of visit to various states in the random walk. To a long bit streams, even when it's truly random, it's possible that a segment

contains a lot of *1*s but a few *0*s or reverse situation. This will cause a temporary great distance in either direction. For better evaluating the randomness, the test set eighteen states which covers the distance from -9 to 9, each conclude a *P-value* separately.

The same parameters with the previous test will follow the same denotations. The initial steps are also normalizing tested stream ε to *X*, and record the distance S_i after each step, then form the set *S* and $S' = \{0, S_1, S_2, \cdots, S_n, 0\}$. The $\xi(x)$ is used to denote the frequency of occurrence of the given state *x*. As just introduced, $x \in \{-9, -8, -7, -6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For each *x*, calculate $\xi(x)$. Then for each $\xi(x)$, compute *P-value*:
by Complementary Error Function:

$$P\text{-}value = \boldsymbol{erfc}\left(\frac{|\xi(x)-J|}{\sqrt{2J(4|x|-2)}}\right)$$

If the *P-value* is falling into (0.01, 0.99), the stream passes the test, otherwise fails. The same as the previous test, if some of them are within (0.01, 0.99) yet some are not, further streams should be examined to determine whether or not this behavior is typical of the generator.

DIEHARD

DIEHARD test suite is developed by George Marsaglia. It's first published on a CD-ROM of random numbers in 1995. All the 15 tests in this suite is concerning on concrete statistical tests. They are using different statistical estimators to calculate p-values. Some of them are also implying K-S test.

- Birthday Spacing Test

This test is based on an analogy: choosing *m* birthdays from a year of *n* days, then record the distance, also called spacing, between every two neighbor birthdays. Using *j* to denote the number of spacing occurred more than once, *j* is asymptotically Poisson distribution with mean $\lambda = m^{\frac{3}{4n}}$. According to the principle of birthday attack, the size of container, *n*, should be large enough. The experience also supports this point. Normally, *n* is required to be larger than 2^{18}. In this test suite, $n = 2^{24}, m = 2^9$, so that the underlying distribution for j is taken to be Poisson with $\lambda = m^{\frac{3}{4n}} = 2$.

To implement this test, a sample of 500 j's is taken. It's using chi-square test to provide a p-value. The The first test uses bits 1-24 (counting from the left) from integers in the specified file.

Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a K-S test.

- The Overlapping 5-Permutation Test

This test requires a sequence of one million 32-bit random integers. Taking five consecutive integers and compare them, and number each integer by the order. For example, the sequence *(11,2,4,5,3)* could be numbered as *(5,1,3,4,2)*. Consequently, there will be *5!=120* permutations of possible orders. As many thousands of state transitions are observed, cumulative counts are made of the number of occurrences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99).

- Binary Rank Test

There are two scales of binary rank test, one is 32x32 matrices and another is 6x8 matrices. The test for 32x32 matrices is almost the same with the Binary Matrix Rank Test in Testu01 suite, except a specified number -- 40,000 matrices are implemented in this suite. As to the 6x8 matrices, it's also very similar with the previous one. However, the ranks 0, 1, 2, 3 are rare, so all of them are pooled with those for rank 4. This test will take 100,000 6x8 random matrices and chi-square test is performed on counts for ranks 4, 5, and 6.

- The Bitstream Test

In this test, the tested sequence is bit streams. Treat every bit as a 'letter', and a consecutive of 20 bits is grouped as a 'word'. Here it's using overlapping, which means from the first bit to the twentieth bit is one letter, from the second to the twenty-first forms another one, and so on so forth. Obviously, there will be totally 2^{20} possible words. After composing all the words, it will observe how many words have never appeared.

The bit stream is required as long as to be enough for composing 2^{21} overlapping 20-bit words, so the minimum string will contain 2^{21} + 19 bits. In this case, the number of missing words j should be (very close to) normally distributed with $\mu = 141{,}909 \; and \; \sigma = 428$. Thus $\frac{j-141909}{428}$

should be a standard normal variate that leads to a uniform $[0, 1)$ p-value. The test is repeated twenty times.

- The OccupancyTest

There are totally three tests in Occupancy test: OPSO (Overlapping-Pairs-Sparse-Occupancy), OQSO (Overlapping-Quadruples-Sparse-Occupancy) and DNA. These tests are very similar as Bitstream Test. However, they are taking shorter word length and letters are chosen from a big pool.

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

OQSO means Overlapping-Quadruples-Sparse-Occupancy. The test OQSO is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^{21} four-letter words, (2^{21}+3 "keystrokes"), is again $\mu = 141{,}909\ and\ \sigma = 290$. The μ is based on theory; σ comes from extensive simulation.

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

This test is also known as a kind of Monkey test. A monkey test means the test runs with no specific test in mind, like a monkey invented unintentionally.

- Count-The-1's Test

This test requires a 32-bit integer stream, but only uses byte of each integer, for example, the leftmost byte. Each byte contains 8 bits. The number of *1*s in each byte may be 0, 1, 2, 3, 4, 5, 6, 7, 8 with possibility $\frac{1}{256}, \frac{8}{256}, \frac{28}{256}, \frac{56}{256}, \frac{70}{256}, \frac{56}{256}, \frac{28}{256}, \frac{8}{256}, \frac{1}{256}$, respectively. Then denote different occurrence of *1*s as a letter. In order to balance the possibilities, both the head and tail occurrences are pooled into one letter, say 0, 1, 2 → A, 3 → B, 4 → C, 5 → D, 6, 7, 8 → E, with possibility of $\frac{37}{256}, \frac{56}{256}, \frac{70}{256}, \frac{56}{256}, \frac{37}{256}$, respectively.

Now there are two ways to implement this test. The first is to takes five specific bytes from five successive integers, and the second one is to take five bytes continuously from the byte streams. Then count *1*s in each byte and compose a word by the corresponding letters. So there will be totally 25 possible compositions. A sample of 256,000 overlapping words is implemented and counts are made on frequencies of each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test. The difference of the naive Pearson sums of $\frac{(\text{observed}-\text{expected})^2}{\text{expected}}$ on counts for 5-letter and 4-letter cell counts.

- Parking Lot Test

Suppose there is a square parking garage with each side 100 parking lots. There are totally 10,000 parking lots. Once a car is coming, it attempts to park in a random lot. If the lot is empty, then park it there and increase the successful parking list. OR, the lot has been taken by another car, the crash increase one and it needs to find another parking lot. If plotting the number of attempts *n* versus the number of successfully parked *k*, the curve on the graph should be similar to those provided by a perfect RNG.

However, the graph display is not provided in this test suite. A simple characterization o random experiment is used. Suppose there are *k* cars parking successfully after $n = 12{,}000$ attempts. Obviously, k is following normal distribution and Simulations shows that $\mu = 3{,}523\ and\ \sigma = 21.9$. Thus $\frac{(k-3523)}{21.9}$ should be a standard normal variable. This will be executed for ten times, all the results are passed into a K-S test.

- The Minimum Distance Test

In this test, another bigger square with side of 10,000 is provided. Randomly picking $n = 8000$ points on it and find the *d*, the minimum distance between each pairs of points. In a truly and independently random case, the d^2 is exponentially distributed with mean 0.995. Thus $t = 1 - e^{\left(\frac{-d^2}{0.995}\right)}$ is uniform on $[0, 1)$. The test will repeats 100 times and get 100 *d*, then pass 100 *t*s to K-S test.

- The 3D Spheres Test

Similar as the previous test, this test is also based on the minimum distance between two points. However, the execution happens in three-dimension. A cubic with each edge of 1000 is provided. The test randomly chooses 4000 points in it, and to every point, center a sphere which is large enough to reach the closest point. Thus the volume of the smallest such sphere is close to exponentially distributed with mean 40π, and the radius cubed is exponential with mean 30 which is obtained by extensive simulation. The test will repeat 20 times, each time it will generate 4000 such spheres. Each minimum radius cubed leads to a uniform variable by means of $1 - e^{\left(\frac{-r^3}{30}\right)}$, then a K-S test is done on the 20 p-values.

- The Squeeze Test

Given a huge integer, multiplied by a sequence of random floats between $[0, 1)$, how fast will it reduce? This is the point that the test cares about. Initially, the huge integer $k=2^{31}-1=2147483647$. The tested sequence will be converted to floats on $[0, 1)$. Every time, it uses *k* to multiple a number in the float stream, then keep the ceiling of the result to multiple the next one, and so on so forth until the result is 1. This test will repeat 100,000 times, then count the occurrence of how many floats are using in every single test. The number of times is classified as $j: \leq 6,7, \cdots, 47, \geq 48$. *j* is used to provide a chi-square test for cell frequencies.

- The Overlapping Sums Test

This test requires the tested stream to be float on $[0, 1)$. Add overlapping sequences of 100 consecutive floats, for example, $S(1) = U(1) + \cdots + U(100)$, $S(2) = U(2) + \cdots + U(101), \cdots$ where *U(i)* is uniform float. The S's are virtually normal with a certain covariance matrix. A

linear transformation of S's converts them to a sequence of independent standard normal, which are converted to uniform variables or a K-S test. The p-value from ten K-S test are given still another K-S test.

- The Runs Test

This test is the same as the Runs Test in Testu01, so no more explanations here. It tests 10,000 integers and repeats 10 times.

- The Craps Test

This test is concerning on the very popular gambling game Craps. In this game, the gambler will toss two dices, and expects to get a favorable point. In this test, the tested stream is converted to floats on $[0, 1)$. For each float, multiple it by 6, and retrieve the integer part and plus 1 for simulating toss a dice. The Craps will be played for 200,000 times. The wins and number of toss for ending each game is recorded. With the winning ratio of $p = \frac{244}{495}$, the number of wins should be close to a normal with $\mu = 200000p \ and \ \sigma^2 = 200000p(1-p)$. Theoretically, the game could runs infinitely. However, in practice, the game will be completed by tossing within 20 times, so it lumps all tossing times larger than 20 with 21. All the counts of number of throws are passed into a chi-square test.

George Marsaglia provided the source code written by C and FORTRAN, which could be downloaded on http://www.stat.fsu.edu/pub/diehard. More relative information could also be found there.

SPRNG

Scalable parallel pseudorandom number generator (SPRNG) has excellent property on generating large scale random numbers in parallel, so it's suited to parallel Monte Carlo applications. It also contains a RNG test suite. The tests in it is mainly based on Knuth's book [5], in which he describes several empirical tests, and also a few other tests. People could use this test suite to test either a single random number stream or correlations between streams in a parallel random number generator. Besides, the test suite is capsulated well and easy to be handled. Users could also test their own RNGs by this suite. The detailed manual description is published on

http://sprng.cs.fsu.edu, and a test wizard is demonstrated there. The code is in C and FORTRAN. People can download the latest version SPRNG 4.0 on the same webpage. Recently, the SPRNG development group is going to publish a clean version which excluding FORTRAN. Then it'll be easily compiled only by C compiler. The newest version is still in the trail, and could be downloaded in http://ww2.cs.fsu.edu/~brailsfo/. The applicable systems which have been tested including Ubuntu 11.10 and Fedora 17.

In SPRNG test suite, the sequential tests include collisions test, coupon collector's test, equal distribution test, gap test, maximum-of-t test, permutations test, poker test, runs up test, and serial test. It also contains an inherently parallel test – sum of independent distributions test. Furthermore, two physical model tests are also included in this suite, which are Ising model tests and Random walk tests [6, 7]. This paper will not go through there.

- Collisions Test

The tested stream is converted to integers on interval $[0, \mathrm{d}-1]$. Every integer could be treated as a 'letter'. A 'word' is composed by taking *md* consecutive letters. If two words are same, a collision happens. The test takes *n* words and counts the number of distinct words, and use *n* to subtract this number to get the number of collisions.

This test is like throw balls into urns. There are d^{md} urns and *n* balls. For getting better statistical results, it requires $log_2 md \times log_2 d < 32$, and *n* should be less than the number of possible words, 2^{32}. Users could set the parameters *n*, $log_2 md$, and $log_2 d$ when calling this test in the test suite.

- Coupon Collector's Test

The tested stream is converted to integers on interval $[0, \mathrm{d}-1]$. The integers could be regarded as coupons. A collector achieves success when he get all the *d* kinds of coupons. For example, $= \varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6 \cdots = 3\ 1\ 2\ 1\ 0\ 2\ \cdots$, when $d = 4$. The collector could not finish the game when he has collected $\varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4$, because the coupon '0' is not in his hand. After taking $\varepsilon_5 = 0$, the collector successes and game over. The next round will start from ε_6.

This test needs to finish *n* round games. The occurrence of lengths in every round will be compared with expected distribution. In some unfortunate case, one round may run very far when all the coupons could be collected. In order to avoid a few extreme cases, lengths larger

than *t* will be lumped into case *t*. Users could set the parameters *n, t,* and *d* when calling this test in the test suite.

- Equal-distribution Test

This test is simply testing the uniformity. The tested stream is converted to integers on interval $[0, \mathrm{d}-1]$, and the test evaluate if every integer has equal probability. The parameters *d* and *n* could be set by users.

- Gap Test

The tested stream is converted to floats on interval $[0,1)$. A measuring gap $[\mathrm{a},\mathrm{b}] \in [0,1)$ has been preset. The test will count how many successive floats will fall into the gap. For example, $[\mathrm{a},\mathrm{b}] = [0.2,0.5]$, and $\varepsilon = \varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6\cdots = 0.1\ 0.2\ 0.4\ 0.2\ 0.6\ 0.2\ \cdots$, then the first round will start from ε_2 and count the consecutive 3 floats. Ignoring the ε_5 who is out of gap, the next round will start from ε_6.

The test will run *n* rounds and record the lengths of each round. Then it counts the occurrence of each length. All the lengths larger than *t* will be lumped into case *t*. Users could set the parameters *n, t, a,* and *b* when calling this test in the test suite. Attention: $\mathrm{a} < b, and\ [\mathrm{a},\mathrm{b}] \in [0,1)$.

- Maximum-of-t Test

The tested stream is converted to floats on interval $[0,1)$. The test cut the tested stream into *n* non-overlapping subsequence of each with length *t*. Then for every subsequence, record the largest float. The n numbers should be compared with exponential distribution. The parameters *n* and *t* could be preset by users.

- Permutations Test

The tested stream is converted to floats on interval $[0,1)$. Every non-overlapping *m* successive floats compose a subsequence. A total of *n* subsequences are composed. In each subsequence, rank the numbers according to their magnitude, that is, the smallest float is ranked 1, ⋯, the biggest float is ranked m. For example, when *m = 6*:

$$\varepsilon = \varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6 = 0.1\ 0.9\ 0.4\ 0.2\ 0.6\ 0.3 \quad \rightarrow \quad (1,6,4,2,5,3)$$

There will be *m!* possible permutations. The test counts the occurrence of each permutation, and checks if everyone is roughly equally probable. *m* and *n* could be preset by users.

- Poker Test

The tested stream is converted to integers on interval $[0, \mathrm{d}-1]$. This test will separate the tested stream into non-overlapping subsequences with equal length *k* and count the numbers of distinct numbers obtained, denoted by *l*. For example, if $d = 3$ and $k = 4$,

$$\varepsilon = \varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6\varepsilon_7\varepsilon_8\cdots = 1\,0\,1\,2\,1\,1\,0\,0\cdots$$

then in the first subsequence, there are 3 distinct integers, whereas in the second one only have 2. The test implements *n* subsequences and counts the occurrence of *l*s. Users could set the parameters *n, k,* and *d* when calling this test in the test suite.

- Runs Up Test

This test is same with Runs Test in Testu01. In this test suite, it only has ascendant mode. The total number of ascendant sequences *n* and upper bound of group *t* could be predefined by users. The runs who are longer than *t* will be lumped into group *t*. No further explanation will be narrated here.

- Serial Test

The tested stream is converted to integers on interval $[0, \mathrm{d}-1]$. This test simply generates *n* pairs of integers. Each of the d^2 pairs should be equally likely occur. *n* and *d* could be predefined by users.

TestU01

TestU01 is a software library implemented in ANSI C language. It is organized by four modules -- RNGs, statistical tests, pre-defined batteries of tests, and tools for applying tests to entire families of generators. In the first two parts, it covers most existing RNGs and RNG tests. The third part involves three testing batteries, composed by different tests and different size of random numbers. The forth part is designed to perform systematic studies of the interaction between certain types of tests and the structure of the point sets produced by given families of RNGs, and to see at which sample size n_0 the test starts to reject the RNG decisively.

Another good attribute of TestU01 is that it provides strong API for other RNGs and PRNGs. The users who want to test their own RNGs or PRNGs could easily connect to respective interface to feed the particular tests in TestU01. The API could accept three kinds of RNs:

*unif01_Gen *unif01_CreateExternGen01 (char *name, double (*gen01)(void));*

implements a pre-existing external generator gen01 that is not part of TestU01. It must be a C function taking no argument and returning a double in the interval [0, 1). Parameter name is the name of the generator. No more than one generator of this type can be in use at a time.

*unif01_Gen *unif01_CreateExternGenBits (char *name, unsigned int (*genB)(void));*

implements a pre-existing external generator genB that is not part of TestU01. It must be a C function taking no argument and returning an integer in the interval $[0, 2^{32}-1]$. If the generator delivers less than 32 bits of resolution, then these bits must be left shifted so that the most significant bit is bit 31 (counting from 0). Parameter name is the name of the generator. No more than one generator of this type can be in use at a time.

*unif01_Gen *unif01_CreateExternGenBitsL (char *name, unsigned long (*genB)(void));*

similar to unif01_CreateExternGenBits, but with unsigned long instead of unsigned int.
The generator genB must also return an integer in the interval $[0, 2^{32}- 1]$.

Besides, this suite could also test the random number files. The files could be either floating-point number in [0, 1) or binary format. Users could use the functions below to call file tests:

*unif01_Gen * ufile_CreateReadText (char *fname, long nbuf);*

reads numbers (assumed to be in text format) from input file fname. The numbers must be floating-point numbers in [0, 1), separated by whitespace characters. Numbers in the file can be grouped in any way: there may be blank lines, some lines may contain many numbers, others only one. The file must contain only valid real numbers, nothing else. The numbers are read in batches of nbuf at a time and kept in an array (if nbuf is very large, a smaller but still large array will be used instead).

void ufile_InitReadText (void);

ieinitializes the generator obtained from ufile_CreateReadText to the beginning of the file.

*unif01_Gen * ufile_CreateReadBin (char *fname, long nbuf);*

reads numbers from input file fname. This file is assumed to be in binary format. The numbers are read in batches of 4 nbuf unsigned char's at a time, transformed into nbuf unsigned 32-bit integers and kept in an array (if nbuf is very large, a smaller but still large array will be used instead). This function is used in order to test (random) bit sequences kept in a file.

void ufile_InitReadBin (void);

reinitializes the generator obtained from ufile_CreateReadBin to the beginning of the file.

All the functions above are included in head file 'unif01.h'.

