

Institutionen för systemteknik

Department of Electrical Engineering

Examensarbete

Theory, Methods and Tools for Statistical Testing of Pseudo and Quantum Random Number Generators

Examensarbete utfört i tillämpad matematik
vid Tekniska högskolan vid Linköpings universitet
av

Krister Sune Jakobsson

LiTH-ISY-EX-14/4790-SE

Linköping 2014



Linköpings universitet
TEKNISKA HÖGSKOLAN

Theory, Methods and Tools for Statistical Testing of Pseudo and Quantum Random Number Generators

Examensarbete utfört i tillämpad matematik
vid Tekniska högskolan vid Linköpings universitet
av


Krister Sune Jakobsson

LiTH-ISY-EX-14/4790-SE

Handledare: **Jonathan Jogenfors**
isy, Linköpings universitet

Examinator: **Jan-Åke Larsson**
isy, Linköpings universitet

Linköping, 15 augusti 2014

	Avdelning, Institution Division, Department Information Coding (ICG) Department of Electrical Engineering SE-581 83 Linköping	Datum Date 2014-08-15
---	--	--

Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LiTH-ISY-EX-14/4790-SE <table style="width: 100%;"> <tr> <td style="width: 60%;">Serietitel och serienummer</td> <td style="width: 40%;">ISSN</td> </tr> <tr> <td>Title of series, numbering</td> <td>_____</td> </tr> </table>	Serietitel och serienummer	ISSN	Title of series, numbering	_____
Serietitel och serienummer	ISSN					
Title of series, numbering	_____					

URL för elektronisk version http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-XXXXX	
---	--

Titel Title	Teori, metodik och verktyg för utvärdering av pseudo- och kvant slumpgeneratorer Theory, Methods and Tools for Statistical Testing of Pseudo and Quantum Random Number Generators
Författare Author	Krister Sune Jakobsson

Sammanfattning Abstract	<p>Statistical random number testing is a well studied field focusing on pseudo-random number generators, that is to say algorithms that produce random-looking sequences of numbers. These generators tend to have certain kinds of flaws, which have been exploited through rigorous testing. Such testing has led to advancements, and today pseudo random number generators are both very high-speed and produce seemingly random numbers. Recent advancements in quantum physics have opened up new doors, where products called quantum random number generators that produce acclaimed true randomness have emerged.</p> <p>Of course, scientists want to test such randomness, and turn to the old tests used for pseudo random number generators to do this. The main question this thesis seeks to answer is if publicly available such tests are good enough to evaluate a quantum random number generator. We also seek to compare sequences from such generators with those produced by state of the art pseudo random number generators, in an attempt to compare their quality.</p> <p>Another potential problem with quantum random number generators is the possibility of them breaking without the user knowing. Such a breakdown could have dire consequences. For example, if such a generator were to control the output of a slot machine, a malfunction could cause the machine to generate double earnings for a player compared to what was planned. Thus, we look at the possibilities to implement live tests to quantum random number generators, and propose such tests.</p> <p>Our study has covered six commonly available tools for random number testing, and we show that in particular one of these stands out in that it has a series of tests that fail our quantum random number generator as not random enough, despite passing an pseudo random number generator. This implies that the quantum random number generator behave differently from the pseudo random number ones, and that we need to think carefully about how we test, what we expect from an random sequence and what we want to use it for.</p>
-----------------------------------	---

Nyckelord Keywords	random number testing, quantum random number generator, pseudo-random number generator, randomness, random number theory, random number testing
------------------------------	---

Abstract

Statistical random number testing is a well studied field focusing on pseudo-random number generators, that is to say algorithms that produce random-looking sequences of numbers. These generators tend to have certain kinds of flaws, which have been exploited through rigorous testing. Such testing has led to advancements, and today pseudo random number generators are both very high-speed and produce seemingly random numbers. Recent advancements in quantum physics have opened up new doors, where products called quantum random number generators that produce acclaimed true randomness have emerged.

Of course, scientists want to test such randomness, and turn to the old tests used for pseudo random number generators to do this. The main question this thesis seeks to answer is if publicly available such tests are good enough to evaluate a quantum random number generator. We also seek to compare sequences from such generators with those produced by state of the art pseudo random number generators, in an attempt to compare their quality.

Another potential problem with quantum random number generators is the possibility of them breaking without the user knowing. Such a breakdown could have dire consequences. For example, if such a generator were to control the output of a slot machine, an malfunction could cause the machine to generate double earnings for a player compared to what was planned. Thus, we look at the possibilities to implement live tests to quantum random number generators, and propose such tests.

Our study has covered six commonly available tools for random number testing, and we show that in particular one of these stands out in that it has a series of tests that fail our quantum random number generator as not random enough, despite passing an pseudo random number generator. This implies that the quantum random number generator behave differently from the pseudo random number ones, and that we need to think carefully about how we test, what we expect from an random sequence and what we want to use it for.

Sammanfattning

Statistisk slumpstalstestning är ett väl studerat ämne som fokuserar på så kallade pseudoslumpgeneratorer, det vill säga algoritmer som producerar slumpliknande sekvenser med tal. Sådana generatorer tenderar att ha vissa defekter, som har exploaterats genom rigorös testning. Sådan testning har lett till framsteg och idag är pseudoslumpgeneratorer både otroligt snabba och producerar till synes slumpade tal. Framsteg inom kvantfysiken har lett till utvecklingen av kvantslumpgeneratorer, som producerar vad som hävdas vara äkta slump.

Självklart vill forskare utvärdera sådan slump, och har då vänt sig till de gamla testerna som utvecklats för pseudoslumpgeneratorer. Den här uppsatsen söker utvärdera hurvida allmänt tillgängliga slumptester är nog bra för att utvärdera kvantslumpgeneratorer. Vi jämför även kvantslumpsekvenser med pseudoslumpsekvenser för att se om dessa väsentligen skiljer sig från varandra och på vilket sätt.

Ett annat potentiellt problem med kvantslumpgeneratorer är möjligheten att dessa går sönder under drift. Om till exempel en kvantslumpgenerator används för att slumpgenerera resultatet hos en enarmad bandit kan ett fel göra så att maskinen ger dubbel vinst för en spelare jämfört med planerat. Därmed ser vi över möjligheten att implementera live-tester i kvantslumpgeneratorer, och föreslår några sådana tester.

Vår studie har täckt sex allmänt tillgängliga verktyg för slumpstalstestning, och vi visar att i synnerhet ett av dessa står ut på så sätt att det har en serie av tester som slumpalen från vår kvantslumpgenerator inte anser är nog slumpade. Trots det visar samma test att sekvensen från pseudoslumpgeneratorerna är bra nog. Detta antyder att kvantslumpgeneratorn beter sig annorlunda mot pseudoslumpgeneratorerna, och att vi behöver tänka över ordentligt kring hur vi testar slumpgeneratorer, vad vi förväntar oss att få ut och hurvida detta påverkar det vi skall använda slumpgeneratorn till.

Acknowledgments

I would like to thank my supervisor Jonathan Jogenfors and my examiner Jan-Åke Larsson for their continuous support and valuable feedback. Without them this would not have been possible.

A great deal of gratitude to my family, who is always there for me and supports me. Without their support I would never have come this far. In particular to my brother, who is always open for discussing various topics in science with me.

A few inspirational words from Carl Sagan:

Somewhere something incredible is waiting to be known.

To sum it up, thanks to my family, girlfriend, dog, friends, pretty much everyone I ever met, everyone I will ever meet and, by extension, the whole universe. You guys are the best!

Linköping, August 2014
Krister Sune Jakobsson

Contents

Notation	xiii
1 Introduction	1
1.1 Historical Overview	1
1.2 Background	2
1.3 Purpose	3
1.4 Delimitations	3
1.5 Method	4
1.6 Structure	4
1.7 Contributions	4
2 Random Number Generators	7
2.1 Universal Turing Machine	7
2.2 Pseudo Random Number Generators	8
2.2.1 Characteristics	8
2.2.2 Mersenne Twister	9
2.2.3 Other Generators	9
2.3 True Random Number Generators	10
2.3.1 Quantum Indeterminacy	10
2.3.2 Quantum Random Number Generation	11
2.3.3 Post-processing	11
2.3.4 QUANTIS Quantum Random Number Generator	12
3 Random Number Testing Methods	13
3.1 Theory for Testing	13
3.1.1 Online and Offline Testing	13
3.1.2 Shannon Entropy	14
3.1.3 Central Limit Theorem	14
3.1.4 Hypothesis Testing	15
3.1.5 Single-level and Two-level Testing	16
3.1.6 Kolmogorov Complexity Theory	17
3.2 Goodness-of-fit Tests	21
3.2.1 Pearson's Chi-square Test	22

3.2.2	Multinomial Test	24
3.2.3	Kolmogorov-Smirnov Test	27
3.2.4	Anderson-Darling Test	28
3.2.5	Shapiro-Wilk Normality Test	29
3.3	Dense Cells Scenario	29
3.3.1	Frequency Test	29
3.3.2	Frequency Test within a Block	30
3.3.3	Non-Overlapping Serial Test	30
3.3.4	Permutation Test	31
3.3.5	Maximum-of-t Test	31
3.3.6	Overlapping Serial Test	31
3.3.7	Poker Test	32
3.3.8	Coupon Collector's Test	32
3.3.9	Gap Test	33
3.3.10	Runs Test	34
3.3.11	Bit Distribution Test	37
3.4	Monkey-related Tests	37
3.4.1	Collision Test	38
3.4.2	Bitstream Test	39
3.4.3	Monkey Typewriter Tests	41
3.4.4	Birthday Spacings Test	42
3.5	Other Statistical Tests	42
3.5.1	Serial Correlation Test	42
3.5.2	Rank of Random Binary Matrix Test	43
3.5.3	Random Walk Tests	43
3.5.4	Borel Normality Test	45
3.6	Spectral Tests	46
3.6.1	NIST Discrete Fourier Transform Test	46
3.6.2	Erdmann Discrete Fourier Transform Test	47
3.6.3	Discrete Fourier Transform Central Limit Theorem Test	47
3.7	Close-point Spatial Tests	47
3.8	Compressibility and Entropy Tests	48
3.8.1	Discrete Entropy Tests	48
3.8.2	Maurer's Universal Test	49
3.8.3	Linear Complexity Tests	49
3.8.4	Lempel-Ziv Compressibility Test	50
3.8.5	Run-length Entropy test	51
3.9	Simulation Based Tests	53
3.9.1	Craps Test	53
3.9.2	Pi Estimation Test	54
3.9.3	Ising Model Tests	56
3.10	Other Tests	57
3.10.1	Bit Persistence Test	58
3.10.2	GCD Test	58
3.10.3	Solovay-Strassen Probabilistic Primality Test	58
3.10.4	Book Stack Test	59

3.11 Live Tests	60
3.11.1 Sliding Window Testing	60
3.11.2 Convergence Tests	61
4 Random Number Test Suites	63
4.1 Diehard	63
4.1.1 Tests	64
4.2 NIST STS	65
4.2.1 Tests	66
4.3 ENT	67
4.3.1 Tests	67
4.4 SPRNG	68
4.4.1 Tests	68
4.5 TestU01	69
4.5.1 Tests	69
4.5.2 Batteries	70
4.5.3 Tools	71
4.6 DieHarder	71
4.6.1 Tests	73
5 Testing	75
5.1 Methods for Testing	75
5.2 Test Results	77
5.2.1 Plot Test	77
5.2.2 Bit Distribution Test	77
5.2.3 DieHarder Tests	77
5.2.4 Crush Battery	80
5.2.5 Alphabit Battery	80
5.2.6 Sliding Window Run-length Coding Test	81
6 Conclusion and Future Work	83
6.1 Conclusions	83
6.1.1 Random Number Testing	83
6.1.2 Quantis	84
6.1.3 Pseudo Random Number Generators	85
6.1.4 Test Suites	85
6.1.5 Live Testing	86
6.2 Future work	86
6.3 Summary	87
A Distribution functions	91
A.1 Discrete distributions	91
A.2 Continuous Distributions	92
B Mathematical Definitions	93
C Raw Result Data from Tests	95

C.1 DieHarder Tests	95
C.2 Crush Tests	110
C.3 Alphabit Tests	115
Bibliography	119

Notation

SOME SETS

Notation	Meaning
\mathbb{N}	Set of natural numbers
\mathbb{N}_+	Set of positive natural numbers
\mathbb{Z}	The set of integers
\mathbb{F}_2	The finite field of cardinality 2 with elements $\{0,1\}$
A^C	The complement of a set A
\mathcal{X}	Set of symbols in an alphabet

SAMPLES

Notation	Meaning
b_i	The i th sample bit with binary equidistribution
u_i	The i th sample uniform on $[0, 1)$
Y_i	The i th equidistributed sample integer on $0, \dots, k - 1$ for some integer k
X_i	Non-overlapping samples in the i th sample space
$X_i^{(o)}$	Overlapping samples in the i th sample space

DISTRIBUTION FUNCTIONS

Notation	Meaning
$U(a, b)$	Uniform distribution on the interval $[a, b)$
$N(\mu, \sigma^2)$	Normal distribution with mean μ and standard deviation σ^2
χ_k^2	The chi-square distribution with k degrees of freedom
Bernoulli(Θ)	Bernoulli distribution with parameter Θ
Po(λ)	Poisson distribution with parameter λ

TEST STATISTICS

Notation	Meaning
Y	General test statistic
X^2	Test statistic for χ^2 test
D_δ	The power divergence test statistic
G^2	The loglikelihood test statistic
$-H$	The negative entropy test statistic
N_b	Number of cells with exactly b points test statistic
W_b	Number of cells with at least b points test statistic
N_0	Number of empty cells test statistic
C	Number of collisions test statistic
K_n	Kolmogorov-Smirnov test statistic
A^2, A^{2*}	Original and modified Anderson-Darling test statistics
W	Shapiro-Wilk Normality test statistic

MATHEMATICAL FUNCTIONS

Notation	Meaning
$S(n, k)$	Stirling number of the second kind, see appendix B
$\ln(x)$	The natural logarithm of x
$\log_2(x)$	The 2-logarithm of x
$E(X)$	The expected value of X
$\text{Var}(X)$	The variance of X
$K_{\mathcal{U}}(x)$	The Kolmogorov complexity of a string x with respect to universal computer \mathcal{U}
$ f _{\text{mod}}$	The modulus operator of a function f
$ f $	The absolute value of a function f
$ A $	The determinant of a matrix A
$I(A)$	The indicator function to a set A
$\ X\ _p$	The L_p -norm of X , see appendix B
$a \equiv b(\text{mod } n)$	a and b congruent modulo n
$\Phi(x)$	The cumulative distribution function (CDF) of the standard normal distribution
$\lfloor x \rfloor$	The floor function for x
$\lceil x \rceil$	The ceil function for x
$\text{GCD}(a, b)$	The greatest common divisor of a and b
δ_{ij}	Delta function, equals to 1 if $i = j$, otherwise 0

OTHER NOTATION

Notation	Meaning
α	Significance level
H_0	The null hypothesis
H_1	The alternative hypothesis to H_0
$\neg H$	The inverse of a statement H
\xrightarrow{p}	...converges in probability to...
\xrightarrow{d}	...converges in distribution to...
\gtrless	Greater to, equal to or less than
$O(n)$	Ordo of order n

ABBREVIATION

Abbreviation	Meaning
XOR	Exclusive Or
RNG	Random Number Generator (in general)
PRNG	Pseudo Random Number Generator
TRNG	True Random Number Generator
HRNG	Hardware Random Number Generator
GCD	Greatest Common Divisor
DFT	Discrete Fourier Transform
CLT	Central Limit Theorem
NIST	National Institute of Standards and Technology
ANSI	American National Standards Institute
METAS	Federal Office of Metrology in Switzerland
OEM	Original Equipment Manufacturer
MWC	Multiply-with-carry
AES	Advanced Encryption Standard
OFB	Output Feedback Mode
OPSO	Overlapping Pairs Spatse Occupancy
OTSO	Overlapping Triplets Spatse Occupancy
OQSO	Overlapping Quadruplets Spatse Occupancy
DNA	Deoxyribonucleic Acid

1

Introduction

This chapter outlines the background for this paper, as well to what extent the study was performed.