In Chapter 4, I'll try to use the TestU01 to test the philox4x32, which is one of Random123 generator. A simple wrapper could help to connect the generating function in philox4x32 to BigCrush test in TestU01. The wrapper code could be found in APPENDIX I and testing result could be found in APPENDIX II.

Also in the same chapter, a 915GB binary file, which is generated by RdRand, is fed to BigCrush test in TestU01. The wrapper code could be found in APPENDIX I and testing result could be found in APPENDIX III.

The source code and users' manual could be downloaded on http://www.iro.umontreal.ca/~simardr/testu01/tu01.html.

Relationships between RNG testing suites

In different testing suites, some tests share the same or similar properties. So here comes an idea that if people could call all these tests within one suite, then they can save a lot of troubles, such as installing other suites, learn how to call testing functions in different suites, find the interface of random number streams in different modules, or even learn various programming languages. This is significant to people who is working on RNGs and relative fields.

As just introduced, TestU01 contains most existing RNGs and statistical tests. Besides, it break limitations other suites suffered, such as the test function parameters are fixed in package, the library directory need to be prefixed before calling tests, the testing integer is required to be in specific length, poor portability, etc.. So it could be used to play a role as platform, on which almost all the tests could be implemented. Below I will try to show the correspondence of tests in TestU01 and other suites. (Table2, Table3)

For tests in FIPS, TestU01 has already composed functions in which all the four tests could be implemented as a group:

*void bbattery_FIPS_140_2 (unif01_Gen *gen);*
*void bbattery_FIPS_140_2File (char *filename);*

The difference between them is the input of the first one is a stream of 1s and 0s, whereas of the second one is a binary data file's name. All the four tests are also included in NIST test suite, so no repeated list here.

TestU01 has also composed a battery function,

*void bbattery_pseudoDIEHARD (unif01_Gen *gen),*

for calling correspond tests in DIEHARD. However, the developer doesn't recommend users to call this function as the DIEHARD can allow many mediocre generators pass it. Here the contents in this battery will not be listed. Parameters that decide the size of testing random number stream and the testing procedure has been pre-set, and could be reset as users" requirement when calling each test. People who are going to implement DIEHARD tests by TestU01 only need to call this function directly.

Although SPRNG tests are based on Knuth's tests, they could also be orientated to parallel tests. In testing procedure, SPRNG could get the random number sequences in different streams in parallel, which will be explained in detail in section 4 of this paper. There is not an exact correspondence between SPRNG and TestU01. The Table 4 only cites the analogous functions

Table 3 The correspondence between tests in NIST and TestU01

Tests in NIST	Test purpose	Testing functions in TestU01	Backup
Monobit test	to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.	sstring_HammingWeight2	L = n
Frequency test within a block	to determine whether the frequency of ones in an M-bit block is approximately M/2	sstring_HammingWeight2	
Runs test	to determine whether the number of runs of ones and zeros of various lengths is as expected	sstring_Run	
Test for the Longest Run of Ones in a Block	to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected	sstring_LongestHeadRun	
Binary Matrix Rank Test	to check for linear dependence among fixed length substrings of the original sequence.	smarsa_MatrixRank	
Discrete Fourier Transform (Spectral) Test [15]	to detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness.	sspectral_Fourier1	
Non-overlapping Template Matching Test	to detect generators that produce too many occurrences of a given non-periodic (aperiodic) pattern	smarsa_MonkeyBits	The difference is whether the window moves one bit or the length of window when matching is found
Overlapping Template Matching test		Not a exact matching function in TestU01, but smultin_MultinomialBitsOver is similar and even more powerful	
Maurer's "Universal Statistical" Test	to detect whether or not the sequence can be significantly compressed without loss of information	svaria_AppearanceSpacings	
Linear Complexity Test	to determine whether or not the sequence is complex enough to be considered random	scomp_LinearComp	the Berlekamp-Massey algorithm [8] is used
Serial Test	to determine whether the number of occurrences of the 2m m-bit overlapping patterns is approximately the same	smultin_MultinomialBitsOver	Delta = 1

Table 3 – continued

Tests in NIST	Test purpose	Testing functions in TestU01	Backup
Approximate Entropy	to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and m+1)	smultin_MultinomialBitsOver	Delta = 0
		sentrop_EntropyDiscOver	
		sentrop_EntropyDiscOver2	
Cumulative Sums (Cusum) Test	to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small	swalk_RandomWalk1	
Random Excursions Test	to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence	Not a exact matching function in TestU01, but similar with swalk_RandomWalk1	
Random Excursions Variant Test	to detect deviations from the expected number of visits to various states in the random walk.	Not a exact matching function in TestU01, but similar with swalk_RandomWalk1, Based on the times that the excursion returns to 0.	

Table 4 The correspondance between tests in SPRNG and TestU01

Tests in SPRNG	Testing Functions in Testu01	Parameters	Backup
Collisions test	void sknuth_CollisionPermut and void smultin_Multinomial	if sknuth_CollisionPermut, $2 \leq t \leq 18$ and $t!/n < 2^{31}$. if smultin_Multinomial, Sparse = TRUE and smultin_GenerCell = smultin_GenerCellSerial	2nd method needs to call setup functions
Coupon collector's test	void sknuth_CouponCollector	$1 < d < 62$ d means the RNG range is in interval [0, d-1]	if d is too large for a given n, there will be only 1 class for the chi-square and the test will not be done
Equidistribution test	void smultin_Multinomial	Sparse = FALSE and smultin_GenerCell = smultin_GenerCellPermut set division to be 1 use par->GenerCell to generate sequence [0 .. d - 1]	Equals to apply multinomial tests on single numbers which are from 0 to d - 1
Gap test	void sknuth_Gap	$0 < a < b < 1$ a,b are fraction numbers, means the upper and lower boundary	
Maximum-of-t test	void sknuth_MaxOft	$n/d \geq$ gofs_MinExpected. n means number of groups, each group has t numbers d means values of X are partitioned into d categories	
Permutations test	1. void sknuth_Permutation 2. smultin_Multinomial	$n/t! \geq$ gofs_MinExpected and $2 \leq t \leq 18$ if 2. Sparse = FALSE and smultin_GenerCell = smultin_GenerCellPermut	2nd method needs to call setup functions
Poker test	void sknuth_SimpPoker	$d < 128$ and $k < 128$ d means the RNG range is in interval [0, d-1] k means k numbers in each group	It's strongly recommended that k and d should be equivalent

Table 4 – continued

Tests in SPRNG	Testing Functions in Testu01	Parameters	Backup
Runs up test	void sknuth_RunIndep	The last parameter in function should be set to TRUE, which means ascending..	In this test function, it only counts the sequences that longer than 6
Serial test	sknuth_Serial	The last parameter, the dimension, should be set to 2.	

.

A testing instance

I'll try to show how to implement a Runs Test on a given bit stream. Runs test is a common test being included in NIST, TestU01 and DIEHARD test suite. It counts the uninterrupted sequence of identical bits, and determines whether the oscillation between 0s and 1s is too fast or too slow.

Let n denote the length of the testing sequence, and ε_i $(0 \le i \le n)$ denote the i^{th} bit in this stream, thus the whole stream is $\varepsilon=\varepsilon_1\varepsilon_2\ldots\varepsilon_n$. $V_n(obs)$ is used to denote the total number of runs. The distribution of $V_n(obs)$ should obey the χ^2 distribution.

Suppose the tested stream has 100 bits (n=100):

$$\varepsilon = 11101010011010110101101010010011010101 10$$
$$11010100100110111010011001101101001101 00$$

It's required to pre-test if this stream could pass the Monobit Test. If the difference between proportion of 1s (denoted as π) and $1/2$ greater than a preset threshold, usually is $2/\sqrt{n}$, then the stream could not pass the Monobit test. In this situation, the stream is rejected by Runs Test as well by default, further test is unnecessary, and p-value could be directly set to 0.00000. In this instance:

$$|\pi - 1/2| = |43/100 - 1/2| = 0.07 < 2/\sqrt{n} = 0.2$$

Thus, further test is applicable. Next step is computing the test statistic $V_n(obs)=\sum_{k=1}^{n-1} r(k) + 1$, where $r(k)=0$ if $\varepsilon_k=\varepsilon_{k+1}$, and $r(k)=1$ otherwise. In this instance:

$$V_n(obs) = (0+0+1+1+\ldots+1+0) = 55$$

After getting $V_n(obs)$, P-value could be calculated:

$$P\text{-}value = erfc\left(\frac{|Vn(obs) - 2n\pi(1-\pi)|}{2\pi(1-\pi)\sqrt{2n}}\right) = erfc\left(\frac{\left|55 - \left(2*100*\frac{43}{100}*\frac{57}{100}\right)\right|}{2*\sqrt{2*100}*\frac{43}{100}*\frac{57}{100}}\right) \approx 0.222 > 0.01$$

Consequently, the stream ε is accepted as random by Runs Test.

CHAPTER THREE

RANDOM NUMBER GENERATORS

Before executing RNG tests, I'd like to introduce the RNGs and PRNGs contained in this article.

Pseudo-random numbers are important in practice for their speed in number generation and their reproducibility. The numbers could be floats, integers or bits of 0 or 1. Actually, they could be treated as same situation by using transformation among float, integer and binary. The integers could not be unlimited large because of the computer storage limitations. Meanwhile, the period is required to be as long as possible. Otherwise, the random numbers will be used out very soon. Another defect is that this stream will be easily guessed by other people, and could not be applied in confidential usages. In applications, people usually use a large prime number as modulus, because theoretically, the longest period under this situation is as large as the prime itself. In the latter case, the good news is bits include only 0 and 1, so there will not exist upper bound. However, it also needs to meet the long period requirement.

PRNG

Let's focus on the PRNGs first. After acquiring the basic requirements, people keep on searching the PRNGs with better and better attributes. The existing applications are included in different areas such as Cryptography, Molecular Physics, Monte Carlo simulations, etc. However, along with the increasing processing speed, the larger period and faster generating speed is required. Many PRNGs have been outdated and thrown into trash bin. Parallel PRNGs are gradually stepping onto stage. In this chapter, I'll cite several PRNGs who are currently still in use, explain their working principles, show the different views, and try to analysis their potential development orientation in future. Because in next chapter, the statistical tests are mainly executed by functions in suite Testu01, I'll also illustrate how to call the following PRNGs by random number generator functions in the suite. All of these PRNGs has been fed to BigCrush

test. The result of LCG with module 2^{32} and MLFG are very representative, and illustrated in APPENDIX III.

Linear Congruential Generators

LCG is the one of the oldest and best known PRNG. It generates integer streams. All the LCGs are sharing a same pattern:

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

Where the

X_n is the output random integers;

m is the modulus, it's always a large prime;

a is the multiplier, naturally a could be any integer, however, for saving calculating time, a is usually taken from the interval $(0 , m)$, besides, in the example below, we'll see how important of picking the a;

c is an additive, the different value of c could not intervene the period, but only decide different order of random numbers;

Before starting a LCG, we need to preset the value of parameters, including m, a, c. Then we need to feed it a seed X_0. Seed is a very important value in all the PRNGs. The bad seed could lead the result generated by a good PRNG into a disaster. Usually, the seeds are required to be taken from truly random numbers. FIPs [4] stipulated the restrictions of seeds for feeding PRNGs whose result would be for Cryptography applications. In the following part, we will see the importance of choosing seeds by comparing the expression by picking different random number streams as seeds. However, to a single value, it lost the randomness. We could pick any number as a seed.

The working procedure of LCG is not hard to understand. To show how exactly the large scale LCG works, I'd like to explain the principle by demonstrating a small scale one.

Let's preset the parameters and seeds before starting the LCG:

$$m = 13$$

$$a = 6$$

$$c = 0$$

$$X_0 = 8$$

Then we could get a number sequence:

$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 9, 2, 12, 7, 3, 5, 4, 11, 1, 6, 10, 8)$$
$$[Stream\ 1]\ (m, a, c\ X_0) = (13, 6, 0, 8)$$

Now $X_0 = X_{12} = 8$, the following sequence will repeat the segment again and again, If we set X_0 to be 1, the output stream will be:

$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1)$$
$$[Stream\ 2]\ (m, a, c\ X_0) = (13, 6, 0, 1)$$

It's very clear that the sequence order in stream 2 is the same with that in stream 1. The difference is initial number. How about the additive value c? How will it affect the sequence? In Condition 1, if we change the additive c to another number, say from 1 to 12, the output stream will be:

$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 10, 9, 3, 6, 11, 2, 0, 1, 7, 4, 12, 8)$$
$$[Stream\ 3]\ (m, a, c\ X_0) = (13, 6, 1, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 11, 3, 7, 5, 6, 12, 9, 4, 0, 2, 1, 8)$$
$$[Stream\ 4]\ (m, a, c\ X_0) = (13, 6, 2, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 12, 10, 11, 4, 1, 9, 5, 7, 6, 0, 3, 8)$$
$$[Stream\ 5]\ (m, a, c\ X_0) = (13, 6, 3, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 0, 4, 2, 3, 9, 6, 1, 10, 12, 11, 5, 8)$$
$$[Stream\ 6]\ (m, a, c\ X_0) = (13, 6, 4, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 1, 11, 6, 2, 4, 3, 10, 0, 5, 9, 7, 8)$$
$$[Stream\ 7]\ (m, a, c\ X_0) = (13, 6, 5, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 2, 5, 10, 1, 12, 0, 6, 3, 11, 7, 9, 8)$$
$$[Stream\ 8]\ (m, a, c\ X_0) = (13, 6, 6, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 3, 12, 1, 0, 7, 10, 2, 6, 4, 5, 11, 8)$$
$$[Stream\ 9]\ (m, a, c\ X_0) = (13, 6, 7, 8)$$
$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 4, 6, 5, 12, 2, 7, 11, 9, 10, 3, 0, 8)$$

[Stream 10] (m, a, c X_0) = (13, 6, 8, 8)

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 5, 0, 9, 11, 10, 4, 7, 12, 3, 1, 2, 8)

[Stream 11] (m, a, c X_0) = (13, 6, 9, 8)

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 6, 7, 0, 10, 5, 1, 3, 2, 9, 12, 4, 8)

[Stream 12] (m, a, c X_0) = (13, 6, 10, 8)

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 7, 1, 4, 9, 0, 11, 12, 5, 2, 10, 6, 8)

[Stream 13] (m, a, c X_0) = (13, 6, 11, 8)

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8)

[Stream 14] (m, a, c X_0) = (13, 6, 12, 8)

From the Stream 1 to Stream 13, except Stream 2, we can see that different additive will lead to a different number sequence. A further observation reveal that these sequences are not linear correlated. Consequently, to a particular LCG, if people want to reuse it for several times, changing *c* may be a good idea. It could effectively avoid correlations between different generated streams by a same LCG. However, in Stream 14, the output stream is an unique number. According to number theory, this situation will always happen whenever the *(m, a, X_0)* is given. Excluding this situation, the *c* value will have no effect on the period of a LCG. People should pay special attention to this when they choose parameters.

The upper examples show us that changing X_0 could not affect the period of a LCG, and various additive *c* will lead to various number sequences. Then what will happen if multiplier *a* is changed? In Condition 1, if we set *a* to be 4, the output stream will be:

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 6, 11, 5, 7, 2, 8, 6, 11, 5, 7, 2, 8)

[Stream 15] (m, a, c X_0) = (13, 4, 0, 8)

The period is shortened into half of original length. For verifying the conclusion we get above, in this situation, if set c to be 5, the output stream will be:

(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 11, 10, 6, 3, 4, 8, 11, 10, 6, 3, 4, 8)

[Stream 16] (m, a, c X_0) = (13, 4, 5, 8)

The period is still half of original length. This strengthened that the c will not affect the period. However, the situation will get even worse if we give another *a*, say 3:

$$(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}) = (8, 11, 7, 8, 11, 7, 8, 11, 7, 8, 11, 7, 8)$$

$$[Stream\ 17]\ (m, a, c\ X_0) = (13, 3, 0, 8)$$

The period in Stream 17 is 1/4 of the original length. From the demonstrations, we realize how important the multiplier *a* is: the period of the LCG may turn into half or even smaller value. In currently using LCGs, the smallest modulus *m* is larger than 2^{30}. It's more and more difficult for people to find a prime when the number is getting bigger and bigger. Some companies are using LCGs with composite modulus under another requirement: the multiplier *a* and modulus *m* should be relatively prime. This could definitely not get a stream as long as the modulus *m* is, however, it save a lot trouble of finding huge primes. If choosing m and a properly, it still could generate a stream with long period and good randomness.

Multiple Recursive Generators

Multiple Recursive Generators are generating integer based on a linear recurrence. The new state is generated according to its former *k* states. They are sharing a common pattern:

$$X_n = (a_1 X_{n-1} + \cdots + a_k X_{n-k}) \bmod m$$

k is called the order of the recurrence. $a_1, \cdots, a_k$ are parameters. For generating integer sequences, they should be integers as well. Knuth proposed that the big prime modulus *m* will bring better quality [15]. To a particular prime module, the parameter and order will affect the period. Let's make some tests in small scales. First we give the parameters as follow:

$$[Stream\ 18]\ (a_1, a_2, a_3, X_1, X_2, X_3, m) = (1, 1, 1, 1, 1, 1, 3)$$

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, \cdots, X_{40}, X_{41}, X_{42}) =$$

$$(1, 1, 1, 0, 2, 0, 2, 1, 0, 0, 1, 1, 2, 1, 1, 1, 0, 2, 0, 2, 1, 0, 0, 1, 1, 2, 1, 1, 1, 0, 2, 0, 2, 1, 0, 0, 1, 1, 2, 1, 1, 1)$$

The tuple $(X_1, X_2, X_3) = (X_{40}, X_{41}, X_{42}) = (1,1,1)$, then we could guess the following sequence will be a repeat of $(X_1, \cdots, X_{39})$. The period is 39. If we are using another arbitary set of parameters:

$$[Stream\ 19]\ (a_1, a_2, a_3, X_1, X_2, X_3, m) = (4, 7, 10, 10, 10, 6, 3)$$

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}) = (10, 10, 6, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2)$$

The tuple $(X_4, X_5, X_6) = (X_{12}, X_{13}, X_{14}) = (2,0,2)$, and the following sequences could also be predicted as the same segment as $(X_4, \cdots, X_{11})$, and the period shrunk to 8. Actually according to the congruence properties, the parameter tuple in Stream 19, (4, 7, 10, 10, 10, 6, 3), will lead to a same sequence as using tuple (1, 1, 1, 1, 1, 0, 3).

Will enlarging the modulus help to extend the period? If we increase the modulus to 5, and observe the generated sequence:

$$[Stream\ 20]\ (a_1, a_2, X_1, X_2, X_3, X_4, m) = (1, 1, 1, 1, 1, 1, 5)$$

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}) =$$
$$(1, 1, 1, 1, 2, 3, 0, 3, 3, 1, 4, 0, 4, 4, 3, 2, 0, 2, 2, 4, 1, 0, 1, 1)$$

The tuple $(X_3, X_4) = (X_{23}, X_{24}) = (1,1)$, and the following sequence will be a copy of $(X_4, \cdots, X_{11})$. The period here is only 20, even less than the Stream 18, the situation when modulus is 3. Let's try another direction, remain the same seeds increase the order a little bit:

$$[Stream\ 21]\ (a_1, a_2, a_3, X_1, X_2, X_3, X_4, m) = (1, 1, 1, 1, 1, 1, 1, 5)$$

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, \cdots, X_{97}, X_{98}, X_{99}) = (1, 1, 1, 1, 3, 0, 4, 2, \cdots, 3, 0, 4)$$

The tuple $(X_5, X_6, X_7) = (X_{97}, X_{98}, X_{99}) = (3,0,4)$, the period make a considerable progress. In fact, both the coefficient and the order could affect the length of period, but the latter is crucial. To a particular modulus m, there will be m^k permutations. Only if the k-tuple in the initial part appears again in the preceding sequence, the generator get into a circle. Consequently, the period

of MRG could achieve $m^k - 1$, theoretically. However, like what we calculated in the streams above, the peak period is not easy to get unless the coefficients are properly set. Once it does, it will get a much better properties than the MLCGs which are using the same modulus. Besides, MRGs are calculating fast and easy to implement.

In around 20 years ago, P. L'Ecuyer suggest the order k be in the interval [1,7], and modulus $m \in \{2^{15} - 19, 2^{31} - 1\}$. He even implemented the CMG by Pascal and C. Along with the increasing speed of calculation, the modulus has been achieved around 2^{64}, and order is choosing loosely, either. For improving the performance, people turn their research interest to combined multiple recursive generators, which combine two or more MRGs of the same order together. I'll not go through it in this paper.

We've introduced the effects to period from both coefficient and order. Because this generator is in view of number sequences, people also want to exploit better properties by setting the seeds. Fibonacci sequence has the characteristic: each member is the summation of its previous two member, excluding the first two numbers. Usually, this sequence starts from 0, 1.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \cdots$$

It is famous for its wonderful and mysterious features, especially the approximation of golden ratio between neighbor members.

Additive Lagged Fibonacci Generators are based on this sequence. They are also MRGs, but usually only two terms are used. They have the common form:

$$F_n = (F_{n-s} + F_{n-t}) \bmod m, \quad 0 < j < t$$

The seed $(F_0, \cdots, F_{s+t-1})$ are from Fibonacci sequence, and should be preset. m is usually a power of 2, like 2^{32} or 2^{64}. The addition could also be substituted by subtraction. Under either situation, the generator will have maximum period: $2^{M-1}(2^t - 1)$. From the expression, we could guess the bigger the order is, the longer the period will have. However, parameter couple (s, t) must be given carefully, or the generated sequence will unfortunately get into recurrence very soon.

Here is a trick of the trade for choosing *(s, t)*. If people want to achieve the maximum period, the polynomial:

$$F(x) = x^s + x^t + 1$$

must be primitive, which means it's irreducible. Popular pairs which have been proved to be good performance including:

(24,55), (38,89), (37,100), (30,127), (83,258), (107,378), (273,607), (1029,2281), ⋯

and see also on page 29 of Volume 2 of The Art of Computing Programming [15].

Multiplicative Lagged Fibonacci Generator

Similar with additive LFGs, multiplicative LFGs are also using members in Fibonacci sequence as seeds. The difference is instead of addition, the MLFGs operate multiplication on two integers. They could be denoted by the common pattern:

$$F_n = (F_{n-k} \times F_{n-l}) \bmod 2^{\mathrm{M}},\ 0 < k < l$$

k and *l* are called the lags. M is often chosen to be 64, which will lead it to be very easy to calculate – ignoring the most significant bit. However, this setting will bring an obvious defect. To show this point, let's start from some small scale generators again. Suppose *(k,l,m) = (3,7,4)*, the seed sequence should be preset:

$$(F_1, F_2, F_3, F_4, F_5, F_6, F_7) = (1, 2, 3, 5, 8, 13, 21)$$

Then we can start generating integer sequences:

[Stream 22] (k,l,m) = (3,7,4)

$$(F_8,F_9,F_{10},F_{11},F_{12},F_{13},F_{14},\ F_{15},F_{16},F_{17},F_{18},F_{19},F_{20},\cdots) = (8,10,15,8,0,3,8,0,14,8,0,0,0,\cdots)$$

It's not hard to imagine once a *0* appears in the sequence, the following part will be nothing but a series of *0*s. Even when we change the modulus to a huge number, say 2^{32}, the problem cannot be resolved. Actually it will happen very soon, within generating less than 100 numbers. What causes this disaster? The answer is even numbers. When generating a new number, once a

multiplier with a factor of exponent of 2, it could not be diminished by mod another exponent of 2, but accumulated. When the exponent of 2 contained within two multipliers is bigger than *M*, the first *0* will appear, and will lead the whole sequence to *0*s. To avoid this situation, the seeds are required to be odd numbers only. Because the multiplication of two odd numbers then subtract an even number is still an odd number. So, as you could guess, the whole sequence contains only odd numbers. The maximum period of multiplicative LFGs could reach $2^{M-3}(2^l - 1)$. Even with a small lag, they still have good properties.

Multiplicative LFGs obtains almost all the advantages of additive LFGs, furthermore, the multiplication will increase more significantly, and consequently lead the current state to a random number, which could effectively avoid fractional linear correlations. In fact, the statistical test results that will be shown in next chapter can also help us confirm this.

Multiplicative LCGs could be easily applied in parallel random number generations, and perform very well. They could generate $2^{(M(l-1)-1)}$ distinct streams, and pass almost all the inter-correlation and inner-correlation tests. Dr.Mascagni and Dr.Srinivasan made deep research on it and explained in detail in the paper Parameterizing parallel multiplicative lagged-Fibonacci generators. They also implemented it in the SPRNG suite.

XORshift Generator

Xorshift generator was first proposed by Marsaglia [13], and after him, Panneton and L'′Ecuyer made deeper research [24]. Because all the operations are based directly on bits, the generating speed is very fast.

The main operations including left shift (denoted as '<<'), right shift (denoted as '>>'), and xor (exclusive or). Just for reminder, exclusive or computing follows:

$$1 \oplus 1 = 0 \qquad 0 \oplus 0 = 0 \qquad 1 \oplus 0 = 1 \qquad 0 \oplus 1 = 1$$

Which is very similar to bit addition without carry.

A xorshift operation is defined as follows: replace the w-bit block by a bitwise xor (exclusiveor) of the original block with a shifted copy of itself by *a* positions either to the right or to the left, where $0 < a < w$.

The seed should be pre-given. It's usually a vector with 32 bits or 64 bits, depends on the generator. In every step, one or more xorshift operations are executed on the vector and get the next state.

Obviously, the xorshift operations are linear. So we could use a matrix to denote the combined operations. Suppose the generator is operated on a vector x of 64 bits. x could be denoted as:

$$(v_1 \quad v_2 \quad \dots \quad v_{64})^T$$

If we want to execute left shift one bit, we could use the L left multiple x, where L is a 64 x 64 matrix with 1s on its main sub-diagonal and 0s otherwise (Figure 4). Similarly, x >> 1 could be denoted as Rx, where R is a 64 x 64 matrix with 1s on its main super-diagonal and 0s otherwise (Figure 5).

$$\begin{bmatrix} 0 & 1 & 0 & & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & 0 & & 0 & 0 & 0 \end{bmatrix}$$

Figure 6

$$\begin{bmatrix} 0 & 0 & 0 & & 0 & 0 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & & 0 & 1 & 0 \end{bmatrix}$$

Figure 7

For moving n times, we only need to multiple L or R by n times respectively, and left multiple x. The last step is to calculate the xor between the original vector x and the new vector x_0 we get after shifting, which is L^sR^tx (suppose x << s and then >>t). According to the xor computing principle, we only need to add x itself to x_0. So the current state x_c could be calculated by:

$$x_c = (I + L^s R^t)x$$

Where I is the 64 x 64 unit matrix.

After acquiring the basic knowledge, let's see how is a XORshift generator working. There is not a universal pattern, different XORshift show various of effect. Specifically, some ways for generating better random bit streams has been found [14]. Here I'll illustrate the generator described in suite Testu01, which is representative. The generator is defined by a recurrence of form:

$$v_i = \sum_{j=1}^{p} A_j v_{i-m_j}$$

The summation in $\sum$ is using bit addition without carry. v_i's are 32 or 64 bits vectors, A_j is either the unit or the product of several *L* and *R* matrixes, and p is a positive integer. At step *i*, the generator's state is demonstrated in Figure 6, where *r* is the number of vectors the generator could store.

$$\begin{bmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,63} \\ v_{1,0} & v_{1,1} & \cdots & v_{1,63} \\ \vdots & \cdots & \cdots & \vdots \\ v_{i-r+1,0} & v_{i-r+1,1} & \cdots & v_{i-r+1,63} \end{bmatrix}$$

Figure 8

And the output is:

$$u_i = \sum_{l=1}^{64} v_{i,l-1} 2^l$$

Where $v_i = (v_{i,0}, \quad \cdots \quad v_{i,63})^T$.

Random123

All the PRNGs I introduced above are sequentially dependent, which means the new status is based on the former statuses. Thus, the generators have to maintain a data list for both calling the former statuses from it and storing new status into it. This procedure doesn't consume a long time if just execute once, however, to generate Terabytes will be a tremendous consuming of time. Random123 RNG suite is designed to avoid this great waste.

Random123 is a composition of counter-based PRNGs, which means the current status doesn't rely on the former statuses any more. Instead, it will depend on the pre-set key-dependent function and counters bijection.

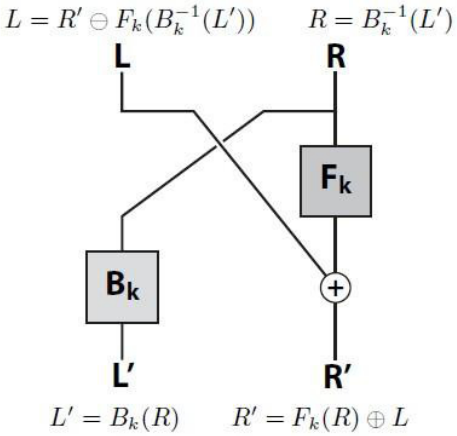

Figure 9

Figure 6 illustrates the working function of Random123. It is similar to traditional Feistel function, whose function is to map input (L,R) to output (L′, R′). F_k is an arbitrary key-dependent function. In addition to Feistel function, a bijection function B_k is used to map R to L′. The inner scope of the pre-constructed function could be simulated as Figure 7.

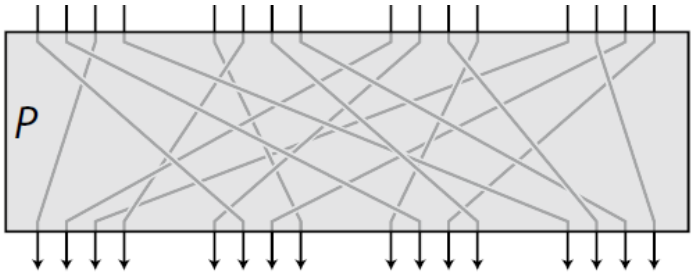

Figure 10

Figure 7 shows a 16-to-16 bijection box. There will be totally 16! different 16-to-16 bijection boxes with equal possibilities. In applications, usually a 32-to-32 or 64-to-64 bijection is called. More bits bijection means more bijection possibilities, and consequently increase the space for randomness.

The Random123 PRNG suite is mainly inspired by and designed for confidential applications. It contains three major families of counter-based PRNGs: Threefry which is extended from threefish and designed to perform a relatively large number of very simple round functions, Philox which is designed as a supplementary to run a smaller number of more complex round functions, and ARS which is extended from AES. More specifically, each family has various PRNGs, including threefry2x32, threefry4x32, threefry2x64, threefry4x64, philox2x32, philox2x64, philox4x32, philox4x64, ars4x32. All these PRNGs obtain very good randomness and could pass most part of BigCrush test. The testing result of philox4x32 and threefry2x32 is attached in APPENDIX II.

RNG

RNGs are commonly generated by hardware. Compared to PRNGs, the current status depends on real-time physical environment, such as entropy, sound, photon, etc. instead of calculating former status, so they are generating quite faster than PRNGs, and appear more randomness. It has also been proved in the tests. However, the disadvantage is being not reproducible. Consequently, they are mainly utilized to Monte Carlo simulation, completely randomized design, providing seeds for PRNGs, etc.

In this segment, I'll introduce two hardware RNGs, RdRand and Letech GRANG box. Each of them have a very strong RN generating ability, and the RNs could pass the BigCrush test easily.

RdRand

RdRand is an instruction for generating RNs from an on-chip RNG. It origins from an Intel hardware RNG plan which is committed to solve the 'platform entropy problem'. Entropy is valuable for confidential uses. As a scarce resource, entropy had to be accumulated and used to seed/reseed a software PRNG that could cryptographically spread that entropy resource out over numerous requests with acceptable performance. Entropy was slowly gathered in small quantities from sources of true entropy at slow rates, however, the demands increasing very fast. Based on this contradiction, Intel creates the Digital Random Number Generator (DRNG). The basic function for DRNG is to build a circuit that is reusable across all Intel process, design, and manufacturing environments, and embed it in each family of Intel silicon products in the most effective way for that family (Figure 8).

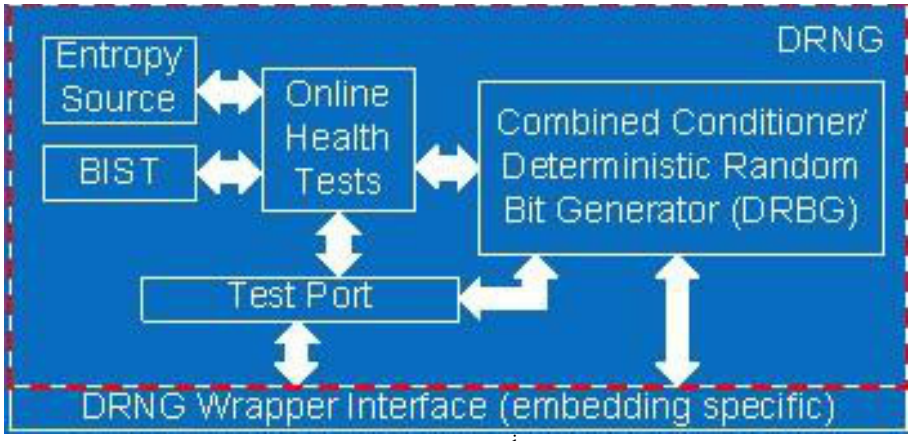

Figure 11[i]

The RdRand provides an instruction for the software interface. It retrieves a hardware generated random value from DRNG and stores it in the destination register given as an argument to the instruction.

The Rdrand instruction code is called Bull Mountain. It could be downloaded from http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/. The code is fairly succinct and API is very straightforward. Users could

[i] Red dotted line is the DRNG's FIPS boundary

also generate a RN file in command line. Recently, Intel released Ivy Bridge (IVB) processors which strongly support the DRNG. IVB processors will be backwards-compatible with the Sandy Bridge platform, but might require a firmware update. It has built Bull Mountain inside. The generating speed is impressively fast. To generate 1GB binary file only cost less than 3 seconds[ii]. In APPENDIX I, I've listed the code for connecting RdRand to BigCrush test and in APPENDIX III the testing result could be found.

Letech GRANG

Genuine Random Number Generator (GRANG) is developed by Letech company. In view of a fact that there is no guarantee that the pulse sequence generated via an electric procedure is truly random even if thermal noise from a physical element like resistor is really random phenomena, Letech developers intended to use the obedience to the principle in physics that time intervals of a random events is given by a Poisson arrival time distribution. GRANG comes with a self-testing function to verify the randomness. A comparison between the generated random numbers and a Poisson arrival time distribution is executed by this function to assure the randomness of physical random numbers. On the other hand, it is important to detect natural environmental noises like a radio wave, an electric discharge and lightning, as well as a malicious attack [29].

GRANG create a GUI for both Windows OS and Linux-like system (Figure 9). Users could setup parameters directly on the control panel and generate random number blocks to specific location. Furthermore, the real-time generating status and self-test result could also be observed on the panel. By this GUI, Users could achieve their intentions easily.

The products include different sizes corresponding to different RN generating speed, which varies from 0.3MB/s to 550MB/s. All size of the chips could be connected to computer either by inserting to motherboard or through USB. I'm using GRANG-SATA-8CH in my test, which has 50MB/s generating speed. It has been proved that this hardware could be successfully utilized under windows 7, Fedora 17 and Ubuntu 11.10. However it could only generate 1GB file every time. Users could execute 'cat' command to heap up the 1GB files. I did the same way and generated a 143GB file finally, then feed it to BigCrush test. The result could be found in APPENDIX II.