1.1 Historical Overview

In 1927 Leonard H. C. Tippett published a table with 41600 digits taken at random with Cambridge University Press. It took 10 years to deem this sample inadequate for large sampling experiments, and around 1940 authors published various tables of random numbers based on spinning roulette wheels, lotteries, card decks and the likes. These methods all gave a certain bias, and to combat this the method known as *compounding bias* was developed by H. Burke Horton. Horton showed that random digits could be produced from sums of other random numbers, and that the compounding of the randomization process created a sequence with less bias than the original sequences. In other words, he showed that the exclusive or, XOR, of two binary sequences would be more random than either of the original ones. [1]

With the growing availability of computers the need for random numbers increased, especially with methods such as Monte Carlo growing more and more popular in different fields of research. As larger and larger amounts of random numbers were needed in the computers, the idea of generating random numbers on the computer seemed appealing. This gave rise to what is now known as Pseudo Random Number Generators (PRNG), which is the name of arithmetical operations that produce seemingly random numbers, called pseudo random numbers. Many methods for generating these were proposed from 1951 until present, often based on recursive functions of various levels of ingenuity. Of course, with

more complex ways of generating random numbers, the need for testing if the random number were truly random also grew. Throughout these past 60 years, new methods were thought of and readily regarded to be, in one way or another, not good enough.

1.2 Background

According to the *Oxford English Dictionary* the word random is defined as: [46]

Made, done, or happening without method or conscious decision.

This is not a mathematical definition, but it brings to light what people consider to be random. Consider a simple coin flip. Is the outcome of this operation random? The simple answer would be that it depends. First one has to consider if the person performing the flips can somehow predict the outcome. This is reflected in the definition above, and if the person can flip the coin with some method that lets him predict the outcome, then it is not random. To distinguish, we will refer to this kind of randomness as *unpredictability*. What is not reflected in unpredictability is the concept of *statistical randomness*, which concerns the distribution of sequences of supposedly random data. Lets say that we flip the coin a hundred times, and that the coin shows heads and tails with equal probability. We expect approximately the same number of heads as tails. A large deviation from this expectation would naturally make us doubt the assumption that the head is equally probable to the tail, even though any outcome of flips is equally probable to any other.

In many applications a good source of random numbers is needed, but what defines good might differ between applications. For example, when doing Monte Carlo simulations the statistical randomness is in focus. In simulations, the predictability of a PRNG is often utilized in order to compare different results of any one test with various methods. In applications regarding cryptography, however, the unpredictability is of utmost importance, and unpredictability is often more valued than statistical randomness. When the term random is used in this paper, it refers to both statistical randomness and unpredictability unless otherwise stated. Quantum theory postulates irreducible indeterminacy of individual quantum processes, which have recently lead to the construction of Quantum Random Number Generators (QRNG). An example of such an device is the Quantis from ID Quantique, which is being marketed as a source of truly random numbers QRNGs have been experimentally shown to be incomputable [8], which is to say that no algorithm can reproduce their results. On the other hand, modern state of the art algorithms for pseudo random number generation, such as the Mersenne Twister, offers similar effects at much higher bit-rates.

In practice, we would like to have some means by which we can evaluate if a given random number generator is good or bad. It is easy to measure the bit-rate of an generator, but it is hard to evaluate the generated randomness. There have been several methods for this developed, but the first software to reach wider

success was the *Diehard battery of tests* released on a CD-ROM in 1995. Before making the Diehard test suite, the developer George Marsaglia stated as one of the motivations behind creating the test suite: [30]

The same simple, easily passed tests are reported again and again.
Such is the power of the printed word.

This primarily refers to the tests presented in the second edition of the book *The Art of Computer Programming: Semi-numerical Algorithms* by Donald E. Knuth, which outlines several tests for random number testing and algorithms for implementing them.

Currently the Diehard test suite has become an widely used standard alongside the more up-to-date SIS suite developed by the *National Institute of Standards and Technology* (NIST). To exemplify the trust put into Diehard, the Federal Office of Metrology METAS in Switzerland used it as a tool in certifying random number generators in 2010. [37]

1.3 Purpose

Some researchers have noted several flaws with the Diehard test suite, and many alternative suites have been developed over the past few years. This paper seeks to give an overview of the tests and test suites that are commonly available and to outline the basic theory behind random number testing. We also seek tests that could be good additions to the existing test suites.

This work will also analyze the QRNG ID Quantique and the PRNGs Mersenne Twister, AES OFB and RANDU, to see in what way they differ. We seek to outline some simple tests that could be used for live-testing of a hardware random number generator (HRNG). Such tests are developed to find out if a random number generator has somehow failed while being used, which is a growing concern as HRNGs are becoming more widespread.

1.4 Delimitations

This work has focused on mapping out and understanding the common tests used for random number testing and its underlying theory, the publicly available test suites for this and how to perform simple live evaluation of a HRNG. The tests included have been selected based primarily on how commonly used they are, how interesting the underlying theory is and to what extent it adds something new to the study.

We have included the following test suites in our analysis:

- Diehard
- NIST STS
- ENT

- SPRNG
- TestU01
- DieHarder

The SPRNG test suite has not been tested, due to the source code not compiling on our computers. The theory for some of the tests implemented in SPRNG have however been explained. A test suite sometimes cited in other works is Crypt-X, this suite has not been included here since it does not seem to publicly available at the time of writing. In the paper *Experimental Evidence of Quantum Randomness Incomputability* [8] several tests are implemented, and the authors claim the source code is readily available. We have not been able to find this code, and so even though many of the tests used in their work are explained they have not been applied. This work has not concerned the deeper theoretical aspects of quantum physics or internal testing of any QRNG.

1.5 Method

Both ID Quantique and Mersenne Twister have already been shown to produce good results in [8] and [33] respectively. We do not seek to prove this, we instead use knowledge about them to see to what extent, if any, test suites can distinguish between their characteristics. We also include two other PRNG, AES OFB and RANDU. The former since it is commonly used in cryptology, and the latter because it is famous for being bad. A new test called the Run-length Entropy test is also proposed and implemented in Matlab. In this work we use two different Quantis USB devices. Quantis 1 has serial number 132194A410 and Quantis 2 has serial number 132193A410.

1.6 Structure

Chapter 2 gives an overview of the RNGs considered in this paper, with a focus on Mersenne Twister and Quantis. Chapter 3 outlines theory for random number testing and several specific tests. Chapter 4 introduces the available test suites and their various features. Chapter 5 explains what tests were run and how, as well as their results. Lastly we present our conclusions, proposal for future work and a summary in 6.

1.7 Contributions

We have given an overview of tests, test suites and batteries of tests. The work covers both tests that are applied in various software and also those tests that are proposed in other research.

More specifically, we present an alternative derivation of the distribution of the test statistic in the Bitstream test, originally due to George Marsaglia, see subsec-

tion 3.4.2. The concept of live testing is also introduced, see subsection 3.1.1, and we provide examples of its uses and applications.

A new random number test is also proposed, the Run-length Entropy test, see subsection 3.8.5. Through a simple Matlab simulation we see that it is capable of clearly capturing an probability error of 5%.

We also show that the QRNG Quantis show some bit-level correlations that are not present in some of our tested PRNGs. A possible reason for this that is mentioned is that if there is a time-dependent error in the underlying probability distribution, then the use of the post-processing technique due to John von Neumann (see subsection 2.3.3) could be to blame. See the chapters 5 and 6 for more on this.

2

Random Number Generators

A *Random Number Generator* (RNG) is a generator of a random outcome. Historically dice, coins and cards have been used to generate seemingly random outcomes, however these methods are generally speaking not useful in modern scientific applications. Here we shortly explain a few modern RNGs and how they work. [1]

2.1 Universal Turing Machine

Before we start to introduce RNGs, we introduce the concept of a *universal Turing machine*, which is the conceptually simplest universal computer. [10]

Consider a computer that is a finite-state machine operating on a finite symbol set, fed with a tape on which some program is written in binary. At each point in time, the machine inspects the program tape, write to a work tape, change state according to the transition table and calls for more program code. Alan Turing believed that the computational ability of human beings could be mimicked by such a machine, and to date all computers could be reduced to a Turing machine.

The universal Turing machine reads the program tape in one direction only, never going back, meaning that the programs are prefix-free. This restriction leads immediately to Kolmogorov complexity theory, which is formally analogous to information theory. More on this in subsection 3.1.6. [10, p. 464-465]

In the context of random numbers, it is worth noting that algorithmic information theory introduces degrees of algorithmic randomness. A random sequence is said to be *Turing computable* if it can be generated by a universal Turing machine. Examples of such sequences are most commonly used PRNGs, such as the

Mersenne Twister. More interestingly, digits of π are also Turing computable, and furthermore not cyclic. This in contrast to Mersenne Twister, which will be further explained in subsection 2.2.2. [8]

2.2 Pseudo Random Number Generators

Currently, computers mainly use so called *Pseudo Random Number Generators* (PRNG), that is to say algorithms that produce seemingly random numbers but which can be reproduced by anyone knowing the original algorithm and the seed with which the result was produced. This section outlines some characteristics of such generators and introduces the PRNGs Mersenne Twister, AES OFB and RANDU.

2.2.1 Characteristics

For the purpose of evaluating a pseudo-random number sequence certain characteristics are of importance. We start by defining the period of a pseudo-random sequence, taken from [21, p. 10].

Definition 2.2.1. *For all pseudo-random sequences having the form $X_{n+1} = f(X_n)$, where f transforms a finite sequence into itself, the period of the sequence is the length of the cycle of numbers that is repeated endlessly.*

The period of a generator is derived from this definition, and commonly set to be equal to the longest period over all possible pseudo-random sequences that a generator can produce. This relates to a simple but very important concept, we can not expect a PRNG to generate infinitely unrepentive sequences.

Next we would like to talk about equidistribution properties. We use the definition of the k -distribution as a measure for the distribution, taken from [33, p. 4].

Definition 2.2.2. *A pseudo-random sequence x_i of n -bit integers of period P , satisfying the following condition, is said to be k -distributed to v -bit accuracy: Let $\text{trunc}_v(x)$ denote the number formed by the leading v bits of x and consider P of the kv -bit vectors:*

$$((\text{trunc}_v(x_i)), (\text{trunc}_v(x_i + 1)), \dots, (\text{trunc}_v(x_i + k - 1))), (0 \leq i \leq P).$$

Then, each of the 2^{kv} possible combinations of bits occurs the same number of times in a period, except for the all-zero combination that occurs once less often. For each $v = 1, 2, \dots, n$, let $k(v)$ denote the maximum number such that the sequence is $k(v)$ -distributed to v -bit accuracy.

As all modern computers are universal Turing machines it should be apparent that generating truly random sequences is impossible. The closest thing would be to generate pseudo-random sequences, that is to say, a sequence that is "random enough" for the intended application. In this work we use the definition of

pseudo-random sequences that can be found in the book Applied Cryptography by Bruce Schneier. [6, p. 44-46]

Definition 2.2.3. *A pseudo-random sequence is one that looks random. The sequence's period should be long enough so that a finite sequence of reasonable length is not periodic. Furthermore, the sequence is said to be cryptographically secure if it is unpredictable given complete knowledge about the algorithm or hardware generating the sequence and all of the previous bits in the stream.*

2.2.2 Mersenne Twister

The Mersenne Twister is a PRNG developed by Makoto Matsumoto and Takuji Nishimura. The generator has become the standard for many random number functions, for example Matlab's rand function. We will not dig deeply into the underlying theory of the Mersenne Twister, for that we refer you to the original paper by Makoto Matsumoto and Takuji Nishimura [33]. Instead, we focus on its characteristics. [33]

Mersenne Twister is an \mathbb{F}_2 -generator with a 623-dimensional equidistribution property with 32-bit accuracy, which means that it is k -distributed with 32-bit accuracy for $1 \leq k \leq 623$. Furthermore it has a prime period of $2^{19937} - 1$ despite only consuming a working area of 624 words. Speed comparison of the generator has shown that it is comparable to other generators but with many other advantages. Through experiments its characteristic polynomial has been shown to commonly have approximately 100 terms.

The number of terms in the characteristic polynomial for the state transition function relates to how random a sequence of \mathbb{F}_2 -generators can produce. \mathbb{F}_2 -generators with both high k -distribution properties (for each v) and characteristic polynomials with many terms are known to be good generators, which Mersenne Twister is an example of.

Despite this, Mersenne Twister does not generate cryptographically secure random numbers. By a simple linear transformation the output becomes a linearly recurring sequence, from which one can acquire the present state from sufficiently large output. It is developed with the intention of producing $[0,1)$ -uniform real random numbers, with focus on the most significant bits, making it ideal for Monte Carlo simulations.

2.2.3 Other Generators

We will briefly mention two other PRNGs used in the testing. These are included mainly for comparison of results, and both have quite well documented properties.

RANDU

RANDU is the name of an notoriously bad PRNG, in fact, that is the main reason for including it. It is an linear congruential generator defined by the recurrence

$$x_i = (655539x_{i-1}) \bmod 2^{31}, \quad (2.1)$$

where the output at step i is $u_i = x_i/m$. [25]

The generator has been noted as an exceptionally bad one, with a period of only 2^{29} , by several authors, including Donald E. Knuth [21] and the developer of the test suite DieHarder, Robert G. Brown [5]. We hope that our testing will clearly reflect this.

AES OFB

The *Advanced Encryption Standard* (AES) is a standard set by the NIST for what should be exceptionally secure algorithms. The standard utilizes Rijndael cipher. It is noted to be "state of the art" by Robert G. Brown in [5].

The algorithm as used here takes a key of 16, 24 or 32 bytes, as well as a seed. At each encryption step the algorithm is applied on the input block to obtain a new block of 128 bits. Of these, the first r bytes are dropped and only the following s bytes are used. These parameters are set by the user. Furthermore the algorithm runs in several modes, denote $E(K, T)$ as the AES encryption operation with key K on plain text T resulting in encrypted text C . The Output Feedback Mode will generate new blocks of 128 bits C_j through the relation $C_j = E(K, C_{j-1})$. [25, p. 78]

In contrast to Mersenne Twister, we expect AES OFB to be cryptographically secure. Thus, our reason for adding it is to see if the random number tests applied will be able to distinguish this.

2.3 True Random Number Generators

True Random Number Generators (TRNG) are based on some seemingly truly random physical phenomenon. Examples of such phenomenon are for example background radiation, photonic emission in semiconductors, radioactive decay and quantum vacuum. In this section we seek to understand the basic concepts of TRNGs based on Quantum theory, which in one sense might be the only really true random number generators. The major other branch of TRNGs are mainly those based on chaos theory, these will not be considered.

2.3.1 Quantum Indeterminacy

We use the following definition taken from the book *Applied Cryptography* [6, p. 44-46] by Bruce Schneier for true random sequences:

Definition 2.3.1. *A true random sequence is one that not only satisfies the definition of pseudo-random sequence (see definition 2.2.3), but can also not be reliably repro-*

duced. If you run a true random sequence generator twice with the exact same input, you will get two completely unrelated random sequences.

Probabilistic behavior is one of the main concepts of quantum physics. It is postulated that randomness is fundamental to the way nature behaves, which means that it is impossible to, both in theory and in practice, predict the future outcome of an experiment. [18, p. 54]

Quantum indeterminacy is nowhere formally defined, but it gives rise to what could be considered as theoretically perfect randomness. This has given rise to quantum random number generators, whose source of entropy (supposedly) are "oracles" of an quantum indeterminate nature. In contrast chaotic processes such as coin flipping are only random in practice, but not in theory, due them being predictable with enough knowledge about their surroundings. [8, p. 4]

2.3.2 Quantum Random Number Generation

The first common example of *Quantum Random Number Generators* (QRNG) are those based on nuclear decay of some element, often read using a Geiger counter. These random number generators have historically been used with great success, however the health risks involved with handling a radioactive material as well as the bulkiness of products have limited its success. [39, p. 7] Two famous techniques for generating random numbers based on photons have been proposed in quantum physics.