[ii] By Ivy Bridge/ Intel Core i5-3550 3.30GHz x 4 / 3.7GB RAM

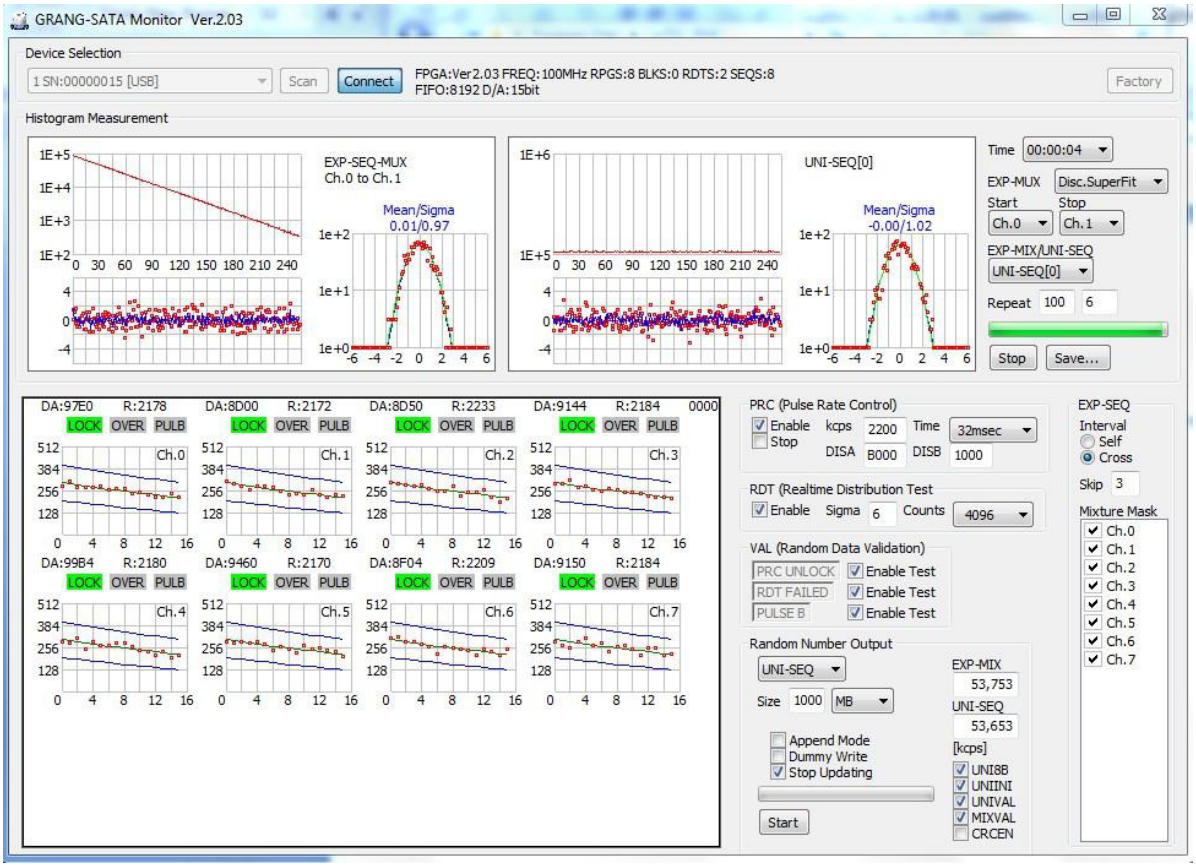
GRANG-SATA Monitor Ver.2.03
Device Selection
1 SN:00000015 [USB]
Scan
Connect
FPGA:Ver2.03 FREQ:100MHz RPGS:8 BLKS:0 RDTS:2 SEQS:8
FIFO:8192 D/A:15bit
Factory
Histogram Measurement
EXP-SEQ-MUX
Ch.0 to Ch.1
Mean/Sigma
0.01/0.97
UNI-SEQ[0]
Mean/Sigma
-0.00/1.02
Time
00:00:04
EXP-MUX
Disc.SuperFit
Start
Stop
Ch.0
Ch.1
EXP-MIX/UNI-SEQ
UNI-SEQ[0]
Repeat
100
6
Stop
Save...
DA:97E0 R:2178
DA:8D00 R:2172
DA:8D50 R:2233
DA:9144 R:2184
DA:99B4 R:2180
DA:9460 R:2170
DA:8F04 R:2209
DA:9150 R:2184
LOCK OVER PULB
Ch.0
Ch.1
Ch.2
Ch.3
Ch.4
Ch.5
Ch.6
Ch.7
PRC (Pulse Rate Control)
Enable
Stop
kcps
2200
Time
32msec
DISA
B000
DISB
1000
RDT (Realtime Distribution Test
Enable
Sigma
6
Counts
4096
VAL (Random Data Validation)
PRC UNLOCK
RDT FAILED
PULSE B
Enable Test
Random Number Output
UNI-SEQ
Size
1000
MB
Append Mode
Dummy Write
Stop Updating
Start
EXP-MIX
53,753
UNI-SEQ
53,653
[kcps]
UNI8B
UNIINI
UNIVAL
MIXVAL
CRCEN
EXP-SEQ
Interval
Self
Cross
Skip
3
Mixture Mask
Ch.0
Ch.1
Ch.2
Ch.3
Ch.4
Ch.5
Ch.6
Ch.7

Figure 12

CHAPTER FOUR

TESTING RESULTS

According to the correspondence between tests in TestU01 and other test suites, I'll try to show the testing results by calling these testing functions. The Confidence interval for p-value is (0.001, 0.999). And eps is used to define when p-value < 1.0e-300, eps1 is used to define when p-value < 1.0e-15. Notice that some tests in TestU01 also calculate Two-level tests such as D_N^+, D_N^-, Anderson-Darling statistic, etc., which are not discussed in this paper. In the following tests, only the p-values for χ^2 are listed. LCG (Linear Congruential Generator) with module 2^{32}, MRG (Multiple Recursive Generator) and Xorshift generator are tested. Due to the space limitation, the BigCrush testing results could not be totally posted here. In APPENDIX II and APPENDIX III, readers could find testing results for some RNGs and PRNGs[iii].

LCG is simply defined by the recurrence:

$$x_i = (ax_{i-1} + c) \bmod m$$

where the multiplier a, increment c and modulus m are set differently in order to getting various sequences. For better performance, two LCGs could be combined together as a new sequence [16]. The widely used configuration contains (Table 5):

Table 5 Parameters of some common LCGs

m	A	c	Reference
2^{24}	1140671485	12820163	Microsoft VB
2^{31}-1	16807	0	IBM[11]
2^{31}	134775813	1	Turbo Pascal
2^{31}	1103515245	12345	rand() in ANCI C
2^{35}	5^{13}	0	Apple
2^{48}	25214903917	11	Unix's rand48()

Here I'll only test the property of the second LCG, which means $a = 16807$, $c = 0$, $m = 2^{31}-1$. The others could be tested by simply modifying parameters. Testing results are listed in Table 6:

iii

Table 6 Testing results for LCG

Test	Statistic	p-value	Acc/Rej
NIST			
Monobit	χ2	0.63	Accept
Monibit test within a blocik	χ2	1-eps1	Reject
Runs	χ2	eps	Reject
	normal	eps	Reject
Longest Run of Ones in a Block	χ2	0.04	Accept
Binary Matrix Rank	χ2	0.7	Accept
Discrete Fourier Transform	normal	0.75	Accept
Non-overlapping Template Matching	Poisson	1-eps1	Reject
Overlapping Template Matching	Poisson	0.24	Accept
	normal	0.17	Accept
Maurer's Universal Statistical	normal	0.9928	Accept
Lempel-Ziv Compression	normal	0.28	Accept
Linear Complexity	χ2	0.94	Accept
	normal	0.34	Accept
Serial	normal	0.84	Accept
Approximate Entropy	χ2	0.44	Accept
Cumulative Sums test	χ2	eps for H,M,J,R 0.0033 for S	Reject for H,M,J,R
FIPS			
Monobit`	χ2	0.33	Accept
Poker	χ2	0.36	Accept
Runs	χ2		All Accept
Longest Run of Zeros in a Block	χ2	0.50	Accept
Longest Run of Ones in a Block	χ2	0.46	Accept
DIEHARD			
Binary Rank Tests for Matrices	χ2	eps	Reject
All other tests in DIEHARD are passed			

MRGs are based on linear recurrence of order k, module m:

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m$$

where $a_1, a_2, \ldots, a_k$ are fixed parameters, and x_n is generated based on the previous values, then use $u_n = x_n/m$ as output. For better performance, two or more MRGs could be combined together [17, 18] as well. Picking parameters and setting seeds are important, because they will determine the property of MRG [12]. LFGs (Lagged-Fibonacci Generators) are special cases of MRGs, and often used as seeds. I'll preset the Fibonacci sequence from $f_1 = 1$, $f_2 = 1$, to $f_{17} = 1597$, and generate

$$f_n = (f_{n-5} - f_{n-17}) \bmod m$$

for testing. The results are listed in Table 7:

Table 7 Testing results for $f_n = (f_{n-5} - f_{n-17}) \bmod m$

Test	Statistic	p-value	Acc/Rej
NIST			
Monobit	χ2	0.32	Accept
Monibit test within a blocik	χ2	0.24	Accept
Runs	χ2	eps	Reject
	normal	eps	Reject
Longest Run of Ones in a Block	χ2	0.59	Accept
Binary Matrix Rank	χ2	0.13	Accept
Discrete Fourier Transform	normal	0.90	Accept
Non-overlapping Template Matching	Poisson	1-eps1	Reject
Overlapping Template Matching	Poisson	0.06	Accept
	normal	0.23	Accept
Maurer's Universal Statistical	normal	0.68	Accept
Lempel-Ziv Compression	normal	0.54	Accept
Linear Complexity	χ2	0.27	Accept
	normal	0.02	Accept
Serial	normal	0.17	Accept
Approximate Entropy	χ2	0.29	Accept
Cumulative Sums test	χ2	H,M,J,R,C are all within (0.001, 0.999)	Accept
FIPS			
Monobit`	χ2	0.99	Accept
Poker	χ2	0.55	Accept
Runs	χ2		All Accept
Longest Run of Zeros in a Block	χ2	0.50	Accept
Longest Run of Ones in a Block	χ2	0.46	Accept
DIEHARD			
Binary Rank Tests for Matrices	χ2	eps	Reject
BirthdaySpacings	χ2	eps1	Reject
All other tests in DIEHARD are passed			

The upper RNGs are all concerning on integers. Although they could have beautiful randomness, time consuming is a big problem. In practice, generating random number sequences directly on bits is always a good option, because it could save a lot of time.

Table 8 Testing results for Xorshift

Test	Statistic	p-value	Acc/Rej
NIST			
Monobit	χ2	0.11	Accept
Monibit test within a blocik	χ2	0.59	Accept
Runs	χ2	eps	Reject
	normal	eps	Reject
Longest Run of Ones in a Block	χ2	3.6e-6	Reject
Binary Matrix Rank	χ2	eps	Reject
Discrete Fourier Transform	normal	0.61	Accept
Non-overlapping Template Matching	Poisson	1-eps1	Reject
Overlapping Template Matching	Poisson	0.50	Accept
	normal	0.81	Accept
Maurer's Universal Statistical	normal	0.76	Accept
Lempel-Ziv Compression	normal	0.05	Accept
Linear Complexity	χ2	0.96	Accept
	normal	1-eps1	Reject
Serial	normal	0.28	Accept
Approximate Entropy	χ2	0.24	Accept
Cumulative Sums test	χ2	eps for H,M,J,R,C	Reject
FIPS			
Monobit`	χ2	0.99	Accept
Poker	χ2	0.55	Accept
Runs	χ2		All Accept
Longest Run of Zeros in a Block	χ2	0.50	Accept
Longest Run of Ones in a Block	χ2	0.46	Accept
DIEHARD			
Binary Rank Tests for Matrices	χ2	eps	Reject
All other tests in DIEHARD are passed			

Xorshift operates as bits shifting, and under some configurations, it could reach good randomness [13, 14, 19]. The sequence could be generated by calling function

unif01_Gen uxorshift_CreateXorshift64 (int a, int b, int c, ulonglong x);*

which will implement Xorshift generator:

$$y = y_{n-1} \oplus (y_{n-1} H_1 a)$$

$$y = y \oplus (y H_2 b)$$

$$y_n = y \oplus (y H_3 c) \bmod 2^{32}$$

where H_1 will be '>>' if a < 0, '<<' if a > 0, so is H_2 and H_3. a, b and c could not indicate that all the bits are shifted out, so -64 < a,b,c < 64. Here I'll call:

uxorshift_CreateXorshift32 (13, -17, 5, 2463534242)

as generator, and list the results in Table 8.

CHAPTER FIVE

PARALLEL RNGS AND PARALLEL TESTS

For large-scale Monte Carlo simulations, the traditional sequential RNG has been of poor quality. Even generators perform well in standard statistical tests for randomness may be unreliable for particular applications, as has been seen many times in the computational science literature [9]. One reason leads to this problem is that the sequential RNG runs out of its period. For example, a LCG with module of 2^{32} will go repeating when the stream is longer than 2^{32} (which is very easy to achieve nowadays within several seconds), and loss the randomness consequently. Another reason is that generating the random numbers will take a long time when enlarging the period. This seems to be a paradox when executing a RNG on single stream. People are attempting to generate multiple random streams in parallel and finding the better attribute. Actually, single stream is falling into disuse, meanwhile more and more people are concentrating on parallel RNGs.

There are several main techniques for generating random numbers in parallel. A main idea is taking several segments, which are non-correlated, from a tremendously long period stream. Three generating methods are often implemented:

Leap frog – the sequence is partitioned among the N processors in a cyclic fashion. Each processor starts from a different point on the common sequence and keep its own leaping span, which is decided by different parameters of the common RNG.

Sequence splitting – the sequence is divided into non-overlapping, contiguous sections, and each processor executes on a different section.

Independent sequences – this method has a high requirement of choosing the seeds for each processor so that they are producing disjoint subsequences.

Another idea is trying to prolong the period of single RNGs, which have been proven as good randomness. For illustration purpose, suppose there are 3 processors A, B and C. The processor A generates sequence $A_1, A_2, \ldots$, processor B generates sequence $B_1, B_2, \ldots$, and so on. One method is taking a fixed length of each sequence in a cyclic fashion and compose a new random number sequence, such as $A_1, A_2, B_1, B_2, C_1, C_2, A_3, A_4, B_3, \ldots$. Another method is using random numbers in one stream to reorder the other streams. For example, taking the A_1^{th} number in

sequence B to the first place, and A_2^{th} number in the second place and so on.

For either method we are choosing, parallel RNG tests should be executed for ensuring the randomness. Actually, people are used to focusing on developing algorithms for parallel RNGs, however, relatively little research are concerning on applying methods for testing them. From historical experience, the final random number stream generated by parallel RNGs fails the statistical tests are most probably due to the correlations that occur between different generating streams, which is also very intuitive. Hence, the parallel testing is concentrating on if the different RNG streams correlate to each other.

People should pay attention to that when saying a test is 'parallel; it doesn't mean executing the test on each stream (which simply equals to execute the test on single random number stream), but means to test the correlations between parallel RNGs. Consequently, before applying the tests, the numbers generated from different processors are needed to be mixed up by some means.

SPRNG testing suite demonstrates its superiority on parallel testing. Within statistical testing functions discussed before, several streams could be interleaved and produce a new stream by adjusting some parameters, and then apply the test on the new stream (Figure 2). The SPRNG test suite is using K-S percentile to judge the randomness. Similarly like single steam test, if the K-S percentile for testing the new stream is falling into the confidential interval, usually [0.01, 0.99], then we believe the test regard the parallel RNGs as non-correlated.

The latest SPRNG test suite version is SPRNG 4.4. It could be downloaded from http://ww2.cs.fsu.edu/~brailsfo/, and the installing instruction is on http://sprng.cs.fsu.edu/Version4.0/quick-start.html. After correctly installed, all the tests could be found in directory ~/sprng4/TESTS. When calling a test, users need to input a command line as follow format:

test.sprng rngtype nstreams ncombine seed param nblocks skip test_parameters

Test.sprng is the test name. Following seven parameters are common to all the tests, whereas the others are specified to different tests. For example, if user would like to run a permutation test for a 48-bit LCG, the command line should be as follow:

./perm.sprng 1 4 6 0 0 5 10 3 100000

'1' is specified to 48-bit LCG, '4' indicates four new streams are composed by interleaving '6' parallel streams, and test '5' blocks of random numbers from each new stream. The first '0'

implies the RNG are using the 0^{th} encoded seed, and the second '0' is a parameter for the generator. Before testing next block, '10' bits are skipped, which is designed for increasing the testing effect. The rest '3' and '100000' are specified to permutation test. If user plans to test a single stream, the third parameter *ncombine* should be set to 1.

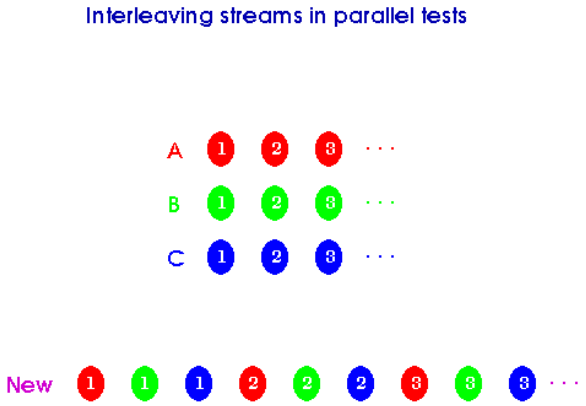

Figure 13

CHAPTER SIX

CONCLUSIONS

RNGs play a significant role not only in scientific disciplines, such as Physics, Statistics, Cryptography, etc., but also in everyday life, like government-run lotteries. Because of these applications, large amount of random numbers are generated. When enjoying the convenience brought us by RNGs, we also need to worry about the 'quality' of these random numbers. Random number testers are like the bureau of quality supervision.

In this article, we mainly focus on statistical tests, and addressed the problem of analysis of statistical results. Many users find it convenient to have predefined suites (or batteries) of more or less standard statistical tests, with fixed parameters, that can be applied to a given RNG. Four popular test suites are introduced with their advantages and disadvantages. Furthermore, due to its comprehensiveness, we illustrated how to execute tests which are in other test suites by calling respective functions or batteries in TestU01, based on the relationship between these testing methods. A wrapper code is also written (here I name it as combination test), but could not post here because of the space limitation. By this mean, people don't need to install several testing packages and execute tests individually. It not only save time and space, but achieve a higher level encapsulation. People who are interested in this wrapper could contact the author directly.

Then we implement three PRNGs on these tests, and list the results in tables in Chapter 4. The more tests accept a PRNG, the more confidence that it could be used to simulate truly random number is built. We could also see the functionality of TestU01 as a platform. One obvious evidence is that G. Marsaglia stated that *[a,b,c,y] = [13,-17,5, 2463534242]* will be a good ideal presetting and will pass all the DIEHARD tests except the binary rank test [13, 21]. The testing result in Table 8 verifies it accurately.

Furthermore, in order to provide comparison, LCG with module 2^{32} and MLFG were fed to BigCrush test. The result in APPENDIX III shows that the former failed most tests whereas the latter passes all tests. From the terrible appearance of the former, and compare with the fact that it could pass most of combination tests, we could come to the conclusion that combination tests are less convincible because the shorter length of testing sequence and less dimension. Besides,

MLFG is a reliable PRNG because it has the same performance with hardware RNGs and meanwhile, it obtains the advantage of PRNGs.

We've also touched several recently emerged PRNGs and RNGs, including Random123, Letech GRANG and RdRand. We were going through into each one to acquire their working functions, APIs and latest products. Then by means of the strong hardware support, I fed the generated numbers to BigCrush in TestU01 and obtained the testing results, partly of which are posted in APPENDIX II and APPENDIX III. It's quite evident that hardware RNGs could pass the BigCrush tests easily. As to Random123, tf2x32, tf4x32, ph2x32 and ph4x32 failed several tests, especially sknuth_Gap test, whereas tf2x64, tf4x64, ph2x64 and ph4x64 pass all tests.

Due to the faster computers and better algorithms, Monte Carlo and other stochastic simulations proposed higher requirements. The PRNGs with longer period, faster generating speed are being chased. Meanwhile, it's also important to search for more precise and varied tests of the randomness properties of these RNGs. In this case, parallel RNGs are the main trend. However, only little parallel tests are developed, which can not follow the pace of inventing parallel RNG algorithms. SPRNG show advantages in parallel testing. So we also showed its working principle and the command line of calling tests in this suite. According to some other materials [10, 11], another kind of test – physical test is more stringent, especially when being operated on large scale random numbers.

Although threefry and philox generators have been tested, I'm still looking for a solution on how to test ARS. Besides, the Letech GRANG has a '1GB roof' in various OS. I need also figure out how to generate a huge file by it directly instead of using 'cat'. After figuring out these questions, I think there will be enough data for me to finish a paper on executing BigCrush test on currently popular RNGs. In future, parallel PRNG is main trend. The random numbers might have not only facing to inter-correlations but also inner-correlations. How to detect them effectively is an important subject. I'll keep working on that area.

APPENDIX I

Wrapper codes for connect users' RNGs to TestU01 BigCrush

In this part, I'll list some codes that I used to make BigCrush test. All the wrappers are inspired by the instructions from manual of TestU01. I was using the version TestU01-1.2.3. So the working space is under directory ~/TestU01-1.2.3.

- Wrapper for testing philox4x32 by TestU01:

  Prerequisites: Random123 should be correctly installed in folder TestU01-1.2.3. The latest Random123 version could be downloaded on http://www.deshawresearch.com/resources_random123.html. The latest version, 1.06 is strongly recommended. Unpack the downloaded file. In order to linking to 'philox.h', the code should be run in folder ~/TestU01-1.2.3/Random123-1.06/include/Random123.

```
#include "unif01.h"
#include  "swrite.h"          // control the result output
#include " bbattery.h"       // BigCrush are included into this header file
#include "philox.h"

unsigned long long ph4x32(void);

int main(void)
{
  unif01_Gen *gen;                                        // All the RNGs have an uniform class "unif01_Gen *"
  gen = unif01_CreateExternGenBits ("ph4x32", ph4x32);    //  Implements a pre-existing external generator genB that is not
                                                          // part of TestU01.
  bbattery_BigCrush(gen);                                 // execute BigCrush test
  unif01_DeleteExternGenBits (gen);                       // clearance function
  return 0;
}

Static unsigned long long c1 = 0;
Static unsigned long long c2 = 0;
Static unsigned long long c3 = 0;
Static unsigned long long c4 = 0;
Static unsigned long long k1 = 0xdeadbeef;
Static unsigned long long k2 = 0xfeedbee;
Static unsigned long long k3 = 0x10101010;
Static unsigned long long k4 = 0x01010101;                // All the initial seeds and keys could be defined by user randomly

unsigned long long ph4x32(void)                           // philox4x32 generator which is written by C.
{
  philox4x32_ctr_t ctr = {{c1, c2, c3, c4}};
  philox4x32_key_t key = {{k1, k2, k3, k4}};
  c1++;
  philox4x32_ctr_t rand = philox4x32(ctr, key);
  return rand.v[0];                                       // Return an integer in the interval [0, 2^32-1] each time.
}
```

- Wrapper for testing binary file which is generated from RdRand by TestU01:

  Prerequisites: A 915GB binary file, GB915.dat, is generated by RdRand and moved to directory ~/TestU01-1.2.3, since all the tests are executed in that directory. Make sure there are still several Gigabytes left on hard disk for temporarily loading the testing bits.

```
#include "unif01.h"
#include  "swrite.h"            // control the result output
#include " bbattery.h"         // BigCrush are included into this header file

int main(void)
{
  unif01_Gen *gen;
  gen = ufile_CreateReadBin ("~/TestU01-1.2.3/GB915.dat", 4);        //Reads numbers from file. The numbers are read in batches of
                                                                      // 4 chars at a time.
  ufile_InitReadBin();                                               // reinitialize the generator to the beginning of the file
  bbattery_BigCrush (gen);
  ufile_DeleteGen(gen);                                               // clearance function
}
```

  RNs generated by Letech GRANG box are tested by the same way.

- Wrapper for feeding RNs generated by RdRand directly to TestU01:

  Prerequisites: An Ivy Bridge system is required. Users could inquire the Intel for internal library file and manual. Then make sure the RdRadn is successfully installed under directory ~/TestU01-1.2.3.

```
#include "unif01.h"
#include  "swrite.h"              // control the result output
#include " bbattery.h"           // BigCrush are included into this header file
#inlcude <unistd.h>
#include "aes128k128d.h"

static unsigned prob;

unsigned retrieve(void)
{
  int status;
  asm("\n\
                rdrand %%eax;\n\
                mov $1, %%edx;\n\
                cmovae %%eax,%%edx;\n\
                move %%edx,%1;\n\
                mov %%eax,%0;":"=r"(prob)."=r"(status)::"%eax"."%edx");
   return prob;
}

int main()
{
  unif01_Gen *gen;
  gen = unif01_CreateExternGenBits("IVB", retrieve);
  bbattery_BigCrush(gen);
  unif01_DeleteExternGenBits(gen);
  return 0;
}
```

APPENDIX II

BigCrush tests results of hardware RNGs

The table 9 lists all the BigCrush tests in TestU01 suite. A binary file whose size is 143GB, generated by Letech GRANG, is tested. Another file whose size is 915GB, generated by RdRand are also tested. However, both of them could not satisfy the minimum data size requirement of BigCrush, that means the tests consume out the data and suspend. Then it's required to reset the tests manually and continue. It's estimated that to run a whole BigCrush test, a file should be as large as 1.9TB. However, limited by the current storage, I could not achieve it. Also, due to the strong API of TestU01 suite and RdRand, the RNs could be generated by RdRand and directly feed BigCrush tests. That's what I showed in the last column – 'On the fly'.

In some tests, there are more than one statistic, so multiple p-values could be calculated to evaluate the quality of randomness. To spread out all the statistics, I create some sub-tables to show them. For example, sub1-2 means the p-values in this blank could be found in the sub-table 1, and labeled 2. P-values are in interval $[0.001,0.9990]$ are considered as regular, in interval $(10^{-15}, 0.001]$ are suspicious and should be tested again, and be marked by green. Eps1 means p-value is in interval $(10^{-300}, 10^{-15})$ whereas eps means in interval $(0, 10^{-300})$. Both of them mean the RNG failed the test, and are marked by red.

Table 9 BigCrush tests on hardware RNGs

Test function	N	n	r	s	Other parameters	P-value of Letech GRANG	P-value of IVB platform	P-value of IVB platform (on the fly)	Acc/Rej
smarsa_SerialOver	1	10^9	0	-	d = 2^8, t = 3	0.47	0.69	0.16	Acc
smarsa_SerialOver	1	10^9	22	-	d = 2^8, t = 3	0.09	0.88	0.62	Acc
smarsa_CollisionOver	30	$2x10^7$	0	2^{21}	t = 2	0.48	0.47	0.45	Acc
smarsa_CollisionOver	30	$2x10^7$	9	2^{21}	t = 2	0.55	0.48	0.07	Acc
smarsa_CollisionOver	30	$2x10^7$	0	2^{14}	t = 2	0.62	0.09	0.10	Acc
smarsa_CollisionOver	30	$2x10^7$	16	2^{14}	t = 2	0.35	0.11	0.85	Acc
smarsa_CollisionOver	30	$2x10^7$	0	64	t = 2	0.01	0.23	0.64	Acc
smarsa_CollisionOver	30	$2x10^7$	24	64	t = 2	0.51	0.78	0.01	Acc

Table 9 - continued

Test function	N	n	r	s	Other parameters	P-value of Letech GRANG	P-value of IVB platform	P-value of IVB platform (on the fly)	Acc/Rej
smarsa_CollisionOver	30	$2x10^7$	0	8	t = 2	0.47	0.43	0.25	Acc
smarsa_CollisionOver	30	$2x10^7$	27	8	t = 2	0.51	0.54	0.69	Acc
smarsa_CollisionOver	30	$2x10^7$	0	4	t = 2	0.97	0.48	4.9e-3	Acc
smarsa_CollisionOver	30	$2x10^7$	28	4	t = 2	0.06	0.89	0.91	Acc
smarsa_BirthdaySpacings	100	10^7	0	2^{31}	t = 2, p = 1	0.05	0.03	0.81	Acc
smarsa_BirthdaySpacings	20	$2x10^7$	0	2^{21}	t = 3, p = 1	0.44	0.38	0.9977	Acc
smarsa_BirthdaySpacings	20	$3x10^7$	14	2^{16}	t = 4, p = 1	0.66	0.11	0.52	Acc
smarsa_BirthdaySpacings	20	$2x10^7$	0	2^9	t = 7, p = 1	0.49	2.2e-3	0.58	Acc
smarsa_BirthdaySpacings	20	$2x10^7$	7	2^9	t = 7, p = 1	0.79	0.85	0.27	Acc
smarsa_BirthdaySpacings	20	$3x10^7$	14	2^8	t = 8, p = 1	0.45	0.30	0.45	Acc
smarsa_BirthdaySpacings	20	$3x10^7$	22	2^8	t = 8, p = 1	0.10	0.55	0.85	Acc
smarsa_BirthdaySpacings	20	$3x10^7$	0	2^4	t = 16, p = 1	0.27	0.71	0.10	Acc
smarsa_BirthdaySpacings	20	$3x10^7$	26	2^4	t = 16, p = 1	0.31	0.17	0.14	Acc
snpair_ClosePairs	30	$6x10^6$	0	-	t = 3, p = 0, m = 30	Sub 1-1	Sub 1-5	Sub 1-9	Acc
snpair_ClosePairs	20	$4x10^6$	0	-	t = 5, p = 0, m = 30	Sub 1-2	Sub 1-6	Sub 1-10	Acc
snpair_ClosePairs	10	$3x10^6$	0	-	t = 9, p = 0, m = 30	Sub 1-3	Sub 1-7	Sub 1-11	Acc
snpair_ClosePairs	5	$3x10^6$	0	-	t = 16, p = 0, m = 30	Sub 1-4	Sub 1-8	Sub 1-12	Acc
sknuth_SimpPoker	1	$4x10^8$	0	-	d = 8, k = 8	0.46	0.85	0.68	Acc
sknuth_SimpPoker	1	$4x10^8$	27	-	d = 8, k = 8	0.54	0.83	0.05	Acc
sknuth_SimpPoker	1	10^8	0	-	d = 32, k = 32	0.41	0.53	0.11	Acc
sknuth_SimpPoker	1	10^8	25	-	d = 32, k = 32	0.05	0.98	0.23	Acc
sknuth_CouponCollector	1	$2x10^8$	0	-	d = 8	0.95	0.13	0.60	Acc
sknuth_CouponCollector	1	$2x10^8$	10	-	d = 8	0.69	0.17	0.36	Acc
sknuth_CouponCollector	1	$2x10^8$	20	-	d = 8	0.38	0.17	0.18	Acc
sknuth_CouponCollector	1	$2x10^8$	27	-	d = 8	0.92	0.64	0.88	Acc
sknuth_Gap	1	$5x10^8$	0	-	Alpha = 0, Beta = 1/16	6.2e-3	0.82	0.33	Acc
sknuth_Gap	1	$3x10^8$	25	-	Alpha = 0, Beta = 1/32	0.42	0.12	0.31	Acc
sknuth_Gap	1	10^8	0	-	Alpha = 0, Beta = 1/128	0.34	0.57	0.76	Acc
sknuth_Gap	1	10^7	20	-	Alpha = 0, Beta = 1/256	0.34	0.93	0.29	Acc
sknuth_Run	5	10^9	0	-	Up = FALSE	Sub 2-1	Sub 2-38	Sub 2-75	Acc
sknuth_Run	5	10^9	15	-	Up = TRUE	Sub 2-2	Sub 2-39	Sub 2-76	Acc
sknuth_Permutation	1	10^9	0	-	t = 3	0.13	0.82	0.38	Acc
sknuth_Permutation	1	10^9	0	-	t = 5	0.52	0.97	0.07	Acc
sknuth_Permutation	1	$5x10^8$	0	-	t = 7	0.23	0.46	0.31	Acc
sknuth_Permutation	1	$5x10^8$	10	-	t = 10	0.30	0.53	0.46	Acc
sknuth_CollisionPermut	20	$2x10^7$	0	-	t = 14	0.67	0.74	0.99	Acc
sknuth_CollisionPermut	20	$2x10^7$	10	-	t = 14	0.60	0.12	0.46	Acc
sknuth_MaxOft	40	10^7	0	-	d =10^5, t = 8	Sub 2-3	Sub 2-40	Sub 2-77	Acc
sknuth_MaxOft	30	10^7	0	-	d =10^5, t = 16	Sub 2-4	Sub 2-41	Sub 2-78	Acc
sknuth_MaxOft	20	10^7	0	-	d =10^5, t = 24	Sub 2-5	Sub 2-42	Sub 2-79	Acc

Table 9 – continued

Test function	N	n	r	s	Other parameters	P-value of Letech GRANG	P-value of IVB platform	P-value of IVB platform (on the fly)	Acc/Rej
sknuth_MaxOft	20	10^7	0	-	d =10^5, t = 32	Sub 2-6	Sub 2-43	Sub 2-80	Acc
svaria_SampleProd	40	10^7	0	-	t = 8	Sub 2-7	Sub 2-44	Sub 2-81	Acc
svaria_SampleProd	20	10^7	0	-	t = 16	Sub 2-8	Sub 2-45	Sub 2-82	Acc
svaria_SampleProd	20	10^7	0	-	t = 24	Sub 2-9	Sub 2-46	Sub 2-83	Acc
svaria_SampleMean	$2x10^7$	30	0	-	-	Sub 2-10	Sub 2-47	Sub 2-84	Acc
svaria_SampleMean	$2x10^7$	30	10	-	-	Sub 2-11	Sub 2-48	Sub 2-85	Acc
svaria_SampleCorr	1	$2x10^9$	0	-	k = 1	0.51	0.60	0.98	Acc
svaria_SampleCorr	1	$2x10^9$	0	-	k = 2	0.79	0.71	0.34	Acc
svaria_AppearanceSpacings	1	-	0	3	Q=10^7,K=10^7 , L=15	0.89	0.12	0.07	Acc
svaria_AppearanceSpacings	1	-	27	3	Q=10^7,K=10^7, L=15	0.93	0.40	0.29	Acc
svaria_WeightDistrib	1	$2x10^7$	0	-	Alpha = 0,Beta = -1/4, k=256	0.32	0.06	0.03	Acc
svaria_WeightDistrib	1	$2x10^7$	20	-	Alpha = 0,Beta = -1/4, k=256	0.11	0.90	0.75	Acc
svaria_WeightDistrib	1	$2x10^7$	28	-	Alpha = 0,Beta = -1/4, k=256	0.16	0.61	0.04	Acc
svaria_WeightDistrib	1	$2x10^7$	0	-	Alpha =0,Beta =-1/16, k=256	0.01	0.43	0.90	Acc
svaria_WeightDistrib	1	$2x10^7$	10	-	Alpha =0,Beta =-1/16, k=256	0.41	0.84	0.64	Acc
svaria_WeightDistrib	1	$2x10^7$	26	-	Alpha =0,Beta =-1/16, k=256	6.5e-9	0.67	0.92	Acc
svaria_SumCollector	1	$5x10^8$	0	-	g = 10	0.54	0.36	0.72	Acc
smarsa_MatrixRank	10	10^6	0	5	L = k = 30	Sub 2-12	Sub 2-49	Sub 2-86	Acc
smarsa_MatrixRank	10	10^6	25	5	L = k = 30	Sub 2-13	Sub 2-50	Sub 2-87	Acc
smarsa_MatrixRank	1	5000	0	4	L = k = 1000	Sub 2-14	Sub 2-51	Sub 2-88	Acc
smarsa_MatrixRank	1	5000	26	4	L = k = 1000	Sub 2-15	Sub 2-52	Sub 2-89	Acc
smarsa_MatrixRank	1	80	15	15	L = k = 5000	Sub 2-16	Sub 2-53	Sub 2-90	Acc
smarsa_MatrixRank	1	80	0	30	L = k = 5000	Sub 2-17	Sub 2-54	Sub 2-91	Acc
smarsa_Savir2	10	10^7	10	-	m = 10^7, t = 30	Sub 2-18	Sub 2-55	Sub 2-92	Acc
smarsa_GCD	10	$5x10^7$	0	30	-	Sub 2-19	Sub 2-56	Sub 2-93	Acc
swalk_RandomWalk1	1	10^8	0	5	L_0 = L_1 = 50	Sub 3-1	Sub 3-7	Sub 3-13	Acc
swalk_RandomWalk1	1	10^8	25	5	L_0 = L_1 = 50	Sub 3-2	Sub 3-8	Sub 3-14	Acc
swalk_RandomWalk1	1	10^7	0	10	L_0 = L_1 = 1000	Sub 3-3	Sub 3-9	Sub 3-15	Acc
swalk_RandomWalk1	1	10^7	20	10	L_0 = L_1 = 1000	Sub 3-4	Sub 3-10	Sub 3-16	Acc
swalk_RandomWalk1	1	10^6	15	15	L_0 = L_1 = 10000	Sub 3-5	Sub 3-11	Sub 3-17	Acc
swalk_RandomWalk1	1	10^6	15	15	L_0 = L_1 = 10000	Sub 3-6	Sub 3-12	Sub 3-18	Acc
scomp_LinearComp	1	$4x10^5$	0	1	-	Sub 4-1	Sub 4-7	Sub 4-13	Acc
scomp_LinearComp	1	$4x10^5$	29	1	-	Sub 4-2	Sub 4-8	Sub 4-14	Acc
scomp_LempelZiv	10	-	0	30	k = 27	Sub 2-20	Sub 2-57	Sub 2-94	Acc
scomp_LempelZiv	10	-	15	15	k = 27	Sub 2-21	Sub 2-58	Sub 2-95	Acc
sspectral_Fourier3	10^5	-	0	3	k = 14	Sub 2-22	Sub 2-59	Sub 2-96	Acc
sspectral_Fourier3	10^5	-	27	3	k = 14	Sub 2-23	Sub 2-60	Sub 2-97	Acc
sstring_LongestHeadRun	1	1000	0	3	L = 10^7	Sub 4-3	Sub 4-9	Sub 4-15	Acc
sstring_LongestHeadRun	1	1000	27	3	L = 10^7	Sub 4-4	Sub 4-10	Sub 4-16	Acc
sstring_PeriodsInStrings	10	$5x10^8$	0	10	-	Sub 2-24	Sub 2-61	Sub 2-98	Acc