Techniques of modern linear optics permit one to generate very short pulses of light in optical fibers. Such pulses can have durations shorter than $\tau \approx 10^{-14}$ sec. Given the total energy of such an optical pulse, one can reduce the wave packet's energy to approximate $\hbar\omega$ by passing it through several wide-bandwidth absorbers. A good approximation will ensure that the wave packet contains only one photon, sometimes called a single-photon state. One of the random number generation techniques is based on this. Originally this method was intended as an experiment for assuring that the photon, despite its wave properties, is also a single, integral particle. A beam splitter consisting of a half-silvered mirror is placed in the path of a single-photon state. In the direction of the unreflected photon beam a single photon detector, detector 1, is placed, and in the trajectory of the reflected beam another detector, detector 2, is placed. Classical physics would suggest the energy split and both detectors should detect the same photon at all times. However, quantum physics suggests that the photon can not split, and will randomly pass through to detector 1 and randomly be reflected to detector 2. This outlines the basics of one kind of QRNG. [3, p. 4-6]

2.3.3 Post-processing

A major problem when dealing with quantum random number generation is bias. *Post-processing* is a term used for methods that deal with such bias. For clarity, we define bias as in definition 2.3.2, taken from [52, p.280].

Definition 2.3.2. An estimator \hat{X} of an parameter x is unbiased if $E[\hat{X}] = x$; otherwise \hat{X} is biased.

Given a RNG that is perfectly unpredictable, but not statistically random, one can use post-processing to improve the result. The easiest method in use is one proposed by John von Neumann in [48, p. 768].

Given a statistical model for observed data $P_X(X_i = 0) = P_X(X_i = 1) = 1/2$, consider a sequence of independent biased estimators \hat{X}_i with probability $P_X(\hat{X}_i = 1) = p$ and $P_X(\hat{X}_i = 0) = 1 - p$ for all i . Let the output function $f(\hat{X}_i, \hat{X}_{i+1})$ be 1 if the two samples of \hat{X}_i gives the sequence 10, 0 if it gives 01 and $f(\hat{X}_{i+2}, \hat{X}_{i+3})$ otherwise. This gives the probability of outputs as in equation 2.2.

$$P(f(\hat{X}_i, \hat{X}_{i+1}) = 1) = p(1 - p) \sum_{n=0}^{\infty} (p^2 + (1 - p)^2)^n = \frac{1}{2} \quad (2.2)$$

We see that even though \hat{X}_i is biased with $E_X[\hat{X}_i] = p$ the probability of the outcomes for the recurring sequence $E[f(\hat{X}_i, \hat{X}_{i+1})] = 1/2$. The drawback of this method is obviously that if p is far from $1/2$ the bit-rate will be severely decreased. Ideally, for $p = 1/2$, the amended process is at most 50% as efficient as the original one. It is worth noting that John von Neumann wrote that it is at most 25% as efficient in his original publication [48, p. 768], he must mean to refer to the average efficiency for using this technique when $p = 1/2$.

2.3.4 QUANTIS Quantum Random Number Generator

Quantis is a QRNG developed by the company ID Quantique. It is certified by the Compliance Testing Laboratory in the UK and by METAS in Switzerland, and used for various applications world-wide. Each Quantis OEM component has a bit-rate of 4 Mbit/s. [39, p.7-8]

The details behind the quantum physical process underlying the product are not officially disclosed in the Quantis Whitepaper published by ID Quantique, however the usage of a semi-transparent mirror and single-photon detectors suggests it is using the method explained in subsection 2.3.2.

There is also a processing and interfacing subsystem. The unbiasing method used is the one due to John von Neumann described in subsection 2.3.3. Some live verification of functionality is carried out, the device continuously checks the light source and detectors and that the raw output stream statistics are within certain bounds.

It is worth noting that the certificate issued by the Compliance Testing Laboratory [22] does not disclose what testing method is used. In a related press release [36] it is explained that the product had been subject to the "*most stringent industry standard randomness tests*", with no further explanation. The certificate issued by METAS [38] clearly state that they used the Diehard battery of tests in the associated annex [37].

3

Random Number Testing Methods

The idea behind testing if a sequence of numbers is truly random might seem paradoxical. Testing if a generator of numbers is truly a random number generator would require testing it infinitely, which is not possible. What one can do is to test if a generated sequence is random enough for whatever it is intended to use with. This chapter begins by deriving theory for such testing, and then explain both theory and implementation for several such specific tests. Lastly we propose a few methods for implementing live testing of HRNGs.

3.1 Theory for Testing

This section outlines various theory used in random number tests. Each test is applied to either a uniform equidistributed sequence of real numbers u_0, u_1, \dots , binary numbers b_1, \dots, b_n or integers Y_0, Y_1, \dots defined by the rule $Y_n = \lfloor k u_n \rfloor$. The number k is chosen for convenience, for example $k = 2^6$ on a binary computer, so that Y_n represents the six most significant bits of the binary representation of u_n . Note that Y_n is independently and uniformly distributed between 0 and $k - 1$, and that $Y_i = b_i$ for $k = 2$.

3.1.1 Online and Offline Testing

In computer science *online* and *offline* is used to indicate a state of connectivity. In the context of RNGs, an online test will be one where the RNG is tested while running and an offline test one where data is first collected and subsequently analyzed afterwards. Generally speaking applying offline tests is preferred, since most tests consider different aspects of the sequence, and if a fairly large sequence generated by an RNG is random enough to pass all tests we will assume, unless given reason to think otherwise, that all future sequences are the same. We will

call a subcategory of online tests for live tests. We distinguished live tests in that the same test is applied many times while an RNG is running, with the intention of exposing any change in behaviour over time of a given RNG. For such an test to be efficient, it must be very computationally cheap, as most RNGs have a high bitrate. Live tests are particularly interesting for an HRNG, as it is often very hard to show that an HRNG has broken during operation. Live testing is distinguished from post-processing in that it does not change the output of the generator, it only indicates if the characteristics of the output has strayed from what is to be expected.

3.1.2 Shannon Entropy

The concept of entropy has uses in many areas, and is a measure of uncertainty in a random variable. In information theory, it is often defined as in definition 3.1.1. [10, p. 13-16]

Definition 3.1.1 (Shannon Entropy). *The Shannon entropy of a discrete random variable X with alphabet \mathcal{X} and probability mass function $p_X(x) = P(X = x)$, $x \in \mathcal{X}$ is defined as*

$$H_b(X) = - \sum_{x \in \mathcal{X}} p_X(x) \log_b p_X(x) \quad (3.1)$$

If the base b of the logarithm equals e , the entropy is measured in nats. Likewise, if it is 2, the entropy is measured in bits and if it is 10 it is measured in digits. If b is not specified it is usually assumed to be 2, and the index b is dropped.

We also define the *binary entropy* in definition 3.1.2.

Definition 3.1.2 (Binary Entropy). *The binary entropy of a discrete random variable X with alphabet $\mathcal{X} = \{0, 1\}$ and probability mass function $P_X(X = 1) = p$ and $P_X(X = 0) = 1 - p$ is defined as*

$$H(p) = -p \log_2 p - (1 - p) \log_2(1 - p) \quad (3.2)$$

Binary entropy is measured in bits.

3.1.3 Central Limit Theorem

The *Central Limit Theorem* (CLT) is one of the most powerful statistical tools available when estimating distributions. It also explains why so many random phenomena produce data with normal distribution. We begin with the main theorem taken from [52].

Theorem 3.1.1 (Central Limit Theorem). *Given a sequence of independent and identically distributed variables X_1, X_2, \dots with expected value μ and variance σ^2 , the cu-*

mulative distribution function

$$Z_n = \frac{\sum_{i=1}^n X_i - n\mu}{\sqrt{n\sigma^2}}, \quad (3.3)$$

has the property

$$\lim_{n \rightarrow \infty} F_{Z_n}(z) = \Phi(z). \quad (3.4)$$

The proof of this theorem is quite extensive and will not be included here. The theorem inspires the following definition.

Definition 3.1.3 (Central Limit Theorem Approximation). *Let $W_n = X_1 + \dots + X_n$ be the sum of independent and identically distributed random variables, each with expected value μ and variance σ^2 . The Central Limit Theorem Approximation to the cumulative distribution function of W_n is*

$$F_{W_n}(w) \approx \Phi\left(\frac{w - n\mu}{\sqrt{n\sigma^2}}\right). \quad (3.5)$$

This definition is often called the normal or Gaussian approximation for W_n .

From this we can derive the following useful relation between normal and Poisson distribution.

Theorem 3.1.2. *For X Poisson distributed with parameter λ , where λ sufficiently large, we have*

$$X \approx N(\mu, \sigma^2), \quad (3.6)$$

where $\mu = \lambda$ and $\sigma^2 = \lambda$.

Proof. Consider $X_1, X_2, \dots, X_\lambda$ independently identically distributed $Po(1)$, and define the sum $Y = X_1 + \dots + X_\lambda$. We know that for independent variables the corresponding characteristic functions satisfy $\phi_Y = \phi_{X_1} \dots \phi_{X_\lambda}$, and $\phi_W(s) = e^{\alpha(e^s - 1)}$ for $W \sim Po(\alpha)$ (see appendix A). Thus, $Y \sim Po(\lambda)$, but recall that according to the Central Limit Theorem Approximation, for large enough λ we will have

$$\frac{Y - \lambda}{\sqrt{\lambda}} \xrightarrow{d} N(0, 1). \quad (3.7)$$

This proves the theorem.

3.1.4 Hypothesis Testing

Hypothesis testing is a frequently used tool in various statistical applications. In a hypothesis test, we assume that the observations result from one of M hypothetical probability models H_0, H_1, \dots, H_{M-1} . For measuring accuracy we use the probability that the conclusion is H_i when the true model is H_j for $i, j = 0, 1, \dots, M - 1$. [52]

A *significance test* is a hypothesis test which tests an hypothesis H_0 that a certain

probability model describes the observations of an experiment against its alternative $\neg H_0$. If there is a known probability model for the experiment H_0 is often referred to as the *null hypothesis*. The test addresses if the hypothesis should be accepted or rejected to a certain *significance level* α . The sample space of the experiment S is divided into an acceptance set A and a rejection set $R = A^c$. For $s \in A$, accept H_0 , otherwise reject the hypothesis. Thus, the significance level is as in equation 3.8. The significance test starts with a value of α and then determines a set R that satisfies 3.8.

$$\alpha = P(s \in R) \quad (3.8)$$

There are two kinds of errors related to significance tests, the probability of rejecting H_0 when H_0 is true and to accept H_0 when H_0 is false. The probability for the first is simply α , but with no knowledge of the alternative hypothesis $\neg H_0$ the probability of the latter can not be known. Because of this, we instead consider another approach.

An alternative to the above is thus the *Binary Hypothesis test*, which has two hypothetical probability models, H_0 and H_1 . The conclusions are thus to either accept H_0 as the true model or accept H_1 . Let the *a priori* probability of H_0 and H_1 respectively be $P(H_0)$ and $P(H_1)$, where $P(H_1) = 1 - P(H_0)$. These reflect the state of knowledge about the probability model before an outcome is observed.

To perform a binary hypothesis test, we first choose a probability model from sample space $S' = \{H_0, H_1\}$. Let S be the sample state for the probability models H_0 and H_1 , the second step is to produce an observation corresponding to an outcome $s \in S$. When the observation leads to a random vector \mathbf{X} we call \mathbf{X} the test static, which often is just a random variable X . For the discrete case the *likelihood functions* are the conditional probability mass functions $P_{\mathbf{X}|H_0}(\mathbf{x})$ and $P_{\mathbf{X}|H_1}(\mathbf{x})$, and for the continuous case the conditional probability density functions $f_{\mathbf{X}|H_0}(\mathbf{x})$ and $f_{\mathbf{X}|H_1}(\mathbf{x})$.

The test divides S into to sets A_0 and $A_1 = A_0^c$. For $s \in A_0$ we accept H_0 , otherwise accept H_1 . In this case we have two error probabilities, $\alpha = P(A_1|H_0)$, the probability of accepting H_0 when H_1 is true, and $\beta = P(A_0|H_1)$, the probability of accepting H_0 when H_1 is true.

3.1.5 Single-level and Two-level Testing

The null hypothesis H_0 is the hypothesis of perfect random behavior. For the $U(0, 1)$ case H_0 is equivalent to the vector u_0, \dots, u_{t-1} being uniformly distributed over the t -dimensional unit cube $[0, 1]^t$. [25]

Given a sample of size n , under the null hypothesis the errors α and β are related with n in such a way that if two of the three values are specified, the third value is automatically determined. Since β can take on many different values due to the infinite number of ways a data stream can be non-random, we thus choose to specify α and n . [43]

A *first-order* or *single-level* test observes the value of Y , let us denote it y , and rejects the null hypothesis H_0 if the significance level p from equation 3.9 is much too close to either 0 or 1. Usually this distance is compared to some test-specific significance level α . Sometimes this p -value can be thought of as a measure of uniformity, where excessive uniformity will generate values close to 1 and lack of uniformity will generate values close to 0. [25]

$$p = P(Y \geq y|H_0) \quad (3.9)$$

When Y has a discrete distribution under H_0 we have right p -values $p_R = P(Y \geq y|H_0)$ and left p -value $p_L = P(Y \leq y|H_0)$, the combined p -value is given in equation.

$$p = \begin{cases} p_R & \text{if } p_R < p_L \\ 1 - p_L & \text{if } p_R \geq p_L \text{ and } p_L < 0.5 \\ 0.5 & \text{otherwise.} \end{cases} \quad (3.10)$$

Second-order or *two-level* tests generates N "independent" test statistics Y, Y_1, \dots, Y_N , by replicating the first-order test N times. Denote the theoretical distribution function of Y under H_0 as F , for continuous F the observations $U_1 = F(Y_1), \dots, U_N = F(Y_N)$ should be independently identically distributed uniform random variables. A common way of performing two-level tests is thus to look at the empirical distribution of these U_j and use a *goodness-of-fit* test to compare it with uniform distribution. Examples of such goodness-of-fit tests include the Kolmogorov-Smirnov test and Anderson-Darling test.

Another approach to second-order testing is to add the N observations of the first level and reject H_0 for too large or too small sums. For most common tests the test statistic Y is either χ^2 , normally or Poisson distributed, meaning that the sum $\tilde{Y} = Y_1 + \dots + Y_N$ has the same type of distribution. For $Y \sim \chi_k^2$ we have $\tilde{Y} \sim \chi_{Nk}^2$, $Y \sim N(\mu, \sigma)$ we have $\tilde{Y} \sim N(N\mu, N^2\sigma^2)$ and $Y \sim \text{Po}(\lambda)$ we have $\tilde{Y} \sim \text{Po}(N\lambda)$. A goodness-of-fit test is applied to \tilde{Y} and the sum of the samples.

A third approach would be to use the Approximate Central Limit Theorem from subsection 3.1.3, this however requires that N is relatively large which is often not the case.

3.1.6 Kolmogorov Complexity Theory

Kolmogorov complexity is essential to algorithmic information theory, and links much of the underlying theory for tests described later in this paper as well as gives the fundamental reasoning for random number testing. This subsection summarize parts of chapter 14 in [10], see their work for proofs of most theorems.

Kolmogorov complexity, sometimes referred to as algorithmic complexity, is the length of the shortest binary computer program that describes an object. One would not use such an computer program in practice since it may take infinitely

long to identify it, but it is useful as a way of thinking. For a finite-length binary string x and universal computer \mathcal{U} , let $l(x)$ denote the length of string x and $\mathcal{U}(p)$ the output of computer \mathcal{U} with program p , then Kolmogorov complexity is defined as in definition 3.1.4. [10]

Definition 3.1.4. Kolmogorov complexity $K_{\mathcal{U}}(x)$ of a string x with respect to a universal computer \mathcal{U} is defined as

$$K_{\mathcal{U}}(x) = \min_{p: \mathcal{U}(p)=x} l(p), \quad (3.11)$$

the minimum length over all programs that print x and halt. If we assume that the computer already know the length of x , we define the conditional Kolmogorov complexity knowing $l(x)$ as

$$K_{\mathcal{U}}(x|l(x)) = \min_{p: \mathcal{U}(p, l(x))=x} l(p). \quad (3.12)$$

An important bound of conditional complexity is in theorem 3.1.3. [10]

Theorem 3.1.3 (Conditional complexity is less than the length of the sequence).

$$K(x|l(x)) \leq l(x) + c \quad (3.13)$$

Proof. Consider the program "Print the l -bit sequence: $x_1x_2...x_{l(x)}$ ". Since l is given no bits are needed to describe l , and the program is self-delimiting since the end is clearly defined. The length of the program is $l(x) + c$.

Below are three illustrative examples of Kolmogorov complexity, we consider a computer that can accept unambiguous commands in English with numbers given in binary notation.