Table 9 – continued

Test function	N	n	r	s	Other parameters	P-value of Letech GRANG	P-value of IVB platform	P-value of IVB platform (on the fly)	Acc/Rej
sstring_PeriodsInStrings	10	$5x10^8$	20	10	-	Sub 2-25	Sub 2-62	Sub 2-99	Acc
sstring_HammingWeight2	10	10^9	0	3	$L = 10^8$	Sub 2-26	Sub 2-63	Sub 2-100	Acc
sstring_HammingWeight2	10	10^9	27	3	$L = 10^8$	Sub 2-27	Sub 2-64	Sub 2-101	Acc
sstring_HammingCorr	1	10^9	10	10	L = 30	0.07	0.35	0.36	Acc
sstring_HammingCorr	1	10^9	10	10	L = 300	0.60	0.31	0.45	Acc
sstring_HammingCorr	1	10^9	10	10	L = 1200	0.27	0.69	0.42	Acc
sstring_HammingIndep	10	$3x10^7$	0	3	L = 30, d = 0	Sub 2-28	Sub 2-65	Sub 2-102	Acc
sstring_HammingIndep	10	$3x10^7$	27	3	L = 30, d = 0	Sub 2-29	Sub 2-66	Sub 2-103	Acc
sstring_HammingIndep	1	$3x10^7$	0	4	L = 300, d = 0	Sub 2-30	Sub 2-67	Sub 2-104	Acc
sstring_HammingIndep	1	$3x10^7$	26	4	L = 300, d = 0	Sub 2-31	Sub 2-68	Sub 2-105	Acc
sstring_HammingIndep	1	10^7	0	5	L = 1200, d = 0	Sub 2-32	Sub 2-69	Sub 2-106	Acc
sstring_HammingIndep	1	10^7	25	5	L = 1200, d = 0	Sub 2-33	Sub 2-70	Sub 2-107	Acc
sstring_Run	1	$2x10^9$	0	3	-	Sub 4-5	Sub 4-11	Sub 4-17	Acc
sstring_Run	1	$2x10^9$	27	3	-	Sub 4-6	Sub 4-12	Sub 4-18	Acc
sstring_AutoCor	10	10^9	0	3	d = 1	Sub 2-34	Sub 2-71	Sub 2-108	Acc
sstring_AutoCor	10	10^9	0	3	d = 3	Sub 2-35	Sub 2-72	Sub 2-109	Acc
sstring_AutoCor	10	10^9	27	3	d = 1	Sub 2-36	Sub 2-73	Sub 2-110	Acc
sstring_AutoCor	10	10^9	27	3	d = 3	Sub 2-37	Sub 2-74	Sub 2-111	Acc

Sub-table 1

Lable	Testing name	AD	A2	Nm	Y	AD(mNP2)	AD(mnp2-S)
1	snpair_ClosePairs	0.78	0.08	0.61	0.48	0.39	0.08
2	snpair_ClosePairs	0.42	0.52	0.72	0.91	0.02	0.81
3	snpair_ClosePairs	0.79	0.81	0.87	0.62	0.47	0.26
4	snpair_ClosePairs	0.19	0.16	0.86	0.35	0.23	0.26
5	snpair_ClosePairs	0.94	0.12	0.62	0.16	0.66	0.15
6	snpair_ClosePairs	0.68	0.70	0.46	0.52	0.89	0.78
7	snpair_ClosePairs	0.76	0.61	0.53	0.71	0.16	0.61
8	snpair_ClosePairs	0.56	0.34	0.25	0.83	0.15	0.21
9	snpair_ClosePairs	0.82	0.90	0.59	0.27	0.90	0.09
10	snpair_ClosePairs	0.35	0.56	0.26	0.91	0.55	4.7e-3
11	snpair_ClosePairs	0.36	0.53	0.34	0.06	0.75	0.75
12	snpair_ClosePairs	0.35	0.09	0.27	0.99	0.94	0.79

Sub-table 2

Lable	Test name	K − S statistic				K − S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
1	sknuth_Run	0.44	0.27	0.46	0.29	-	-	-	-
2	sknuth_Run	0.63	0.52	0.98	0.44	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			
		D+	D-	A2	χ^2	D+	D-	A2	
3	sknuth_MaxOft	0.02	0.99	0.06	0.97	0.71	0.12	0.29	-
4	sknuth_MaxOft	0.62	0.14	0.31	0.21	0.04	0.88	0.13	-
5	sknuth_MaxOft	0.08	0.72	0.22	0.84	0.36	0.77	0.82	-
6	sknuth_MaxOft	0.75	0.30	0.72	0.26	0.93	8.2e-3	0.02	-
		End of MaxOft test							
7	svaria_SampleProd	0.25	0.20	0.24	-	-	-	-	-
8	svaria_SampleProd	0.42	0.08	0.37	-	-	-	-	-
9	svaria_SampleProd	0.95	5.1e-3	0.02	-	-	-	-	-
10	svaria_SampleMean	0.30	0.91	0.69	-	-	-	-	-
11	svaria_SampleMean	0.14	0.9934	0.17	-	-	-	-	-
12	smarsa_MatrixRank	0.15	0.80	0.42	0.81	-	-	-	-
13	smarsa_MatrixRank	0.88	0.17	0.45	0.26	-	-	-	-
14	smarsa_MatrixRank	-	-	-	0.80	-	-	-	-
15	smarsa_MatrixRank	-	-	-	0.78	-	-	-	-
16	smarsa_MatrixRank	-	-	-	0.65	-	-	-	-
17	smarsa_MatrixRank	-	-	-	0.41	-	-	-	-
18	smarsa_Savir2	0.78	0.02	0.14	0.18	-	-	-	-
19	smarsa_GCD	0.03	0.94	0.21	0.92	-	-	-	-
20	scomp_LempelZiv	0.51	0.62	0.84	0.52	-	-	-	0.38
21	scomp_LempelZiv	0.71	0.09	0.27	0.13	-	-	-	0.71
22	sspectral_Fourier3	0.46	0.64	0.83		-	-	-	-
23	sspectral_Fourier3	0.39	0.25	0.47		-	-	-	-
24	sstring_PeriodsInStrings	0.52	0.33	0.60	0.55	-	-	-	-
25	sstring_PeriodsInStrings	0.08	0.79	0.15	0.83	-	-	-	-
26	sstring_HammingWeight2	0.83	0.05	0.18	0.13	-	-	-	-
27	sstring_HammingWeight2	0.64	0.57	0.95	0.57	-	-	-	-
28	sstring_HammingIndep	0.61	0.58	0.91	0.42	-	-	-	-
29	sstring_HammingIndep	0.92	0.32	0.32	0.07	-	-	-	-
30	sstring_HammingIndep	-	-	-	0.32	-	-	-	-
31	sstring_HammingIndep	-	-	-	0.85	-	-	-	-
32	sstring_HammingIndep	-	-	-	0.02	-	-	-	-
33	sstring_HammingIndep	-	-	-	0.9914	-	-	-	-
34	sstring_AutoCor	0.84	0.17	0.35	0.20	-	-	-	0.29
35	sstring_AutoCor	0.18	0.95	0.33	0.90	-	-	-	0.59
36	sstring_AutoCor	0.70	0.53	0.91	0.38	-	-	-	0.64
37	sstring_AutoCor	0.74	0.21	0.71	0.34	-	-	-	0.87

Sub-table 2 – continued

Lable	Test name	K − S statistic				K − S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
39	sknuth_Run	0.31	0.11	0.26	0.53	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			
		D+	D-	A2	χ^2	D+	D-	A2	
40	sknuth_MaxOft	0.36	0.71	0.79	0.64	0.20	0.78	0.52	-
41	sknuth_MaxOft	0.83	0.42	0.72	0.24	0.60	0.29	0.43	-
42	sknuth_MaxOft	0.55	0.36	0.72	0.51	0.63	0.48	0.97	-
		D+	D-	A2	χ^2	D+	D-	A2	
43	sknuth_MaxOft	0.78	0.42	0.93	0.47	0.12	92	0.11	-
		End of MaxOft test							-
44	svaria_SampleProd	0.37	0.46	0.48	-	-	-	-	-
45	svaria_SampleProd	0.79	0.18	0.26	-	-	-	-	-
46	svaria_SampleProd	0.17	0.23	0.19	-	-	-	-	-
47	svaria_SampleMean	0.40	0.80	0.82	-	-	-	-	-
48	svaria_SampleMean	0.29	0.17	0.14	-	-	-	-	-
49	smarsa_MatrixRank	0.64	0.69	0.29	0.64	-	-	-	-
50	smarsa_MatrixRank	0.26	0.69	0.27	0.85	-	-	-	-
51	smarsa_MatrixRank	-	-	-	0.65	-	-	-	-
52	smarsa_MatrixRank	-	-	-	0.99	-	-	-	-
53	smarsa_MatrixRank	-	-	-	0.17	-	-	-	-
54	smarsa_MatrixRank	-	-	-	0.09	-	-	-	-
55	smarsa_Savir2	0.51	0.64	0.89	0.44	-	-	-	-
56	smarsa_GCD	0.73	0.12	0.23	0.20	-	-	-	-
57	scomp_LempelZiv	0.97	0.40	0.77	0.20	-	-	-	0.51
58	scomp_LempelZiv	0.88	0.12	0.18	0.07	-	-	-	0.74
59	sspectral_Fourier3	0.79	0.37	0.77	-	-	-	-	-
60	sspectral_Fourier3	0.41	0.39	0.76	-	-	-	-	-
61	sstring_PeriodsInStrings	0.62	0.82	0.9982	0.57	-	-	-	-
62	sstring_PeriodsInStrings	0.47	0.79	0.92	0.67	-	-	-	-
63	sstring_HammingWeight2	0.75	0.41	0.44	0.16	-	-	-	-
64	sstring_HammingWeight2	0.45	0.28	0.62	0.55	-	-	-	-
65	sstring_HammingIndep	0.50	0.31	0.68	0.37	-	-	-	-
66	sstring_HammingIndep	0.63	0.32	0.69	0.50	-	-	-	-
67	sstring_HammingIndep	-	-	-	0.05	-	-	-	-
68	sstring_HammingIndep	-	-	-	0.24	-	-	-	-
69	sstring_HammingIndep	-	-	-	0.64	-	-	-	-
70	sstring_HammingIndep	-	-	-	0.80	-	-	-	-
71	sstring_AutoCor	0.13	0.24	0.28	0.56	-	-	-	0.9950
72	sstring_AutoCor	0.36	0.94	0.62	0.82	-	-	-	0.38
73	sstring_AutoCor	0.74	0.30	0.70	0.30	-	-	-	0.88
74	sstring_AutoCor	0.76	0.20	0.68	0.27	-	-	-	0.51

Sub-table 2 – continued

Lable	Test name	K − S statistic				K − S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
76	sknuth_Run	0.88	0.34	0.60	0.23	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			
		D+	D-	A2	χ^2	D+	D-	A2	
77	sknuth_MaxOft	0.87	0.10	0.14	0.09	0.66	0.28	0.76	-
78	sknuth_MaxOft	0.49	0.32	0.40	0.28	0.91	0.06	0.16	-
79	sknuth_MaxOft	0.05	0.94	0.12	0.95	0.51	0.03	0.08	-
80	sknuth_MaxOft	0.90	0.06	0.19	0.07	0.28	0.81	0.71	-
	End of MaxOft test								
81	svaria_SampleProd	0.61	0.50	0.96	-	-	-	-	-
Lable	Test name	K − S statistic				K − S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
83	svaria_SampleProd	0.22	0.40	0.47	-	-	-	-	-
84	svaria_SampleMean	0.86	0.17	0.54	-	-	-	-	-
85	svaria_SampleMean	0.23	0.65	0.62	-	-	-	-	-
86	smarsa_MatrixRank	0.91	0.24	0.18	0.10	-	-	-	-
87	smarsa_MatrixRank	0.58	0.49	0.90	0.42	-	-	-	-
88	smarsa_MatrixRank	-	-	-	0.94	-	-	-	-
89	smarsa_MatrixRank	-	-	-	0.91	-	-	-	-
90	smarsa_MatrixRank	-	-	-	0.48	-	-	-	-
91	smarsa_MatrixRank	-	-	-	0.72	-	-	-	-
92	smarsa_Savir2	0.91	0.02	0.09	0.07	-	-	-	-
93	smarsa_GCD	0.82	0.21	0.35	0.17	-	-	-	-
94	scomp_LempelZiv	0.59	0.10	0.20	0.14	-	-	-	0.12
95	scomp_LempelZiv	0.04	0.82	8.4e-3	0.9939	-	-	-	0.02
96	sspectral_Fourier3	0.78	0.80	0.9983	-	-	-	-	-
97	sspectral_Fourier3	0.29	0.28	0.29	-	-	-	-	-
98	sstring_PeriodsInStrings	0.25	0.84	0.60	0.74	-	-	-	-
99	sstring_PeriodsInStrings	0.86	0.25	0.54	0.20	-	-	-	-
100	sstring_HammingWeight2	0.84	0.08	0.23	0.11	-	-	-	-
101	sstring_HammingWeight2	0.26	0.95	0.35	0.89	-	-	-	-
102	sstring_HammingIndep	0.13	0.87	0.43	0.83	-	-	-	-
103	sstring_HammingIndep	0.48	0.64	0.88	0.62	-	-	-	-
104	sstring_HammingIndep	-	-	-	0.31	-	-	-	-
105	sstring_HammingIndep	-	-	-	0.95	-	-	-	-
106	sstring_HammingIndep	-	-	-	0.26	-	-	-	-
107	sstring_HammingIndep	-	-	-	0.34	-	-	-	-
108	sstring_AutoCor	0.84	0.24	0.59	0.20	-	-	-	0.74
109	sstring_AutoCor	0.01	0.60	0.10	0.86	-	-	-	0.90
110	sstring_AutoCor	0.64	0.74	0.9954	0.52	-	-	-	0.29
111	sstring_AutoCor	0.96	0.21	0.40	0.10	-	-	-	0.49

Sub-table 3

Lable	Test name	Test on the value of the statistic				
		H	M	J	R	C
1	swalk_RandomWalk1	0.06	0.86	0.98	0.72	0.87
2	swalk_RandomWalk1	0.36	0.36	0.79	0.62	0.58
3	swalk_RandomWalk1	0.75	0.29	0.25	0.21	0.26
4	swalk_RandomWalk1	0.39	0.04	0.39	0.43	0.04
5	swalk_RandomWalk1	0.36	0.67	0.09	0.04	0.71
6	swalk_RandomWalk1	0.08	0.19	6.6e-3	0.62	0.17
8	swalk_RandomWalk1	0.17	0.52	0.67	0.20	0.25
9	swalk_RandomWalk1	0.02	0.34	0.77	0.87	0.19
10	swalk_RandomWalk1	0.86	0.26	0.09	0.11	0.53
11	swalk_RandomWalk1	0.59	0.91	0.30	0.13	0.10
12	swalk_RandomWalk1	0.29	0.69	0.64	0.19	0.81
13	swalk_RandomWalk1	0.71	0.19	0.45	0.77	0.70
14	swalk_RandomWalk1	0.03	0.19	0.65	0.03	0.30
15	swalk_RandomWalk1	0.69	0.73	0.46	0.91	0.88
16	swalk_RandomWalk1	0.75	0.73	0.09	0.54	0.41
17	swalk_RandomWalk1	0.12	0.79	0.32	0.25	0.96
18	swalk_RandomWalk1	0.73	0.19	0.78	0.16	0.12
19	swalk_RandomWalk1	0.52	0.51	0.85	0.74	0.44
20	swalk_RandomWalk1	0.35	0.71	0.40	0.87	0.95
21	swalk_RandomWalk1	0.91	0.27	0.15	0.83	0.44
22	swalk_RandomWalk1	0.48	0.42	0.09	0.59	0.65
23	swalk_RandomWalk1	0.64	0.71	0.92	0.50	0.92
24	swalk_RandomWalk1	0.39	2.5e-3	0.21	0.77	0.25

Sub-table 4

Lable	Test name	Statistic	
		χ^2	Normal
1	scomp_LinearComp	0.29	0.22
2	scomp_LinearComp	0.55	0.34
3	sstring_LongestHeadRun	0.55	0.90
4	sstring_LongestHeadRun	0.97	0.69
5	sstring_Run	0.90	0.63
6	sstring_Run	0.58	0.77
7	scomp_LinearComp	0.15	0.52
8	scomp_LinearComp	0.48	0.19
9	sstring_LongestHeadRun	0.77	0.50
10	sstring_LongestHeadRun	0.08	0.44
11	sstring_Run	0.09	0.84
12	sstring_Run	0.12	0.59
13	scomp_LinearComp	0.47	0.74
14	scomp_LinearComp	0.62	0.17
15	sstring_LongestHeadRun	0.65	0.69
16	sstring_LongestHeadRun	0.52	0.69
17	sstring_Run	1-6.3e-5	0.45
18	sstring_Run	0.9997	0.51
19	scomp_LinearComp	0.94	0.35
20	scomp_LinearComp	0.35	0.99
21	sstring_LongestHeadRun	0.25	0.25
22	sstring_LongestHeadRun	0.55	0.02
23	sstring_Run	0.60	0.57
24	sstring_Run	0.58	0.18

APPENDIX III

BigCrush tests results of PRNGs

The table 10 lists all the BigCrush tests in TestU01 suite. The PRNGs tested here include threefry2x32 and philox4x32 in Random123, LCG with module 2^{32} and MLFG.

In some tests, there are more than one statistic, so multiple p-values could be calculated to evaluate the quality of randomness. To spread out all the statistics, I create some sub-tables to show them. For example, sub1-2 means the p-values in this blank could be found in the sub-table 1, and labeled 2. P-values are in interval $[0.001,0.9990]$ are considered as regular, in interval $(10^{-15}, 0.001]$ are suspicious and should be tested again, and be marked by green. Eps1 means p-value is in interval $(10^{-300}, 10^{-15})$ whereas eps means in interval $(0, 10^{-300})$. Both of them mean the RNG failed the test, and are marked by red.