Example 1. Consider the simple sequence $01010101...01$, where 01 is repeated k times. The program "Repeat 01 k times.", consisting of 18 letters, will do the trick, and the conditional complexity on knowing k will be constant.

$$K(0101...01|n) = c \text{ for all } n. \quad (3.14)$$

Example 2. As an interesting example of Kolmogorov complexity, consider digits of π . The first n bits of π can be computed using a series expansion, so the program "Calculate π to precision n using series expansion." and infinitely many other programs of constant length could be used for this. Given that the computer already knows n , this program has a constant length

$$K(\pi_1, \pi_2, ..., \pi_n|n) = c \text{ for all } n. \quad (3.15)$$

This shows that π is Turing computable, which was mentioned in section 2.1.

For the next example we need to use the Strong form of Stirling's approximation explained in theorem 3.1.4.

Theorem 3.1.4 (Strong form of Stirling's approximation). For all $n \in \mathbb{N}_+$ we have

the following relation:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}} \quad (3.16)$$

The proof of theorem 3.1.4 is outside the scope of this paper, we instead focus on the following corollary.

Corollary. For $0 < k < n, k \in \mathbb{N}_+$ we have

$$\sqrt{\frac{n}{8k(n-k)}} 2^{nH(k/n)} \leq \binom{n}{k} \leq \sqrt{\frac{n}{\pi k(n-k)}} 2^{nH(k/n)}. \quad (3.17)$$

Here $H(k/n)$ is the binary entropy function from equation 3.2.

Example 3. For the purpose of random number generation, we would like to know if we can compress a binary sequence of n bits with k 1's. Consider the program "Generate all sequences with k 1's in lexicographic order; Of these sequences, print the i th sequence." We have variables k with known range $\{0, 1, \dots, n\}$ and i with conditional range $\{1, 2, \dots, \binom{n}{k}\}$. The total length of this c letters long program is thus:

$$l(p) = c + \log n + \log \binom{n}{k} \leq c' + \log n + nH\left(\frac{k}{n}\right) - \frac{1}{2} \log n, \quad (3.18)$$

where the last inequality follows from equation 3.17. The representation of k has taken $\log n$ bits, so if $\sum_{i=1}^n x_i = k$ the condition complexity according to theorem 3.1.3 is,

$$K(x_1, x_2, \dots, x_n | n) \leq nH\left(\frac{k}{n}\right) + \frac{1}{2} \log n + c. \quad (3.19)$$

This example is in particular interesting, as it gives an upper bound for the Kolmogorov complexity of a binary string. Now we shift our attention to an Bernoulli process. We would like to understand the complexity of a random sequence, and thus look at theorem 3.1.5.

Theorem 3.1.5. Let X_1, X_2, \dots, X_n be drawn according to a Bernoulli($\frac{1}{2}$) process. Then

$$P(K(X_1, X_2, \dots, X_n | n) < n - k) < 2^{-k}. \quad (3.20)$$

This shows that most sequences have complexity close to their length, and that the number of sequences with low complexity are few. This corresponds well with the human notion of randomness, and serves as motivation for the next definitions, *algorithmically random* sequences and *incompressible strings*.

Definition 3.1.5 (Algorithmical Randomness). A sequence x_1, x_2, \dots, x_n is said to be algorithmically random if

$$K(x_1 x_2 \dots x_n | n) \geq n. \quad (3.21)$$

Definition 3.1.6 (Incompressible Strings). We call an infinite string x incompress-

ible if

$$\lim_{n \rightarrow \infty} \frac{K(x_1 x_2 \dots x_n | n)}{n} = 1. \quad (3.22)$$

These definitions set randomness as something depending on the understanding of the observer. To a computer, something random is simply something more complex than what it can understand from observing it. This can be related to the ideas of "oracles" and true randomness in quantum physics, which we so far can not understand beyond the fact that they seem to imply true randomness. Incompressibility follows from algorithmic randomness when we consider growing algorithmically random sequences. Given an algorithmically random sequence, if adding digits to the sequence results in a sequence that can be compressed we conclude that it was compressible, and compress it. We then continue adding and compressing indefinitely. The resulting sequence should be incompressible, which motivates the definition. The above definitions lead to theorem 3.1.6, for a proof see [10, p. 478].

Theorem 3.1.6 (Strong law of large numbers for incompressible sequences). *If a string $x_1 x_2 \dots$ is incompressible, it satisfies the law of large numbers in the sense that*

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n x_i \rightarrow \frac{1}{2}. \quad (3.23)$$

This shows that, for an incompressible infinite string, the number of 0's and 1's should be almost equal, and theorem 3.1.7 follows.

Theorem 3.1.7. *If a sequence is incompressible, it will satisfy all computable statistical tests for randomness.*

Proof. *Identify a test where the incompressible sequence x fails, this reduces the descriptive complexity of x , which contradicts the assumption that x is incompressible.*

This theorem is of utmost importance. It relates statistical testing to incompressibility, which is the basic idea for several of the random number tests. Now for a theorem that relates entropy with algorithmically random and incompressible sequences.

Theorem 3.1.8. *Let X_1, X_2, \dots, X_n be drawn i.i.d. $\sim \text{Bernoulli}(\theta)$. Then*

$$\frac{1}{n} K(X_1 X_2 \dots X_n | n) \xrightarrow{P} H(\theta). \quad (3.24)$$

where $H(\theta)$ is the binary entropy function from definition 3.1.2.

Notably, for large n and $\Theta = 1/2$ the binary entropy function will approach 1 and if it does not then $K(X_1, X_2, \dots, X_n | n) < 1$, which is to say that a randomly generated sequence is algorithmically random and incompressible if and only if it approach equal probability of outcomes 0 and 1 when the sequence grows.

Now we consider a monkey typing keys on a typewriter at random, or equivalently flipping coins and inputting the result into a universal Turing machine. Remember that most sequences of length n have complexity near n . Furthermore the probability of an randomly input program p is related to its length $l(p)$ through $2^{-l(p)}$. From these two facts, we see that shorter programs are more probable than long ones, and that at the same time short programs produce strings with simply described nature. Thus, the probability distribution of the output strings are clearly not uniform. From this we define a universal probability distribution.

Definition 3.1.7. *The universal probability of a string x is*

$$P_{\mathcal{U}}(x) = P(\mathcal{U}(p) = x) = \sum_{p: \mathcal{U}(p)=x} 2^{-l(p)}, \quad (3.25)$$

the probability that a program randomly drawn with binary equal distribution p_1, p_2, \dots will print out the string x .

The following theorem shows the universality of the probability mass function above.

Theorem 3.1.9. *For every computer \mathcal{A} ,*

$$P_{\mathcal{U}}(x) \geq c_{\mathcal{A}} P_{\mathcal{A}}(x), \quad (3.26)$$

for every binary string x , where the constant $c'_{\mathcal{A}}$ depends only on \mathcal{U} and \mathcal{A} .

This theorem is interesting in particular because if we present the hypothesis that X is drawn according to $P_{\mathcal{U}}$ versus the hypothesis that it is drawn according to $P_{\mathcal{A}}$, it will have a bounded likelihood ratio $P_{\mathcal{U}}/P_{\mathcal{A}}$ not equal to zero or infinity. This in contrast to other hypothesis tests, where it typically goes to 0 or ∞ . Our universal probability distribution can thus never be rejected completely as the true distribution of any data drawn according to some computable probability distribution. Using the monkey analogy, we can not reject that the universality is the output of monkeys typing on a computer, but we see that, from a universal computers perspective, we can reject the hypothesis that the universe is random. Thus, a computer can never generate a truly random sequence.

3.2 Goodness-of-fit Tests

This section outlines test statistics used in many tests. A random number test can be split into three parts. First, identify some characteristic of the sequence that you would like to test, the next step is to find a test statistic with known distribution, and lastly to apply some theoretical comparison, which is usually a goodness-of-fit test. This section will show several test statistics and associated goodness-of-fit tests.

3.2.1 Pearson's Chi-square Test

Pearson's χ^2 test is a common statistical tool used in various applications. The test as described here gives a probabilistic answer as to if a given sequence of data is χ^2 distributed under the null hypothesis or not. This test is usually applied when we have naturally categorized, often called binned, data. [21, p. 42-48]

The following derivation is partly taken from Donald E. Knuth's book [21]. Suppose that we have k categories of observable events. Given a fairly large number n of independent observations, let p_j be the probability that each observation falls into category j , and X_j the number of observations that actually do fall into category j . We see that (X_0, \dots, X_{k-1}) are multinomially distributed with parameters $(k, 1/n, \dots, 1/n)$. The corresponding probability mass function is given as

$$P(X_0 = x_0, \dots, X_{k-1} = x_{k-1}) = \frac{n!}{x_0! \dots x_{k-1}!} p_0^{x_0} \dots p_{k-1}^{x_{k-1}}. \quad (3.27)$$

Now, since the Poisson distribution expresses the probability of a given number of events occurring in a fixed interval, we suspect that X_j has Poisson distribution. Assume that X_j has a Poisson distribution with probability mass function

$$P(X_j = x_j) = \frac{e^{-np_j} (np_j)^{x_j}}{x_j!}, \quad (3.28)$$

and furthermore that X_j are independent. Under these assumptions, we can multiply the independent probabilities to get

$$P(X_0 = x_0, \dots, X_{k-1} = x_{k-1}) = \prod_{j=0}^{k-1} P(X_j = x_j) = \prod_{j=0}^{k-1} \frac{e^{-np_j} (np_j)^{x_j}}{x_j!}. \quad (3.29)$$

Now consider the probability of the sum of all X_j equaling n ,

$$P(X_0 = x_0, \dots, X_{k-1} = x_{k-1}, X_0 + \dots + X_{k-1} = n) = \sum_{\substack{x_0 + \dots + x_{k-1} = n \\ x_0, \dots, x_{k-1} \geq 0}} \prod_{j=0}^{k-1} \frac{e^{-np_j} (np_j)^{x_j}}{x_j!} = \frac{e^{-n} n^n}{n!}, \quad (3.30)$$

we can calculate the conditional probability

$$P(X_0 = x_0, \dots, X_{k-1} = x_{k-1} | X_0 + \dots + X_{k-1} = n) = \frac{\prod_{j=0}^{k-1} \frac{e^{-np_j} (np_j)^{x_j}}{x_j!}}{\frac{e^{-n} n^n}{n!}} = \frac{n!}{x_0! \dots x_{k-1}!} p_0^{x_0} \dots p_{k-1}^{x_{k-1}}. \quad (3.31)$$

We conclude that X_j can be regarded as independently distributed Poisson variables. For convenience, we change variable

$$Z_j = \frac{X_j - np_j}{\sqrt{np_j}} \quad (3.32)$$

We would like to find the distribution for the test statistic

$$X^2 = Z_0^2 + \dots + Z_{k-1}^2, \quad (3.33)$$

so we now focus on the distribution of Z_j . We have already shown that X_j have Poisson distribution, thus it has characteristic function (see appendix A):

$$\phi_{X_j}(t) = E(e^{itX_j}) = \exp(np_j(e^{it} - 1)), \quad (3.34)$$

for $-\infty < t < \infty$. Here $i = \sqrt{-1}$ and j is an index. The standardized Poisson random variable Z_j has characteristic function

$$\phi_{Z_j} = E\left(\exp\left(it \frac{X_j - np_j}{\sqrt{np_j}}\right)\right). \quad (3.35)$$

For n large we get

$$\begin{aligned} \lim_{n \rightarrow \infty} E\left(\exp\left(it \frac{X_j - np_j}{\sqrt{np_j}}\right)\right) &= \lim_{n \rightarrow \infty} \exp(-it\sqrt{np_j}) E\left(\exp\left(\frac{itX_j}{\sqrt{np_j}}\right)\right) \\ &= \lim_{n \rightarrow \infty} \exp(-it\sqrt{np_j}) \exp(np_j(e^{it/\sqrt{np_j}} - 1)) \\ &= \lim_{n \rightarrow \infty} \exp(-it\sqrt{np_j}) \exp(np_j(1 + \frac{it}{\sqrt{np_j}} + \frac{(it/\sqrt{np_j})^2}{2!} + o(n^{-3/2} - 1))) \\ &= \lim_{n \rightarrow \infty} \exp(-t^2/2 + \mathcal{O}(n^{-3/2})) \\ &= \exp(-t^2/2) \end{aligned} \quad (3.36)$$

Which implies that that Z_j is standard normally distributed when $n \rightarrow \infty$, so we say that Z_j is approximately standard normal distributed for n large enough. Since the definition of χ^2 distribution with k degrees of freedom is exactly equation 3.33 for all Z_j standard normally distributed, we see that X^2 has approximate χ^2 distribution. This leads to the χ^2 test statistic X^2 with k categories of data for the observations. The equality follows from that $X_0 + \dots + X_{k-1} = n$ and $p_0 + \dots + p_{k-1} = 1$.

$$X^2 = \sum_{j=0}^{k-1} \frac{(X_j - np_j)^2}{np_j} = \frac{1}{n} \sum_{j=0}^{k-1} \left(\frac{X_j^2}{p_j} \right) - n \quad (3.37)$$

Under the null hypothesis H_0 and for $n \gg k$ the test statistic $X^2 \sim \chi_{k-1}^2$, that is to say χ^2 distributed with $k-1$ degrees of freedom. The reason for this reduction in degrees of freedom is due to that this is conditional on $X_0 + \dots + X_{k-1} = n$, meaning that all the variables are not truly independent. It is worth noticing that calculating the p -value for the χ^2 test can be done with relative ease.

3.2.2 Multinomial Test

We will now show that the χ^2 test described in subsection 3.2.1 is only one of many tests based on multinomial distributions that can be of interest. These tests were thoroughly studied by L'Ecuyer et al. in [27].

When testing independence and uniformity of a random number generator a quite intuitive test is to segment the interval $[0, 1)$ into d equal parts. For dimension t , this gives $k = d^t$ hypercubes, sometimes called cells.

Non-Overlapping Case

First assume that we have calculated X_j , the number of data points falling into cell j , $0 \leq j \leq k-1$, and that we have done so in a way that the observed vector (X_0, \dots, X_{k-1}) has a multinomial distribution with parameters $(n, 1/k, \dots, 1/k)$ under the null hypothesis H_0 . We are interested in knowing the theoretical distribution of the test statistic Y in equation 3.38. [25, p. 96]

$$Y = \sum_{j=0}^{k-1} f_{n,k}(X_j) \quad (3.38)$$

The exact values for $E(Y)$ and $\text{Var}(Y)$ under H_0 are derived in [27], but these values are very expensive to calculate when $\lambda = n/k$ is big. When $\lambda \gg 1$ approximations with $\mathcal{O}(1/n)$ error are provided in [41] and approximations for other cases are covered at the end of this subsection. The possible choices for the real function $f_{n,k}$ are various, the important thing is that the resulting test statistic Y is a measure of clustering that decreases when the points are more evenly distributed

over the cells. Below follows several examples of $f_{n,k}$ and their corresponding statistics Y .

Example 4 (Power Divergence Statistic). *For a real-valued parameter $\delta > -1$ the Power Divergence Statistic is given by*

$$D_\delta = \sum_{j=0}^{k-1} \frac{2}{\delta(1+\delta)} X_j \left[\left(\frac{X_j}{\lambda} \right)^\delta - 1 \right]. \quad (3.39)$$

Here $\lambda = E(X_i)$, $i = 1, \dots, n$, if $n \gg k$ we also have that $E(X_i) \approx n/k$. In particular, two special cases are of interest. The first one being for $\delta = 1$, $D_1 = X^2$, which is the Pearson χ^2 test statistic from subsection 3.2.1, with known distribution under H_0 . The other one being the log-likelihood statistic

$$\lim_{\delta \rightarrow 0} D_\delta = G^2 = \sum_{j=0}^{k-1} 2X_j \ln(X_j/\lambda). \quad (3.40)$$

Example 5 (Negative Entropy Statistic). *The negative entropy statistic $Y = -H$ is given by*

$$-H = \sum_{j=0}^{k-1} (X_j/n) \log_2(X_j/n). \quad (3.41)$$

This relates to the log-likelihood statistic G^2 from equation 3.40 through the relation

$$H = \log_2(k) - G^2/(2n \ln 2). \quad (3.42)$$

Example 6 (Counting Variables). *Let us consider a few cases when $f_{n,k}(x)$ primarily depends on a indicator function $I[\cdot]$, described in table 3.1.*

Y	$f_{n,k}(x)$	Counting
N_b	$I[x = b]$	Number of cells with exactly b points
W_b	$I[x \geq b]$	Number of cells with at least b points
N_0	$I[x = 0]$	Number of empty cells
C	$(x-1)I[x > 1]$	Number of collisions

Table 3.1: The $f_{n,k}(x)$ and corresponding Y for some Interval Statistics.

These statistics satisfies the following properties;

$$\begin{aligned} N_0 &= k - W_1 = k - n + C \\ W_b &= N_b + \dots + N_n \\ C &= W_2 + \dots + W_n. \end{aligned} \quad (3.43)$$

The exact distribution of C under H_0 is derived in subsection 3.4.1.

The by far most used of these test statistics when $n \gg k$ is the Pearson χ^2 test statistic, however the other test statistics mentioned can also be used in many cases and some of them have proven to be better for finding deficiencies in random number generators. [25, p. 96]

In [27] approximations for Y under various circumstances are provided through a series of theorems. Begin by defining $\sigma_N^2 = \text{Var}(Y)$, $\sigma_C^2 = \text{Var}(Y)/(2(k-1))$,

$$Y^{(N)} = \frac{Y - k\mu}{\sigma_N}, \quad (3.44)$$

$$E(Y^{(N)}) = 0, \text{Var}(Y^{(N)}) = 1,$$

as well as

$$Y^{(C)} = \frac{Y - k\mu + (k-1)\sigma_C}{\sigma_C}, \quad (3.45)$$

$$E(Y^{(C)}) = k - 1, \text{Var}(Y^{(C)}) = 2(k - 1).$$

Theorem 3.2.1 (Power Divergence Statistics). *Under the assumption of the null hypothesis H_0 , for $\delta > -1$,*

1. [Dense case] If k is fixed and $n \rightarrow \infty$, $D_\delta^{(C)} \xrightarrow{d} \chi_{k-1}^2$,
2. [Sparse case] If $k \rightarrow \infty$, $n \rightarrow \infty$, and $n/k \rightarrow \lambda_0$ where $0 < \lambda_0 < \infty$, then $D_\delta^{(N)} \xrightarrow{d} N(0, 1)$.

Now we consider the counting variable statistics from example 6. If k is fixed and $n \rightarrow \infty$ eventually N_b becomes 0 and W_b equals k . For both n and k large each X_j is approximately Poisson distributed $\text{Po}(\lambda)$, so for $b \geq 0$

$$P(X_j = b) \approx e^{-\lambda} \lambda^b / b! \quad (3.46)$$

If k is large and $P(X_j = b)$ is small N_b is approximately Poisson distributed with mean

$$E(N_b) = \frac{n^b e^{-\lambda}}{k^{b-1} b!} \quad (3.47)$$

Theorem 3.2.2 (Counting Variable Statistics). *Under the assumption of the null hypothesis H_0 , let $k \rightarrow \infty$ and $n \rightarrow \infty$, for positive constants λ_∞ , γ_0 and λ_0 we have,*

1. [Very sparse case] If $b \geq 2$ and $n^b / (k^{b-1} b!) \rightarrow \lambda_\infty$, then $W_b \xrightarrow{d} N_b \xrightarrow{d} \text{Po}(\lambda_\infty)$.
For $b = 2$, one also has $C \xrightarrow{d} N_2$.
2. For $b = 0$, if $n/k - \ln(k) \rightarrow \gamma_0$, then $N_0 \xrightarrow{d} \text{Po}(e^{-\gamma_0})$.

3. [Sparse case] If $k \rightarrow \infty$ and $n/k \rightarrow \lambda_0 > 0$, for $Y = N_b, W_b$ or C , one has $Y^{(N)} \xrightarrow{d} N(0, 1)$.

Overlapping Case

Define $X_{t,j}^{(o)}$ as the number of overlapping t -dimensional vectors $v_i = u_i, \dots, u_{i+t}$ with cyclic wrapping, $i = 0, \dots, n-1$, falling into cell number j . For $\delta > -1$ we have the Power Divergence Statistic for the t -dimensional overlapping vectors as

$$D_{\delta,(t)} = \sum_{j=0}^{k-1} \frac{2}{\delta(1+\delta)} X_{t,j}^{(o)} \left[(X_{t,j}^{(o)}/\lambda)^\delta - 1 \right]. \quad (3.48)$$

Define $\tilde{D}_{\delta,(t)} = D_{\delta,(t)} - D_{\delta,(t-1)}$, from [27] we have theorem 3.2.3 and proposition 3.2.1. Note that $k = d^t$ and $k' = d^{t-1}$.

Theorem 3.2.3 (Dense case). Assuming H_0 true, if k is fixed and $n \rightarrow \infty$, then

$$\tilde{D}_{\delta,(t)} \xrightarrow{d} \chi_{k-k'}^2. \quad (3.49)$$

Proposition 3.2.1 (Sparse case). For $k, n \rightarrow \infty$ and $n/k \rightarrow \lambda_0$ where $0 < \lambda_0 < \infty$ simulations suggest that

$$\frac{\tilde{D}_{1,(t)}^2 - (k - k')}{\sqrt{2(k - k')}} \xrightarrow{d} N(0, 1). \quad (3.50)$$

Note that this has not been formally proven.

Despite there being no formal proof, simulations suggest that 1 and 2 in theorem 3.2.2 are valid for the overlapping case as well. [26]

3.2.3 Kolmogorov-Smirnov Test

Often one would like to evaluate random quantities that range over infinitely many values, such as a real random number between 0 and 1. Then a *Kolmogorov-Smirnov test* computes how much an observed probability distribution differs from a hypothesized one. This in contrast with the previous tests that considered binned data. [21, p. 48-55]

The cumulative distribution function for a random quantity X is calculated as:

$$F(x) = P(X \leq x) \quad (3.51)$$

For n independent observations X_1, X_2, \dots, X_n , the *empirical distribution function* is defined in equation 3.52.

$$F_n(x) = \frac{\text{Number of } X_1, X_2, \dots, X_n \text{ that are } \leq x}{n} = \frac{1}{n} \sum_{i=1}^n I(x_i \leq x) \quad (3.52)$$

where $I(A)$ is the indicator function the set A .

The Kolmogorov-Smirnov test may be used when $F(x)$ has no jumps. Given a bad source of random numbers, it is probable that the empirical distribution functions $F_n(x)$ will not approximate $F(x)$ well. We form the Kolmogorov-Smirnov test statistics in equation 3.53.

$$K_n = \sqrt{n} \max_{-\infty < x < \infty} |F_n(x) - F(x)| \quad (3.53)$$

The Kolmogorov-Smirnov test statistic in equation 3.53 is then compared with the Kolmogorov distribution of K_n , which can be found in various tables. [21]

It is not always easy to turn a Kolmogorov-Smirnov result into an actual p -value, it tends to be more sensitive to one end or the other of an empirical distribution. This and other reasons prompt the use of the Anderson-Darling test described in subsection 3.2.4 on the Kolmogorov-Smirnov test statistic, which is the common way of using the Kolmogorov-Smirnov goodness-of-fit test. [5, p. 2]

3.2.4 Anderson-Darling Test

The *Anderson-Darling test* is an goodness-of-fit test that has shown to perform very well in most situations. It gives more weight to the tails of the distribution than the Kolmogorov-Smirnov test. [45]

With cumulative distribution function as in equation 3.51 and ordered observed data Y_i , $i = 1, \dots, N$, the Anderson-Darling test statistic A^2 is often defined as

$$A^2 = -N - \sum_{i=1}^N \frac{(2i-1)}{N} [\ln F(Y_i) + \ln(1 - F(Y_{n+1-i}))], \quad (3.54)$$

This definition differs slightly from the original one, but is widely used. A further modification which takes into account the sample size n is sometimes preferred

$$A^{2*} = A^2(1.0 + 0.75/n + 2.25/n^2). \quad (3.55)$$

Depending on what distribution is being tested the critical range for A^{2*} and the formula for the p -value will differ, ranges and p -values for several distributions can be found in various publications. Note that in this paper, when a test applies the Kolmogorov-Smirnov test it means that the Kolmogorov-Smirnov test statistic is evaluated with the Anderson-Darling goodness-of-fit test.

3.2.5 Shapiro-Wilk Normality Test

The *Shapiro-Wilk test for normality* tests if sequence is normal. Given ordered observed data X_1, \dots, X_n , the test statistic is defined as

$$W = \frac{(\sum_{i=1}^n a_i X_i)^2}{\sum_{i=1}^n (X_i - E(X))^2}, \quad (3.56)$$

where

$$(a_1, \dots, a_n) = \frac{m^T V^{-1}}{(m^T V^{-1} V^{-1} m)^{1/2}}. \quad (3.57)$$

Here m is an vector (m_1, \dots, m_n) with the expected values of the order statistics of i.i.d. random variables samples from the standard normal distribution and V is the covariance matrix of those order statistics. Using the algorithm AS R94 the test statistic can be calculated for samples up to sample size $n = 5000$. Small values of W leads to rejection of normality and values near 1 implies normality. [40]

In [40] Nornadiah et al. show that the Shapiro-Wilk normality test statistic performs better than the Anderson-Darling test statistic described in subsection 3.2.4 for normal distributions. Thus, when presented with normal distributions of less than 5000 samples we would prefer to use the Shapiro-Wilk Normality test.

3.3 Dense Cells Scenario

These tests relates to the dense scenario described in subsection 3.2.2, and focus on the theorems related to the dense scenario. This section seeks to explain some of the more common illustrative tests for the dense cells scenario.

3.3.1 Frequency Test

The *frequency test*, also known as the *equidistribution test* or the *monobit test*, tests that the numbers of a generated sequence are uniformly distributed. It illustrates well that the same aspect of a random sequence can be tested in many ways, with different methods. We present three used methods for this test. [21]

Consider first that we have a sample of n random bits b_1, b_2, \dots, b_n , each with probability $1/2$. One way to measure the distribution of the bits would simply be to count the number of 1's minus the number of 0's, that is to say calculate

$$S_n = \sum_{i=1}^n (2b_i - 1). \quad (3.58)$$

For a perfect distribution, we expect to get S_n near zero for sufficiently large n .

Since b_i all follow binomial distribution, the central limit theorem shows that $P(\frac{|S_n|}{\sqrt{n}} \leq z) = 2\Phi(z) - 1$, for z positive. Thus we can use the test statistic

$$S_n = \frac{|X_1 + \dots + X_n|}{\sqrt{n}}, \quad (3.59)$$

with theoretical half normal distribution under H_0 for large n . We can thus apply the Anderson-Darling test. [43]

Assume instead that we generate n uniforms (u_1, \dots, u_n) , and calculate the empirical distribution function $F_n(x)$. It should be relatively intuitive that the corresponding theoretical distribution function is $F(x) = x$, so we simply apply the Anderson-Darling test with the Kolmogorov-Smirnov test statistic described in subsection 3.2.3. [21]

A third approach is to use the sequence of n randomly generated i.i.d. integers Y_1, \dots, Y_n , ranging from 0 to $k - 1$. For each integer r , $0 \leq r \leq k - 1$, count the number of occurrences that $Y_j = r$ and denote it as X_r . Assume H_0 , we see that $p_r = 1/k$ and we have n observations falling into k categories, so the vector (X_0, \dots, X_{k-1}) are multinomially distributed with parameters $(n, 1/k, \dots, 1/k)$. We can thus form the χ^2 test statistic X^2 and apply a χ^2 test. [21]

3.3.2 Frequency Test within a Block

The *Frequency Test within a Block*, sometimes referred to as the *Hamming weight 2 Test*, examines the proportion of 1's within L -bit blocks of non-overlapping random bits. Given n bits it partitions the bit string into a total of $K = \lfloor n/L \rfloor$ blocks. Define X_j as the hamming weight of block j , that is to say the number of 1's in block j , then under H_0 , X_j are i.i.d. binomial with mean $L/2$ and variance $L/4$. We compute the test statistic

$$X^2 = \sum_{j=1}^K \frac{(X_j - L/2)^2}{L/4}, \quad (3.60)$$

which should be approximately χ^2 distributed with K degrees of freedom for large enough values of L . When $L = n$ this test is simply the frequency test described in subsection 3.3.1.

3.3.3 Non-Overlapping Serial Test

This test aims to see if sequences of successive numbers are uniformly distributed in an independent manner or not. [25]

For n random data points u_0, \dots, u_{n-1} , each of dimension t , let X_i be the number of random samples u_i that belongs to the subspace i , $i = 1, \dots, k = d^t$. Under the null hypothesis H_0 , the test statistic (X_0, \dots, X_{k-1}) has a multinomial distribution with parameters $(n, 1/k, \dots, 1/k)$. We can thus apply the multinomial tests described in subsection 3.2.2 under the null hypothesis H_0 .

3.3.4 Permutation Test

Similar to the serial test described in subsection 3.3.3, but with a different mapping of the input uniforms to the sample statistics. [25]

Instead of comparing non-overlapping serial sequences of data we instead divide the input sequence into a total of n groups with t elements each, giving $v_{jt} = (u_{jt}, u_{jt+1}, \dots, u_{jt+t-1})$ for $0 \leq j < n$. We find which of the $t!$ permutations of t object that would reorder the coordinates of v_{jt} in increasing order. Let $k = t!$ number all of the t possible permutations and let X_j be the number of vectors v_{jt} that are reordered by permutation j for $j = 0, \dots, k-1$. The independence of the permutations makes the observed vector (X_0, \dots, X_{k-1}) multinomially distributed with parameters $(n, 1/k, \dots, 1/k)$ under the null hypothesis H_0 . We thus apply the multinomial test from subsection 3.2.2.

3.3.5 Maximum-of-t Test

For this test we define the sequence X_0, \dots, X_{k-1} where

$$X_j = \max(u_{tj}, u_{tj+1}, \dots, u_{tj+t-1}). \quad (3.61)$$

where u_{tj} are sampled i.i.d. t -dimensional uniforms, $0 \leq j < k$. We now look at the distribution function of X_j , since

$$P(\max(u_1, u_2, \dots, u_t) \leq x) = P(u_1 \leq x)P(u_2 \leq x) \dots P(u_t \leq x) = x^t \quad (3.62)$$

we have $F_X(x) = x^t$. Now simply apply the Anderson-Darling test on the Kolmogorov-Smirnov test statistic for (X_0, \dots, X_{k-1}) , $0 \leq x \leq 1$. [21, p. 70]

Alternatively, since (X_0, \dots, X_{k-1}) is multinomially distributed with parameters $(n, 1/k, \dots, 1/k)$ we can apply a χ^2 test with $k = t$ degrees of freedom or perform several such tests and apply a Anderson-Darling test. [25, p. 97]

3.3.6 Overlapping Serial Test

Generating cell numbers that are *dependent* is of interest, but will render the distributions derived in the non-overlapping serial test from subsection 3.3.3 useless. In this test we generate n uniforms u_0, \dots, u_{n-1} and set $v_0 = (u_0, \dots, u_{t-1})$, \dots , $v_{n-t+1} = (u_{n-t+1}, \dots, u_n)$, $v_{n-t+2} = (u_{n-t+2}, \dots, u_n, u_0)$, \dots , $v_n = (u_n, u_0, \dots, u_{t-2})$. Since these points overlap, they are no longer independent, meaning that $(X_0^{(o)}, \dots, X_{k-1}^{(o)})$ is not multinomially distributed. Consider $X_{t,j}^{(o)}$ as the number of overlapping vectors v_i , $i = 0, \dots, n-1$ that belongs to cell j . The test checks to see if the distribution of the data matches the theoretical distribution for either the dense or sparse cases in theorem 3.2.3 and proposition 3.2.1. In applications, this is often done with $\delta = 1$, due to this special case being proven long before the general one, but of course one could equally well use the log-likelihood test statistic ($\delta \rightarrow 0$).

Either case, this will result in a χ^2 distribution for the dense case and standard normal distributions in the sparse case. [25, p. 98]

3.3.7 Poker Test

The poker test, also known as the *partition test*, takes n successive t -tuplets of integers $Y_{tj}, Y_{tj+1}, \dots, Y_{tj+t-1}$ for $0 \leq j < n$, and matches the results to some predetermined pattern to identify if any pattern is more frequent than it should be. The "classical" poker test considers 7 categories of quintuples and a simplified poker test groups some of the less probable outcomes as illustrated in table 3.2.