Table 10 BigCrush tests on PRNGs

Test function	N	n	r	s	Other parameters	Ph4x32	Tf2x32	LCG	MLFG
smarsa_SerialOver	1	10^9	0	-	d = 2^8, t = 3	0.11	0.28	1-eps1	0.84
smarsa_SerialOver	1	10^9	22	-	d = 2^8, t = 3	0.62	0.04	1-eps1	0.50
smarsa_CollisionOver	30	$2x10^7$	0	2^{21}	t = 2	0.47	0.21	1-eps1	0.83
smarsa_CollisionOver	30	$2x10^7$	9	2^{21}	t = 2	0.47	0.60	1-eps1	0.47
smarsa_CollisionOver	30	$2x10^7$	0	2^{14}	t = 2	0.09	0.70	1-eps1	0.50
smarsa_CollisionOver	30	$2x10^7$	16	2^{14}	t = 2	0.89	0.42	1-eps1	0.02
smarsa_CollisionOver	30	$2x10^7$	0	64	t = 2	0.94	0.25	1-eps1	0.91
smarsa_CollisionOver	30	$2x10^7$	24	64	t = 2	0.66	0.01	1-eps1	0.31
smarsa_CollisionOver	30	$2x10^7$	0	8	t = 2	0.30	0.16	1-eps1	0.19
smarsa_CollisionOver	30	$2x10^7$	27	8	t = 2	0.08	0.12	1-eps1	0.99
smarsa_CollisionOver	30	$2x10^7$	0	4	t = 2	0.57	0.13	1-eps1	0.21
smarsa_CollisionOver	30	$2x10^7$	28	4	t = 2	0.46	0.92	1-eps1	0.95
smarsa_BirthdaySpacings	100	10^7	0	2^{31}	t = 2, p = 1	0.66	0.58	eps	0.96
smarsa_BirthdaySpacings	20	$2x10^7$	0	2^{21}	t = 3, p = 1	0.43	0.28	eps	0.72
smarsa_BirthdaySpacings	20	$3x10^7$	14	2^{16}	t = 4, p = 1	0.55	0.13	eps	0.92
smarsa_BirthdaySpacings	20	$2x10^7$	0	2^9	t = 7, p = 1	0.55	0.70	eps	0.85
smarsa_BirthdaySpacings	20	$2x10^7$	7	2^9	t = 7, p = 1	0.78	0.72	eps	0.53
smarsa_BirthdaySpacings	20	$3x10^7$	14	2^8	t = 8, p = 1	0.34	0.87	eps	0.94
smarsa_BirthdaySpacings	20	$3x10^7$	22	2^8	t = 8, p = 1	0.25	0.03	eps	0.42
smarsa_BirthdaySpacings	20	$3x10^7$	0	2^4	t = 16, p = 1	0.22	0.26	eps	0.94
smarsa_BirthdaySpacings	20	$3x10^7$	26	2^4	t = 16, p = 1	0.9999	0.71	eps	0.17

Table 10 – continued

Test function	N	n	r	s	Other parameters	Ph4x32	Tf2x32	LCG	MLFG
snpair_ClosePairs	30	$6x10^6$	0	-	t = 3, p = 0, m = 30	Sub 5-1	Sub 5-5	Sub 5-9	Sub 5-13
snpair_ClosePairs	20	$4x10^6$	0	-	t = 5, p = 0, m = 30	Sub 5-2	Sub 5-6	Sub 5-10	Sub 5-14
snpair_ClosePairs	10	$3x10^6$	0	-	t = 9, p = 0, m = 30	Sub 5-3	Sub 5-7	Sub 5-11	Sub 5-15
snpair_ClosePairs	5	$3x10^6$	0	-	t = 16, p = 0, m = 30	Sub 5-4	Sub 5-8	Sub 5-12	Sub 5-16
sknuth_SimpPoker	1	$4x10^8$	0	-	d = 8, k = 8	0.16	0.20	0.04	0.54
sknuth_SimpPoker	1	$4x10^8$	27	-	d = 8, k = 8	0.90	0.42	0.06	0.67
sknuth_SimpPoker	1	10^8	0	-	d = 32, k = 32	0.50	0.36	8.2e-3	0.87
sknuth_SimpPoker	1	10^8	25	-	d = 32, k = 32	0.76	0.69	9.2e-3	0.52
sknuth_CouponCollector	1	$2x10^8$	0	-	d = 8	0.04	0.96	eps	0.46
sknuth_CouponCollector	1	$2x10^8$	10	-	d = 8	0.21	0.10	eps	0.76
sknuth_CouponCollector	1	$2x10^8$	20	-	d = 8	0.33	0.35	eps	0.79
sknuth_CouponCollector	1	$2x10^8$	27	-	d = 8	0.54	0.27	eps	0.77
sknuth_Gap	1	$5x10^8$	0	-	Alpha = 0, Beta = 1/16	eps	2.2e-16	eps	0.18
sknuth_Gap	1	$3x10^8$	25	-	Alpha = 0, Beta = 1/32	eps	eps	eps	0.46
sknuth_Gap	1	10^8	0	-	Alpha = 0, Beta = 1/128	eps	eps	eps	0.71
sknuth_Gap	1	10^7	20	-	Alpha = 0, Beta = 1/256	eps	eps	eps	0.77
sknuth_Run	5	10^9	0	-	Up = FALSE	Sub 6-1	Sub 6-38	Sub 6-75	Sub 6-112
sknuth_Run	5	10^9	15	-	Up = TRUE	Sub 6-2	Sub 6-39	Sub 6-76	Sub 6-113
sknuth_Permutation	1	10^9	0	-	t = 3	0.86	0.54	0.75	0.9967
sknuth_Permutation	1	10^9	0	-	t = 5	0.07	0.38	1-3.9e-8	0.89
sknuth_Permutation	1	$5x10^8$	0	-	t = 7	0.28	0.58	eps	0.90
sknuth_Permutation	1	$5x10^8$	10	-	t = 10	0.27	0.83	1-eps1	0.86
sknuth_CollisionPermut	20	$2x10^7$	0	-	t = 14	0.27	0.08	1-1.2e-8	2.0e-3
sknuth_CollisionPermut	20	$2x10^7$	10	-	t = 14	0.62	0.02	1-7.2e-10	0.32
sknuth_MaxOft	40	10^7	0	-	d =10^5, t = 8	Sub 6-3	Sub 6-40	Sub 6-77	Sub 6-114
sknuth_MaxOft	30	10^7	0	-	d =10^5, t = 16	Sub 6-4	Sub 6-41	Sub 6-78	Sub 6-115
sknuth_MaxOft	20	10^7	0	-	d =10^5, t = 24	Sub 6-5	Sub 6-42	Sub 6-79	Sub 6-116
sknuth_MaxOft	20	10^7	0	-	d =10^5, t = 32	Sub 6-6	Sub 6-43	Sub 6-80	Sub 6-117
svaria_SampleProd	40	10^7	0	-	t = 8	Sub 6-7	Sub 6-44	Sub 6-81	Sub 6-118
svaria_SampleProd	20	10^7	0	-	t = 16	Sub 6-8	Sub 6-45	Sub 6-82	Sub 6-119
svaria_SampleProd	20	10^7	0	-	t = 24	Sub 6-9	Sub 6-46	Sub 6-83	Sub 6-120
svaria_SampleMean	$2x10^7$	30	0	-	-	Sub 6-10	Sub 6-47	Sub 6-84	Sub 6-121
svaria_SampleMean	$2x10^7$	30	10	-	-	Sub 6-11	Sub 6-48	Sub 6-85	Sub 6-122
svaria_SampleCorr	1	$2x10^9$	0	-	k = 1	0.59	0.32	0.35	0.31
svaria_SampleCorr	1	$2x10^9$	0	-	k = 2	0.84	0.75	0.55	0.08
svaria_Appearancepsacings	1	-	0	3	Q=10^7,K=10^7 , L=15	0.58	0.11	0.12	0.34
svaria_Appearancepsacings	1	-	27	3	Q=10^7,K=10^7, L=15	0.91	0.45	0.37	0.15
svaria_WeightDistrib	1	$2x10^7$	0	-	Alpha = 0,Beta = -1/4, k=256	7.0e-3	0.65	0.03	0.77
svaria_WeightDistrib	1	$2x10^7$	20	-	Alpha = 0,Beta = -1/4, k=256	0.08	0.07	1.7e-4	0.84
svaria_WeightDistrib	1	$2x10^7$	28	-	Alpha = 0,Beta = -1/4, k=256	0.01	0.10	1.5e-5	0.46
svaria_WeightDistrib	1	$2x10^7$	0	-	Alpha =0,Beta =-1/16, k=256	0.31	0.05	9.9e-6	0.46
svaria_WeightDistrib	1	$2x10^7$	10	-	Alpha =0,Beta =-1/16, k=256	0.39	8.4e-4	0.10	0.11

Table 10 – continued

Test function	N	n	r	s	Other parameters	Ph4x32	Tf2x32	LCG	MLFG
svaria_WeightDistrib	1	$2x10^7$	26	-	Alpha =0,Beta =-1/16, k=256	0.03	2.3e-4	0.02	0.87
svaria_SumCollector	1	$5x10^8$	0	-	g = 10	3.6e-10	2.3e-4	eps	0.54
smarsa_MatrixRank	10	10^6	0	5	L = k = 30	Sub 6-12	Sub 6-49	Sub 6-86	Sub 6-123
smarsa_MatrixRank	10	10^6	25	5	L = k = 30	Sub 6-13	Sub 6-50	Sub 6-87	Sub 6-124
smarsa_MatrixRank	1	5000	0	4	L = k = 1000	Sub 6-14	Sub 6-51	Sub 6-88	Sub 6-125
smarsa_MatrixRank	1	5000	26	4	L = k = 1000	Sub 6-15	Sub 6-52	Sub 6-89	Sub 6-126
smarsa_MatrixRank	1	80	15	15	L = k = 5000	Sub 6-16	Sub 6-53	Sub 6-90	Sub 6-127
smarsa_MatrixRank	1	80	0	30	L = k = 5000	Sub 6-17	Sub 6-54	Sub 6-91	Sub 6-128
smarsa_Savir2	10	10^7	10	-	m = 10^7, t = 30	Sub 6-18	Sub 6-55	Sub 6-92	Sub 6-129
smarsa_GCD	10	$5x10^7$	0	30	-	Sub 6-19	Sub 6-56	Sub 6-93	Sub 6-130
swalk_RandomWalk1	1	10^8	0	5	$L_0 = L_1 = 50$	Sub 7-1	Sub 7-7	Sub 7-13	Sub 7-19
swalk_RandomWalk1	1	10^8	25	5	$L_0 = L_1 = 50$	Sub 7-2	Sub 7-8	Sub 7-14	Sub 7-20
swalk_RandomWalk1	1	10^7	0	10	$L_0 = L_1 = 1000$	Sub 7-3	Sub 7-9	Sub 7-15	Sub 7-21
swalk_RandomWalk1	1	10^7	20	10	$L_0 = L_1 = 1000$	Sub 7-4	Sub 7-10	Sub 7-16	Sub 7-22
swalk_RandomWalk1	1	10^6	15	15	$L_0 = L_1 = 10000$	Sub 7-5	Sub 7-11	Sub 7-17	Sub 7-23
swalk_RandomWalk1	1	10^6	15	15	$L_0 = L_1 = 10000$	Sub 7-6	Sub 7-12	Sub 7-18	Sub 7-24
scomp_LinearComp	1	$4x10^5$	0	1	-	Sub 8-1	Sub 8-7	Sub 8-13	Sub 8-19
scomp_LinearComp	1	$4x10^5$	29	1	-	Sub 8-2	Sub 8-8	Sub 8-14	Sub 8-20
scomp_LempelZiv	10	-	0	30	k = 27	Sub 6-20	Sub 6-57	Sub 6-94	Sub 6-131
scomp_LempelZiv	10	-	15	15	k = 27	Sub 6-21	Sub 6-58	Sub 6-95	Sub 6-132
sspectral_Fourier3	10^5	-	0	3	k = 14	Sub 6-22	Sub 6-59	Sub 6-96	Sub 6-133
sspectral_Fourier3	10^5	-	27	3	k = 14	Sub 6-23	Sub 6-60	Sub 6-97	Sub 6-134
sstring_LongestHeadRun	1	1000	0	3	L = 10^7	Sub 8-3	Sub 8-9	Sub 8-15	Sub 8-21
sstring_LongestHeadRun	1	1000	27	3	L = 10^7	Sub 8-4	Sub 8-10	Sub 8-16	Sub 8-22
sstring_PeriodsInStrings	10	$5x10^8$	0	10	-	Sub 6-24	Sub 6-61	Sub 6-98	Sub 6-135
sstring_PeriodsInStrings	10	$5x10^8$	20	10	-	Sub 6-25	Sub 6-62	Sub 6-99	Sub 6-136
sstring_HammingWeight2	10	10^9	0	3	L = 10^8	Sub 6-26	Sub 6-63	Sub 6-100	Sub 6-137
sstring_HammingWeight2	10	10^9	27	3	L = 10^8	Sub 6-27	Sub 6-64	Sub 6-101	Sub 6-138
sstring_HammingCorr	1	10^9	10	10	L = 30	0.02	0.15	0.04	0.70
sstring_HammingCorr	1	10^9	10	10	L = 300	0.90	0.43	0.56	0.93
sstring_HammingCorr	1	10^9	10	10	L = 1200	0.57	0.46	0.50	0.02
sstring_HammingIndep	10	$3x10^7$	0	3	L = 30, d = 0	Sub 6-28	Sub 6-65	Sub 6-102	Sub 6-139
sstring_HammingIndep	10	$3x10^7$	27	3	L = 30, d = 0	Sub 6-29	Sub 6-66	Sub 6-103	Sub 6-140
sstring_HammingIndep	1	$3x10^7$	0	4	L = 300, d = 0	Sub 6-30	Sub 6-67	Sub 6-104	Sub 6-141
sstring_HammingIndep	1	$3x10^7$	26	4	L = 300, d = 0	Sub 6-31	Sub 6-68	Sub 6-105	Sub 6-142
sstring_HammingIndep	1	10^7	0	5	L = 1200, d = 0	Sub 6-32	Sub 6-69	Sub 6-106	Sub 6-143
sstring_HammingIndep	1	10^7	25	5	L = 1200, d = 0	Sub 6-33	Sub 6-70	Sub 6-107	Sub 6-144
sstring_Run	1	$2x10^9$	0	3	-	Sub 8-5	Sub 8-11	Sub 8-17	Sub 8-23
sstring_Run	1	$2x10^9$	27	3	-	Sub 8-6	Sub 8-12	Sub 8-18	Sub 8-24
sstring_AutoCor	10	10^9	0	3	d = 1	Sub 6-34	Sub 6-71	Sub 6-108	Sub 6-145
sstring_AutoCor	10	10^9	0	3	d = 3	Sub 6-35	Sub 6-72	Sub 6-109	Sub 6-146
sstring_AutoCor	10	10^9	27	3	d = 1	Sub 6-36	Sub 6-73	Sub 6-110	Sub 6-147

Table 10 – continued

Test function	N	n	r	s	Other parameters	Ph4x32	Tf2x32	LCG	MLFG
sstring_AutoCor	10	10^9	27	3	d = 3	Sub 6-37	Sub 6-74	Sub 6-111	Sub 6-148

Sub-table 5

Lable	Testing name	AD	A2	Nm	Y	AD(mNP2)	AD(mnp2-S)
1	snpair_ClosePairs	0.53	0.79	0.01	0.88	0.80	0.03
2	snpair_ClosePairs	0.41	0.39	0.64	0.74	0.71	0.85
3	snpair_ClosePairs	0.88	0.46	0.28	0.93	0.25	0.32
4	snpair_ClosePairs	0.65	0.92	0.27	0.32	0.59	0.16
5	snpair_ClosePairs	0.28	0.22	0.81	0.31	0.43	0.49
6	snpair_ClosePairs	0.20	0.28	0.53	0.30	0.39	0.89
7	snpair_ClosePairs	0.56	0.35	0.86	0.48	0.66	0.40
8	snpair_ClosePairs	0.99	0.22	0.78	0.27	0.67	0.33
9	snpair_ClosePairs	eps	eps	eps	1-eps1	-	-
10	snpair_ClosePairs	eps	eps	eps	1-eps1	-	-
11	snpair_ClosePairs	3.2e-157	3.2e-157	eps	1-eps1	-	-
12	snpair_ClosePairs	1.8e-79	1.8e-79	eps	1-eps1	-	-
13	snpair_ClosePairs	0.14	0.82	0.13	0.90	0.20	0.66
14	snpair_ClosePairs	0.65	0.36	0.03	0.94	0.97	0.25
15	snpair_ClosePairs	0.56	0.07	5.1e-3	0.89	0.09	0.61
16	snpair_ClosePairs	0.88	0.95	0.11	0.10	0.24	0.73

Sub-table 6

Lable	Test name	K – S statistic				K – S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
1	sknuth_Run	0.22	0.79	0.62	0.80	-	-	-	-
2	sknuth_Run	0.59	0.13	0.40	0.17	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			-
		D+	D-	A2	χ^2	D+	D-	A2	
3	sknuth_MaxOft	0.20	0.87	0.73	0.77	0.18	0.98	0.33	-
4	sknuth_MaxOft	0.31	0.53	0.70	0.36	0.83	0.66	0.9945	-
5	sknuth_MaxOft	0.13	0.52	0.20	0.83	0.51	0.17	0.42	-
6	sknuth_MaxOft	0.67	0.25	0.60	0.24	0.40	0.78	0.67	-
		End of MaxOft test							
7	svaria_SampleProd	0.46	0.11	0.30	-	-	-	-	-
8	svaria_SampleProd	0.74	0.16	0.35	-	-	-	-	-
9	svaria_SampleProd	0.34	0.73	0.90	-	-	-	-	-
10	svaria_SampleMean	0.42	0.47	0.67	-	-	-	-	-

Sub-table 6 – continued

Lable	Test name	K – S statistic				K – S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
12	smarsa_MatrixRank	0.59	0.58	0.76	0.19	-	-	-	-
13	smarsa_MatrixRank	0.33	0.42	0.51	0.77	-	-	-	-
14	smarsa_MatrixRank	-	-	-	0.68	-	-	-	-
15	smarsa_MatrixRank	-	-	-	0.92	-	-	-	-
16	smarsa_MatrixRank	-	-	-	0.81	-	-	-	-
17	smarsa_MatrixRank	-	-	-	0.03	-	-	-	-
18	smarsa_Savir2	0.15	0.80	0.20	0.73	-	-	-	-
19	smarsa_GCD	0.91	1.1e-3	0.01	0.01	-	-	-	-
20	scomp_LempelZiv	0.09	0.71	0.22	0.90	-	-	-	0.76
21	scomp_LempelZiv	0.58	0.26	0.76	0.45	-	-	-	0.58
22	sspectral_Fourier3	0.21	0.58	0.69	-	-	-	-	-
23	sspectral_Fourier3	0.13	0.83	0.22	-	-	-	-	-
24	sstring_PeriodsInStrings	0.89	0.19	0.40	0.18	-	-	-	-
25	sstring_PeriodsInStrings	0.82	0.46	0.75	0.28	-	-	-	-
26	sstring_HammingWeight2	0.76	0.55	0.99	0.45	-	-	-	-
27	sstring_HammingWeight2	0.05	0.87	0.18	0.93	-	-	-	-
28	sstring_HammingIndep	0.9913	8.1e-3	0.03	0.01	-	-	-	-
29	sstring_HammingIndep	0.47	0.18	0.38	0.57	-	-	-	-
30	sstring_HammingIndep	-	-	-	0.06	-	-	-	-
31	sstring_HammingIndep	-	-	-	0.02	-	-	-	-
32	sstring_HammingIndep	-	-	-	0.36	-	-	-	-
33	sstring_HammingIndep	-	-	-	0.79	-	-	-	-
34	sstring_AutoCor	0.59	0.56	0.96	0.36	-	-	-	0.34
35	sstring_AutoCor	0.31	0.50	0.38	0.59	-	-	-	0.03
36	sstring_AutoCor	0.82	0.15	0.32	0.13	-	-	-	0.77
37	sstring_AutoCor	0.18	0.97	0.20	0.94	-	-	-	0.77
38	sknuth_Run	0.50	0.20	0.56	0.42	-	-	-	-
39	sknuth_Run	0.35	0.67	0.88	0.47	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			-
		D+	D-	A2	χ^2	D+	D-	A2	
40	sknuth_MaxOft	0.21	0.85	0.23	0.93	0.43	0.53	0.63	-
41	sknuth_MaxOft	0.95	0.12	0.14	0.04	0.50	0.08	0.26	-
42	sknuth_MaxOft	0.50	0.13	0.17	0.40	0.79	0.39	0.75	-
43	sknuth_MaxOft	0.51	0.74	0.73	0.76	0.41	0.56	0.76	-
		End of MaxOft test							-
44	svaria_SampleProd	0.05	0.9920	0.03	-	-	-	-	-
45	svaria_SampleProd	0.33	0.69	0.67	-	-	-	-	-
46	svaria_SampleProd	0.72	0.58	0.97	-	-	-	-	-
47	svaria_SampleMean	0.22	0.11	0.24	-	-	-	-	-
48	svaria_SampleMean	0.67	0.31	0.56	-	-	-	-	-