Type	Classical Poker Test	Simplified Poker Test
All different	abcde	5 values
One pair	aabcd	4 values
Two pairs	aabb	3 values
Three of a kind	aaabc	
Full house	aaabb	2 values
Four of a kind	aaaab	
Five of a kind	aaaaa	1 value

Table 3.2: Table of patterns used for two types of the Poker test.

The simplified poker test can easily be generalized, and this is the test commonly used. For n groups of t successive independent numbers we can count the number of t -tuplets with r distinct values. Assuming $n \gg k = d^t$, the various poker hands are multinomially distributed and we can apply a χ^2 test to see if the theoretical frequency of these categories is the same as the actual frequencies.

To get the probability of getting r distinct values we must see how many of the d^t t -tuplets of numbers between 0 and $k - 1$ have exactly r distinct elements, this number is then divided by the total d^t . Since the Stirling number of the second kind (see appendix B) gives us the number of ways to partition a set of t elements into r parts, we simply multiply this with the number of ordered choices of r things from a set of d objects and lastly divide by d^t . This is shown in equation 3.63.

$$p_r = \frac{d(d-1)(d-2)\dots(d-r+1)}{d^t} S(t, r) \quad (3.63)$$

3.3.8 Coupon Collector's Test

Given a sequence Y_0, Y_1, \dots with $0 \leq Y_j < k$ the coupon collector's test looks to find coupon collector segments from the sequence. A coupon collector segment is defined as any subsequence $\{Y_{j+1}, Y_{j+2}, \dots, Y_{j+r}\}$ needed to acquire a "complete set" of integers from 0 to $k - 1$. Let c_r be the number of coupon collector segments of length r where $d \leq r < t$ and let c_t be the number of coupon collector segments of length $\geq t$.

In order to evaluate the sequence for n consecutive coupon collector segments we start by calculating the probabilities q_r , given by equation 3.64, being the probability that the subsequence of length r is incomplete.

$$q_r = 1 - \frac{d!}{d^r} S(r, d) \quad (3.64)$$

By applying equation 3.63, being the probability of r distinct values, to 3.64 we get the relations

$$p_r = \begin{cases} q_{r-1} - q_r & \text{for } d \leq r < t \\ q_{t-1} & \text{for } r = t \end{cases} \quad (3.65)$$

Using the above we get the probabilities of the counts for c_d, \dots, c_t for the n lengths.

$$p_r = \begin{cases} \frac{d!}{d^r} S(r-1, d-1) & \text{for } d \leq r < t \\ 1 - \frac{d!}{d^{t-1}} S(t-1, d) & \text{for } r = t \end{cases} \quad (3.66)$$

Now apply a χ^2 test to the counts c_d, c_{d+1}, \dots, c_t with $k = t - d + 1$ degrees of freedom after n lengths have been counted.

3.3.9 Gap Test

This test examines the length of so called "gaps" between occurrences of the real number u_j , in order to identify if oscillations between gaps of different lengths are too fast or too slow. The test is similar to the binary runs test, often called the STS Runs test, proposed in [43].

Assume α and β to be real numbers with $0 \leq \alpha < \beta \leq 1$. Consider the lengths of the subsequence $u_j, u_{j+1}, \dots, u_{j+r}$ in which $\alpha \leq u_{j+r} < \beta$ but the other values in the subsequence are not, then this subsequence represents a gap of length r . We extract gaps of lengths $0, 1, \dots, t-1$ as well as $\geq t$ from the sequence u_0, u_1, \dots until a total of n gaps have been found, and compare it with the theoretical probability given by a Bernoulli trial as in equation 3.67 using a χ^2 test with $k = t + 1$ degrees of freedom.

$$P(\text{gap of length } r) = P(\alpha \leq u_{j+r} < \beta) P(u_j \notin [\alpha, \beta))^r \quad (3.67)$$

By noting that $P(u_j \in [\alpha, \beta)) = \alpha - \beta$ and denoting this probability p we can simply write as in equation 3.68.

$$p_r = \begin{cases} p(1-p)^r & \text{for } 0 \leq r < t \\ (1-p)^r & \text{for } r = t \end{cases} \quad (3.68)$$

For this test to give good results the choice of parameters is important. Common

values for α and β are $\alpha = 0$ and $\beta = 1$ in order to speed up the computation. The cases when α and β make up the interval between 0 and 0.5 or between 0.5 and 1 gives rise to tests often called "runs above the mean" and "runs below the mean" respectively. The variables n and t should be chosen so that the number of occurrences of each gap with index r , $0 \leq r \leq t$, should be at least 5. [21]

3.3.10 Runs Test

The *Runs test* seeks to evaluate if a random number sequence contains too many increasing or decreasing sequences. Since such sequences are not independent due to the probability of a short run following a long run being high, it serves as a good example of how complicated deriving the distribution for dependent tests can be.

In this test we look at a uniformly drawn sequence (X_1, \dots, X_t) and split it up into portions of increasing and decreasing subsequences, often called "runs up" and "runs down" respectively. Since adjacent runs are not independent, due to the fact that a long run will tend to be followed by a short run, applying a standard χ^2 test to the run counts would not give correct results. Instead we seek to find a different test statistic that does apply to these sequences.

For permutations of n elements, let $Z_{pi} = 1$ if position i is the beginning of a run of length p or more, and $Z_{pi} = 0$ otherwise. For each integer p , runs of length $\geq p$ are calculated by $R'_p = Z_{p1} + \dots + Z_{pn}$ and runs of length exactly equal to p as $R_p = R'_p - R'_{p+1}$. We would like to know the mean value and covariance of R_p , which can be expressed in terms of the mean value for Z_{pi} and $Z_{pi}Z_{qj}$. A permutation P of U_1, \dots, U_n for which we have a "runs up" is shown in 3.69. Note that we only show this for the case when we count "runs up", but that the idea is the same as in the case with "runs down".

$$U_{i-1} > U_i < \dots < U_{i+p-1} \quad (3.69)$$

We would like to know the probability of such a sequence. Consider $i \leq n - p + 1$, there are $p + \delta_{i1}$ ways to order a sequence like 3.69, $\binom{n}{p+1}$ ways to choose the elements for the positions indicated and $(n-p-1)!$ ways to arrange the remaining elements. This gives us the expected value for Z_{pi} as in equation 3.70.

$$E(Z_{pi}) = \frac{1}{n!} \sum_p Z_{pi} = \begin{cases} \frac{p+\delta_{i1}}{(p+1)!} & \text{if } i \leq n - p + 1; \\ 0 & \text{otherwise.} \end{cases} \quad (3.70)$$

The corresponding covariance can be derived for $i > j$ by calculating $E(Z_{pi}Z_{qj}) = \frac{1}{n!} \sum_{i,j \in P} Z_{pi}Z_{qj}$. Since $Z_{pi}Z_{qj} = 1$ if and only if $Z_{pi} = Z_{qj} = 1$, we need to consider the case when $i+p \leq j \leq n-q+1$. For $i+p < j \leq n-q+1$ we have non-overlapping sequences and can simply calculate the covariance as $\sigma(Z_{pi}, Z_{qj}) = E(Z_{pi}Z_{qj})$. The case when $i+p = j \leq n-q+1$ requires us to consider overlapping sequences

of the kind shown in 3.71.

$$U_{i-1} > U_i < \dots < U_{i+p-1} > U_{i+p} < \dots < U_{i+p+q-1} \quad (3.71)$$

There are $\binom{n}{p+q+1}$ ways to choose the elements for the positions indicated in 3.71, we need to know how many ways there are to arrange them. Begin by observing that there are $(p+q+1)\binom{p+q}{p}$ ways to have the arrangement

$$U_{i-1} \geq U_i < \dots < U_{i+p-1} \geq U_{i+p} < \dots < U_{i+p+q-1}. \quad (3.72)$$

Subtract $\binom{p+q+1}{p+1}$ for those ways in which $U_{i-1} < U_i$, $\binom{p+q+1}{1}$ for those in which $U_{i+p-1} < U_{i+p}$ and add 1 for the case that both of the mentioned inequalities hold true, resulting in 3.73. Note that when $i = 1$ we do not subtract the case when $U_{i-1} < U_i$.

$$(p+q+1)\binom{p+q}{p} - (1 - \delta_{i1})\binom{p+q+1}{p+1} - \binom{p+q+1}{1} + (1 - \delta_{i1}) \quad (3.73)$$

There are $(n-p-q-1)!$ ways to arrange the remaining elements. This gives us equation 3.74, calculated by taking equation 3.73 multiplied by $\binom{n}{p+q+1}(n-p-q-1)!$ to get the sum over all partitions of $Z_{pi}Z_{qj}$ and lastly divided by all combinations $n!$.

$$E(Z_{pi}Z_{qj}) = \frac{\sum_{i \in P} Z_{pi}Z_{qj}}{n!} = \begin{cases} \frac{(p+\delta_{i1})q}{(p+1)!(q+1)!} & \text{if } i+p < j \leq n-p+1, \\ \frac{p+\delta_{i1}}{(p+1)!q!} - \frac{p+q+\delta_{i1}}{(p+q+1)!} & \text{if } i+p = j \leq n-p+1, \\ 0 & \text{otherwise.} \end{cases} \quad (3.74)$$

We can derive the equations 3.75 through 3.78 by using equation 3.70 and 3.74. Here $t = \min(p, q)$ and $s = p + q$.

$$E(R'_p) = \frac{(n+1)p}{(p+1)!} - \frac{p-1}{p!} \text{ for } 1 \leq p \leq n \quad (3.75)$$

$$E(R'_p R'_q) = \sum_{1 \leq i, j \leq n} \frac{1}{n!} \sum_P Z_{pi} Z_{qj} \quad (3.76)$$

$$\sigma(R'_p, R'_q) = E(R'_p R'_q) - E(R'_p)E(R'_q) = \begin{cases} E(R'_t) + f(p, q, n), & \text{if } p+q \leq n; \\ E(R'_t) - E(R'_p)E(R'_q), & \text{if } p+q > n. \end{cases} \quad (3.77)$$

$$f(p, q, n) = (n+1) \left(\frac{s(1-pq) + pq}{(p+1)!(q+1)!} - \frac{2s}{(s+1)!} \right) + 2 \left(\frac{s-1}{s} \right) + \frac{(s^2 - s - 2)pq - s^2 - p^2q^2 + 1}{(p+1)!(q+1)!} \quad (3.78)$$

With the expressions from equation 3.75 and 3.77 we can derive 3.79.

$$\begin{aligned} E(R_p) &= E(R'_p) - E(R'_{p+1}), \\ \sigma(R_p, R'_q) &= \sigma(R'_p, R'_q) - \sigma(R'_{p+1}, R'_q), \\ \sigma(R_p, R_q) &= \sigma(R_p, R'_q) - \sigma(R_p, R'_{q+1}). \end{aligned} \quad (3.79)$$

It has been proved that the probabilities $R_1, R_2, \dots, R_{t-1}, R'_t$ become normally distributed as $n \rightarrow \infty$, subject to the mean and covariance from 3.79. See [51] for this proof. This means that we can compute the number of runs R_p of length p for $1 \leq p < t$ as well as the number of runs of length t or more, R'_t and apply a χ^2 as was explained in subsection 3.2.1. Form the matrix C as in equation 3.80.

$$C_{ij} = \begin{pmatrix} \sigma(R_1, R_1) & \sigma(R_1, R_2) & \cdots & \sigma(R_1, R_{t-1}) & \sigma(R_1, R'_t) \\ \sigma(R_2, R_1) & \sigma(R_2, R_2) & \cdots & \sigma(R_2, R_{t-1}) & \sigma(R_2, R'_t) \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \sigma(R_{t-1}, R_1) & \sigma(R_{t-1}, R_2) & \cdots & \sigma(R_{t-1}, R_{t-1}) & \sigma(R_{t-1}, R'_t) \\ \sigma(R'_t, R_1) & \sigma(R'_t, R_2) & \cdots & \sigma(R'_t, R_{t-1}) & \sigma(R'_t, R'_t) \end{pmatrix} \quad (3.80)$$

Now we set $A_{ij} = C_{ij}^{-1}$ as well as the variables in 3.81.

$$Q_1 = R_1 - E(R_1), \dots, Q_{t-1} = R_{t-1} - E(R_{t-1}), Q_t = R'_t - E(R'_t) \quad (3.81)$$

This gives us the test statistic in 3.82 which for large n will be χ^2 -distributed with t degrees of freedom. Recall that $t = \min(p, q)$.

$$V = \sum_{i,j=1}^t Q_i Q_j A_{ij} \quad (3.82)$$

By observing equations 3.77 through 3.79 we see that C can be split into one part multiplied by n and one part that is not. Thus, $C = nC_1 + C_2$. To calculate $A = C^{-1}$, recall that the Neumann Series expansion for a square matrix with $|P| < 1$ is $(I - P)^{-1} = I + P + P^2 + \dots$, which applied on A yields equation 3.83 for n big enough to make $|C_1^{-1}C_2/n| < 1$. Note that C_1^{-1} and C_2 are fixed for fixed t .

$$A = (nC_1 + C_2)^{-1} = n^{-1}C_1^{-1} - n^{-2}C_1^{-1}C_2C_1^{-1} + \dots \quad (3.83)$$

In many applications, runs up to length $t = 6$ are considered, and the following approximation due to Knuth can then be used: $C_1^{-1}C_2C_1^{-1} \approx -6C^{-1}$. This will give rise to an error of $\mathcal{O}(n^{-3})$, and also give a simplified expression for the test statistic V in equation 3.82 that is χ_5^2 distributed.

$$V \approx Q^T C_1^{-1} Q / (n - 6) \quad (3.84)$$

3.3.11 Bit Distribution Test

This is not a new test as much as it is a mapping of relationships between tests. Many tests are quite similar, and they are actually testing the same thing. Robert G. Brown noticed this and therefore made a new test he chose to call the *Bit Distribution test*. [5]

We will argue a hierarchical relationship between this test and some other ones. Begin by taking a long set of binary equidistributed random values and treat them like a string of n values without overlap. The test utilizes a window of size w and slides through the sample sequence one window at a time, incrementing a frequency counter indexed by the w -bit integer that appears in the window. Since we expect the pattern 2^w out of n bits to appear $n/2^w$ times, the frequencies observed should distributed around this value.

This test implements the dense case of the non-overlapping serial test from subsection 3.3.3 for binary sequences, so $d = 2$ and $t = w$. Thus we will implement a χ^2 test to see if the frequencies of the counts are as expected.

Now suppose that w and w' are the size of bit windows used to generate p -values p_w and $p_{w'}$. Then passing the test at $w > w'$ is sufficient to conclude that the sequence will also pass the test at w' . That is to say, if 4-bit integers in a sequence occurs with expected frequencies within the bounds permitted then so does the 3-bit, 2-bit and 1-bit ones. Obviously the converse is not true, but passing at w is a necessary condition for passing at $w' > w$.

Thus, if we accept the null hypothesis for the bit distribution test for $w = 4$ we also accepted the hypothesis for the binary versions of the frequency test ($w = 1$), the STS runs test ($w = 2$, slightly weaker than the bit distribution test) and the bit distribution test for $w = 3$. We have also satisfied a necessary condition for the $n = 8$ bit distribution test.

This shows that the largest value w_{\max} for which an random number generator passes the bit distribution test should be an important descriptor of the quality of the generator. We can say that such an generator is "random up to w_{\max} bits".

3.4 Monkey-related Tests

These are tests related in some way to the theory outlined by George Marsaglia about monkeys typing randomly on a typewriter, see [28]. Most of the tests cor-

respond to the sparse case of the multinomial test from subsection 3.2.2.

3.4.1 Collision Test

The collision test due to Donald E. Knuth is an interesting example of how we can use the counting variables introduced in subsection 3.2.2. For test applying the χ^2 test we always had to assume that we had many more observations than categories to put them in, called a dense case. We will now consider the sparse case. A good source of uniform random numbers should spread out the numbers so that two categories would not get many observations, with high probability. The collision test looks at this.

For the collision test we split n independent observations into k categories, and we assume that $k > n$ holds, often called a sparse case. This implies that most of the times when an observation is categorized, there were no previous observation in the same category.