Sub-table 6 – continued

Lable	Test name	K – S statistic				K – S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
49	smarsa_MatrixRank	0.74	0.06	0.19	0.18	-	-	-	-
51	smarsa_MatrixRank	-	-	-	0.75	-	-	-	-
52	smarsa_MatrixRank	-	-	-	0.40	-	-	-	-
53	smarsa_MatrixRank	-	-	-	0.59	-	-	-	-
54	smarsa_MatrixRank	-	-	-	0.87	-	-	-	-
55	smarsa_Savir2	0.67	0.52	0.98	0.49	-	-	-	-
56	smarsa_GCD	0.03	0.96	0.04	0.98	-	-	-	-
57	scomp_LempelZiv	0.21	0.91	0.44	0.86	-	-	-	0.76
58	scomp_LempelZiv	0.43	0.82	0.68	0.75	-	-	-	0.31
59	sspectral_Fourier3	8.2e-3	0.79	0.06	-	-	-	-	-
60	sspectral_Fourier3	0.36	0.38	0.51	-	-	-	-	-
61	sstring_PeriodsInStrings	0.12	0.41	0.46	0.66	-	-	-	-
62	sstring_PeriodsInStrings	0.41	0.59	0.78	0.54	-	-	-	-
63	sstring_HammingWeight2	0.93	0.12	0.26	0.06	-	-	-	-
64	sstring_HammingWeight2	0.21	0.83	0.61	0.72	-	-	-	-
65	sstring_HammingIndep	0.90	0.41	0.88	0.29	-	-	-	-
66	sstring_HammingIndep	0.35	0.86	0.78	0.77	-	-	-	-
67	sstring_HammingIndep	-	-	-	0.83	-	-	-	-
68	sstring_HammingIndep	-	-	-	0.29	-	-	-	-
69	sstring_HammingIndep	-	-	-	0.65	-	-	-	-
70	sstring_HammingIndep	-	-	-	0.32	-	-	-	-
71	sstring_AutoCor	0.73	0.28	0.63	0.19	-	-	-	0.63
72	sstring_AutoCor	0.15	0.89	0.55	0.80	-	-	-	0.67
73	sstring_AutoCor	0.34	0.13	0.28	0.26	-	-	-	0.97
74	sstring_AutoCor	0.22	0.76	0.34	0.86	-	-	-	0.31
75	sknuth_Run	2.5e-3	0.97	1.7e-3	0.9993	-	-	-	-
76	sknuth_Run	1.0e-7	0.9921	4.9e-7	1-1.3e-7	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			-
		D+	D-	A2	χ^2	D+	D-	A2	
77	sknuth_MaxOft	1.0e-97	1-2.3e-13	3.4e-205	1-eps1	0.98	0.18	0.08	-
78	sknuth_MaxOft	eps	1-eps1	eps	1-eps1	0.90	0.01	0.12	-
79	sknuth_MaxOft	eps	1-eps1	eps	1-eps1	0.9927	0.02	2.2e-3	-
80	sknuth_MaxOft	eps	1-eps1	eps	1-eps1	0.9959	3.6e-4	308e-4	-
	End of MaxOft test								
81	svaria_SampleProd	0.95	1.0e-3	6.3e-3	-	-	-	-	-
82	svaria_SampleProd	0.96	0.01	0.01	-	-	-	-	-
83	svaria_SampleProd	0.98	2.6e-3	0.02	-	-	-	-	-
84	svaria_SampleMean	0.68	0.24	0.48	-	-	-	-	-
85	svaria_SampleMean	0.93	0.33	0.78	-	-	-	-	-
86	smarsa_MatrixRank	0.78	0.28	0.81	0.40	-	-	-	-
87	smarsa_MatrixRank	0.23	0.62	0.70	0.71	-	-	-	-

Sub-table 6 – continued

Lable	Test name	K – S statistic				K – S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
88	smarsa_MatrixRank	-	-	-	0.54	-	-	-	-
89	smarsa_MatrixRank	-	-	-	0.77	-	-	-	-
91	smarsa_MatrixRank	-	-	-	0.29	-	-	-	-
92	smarsa_Savir2	0.53	0.34	0.61	0.45	-	-	-	-
93	smarsa_GCD	0.58	0.68	0.89	0.56	-	-	-	-
94	scomp_LempelZiv	0.94	0.08	0.05	0.02	-	-	-	0.49
95	scomp_LempelZiv	0.97	0.12	0.11	0.03	-	-	-	0.50
96	sspectral_Fourier3	1.4e-40	1.8e-44	3.9e-72	-	-	-	-	-
97	sspectral_Fourier3	1.6e-49	6.7e-48	1.9e-80	-	-	-	-	-
98	sstring_PeriodsInStrings	8.6e-12	0.9979	5.1e-12	1-5.9e-12	-	-	-	-
99	sstring_PeriodsInStrings	2.1e-14	0.9939	5.8e-13	1-2.2e-12	-	-	-	-
100	sstring_HammingWeight2	0.56	0.31	0.67	0.41	-	-	-	-
101	sstring_HammingWeight2	0.60	0.38	0.74	0.62	-	-	-	-
102	sstring_HammingIndep	0.38	0.76	0.81	0.64	-	-	-	-
103	sstring_HammingIndep	0.57	0.67	0.96	0.61	-	-	-	-
104	sstring_HammingIndep	-	-	-	0.27	-	-	-	-
105	sstring_HammingIndep	-	-	-	0.89	-	-	-	-
106	sstring_HammingIndep	-	-	-	0.99	-	-	-	-
107	sstring_HammingIndep	-	-	-	0.32	-	-	-	-
108	sstring_AutoCor	0.34	0.35	0.46	0.57	-	-	-	0.96
109	sstring_AutoCor	0.34	0.57	0.74	0.48	-	-	-	0.83
110	sstring_AutoCor	0.44	0.22	0.41	0.40	-	-	-	0.98
111	sstring_AutoCor	0.52	0.56	0.89	0.61	-	-	-	0.76
112	sknuth_Run	0.77	0.44	0.82	0.27	-	-	-	-
113	sknuth_Run	0.77	0.37	0.80	0.31	-	-	-	-
		χ^2 with 99999 degrees of freedom				Anderson-Darling test			-
		D+	D-	A2	χ^2	D+	D-	A2	
114	sknuth_MaxOft	0.31	0.84	0.77	0.69	0.16	0.35	0.24	-
115	sknuth_MaxOft	0.96	0.29	0.44	0.12	0.38	0.41	0.60	-
116	sknuth_MaxOft	0.72	0.16	0.64	0.48	0.56	0.67	0.92	-
117	sknuth_MaxOft	0.61	0.71	0.83	0.55	0.32	0.83	0.86	-
	End of MaxOft test								
118	svaria_SampleProd	0.37	0.51	0.66	-	-	-	-	-
119	svaria_SampleProd	0.54	0.53	0.84	-	-	-	-	-
120	svaria_SampleProd	0.17	0.74	0.19	-	-	-	-	-
121	svaria_SampleMean	0.39	0.56	0.80	-	-	-	-	-
122	svaria_SampleMean	0.78	0.48	0.69	-	-	-	-	-
123	smarsa_MatrixRank	0.39	0.82	0.84	0.79	-	-	-	-
124	smarsa_MatrixRank	0.94	0.39	0.84	0.31	-	-	-	-
125	smarsa_MatrixRank	-	-	-	0.70	-	-	-	-
126	smarsa_MatrixRank	-	-	-	0.79	-	-	-	-

Sub-table 6 – continued

Lable	Test name	K − S statistic				K − S statistic			Sample Variance
		D+	D-	A2	χ^2	-	-	-	
127	smarsa_MatrixRank	-	-	-	0.74	-	-	-	-
128	smarsa_MatrixRank	-	-	-	0.96	-	-	-	-
129	smarsa_Savir2	0.65	0.50	0.87	0.34	-	-	-	-
131	scomp_LempelZiv	0.93	0.02	0.21	0.12	-	-	-	0.82
132	scomp_LempelZiv	0.12	0.98	0.10	0.97	-	-	-	0.69
133	sspectral_Fourier3	0.54	0.61	0.90	-	-	-	-	-
134	sspectral_Fourier3	0.89	0.10	0.14	-	-	-	-	-
135	sstring_PeriodsInStrings	0.89	0.35	0.55	0.19	-	-	-	-
136	sstring_PeriodsInStrings	3.7e-3	0.91	0.01	0.9917	-	-	-	-
137	sstring_HammingWeight2	0.08	0.92	0.21	0.93	-	-	-	-
138	sstring_HammingWeight2	0.97	0.14	0.20	0.06	-	-	-	-
139	sstring_HammingIndep	0.75	0.52	0.87	0.41	-	-	-	-
140	sstring_HammingIndep	0.47	0.46	0.78	0.43	-	-	-	-
141	sstring_HammingIndep	-	-	-	0.84	-	-	-	-
142	sstring_HammingIndep	-	-	-	0.29	-	-	-	-
143	sstring_HammingIndep	-	-	-	0.35	-	-	-	-
144	sstring_HammingIndep	-	-	-	0.55	-	-	-	-
145	sstring_AutoCor	0.15	0.89	0.20	0.94	-	-	-	0.56
146	sstring_AutoCor	0.94	0.14	0.20	0.08	-	-	-	0.15
147	sstring_AutoCor	0.10	0.60	0.48	0.49	-	-	-	0.34
148	sstring_AutoCor	0.78	0.01	0.04	0.04	-	-	-	0.98

Sub-table 7

Lable	Test name	Test on the value of the statistic				
		H	M	J	R	C
1	swalk_RandomWalk1	0.06	0.86	0.98	0.72	0.87
2	swalk_RandomWalk1	0.36	0.36	0.79	0.62	0.58
3	swalk_RandomWalk1	0.75	0.29	0.25	0.21	0.26
4	swalk_RandomWalk1	0.39	0.04	0.39	0.43	0.04
5	swalk_RandomWalk1	0.36	0.67	0.09	0.04	0.71
6	swalk_RandomWalk1	0.08	0.19	6.6e-3	0.62	0.17
7	swalk_RandomWalk1	0.65	0.59	0.09	0.15	0.56
8	swalk_RandomWalk1	0.17	0.52	0.67	0.20	0.25
9	swalk_RandomWalk1	0.02	0.34	0.77	0.87	0.19
10	swalk_RandomWalk1	0.86	0.26	0.09	0.11	0.53
11	swalk_RandomWalk1	0.59	0.91	0.30	0.13	0.10
12	swalk_RandomWalk1	0.29	0.69	0.64	0.19	0.81
13	swalk_RandomWalk1	0.71	0.19	0.45	0.77	0.70
14	swalk_RandomWalk1	0.03	0.19	0.65	0.03	0.30
15	swalk_RandomWalk1	0.69	0.73	0.46	0.91	0.88
16	swalk_RandomWalk1	0.75	0.73	0.09	0.54	0.41
17	swalk_RandomWalk1	0.12	0.79	0.32	0.25	0.96
18	swalk_RandomWalk1	0.73	0.19	0.78	0.16	0.12
19	swalk_RandomWalk1	0.52	0.51	0.85	0.74	0.44
20	swalk_RandomWalk1	0.35	0.71	0.40	0.87	0.95
21	swalk_RandomWalk1	0.91	0.27	0.15	0.83	0.44
22	swalk_RandomWalk1	0.48	0.42	0.09	0.59	0.65
23	swalk_RandomWalk1	0.64	0.71	0.92	0.50	0.92
24	swalk_RandomWalk1	0.39	2.5e-3	0.21	0.77	0.25

Sub-table 8

Lable	Test name	Statistic	
		χ^2	Normal
1	scomp_LinearComp	0.29	0.22
2	scomp_LinearComp	0.55	0.34
3	sstring_LongestHeadRun	0.55	0.90
4	sstring_LongestHeadRun	0.97	0.69
5	sstring_Run	0.90	0.63
6	sstring_Run	0.58	0.77
7	scomp_LinearComp	0.15	0.52
8	scomp_LinearComp	0.48	0.19
9	sstring_LongestHeadRun	0.77	0.50
10	sstring_LongestHeadRun	0.08	0.44
11	sstring_Run	0.09	0.84
12	sstring_Run	0.12	0.59
13	scomp_LinearComp	0.47	0.74
14	scomp_LinearComp	0.62	0.17
15	sstring_LongestHeadRun	0.65	0.69
16	sstring_LongestHeadRun	0.52	0.69
17	sstring_Run	1-6.3e-5	0.45
18	sstring_Run	0.9997	0.51
13	scomp_LinearComp	0.94	0.35
14	scomp_LinearComp	0.35	0.99
15	sstring_LongestHeadRun	0.25	0.25
16	sstring_LongestHeadRun	0.55	0.02
17	sstring_Run	0.60	0.57
18	sstring_Run	0.58	0.18

REFERENCES

[1] A. Rukhin, J. Soto, J. Nechvatal: "A Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications." *NIST Special Publication 800-22* (May 15, 2011).

[2] *"NIST/SEMATECH e-Handbook of Statistical Methods*." *http://itl.nist.gov/div898/handbook/prc/section1/prc13.htm*. http://www.itl.nist.gov/div898/handbook.

[3] M.Mascagni, A.Srinivasan; "Algorithm 806: SPRNG: a scalable library for pseudorandom number generation." *ACM Transactions on Mathematical Software* (September, 2000): Volume 26 Issue 3.

[4] Information Technology Laboratory, National Institute of Standards and Technology. "FIPS PUB 140-2, Security Requirement For Cryptographic Modules." (December 3, 2002).

[5] Donald E. Knuth. "The Art of Computer Programming, Seminumerical Algorithms." (1997). Volume2, 3rd Edition.

[6] D.V.Boulatov, V.A.Kazakov. "The ising model on a random planar lattice: The structure of the phase transition and the exact critical exponents." *Physics Letter B, Volume 186, Issues 3-4* (Mar 12, 1987): 379-384.

[7] I.Vattulainen, T.Ala-Nissila, K.Kankaala. "Physical Models as Tests of Randomness." *Physical Review E* (1995). *adsabs.harvard.edu*.

[8] A. Menezes, P. Van Oorschot and S. Vanstone. "The Handbook of Applied Cryptography." (1997).

[9] Paul D.Coddington, Sung-Hoon Ko. "Technique for Empirical Testing of Parallel Random Number Generators." (January 31, 1998).

[10] Paul D.Coddington. "Random Number Generators for Parallel Computers." *The NHSE Review, http://nhse.cs.rice.edu/NHSEreview/, 1996 Volume, Second Issue*.

[11] P. A. W. Lewis, A. S. Goodman, and J. M. Miller. "A pseudo-random number generator for the system/360." *IBM System's Journal, 8* (1969): 136-143.

[12] P. L'Ecuyer, F. Blouin, and R. Couture. "A search for good multiple recursive random number generators." *ACM Transactions on Modeling and Computer Simulation, 3(2)* (1993): 87-98.

[13] G. Marsaglia. "Xorshift RNGs." *Journal of Statistical Software, 8(14)* (2003): 1-6.

[14] R. P. Brent. “Note on Marsaglia’s xorshift random number generators.” *Journal of statistical software, 11(5)* (2004): 1-4. 17.

[15] D. E. Knuth. “The Art of Computer Programming, Volume 2: Seminumerical Algorithms.” Addison-Wesley, Reading, Mass, third edition (1998).

[16] P. L’Ecuyer. “Efficient and portable 32-bit random variate generators.” *Proceedings of the 1986 Winter Simulation Conference* (1986): 275-277.

[17] P. L’Ecuyer. “Combined multiple recursive random number generators.” *Operations Research*, *44(5)* (1996): 816-822.

[18] P. L’Ecuyer. “Good parameters and implementations for combined multiple recursive random number generators.” *Operations Research*, *47(1)* (1999): 159-164.

[19] G. Marsaglia. “A current view of random number generators.” In Computer Science and Statistics, *Sixteenth Symposium on the Interface* (1985): 3-10.

[20] G. Marsaglia. ”DIEHARD: a battery of tests of randomness.” *http://stat.fsu.edu/~geo/diehard.html* (1996).

[21] G. Marsaglia and W. W. Tsang. “Some difficult-to-pass tests of randomness.” *Journal of Statistical Software*, *7(3)* (2002): 1-9.

[22] Badger, Lee (April 1994). "Lazarini's Lucky Approximation of π". *Mathematics Magazine* (Mathematical Association of America) **67** (2): 83–91.

[23] Dell, Zachary; Franklin, Scott V. (September 2009). "The Buffon-Laplace needle problem in three dimensions". *Journal of Statistical Mechanics: Theory and Experiment* 09 (09): 010. Bibcode 2009JSMTE..09..010D

[24] F. Panneton and P. L’Ecuyer. “On the xorshift random number generators”. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361.

[25] M Mascagni, A Srinivasan. “Parameterizing parallel multiplicative lagged-Fibonacci generators”. *Parallel Computing, Vol 30, Issue 7* (July 2004): 899-916.

[26] Pal Revesz, “Random Walk in Random and Non-Random Environments”. *Singapore: World Scientific*, 1990.

[27] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone**.** “Handbook of applied cryptography (Fifth painting)”. *CRC Press* (Aug, 2001).

[28] John K. Salmon,Mark A. Moraes, Ron O. Dror, and David E. Shaw “Parallel Random Numbers: As Easy as 1, 2, 3”. *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011). pp. 16:1–16:12.

[29] Takeshi SAITO1, Koichi ISHII2, Isao TATSUNO2, Susumu SUKAGAWA2 and Tomotake YANAGITA1, “Randomness and Genuine Random Number Generator With Self-testing Functions”. *IEEE 10th International Symposium on Applied Machine Intelligence and Informatics*, (Jan, 2012). Page 213.

BIOGRAPHICAL SKETCH

My name is Liang Li. I was born in August 24th, 1982, and grew up in Shanxi Provence, China. In my 18 years old, I was enrolled by Jilin University, which is one of the top ten universities in China. During the undergraduate, I was major in Computer Science, and obtained my B.C. degree in 2004. Then I got a job in China Yellow River TV Station and kept on working there as a system maintenance staff until 2009.

In 2010 spring semester, I was enrolled by Computer Science department in Florida State University. So far, I'm still pursuing my Master degree. Within less than two years, my main interest of research is on random number generation and tests. Recently, I'm mainly focusing on testing properties of parallel random number generators.