The probability that exactly j previous observations were put in a given category is $p_j = \binom{n}{j} k^{-j} (1 - k^{-1})^{n-j}$. When an observation is categorized in an category which have previously had another object categorized, we say that a collision has occurred. This gives us the collision test statistic C , which was already briefly mentioned in subsection 3.2.2 example 6.

The probability of a collision is the same as for the Poker test shown in subsection 3.3.7. The probability that c collisions will occur can be calculated as the probability that $n - c$ categories already contain observations, shown in equation 3.85.

$$P(C = c) = \frac{k(k-1)\dots(k-n+c+1)}{k^n} S(n, n-c) \quad (3.85)$$

We are interested in knowing the expected number of collisions, which is calculated in 3.86, where we use the notation $P(C = k) = p_k$.

$$E(C) = \sum_{j=1}^n (j-1)p_j = \sum_{j=1}^n jp_j - \sum_{j=1}^n p_j = \frac{n}{k} - 1 + p_0 \quad (3.86)$$

By using a standard Taylor series expansion we can calculate

$$p_0 = (1 - k^{-1})^n = 1 - nk^{-1} + \binom{n}{2} k^{-2} - \mathcal{O}(n^3 k^{-3}). \quad (3.87)$$

Combining equation 3.86 and 3.87 and get the average total number of collisions to be

$$E(C) = \sum_{j=1}^n (j-1)p_j = \frac{n(n-1)}{2k} \approx \frac{n^2}{2k}. \quad (3.88)$$

Calculating the exact distribution for C as explained above can be done for n and k large in $O(n \log n)$ by using hashing tables (see [21] page 71 for details), but when n gets very large one would like to use some kind of approximation. Recall equations 3.43 through 3.47 and theorem 3.2.2, which outlines the convergence of C for various extreme cases. We see that for n very large we will want to approximate C with a Poisson distribution with mean $E(C)$ for $n/k \leq 1$ and a normal distribution with the exact mean and standard deviation if $n/k > 1$. In the case of a Poisson distribution we can either apply a Andersson-Darling test for Poisson distribution or approximate it using proposition 3.1.2 and apply a normal distribution test.

3.4.2 Bitstream Test

The *Bitstream test*, due to George Marsaglia, is a popular test used in many test suites. However, the original documentation is somewhat lacking and the distribution in the Diehard implementation does not open up for varying the parameters. In the paper [28] Marsaglia presents several sparse overlapping tests, and derives an equation for standard deviation for the overlapping pairs, triples and quadruples. This formula, when applied to overlaps of length 20, gives results different from what Marsaglia uses in Diehard. It would seem that Marsaglia has derived the exact probabilities for this in some other seemingly unpublished paper. This prompted our attempt at deriving the exact variance.

Our derivation assumes that part 2 of theorem 3.2.2 holds for overlapping vectors. As was pointed out in subsection 3.2.2 this has been motivated through simulation, but not been proved. Assuming it holds true, it would mean that for the sparse case, sampling a long window of overlapping sequences should be almost the same as sampling an equal number of non-overlapping ones. We motivate this with some light reasoning.

Consider a window of length l being moved one bit to the right and remaining the same. There are only two such sequences, a sequence of $l+1$ 1's or 0's, giving the probability of the moved being the same as the original window $2/2^{l+1}$. For moving it two bits, each two-bit sequence in the window should be the same, giving the probability $2^2/2^{l+2}$. Thus, the probability of a sequence repeating when moving it up to l bits is the sum of these probabilities, being $l/2^l$. For l relatively big, $l \approx 10$, this probability is reasonably close to 0, and we can conclude that the probability of getting the same value when sliding over a long window is relatively small. Of course, this only shows that the probability of getting the same number when sliding over a window is low for long windows. It does not constitute a proper motivation, but we leave it at this.

For this test we generate random equidistributed bits b_0, b_1, \dots , and consider the

Source	n -value	Mean μ	Standard deviation σ
Diehard C-code	2^{20}	385750	512
Diehard C-code	2^{21}	141909	428
Diehard C-code	2^{22}	19205	167
Calculations here	2^{20}	385750	621
Calculations here	2^{21}	141909	377
Calculations here	2^{22}	19205	139

Table 3.3: Table of μ and σ for various n . The source code for Diehard can be found in [29].

n samples generated by sliding a window of size t over the bits. Without overlap the theory relates to the collision test, which will be shown below. There are 2^t binary words of length t , and we are interested in seeing how many words are not found when searching the binary sequence. We define our observation variables $X_0^{(o)}, \dots, X_{2^t}^{(o)}$, and are interesting in finding how many of these are empty. We use the counting variable test statistic N_0 from subsection 3.2.2. Since this is a sparse case, we have that $n \ll k$. Assuming that $k = 2^t$ and n are both sufficiently large we apply part 2 of theorem 3.2.2, and get

$$\frac{n}{k} - \ln(k) \approx \gamma_0, N_0 \xrightarrow{d} \text{Po}(e^{-\gamma_0}). \quad (3.89)$$

We now use the approximate from Poisson distribution to normal distribution using proposition 3.1.2. Note that

$$\lambda = e^{-(n/k - \ln(k))} = k e^{-n/k} = k - n + \frac{n^2}{2k} \left(1 - \frac{n}{3k} + \frac{n^2}{12k^2} - \dots\right), \quad (3.90)$$

which since this is a (relatively) sparse case, $n < k$, should be relatively large. This should look fairly familiar, recall from theorem 3.43 that $N_0 = k - n + C$. Applying expected value and recalling that $E(C) \approx n^2/2k$ proves this relation for the non-overlapping case, assuming $\frac{n}{3k}$ is small enough. This connection breaks, however, when $n \approx k$.

Now we simply apply some form of goodness-of-fit test appropriate for the normal distribution, for example the Andersson-Darling test. We can compare our estimation to what can be found in Diehard. In the original Diehard source code this test is implemented with $t = 20$ which gives $k = 2^t = 2^{20} = 1048576$ possible words. Three theoretical results for different n are included as comments to the Diehard source code, we have compared these with our equations above and the results is in table 3.3. Note that in our calculations the e^{γ_0} values are sufficiently large to motivate using the above approximation.

For a truly random sequence N_0 should be closely normally distributed with mean μ and standard deviation σ under H_0 , where the parameters are either as derived here or the ones due to Marsaglia found in table 3.3.

3.4.3 Monkey Typewriter Tests

The basic idea for all *monkey tests* is that if a monkey randomly types on a typewriter, how many letters must he type until he will type a specific word? These are tests proposed by George Marsaglia. They are similar to the other tests in this section, with the key difference being that the tests described in this subsection use an approximation that Marsaglia originally derived in his original paper on Monkey tests. [28], [25]

For n number uniforms u_1, \dots, u_n and d^t cells, we generate n overlapping observations by $(u_0, \dots, u_{t-1}), (u_1, \dots, u_t), \dots, (u_{n-1}, u_n, u_0, \dots, u_{t-3}), (u_n, u_0, \dots, u_{t-2})$ and count the occurrences of different outcomes. Define the density to be $\lambda = (n - t + 1)/d^t$. If n and d^t are large and of the same order of magnitude, then under H_0 the collision test statistic C is approximately normally distributed with $\mu \approx d^t(\lambda - 1 + e^{-\lambda})$ and $\sigma^2 \approx d^t e^{-\lambda}(1 - (1 + \lambda)e^{-\lambda})$. In some applications the approximative variance $\sigma^2 \approx d^t e^{-\lambda}(1 - 3e^{-\lambda})$ is used instead, since this was the original approximation presented by Marsaglia, however it has since been improved. For $n \ll d^t$ the number of collisions should be approximately $\text{Po}(\mu)$ whereas if λ is large enough (approximately $\lambda \geq 6$, compares to $\lambda \approx 1$ for $n = 2^{20}$ in subsection 3.4.2) then the number of empty cells $(d^t - n + C) \sim \text{Po}(d^t e^{-\lambda})$. [25]

There are several subcategories of monkey tests presented in various contexts, we will briefly introduce the most common ones below.

CAT Test

This is a special case of the monkey test where we only look at a single cell of all d^t cells, also meaning it is notably a weaker test. The test will fail if the particular cell is visited too frequently. In this test, we start by deciding the target cell from the first t random values, the target cell should be aperiodic. The test utilizes n random numbers giving a total of $n - t + 1$ points with overlapping coordinates. Let \tilde{Y} be the number of points hitting the target cell, then under H_0 the probability of one random number hitting the cell is distributed $\text{Po}(\lambda)$ where $\lambda = (n - t + 1)/d^t$. The distribution for the sum is thus $Y \sim \text{Po}(N\lambda)$. The non-overlapping version of this test is usually referred to as the *Non-overlapping Template Matching test* [25, p. 114]

OPSO, OTSO and OQSO Tests

Marsaglia presented four overlapping sparse occupancy tests, for overlapping pairs (OPSO), triplets (OTSO) and quadruplets (OQSO). This test is based on a speculation made by Marsaglia and Zaman [28] that the number of empty cells N_0 described in has mean $d^t e^{-\lambda}$ and variance $d^t e^{-\lambda}(1 - 3e^{-\lambda})$. This approximation is relatively accurate for $2 \leq t \leq 5$, and so the test based on N_0 for $t = 2, 3, 4$ is what is commonly referred to as the OPSO, OTSO and OQSO tests respectively. The tests simply checks to see if the theoretical and sampled distributions for N_0 described in 3.2.2 are similar enough. [25, p. 97, 113-114]

DNA Test

Here a 4-letter alphabet C,G,A,T is assumed. A randomly generated sequence will resemble the mapping of a segment of DNA, thus giving it the name. The test is simply a special case of the above general case with alphabet size $d = 4$ and $t = 10$. [25]

3.4.4 Birthday Spacings Test

This test resembles the Collision test. We would like to split n observations into $k = d^t$ cells for dimension t . We can think of the cells as "days of a year" and the observations as "birthdays", thus giving rise to the name of the test. [21]

Suppose we have "birthdays" Y_1, \dots, Y_n , for $0 \leq Y_k < k$ i.i.d. For this test, we start by sorting the random sequence so that $Y_{(1)} \leq Y_{(2)} \leq \dots \leq Y_{(n)}$ and define the spacings as

$$S_1 = Y_{(2)} - Y_{(1)}, \dots, S_{n-1} = Y_{(n)} - Y_{(n-1)}, S_n = Y_{(1)} + m - Y_{(n)}. \quad (3.91)$$

Lastly, these spacings are sorted in order of size so that $S_{(1)} \leq S_{(2)} \leq \dots \leq S_{(n)}$. Let Y be the number of equal spacings, calculated as $S_{(j)} - S_{(j-1)}$ for indices j such that $1 < j \leq n$. Under H_0 , each sample is approximately $\text{Po}(\lambda)$ where $\lambda = n^3/(4k)$. Thus the sum of all values for N replications is approximately $\text{Po}(N\lambda)$. We run an appropriate test statistic.

3.5 Other Statistical Tests

In this section we have collected tests that are of importance, but are neither related to the monkey tests or are special cases of the dense scenario for the multinomial distribution.

3.5.1 Serial Correlation Test

The *Serial correlation coefficient* is defined by equation 3.92 where all sums are taken over $0 \leq j < n$. We exclude the case when the denominator is zero.

$$C = \frac{n \sum (U_j V_j) - (\sum U_j)(\sum V_j)}{\sqrt{(n \sum U_j^2 - (\sum U_j)^2)(n \sum V_j^2 - (\sum V_j)^2)}} \quad (3.92)$$

A correlation coefficient near zero implies that the U_j and V_j are independent of each other, near ± 1 indicates strong dependence. If we set $V_j = U_{j+1}$ we get the serial correlation coefficient, note that since $U_1 U_2$ is correlated with $U_2 U_3$ we do not expect the serial correlation coefficient to be exactly zero even for perfectly random numbers.

For independent uniformly distributed u_i and v_i we expect C to be within the following confidence interval approximately 95% of the time, which is the basic

idea of this test.

$$\mu_n \pm 2\sigma_n \quad (3.93)$$

where

$$\mu_n = \frac{-1}{n-1}, \sigma_n^2 = \frac{n}{(n-1)^2(n-2)} - \frac{25}{5} \frac{1}{n^2} + \mathcal{O}\left(\frac{\ln(n)}{n^{7/3}}\right). \quad (3.94)$$

Consider n uniforms u_0, \dots, u_{n-1} , and let v_j be the cyclic sequence of the n uniforms $u_j, \dots, u_{n-1}, u_0, \dots, u_{j-1}$ for $j = 1, \dots, n-1$. Using fast Fourier transforms we can calculate all these serial correlation coefficients in $\mathcal{O}(n \ln n)$ steps, which is the most common way of applying this test.

3.5.2 Rank of Random Binary Matrix Test

This test looks at finding linear dependence between fixed lengths of substrings from the sequence. We expect a random sequence to not exhibit much dependence, since otherwise we could predict it. [25]

Let us fill a $L \times k$ matrix with random bits by generating a sequence of uniforms and taking s bits from each uniform. Fill the matrix one row at a time using $\lceil k/s \rceil$ uniforms per row. The test generates n such matrices and counts each of their rank. The probability of the rank R of a random matrix is x given by

$$P(R = 0) = 2^{-Lk}$$

$$P(R = x) = 2^{x(L+k-x)-Lk} \prod_{i=0}^{x-1} \frac{(1-2^{i-L})(1-2^{i-k})}{1-2^{i-x}}, \quad 1 \leq x \leq \min(L, k). \quad (3.95)$$

Under the null hypothesis we can compare the empirical distribution from the random number generator with the theoretical one given from the probability distribution above, using a χ^2 test after merging classes if needed. In order to avoid that the probability will be lumped in a single class of the χ^2 , the difference $|L - k|$ should be small.

3.5.3 Random Walk Tests

There are several tests based on random walks proposed in various contexts. Here we outline a general random walk and show several of the most common test statistics used. [26]

Consider a random walk over the set \mathbb{Z} , starting at 0 and moving one unit to left and right at each time step with equal probability. We take steps of size l , where l is an even integer in $[L_0, L_1]$. First generate a random walk of length $l = L_0$, then add two steps to get a random walk of length $l = L_0 + 2$, then repeat by adding 2 steps more until $l = L_1$. Assuming that our random number generator produces uniform random numbers, we take s bits from each uniform to generate binary moves b_i . For the entire walk of length L_1 , we need $\lceil L_1/s \rceil$ uniforms. Let

$X_i = 2b_i - 1$, define $S_0 = 0$ and

$$S_k = \sum_{i=1}^k X_i, k \geq 1. \quad (3.96)$$

This gives us the stochastic process $\{S_k, k \geq 0\}$, which is what is commonly referred to as a random walk over integers. Under the null hypothesis H_0 we have

$$p_{k,y} = P(S_k = y) = \begin{cases} 2^{-k} \binom{k}{(k+y)/2} & \text{if } k+y \text{ is even;} \\ 0 & \text{if } k+y \text{ is odd.} \end{cases} \quad (3.97)$$

For l even, we define several test statistics as follows: We are interested at looking at the number of steps to the right, H , the maximum value reached by the walk, M , the fraction of time spent on the right side of the origin, J , the first passage time at y , P_y , the number of returns to 0, R and the number of sign changes, C . These have the theoretical probabilities under the null hypothesis H_0 equal to

$$P(H = k) = P(S_l = 2k - l) = 2^{-l} \binom{l}{k}, 0 \leq k \leq l, \quad (3.98)$$

$$P(M = y) = p_{l,y} + p_{l,y+1}, 0 \leq y \leq l, \quad (3.99)$$

$$P(J = k) = p_{k,0} p_{l-k,0}, 0 \leq k \leq l, k \text{ even}, \quad (3.100)$$

$$P(P_y = k) = (y/k) p_{k,y}, \quad (3.101)$$

$$P(R = y) = p_{l-y,y}, 0 \leq y \leq l/2, \quad (3.102)$$

$$P(C = y) = 2p_{l-1,2y+1}, 0 \leq y \leq (l-1)/2. \quad (3.103)$$

The test statistics are calculated as

$$H = l/2 + S_l/2 = \sum_{i=1}^l I(X_i = 1), \quad (3.104)$$

$$M = \max\{S_k, 0 \leq k \leq l\}, \quad (3.105)$$

$$J = 2 \sum_{k=1}^{l/2} I(S_{2k-1} > 0), \quad (3.106)$$

$$P_y = \min\{k : S_k = y\}, y > 0, \quad (3.107)$$

$$R = \sum_{k=1}^l I(S_k = 0), \quad (3.108)$$

$$C = \sum_{k=3}^l I(S_{k-2}S_k < 0), \quad (3.109)$$

where I denotes the indicator function.

When performing the random walk test, n random walks are performed and the empirical distribution of the above test statistics are compared to the theoretical ones via a χ^2 test. Several modified random walk tests are used in various test suites, but will not be explained in detail here. Three similar tests are proposed by Rukhin et al in [43], these are the *Cumulative Sums test*, the *Random Excursion test* and the *Random Excursion Variant test*. [25]

3.5.4 Borel Normality Test

Borel introduced a normality property for infinite sequences and explicitly studied randomness. A real infinite sequence is said to be Borel normal if its digits in every base follows a uniform distribution. Borel used his notion of normal to define randomness, but was later disproven. Instead, it has been shown that a string being Borel normal is equivalent to it being incompressible by an information lossless finite-state compressor, and as we argued in subsection 3.1.6, this means that algorithmic random infinite sequences are Borel normal. [8]

In [7] Cristian Calude showed that Borel normality can be transposed to finite strings. We expect a string generated from a PRNG to be Borel normal, but not so much one generated from a truly randomly generated source. Indeed, Calude showed in his work that almost all pseudo-random strings are Borel normal. In a series of experiments [8] Calude et al. further strengthen this claim.

We will start with an definition taken from [7].

Definition 3.5.1 (Borel Normality for Strings). *For a fixed alphabet $\mathcal{X} = \{a_1, \dots, a_Q\}$ we denote \mathcal{X}^* as the free monoid generated by \mathcal{X} under concatenation. We call the elements of \mathcal{X}^* strings, where its elements x have length $|x|$. Every ordering on \mathcal{X} induces an quasi-lexicographical order on \mathcal{X}^* . For $k \in \mathbb{N}$, denote $\mathcal{X}^k = \{x \in \mathcal{X}^* \mid |x| = k\}$, and consider the alphabet $B = \mathcal{X}^k$ and the free monoid $B^* = (\mathcal{X}^k)^*$. Every $x \in B^*$ belongs to \mathcal{X}^* , but the converse does not hold. We denote $|x|_k$ for $x \in B^*$ as the length of x according to B which is $k^{-1}|x|$. Let $N_i^k(x)$ denote the count of occurrences of the lexicographical i th string of length k . Then we have*

1. A non-empty string $x \in \mathcal{X}^*$ is called k -limiting if

$$\left| \frac{N_i^k(x)}{|x|_k} - Q^{-k} \right| \leq \sqrt{\frac{\log_Q |x|}{|x|}}, \quad (3.110)$$

for every $1 \leq i \leq Q^k$.

2. If for every natural k , $1 \leq k \leq \log_Q \log_Q |x|$, x is k -limiting, then we say that x

is Borel normal.

In the binary case, we have the alphabet $B_k = \{0, 1\}^k$ and $Q = 2$. Thus, a string x is Borel Normal if for every integer $1 \leq k \leq \log_2 \log_2 |x|$ we have

$$\left| \frac{N_i^k(x)}{|x|_k} - 2^{-k} \right| \leq \sqrt{\frac{\log_2 |x|}{|x|}}, \quad (3.111)$$

for every $1 \leq i \leq 2^k$.

In this test, as proposed in [8], we take non-overlapping k -bit strings from our samples, $k = 1, \dots, 5$. We count the maximum, minimum and difference of non-overlapping occurrences of these strings, and test equation 3.111. We are particularly interested at seeing how many values are needed to fulfill the Borel normality condition, as we expect pseudo-random sequences to do so for relatively small sequences.

3.6 Spectral Tests

Several tests based on various spectral methods have been proposed and implemented in various test suites. The tests are designed to detect periodic features that are hard to identify through other methods.

These tests calculate *discrete Fourier coefficients* defined by

$$f_l = \sum_{j=0}^{n-1} (2b_j - 1) e^{2\pi i j l / n}, l = 0, 1, \dots, n-1, \quad (3.112)$$

where b_j is the j th bit in a $n = 2^k$ long binary random sequence. Here i denotes the imaginary number $i = \sqrt{-1}$. Denote the modulus operator by $|\cdot|_{\text{mod}}$. [25, p.125]

3.6.1 NIST Discrete Fourier Transform Test

This test was popularized by the NIST in their publication [43]. Note that f_l for $l > n/2$ can be obtained from the f_l with $l \leq n/2$ since $(2b_j - 1)$ are real numbers. We denote the observed number of $|f_l|_{\text{mod}}$ smaller than h for $l \leq n/2$ as O_h . Under H_0 , for large enough n and $h = \sqrt{2.995732274n}$, O_h has approximate normal distribution with mean $\mu = 0.95n/2$ and variance $\sigma^2 = 0.05\mu$. In the test the distribution of N values for the standardized statistic $(O_h - \mu)/\sigma$ is compared with the standard normal distribution. [25, p.125]

3.6.2 Erdmann Discrete Fourier Transform Test

In this test $S_l = |f_l|_{\text{mod}}^2/n$ is computed for $l = 0, 1, \dots, n-1$. Under H_0 , each S_l has mean 1 and variance $1 - 2/n$ for $l \neq 0$, and thus the sum of variables

$$X = \sum_{l=1}^{n/4} S_l, \quad (3.113)$$

should be approximately normally distributed with mean $\mu = n/4$ and variance $\sigma^2 = (n-2)/4$ according to the CLT. [25, p.125]

3.6.3 Discrete Fourier Transform Central Limit Theorem Test

This test builds on the Erdmann's test. Let X_l denote the sum of N copies of S_l from subsection 3.6.2. For N large enough, X_l should be approximately normal with mean N and variance $N(1 - 2/n)$, according to the CLT. In this test, we compare the empirical distribution of the $n/4$ normal variables X_l , $l = 1, 2, \dots, n/4$ to the standard normal distribution. [25, p.126]

3.7 Close-point Spatial Tests

The tests following here are based on the work by L'Ecuyer et al. [23]. The tests look at the distance between the closest points in a sample of n uniformly distributed points in the unit torus $[0, 1)^t$.

These tests are similar to the minimum distance test and the 3-D spheres test proposed by Marsaglia and implemented in Diehard, a generalization of Marsaglia's tests to higher dimensions is available in [14].

The unit torus is defined so that points that are face to face on opposite sides are "close" to each other. For the points X_1, \dots, X_n , let $D_{n,i,j} = \|X_j - X_i\|_p$, where $\|X\|_p$ is the L_p -norm as explained in appendix B. Put $\lambda(n) = n(n-1)V_t(1)/2$, where $V_t(r)$ is the volume of the ball with radius r in t dimensions and let $Y_n(\tau)$ be the number of distinct pairs of points (X_i, X_j) , with $i < j$ so that $D_{n,i,j} \leq (\tau/\lambda(n))^{1/t}$, for each $\tau > 0$.

Theorem 3.7.1. *Under H_0 , for any fixed $\tau_1 > 0$ and $n \rightarrow \infty$, the truncated process $\{Y_n(\tau), 0 \leq \tau \leq \tau_1\}$ converges weakly to a Poisson process with unit rate. Moreover, for $\tau \leq \lambda(n)/2^t$, one has $E(Y_n(\tau)) = \tau$ and $\text{Var}(Y_n(\tau)) = \tau - 2\tau^2/(n(n-1))$.*

The proof for theorem 3.7.1 can be found in [23].

Now we defined the jump times for Y_n , $T_{n,i} = \inf\{\tau \geq 0 | Y_n(\tau) \geq i\}$, $i = 1, 2, 3, \dots$, with $T_{n,0} = 0$. The transformed spacings between these jump times are also of interest, $W_{n,i}^* = 1 - e^{-(T_{n,i} - T_{n,i-1})}$. From theorem 3.7.1 we have that for fixed $m > 0$ and n large enough $W_{n,1}^*, \dots, W_{n,m}^*$ are approximately i.i.d. $U(0, 1)$. Now we apply the Anderson-Darling test to compare the empirical distribution of these random variables with the uniform one. This test is referred to as the *m-nearest-pairs test*.

The acronym *m-nearest pair test* is simply the corresponding two-level test that

applies a goodness-of-fit test to the N values of the Anderson-Darling statistic. An alternative approach is to pool the N batches of m observations $W_{n,i}^*$ in a single sample size Nm and apply a Anderson-Darling test. This test is referred to as the *m-nearest-pairs 1 test*.

We could also superpose N copies of the process Y_n to obtain a process $Y = \sum_{i=1}^n Y_n$. For τ_1 fixed, let J count the number of jumps of Y in the interval $[0, \tau_1]$. Let $T_0 = 0$ and T_1, \dots, T_{T_1} be sorted times of these jumps. Under H_0 , J is approximately Poisson distributed with mean $N\tau_1$ which lies the foundation for the *Njumps test*. Conditional on J the jump times T_j are distributed as J independent uniforms over $[0, \tau_1]$, which can be tested with an Anderson-Darling test on the J observations. This is called the *m-nearest-pairs 2 test*.

Various variants to the above tests are also proposed, for example by applying a "spacings" transformation before applying the Anderson-Darling tests, using a different measure of distance or using different goodness-of-fit tests.

3.8 Compressibility and Entropy Tests

Here we collect tests that focus on compressing a sequence. As was shown in subsection 3.1.6, incompressibility is closely related to randomness.

3.8.1 Discrete Entropy Tests

This test estimates the Shannon entropy of the sequence and compares it to the entropy distribution. Since Shannon entropy quantifies how much expected information is contained in the sequence, it should serve as a good measure of randomness, as was shown in subsection 3.1.6.

Build n blocks of L bits by taking the s least significant bits of from each of the NL/s successive output values from a uniform $[0, 1)$ generator. Then we have a total of $k = 2^L$ L -bit blocks, number these blocks from 0 to $k-1$. Let X_i denote the observed frequency of the i th block. The Negative Entropy Statistic was defined in subsection 3.2.2 example 5 to be

$$-H = \sum_{i=0}^{k-1} X_i \log_2 X_i. \quad (3.114)$$

We now apply theorem 3.2.1, and approximate $-H$ using a normal distribution if $n/2^L \leq 8$ (sparse case) and χ^2 distribution when $n/2^L > 8$ (dense case). [25]

The overlapping case of this test is the same as above, define the n blocks overlapping as in the overlapping serial test subsection 3.3.6 and use the approximative distributions from subsection 3.2.2. An alternative entropy test is described by Dudewicz et al. in [12].

3.8.2 Maurer's Universal Test

Maurer's universal test was developed with testing random number generators for cryptographic applications in mind. The focus of this test is the number of bits between matching patterns, with the purpose to detect if the sequence can be compressed without loss of information. As was explained in subsection 3.1.6, a significantly compressible function is considered to be non-random. [43, p. 2-20]

This test measures the entropy of a binary sequence. We generate $Q + K$ blocks of length L bits, where K is the number of steps of the test and Q is the number of initialization steps. For each of the K blocks we find the number of blocks generated since the most recent occurrence of the same block in the sequence. If it is generating block j and the most recent occurrence was in block $(j - i)$, it sets $A_j = i$; if the j th block is the first of its kind we set $A_j = j$. Then we compute the average

$$T_U = \frac{1}{K} \sum_{j=Q+1}^{Q+K} \log_2 A_j, \quad (3.115)$$

which under H_0 has approximately normal distribution with parameters specified in [34]. [25]

Several improvements to this test has been proposed, see for example Coron et al. [9] for an better formula for the variance and Haje et al. [13] for several proposed modifications of the test.

3.8.3 Linear Complexity Tests

In particular two linear complexity tests are common, the *jump complexity test* and the *jump size test*. Given a binary sequence, consider subsequences of length l . For $0 \leq l \leq n$, a jump is said to occur whenever adding a bit to a subsequence increases the complexity. Assume we have some effective algorithm for calculating the linear complexity of a binary sequence, for example the Berlekamp-Massey algorithm described in [2]. [25, p. 123]

The jump complexity test considers how many jumps occurs in the sequence. Denote the number of jumps J and define R_n to be the parity function of n ($= 1$ if and only if n is odd). Under H_0 , for n sufficiently large, J is approximately normally distributed with with mean and variance

$$\mu = \frac{n}{4} + \frac{4 + R_n}{12} - \frac{1}{32^n} \quad (3.116)$$

$$\sigma^2 = \frac{n}{8} - \frac{2 - R_n}{9 - R_n} + \frac{n}{6(2^n)} + \frac{6 + R_n}{18(2^n)} - \frac{1}{9(2^{2n})}. \quad (3.117)$$

The test compares the empirical and theoretical values.

The jump size test instead looks at the size of the jumps. The size h of a jump

is the amount by which the complexity increases when considering the next bit of the sequence. We count how many jumps of each size occurs, and compare the expected number for the jump sizes to the actual ones using a χ^2 test. It has been shown that under H_0 the jump sizes are independent identically distributed random variables that obey the geometric distribution with parameter $1/2$.

3.8.4 Lempel-Ziv Compressibility Test

This test utilizes the Lempel-Ziv compression algorithms. Lempel-Ziv are popular compression algorithms due to them being proven universally optimal and simple to implement. There are two popular versions, the sliding window Lempel-Ziv first published in 1977 and the tree-structured Lempel-Ziv published in 1978. These coding algorithms and examples are from [10, p. 442].

Consider a string x_1, x_2, \dots that we wish to compress, also assume that it has already been compressed until time $i - 1$. We call a division of the string $x_1 x_2 \dots x_n$ into phrases, separated by commas, a parsing S . Let W be the length of each window looking backward, then the algorithm will start by finding the next phrase being the largest k so that for some j , $i - 1 - W \leq j \leq i - 1$, we have $x_j, \dots, x_{j+k-1} = x_i, \dots, x_{i+k-1}$. This phrase will thus be $x_i \dots x_{i+k-1}$ and represented by the number (F, P, L) , where P indicated the beginning of the match and L its length. Note that the target of a (P, L) pair could extend beyond the window. If no match is found in the window, the next character is sent uncompressed. The flag F will distinguish between these cases, and the uncompressed character will be coded as (F, C) .

Example 7. For $W = 4$ and the sequence is *ABBABBABBBAAABABA* with an empty initial window the parsing will be *A, B, B, ABBABB, BA, A, BA, BA* and the sequence will be coded as: $(0, A), (0, B), (1, 1, 1), (1, 3, 6), (1, 4, 2), (1, 1, 1), (1, 3, 2), (1, 2, 2)$, here $F = 0$ implies no match and $F = 1$ implies a match was found.

The Tree-Structured Lempel-Ziv algorithm can be seen as building a dictionary in the form of a tree, where each node correspond to a phrase seen before. The source is parsed into strings that have not appeared so far. The algorithm look along the input sequence until it comes across the shortest string not yet marked off. The fact that its the shortest means that all its prefixes must have occurred before, meaning that it is possible to build a tree in this manner. The phrase is coded by giving the location of the prefix and the value of the last symbol.

Example 8. String *ABBABBABBBAAABABAA* is parsed as *A, B, BA, BB, AB, BBA, ABA, BAA*, meaning that the example will be coded:
 $(0, A), (0, B), (2, A), (2, B), (1, B), (4, A), (5, A), (3, A), \dots$

This test usually utilizes the latter algorithm. For a string of $n = 2^k$ bits the test counts W distinct patterns in the string in order to determine its compressibility

with Lempel-Ziv. Under H_0 , for large n

$$Z = \frac{W - n/\log_2 n}{\sqrt{0.266n/(\log_2 n)^3}}, \quad (3.118)$$

should have approximately standard normal distribution. Since the approximation has been shown to be bad even for big values of n , some implementations instead considers W as having a normal distribution with empirically derived estimates for mean and standard deviation. [25]

3.8.5 Run-length Entropy test

This test is the only new test both introduced and implemented in this work. It is an entropy test applied to run-length sequences. The test is intended to catch unpredictability in the oscillations, as we expect an random sequence to oscillate in an unpredictable manner.

The concept of run-length code is simple, we code 1_2 as 0_{10} , 01_2 as 1_{10} , ..., $00...01_2$ as $M - 1_{10}$ where M is the number of digits in the string, and lastly $00...00_2$ as M . Here A_k denotes number A in base k . This corresponds to the binary case for the runs of length M in the runs test. This is not an optimal code, in contrast to the Lempel-Ziv codes we illustrated in subsection 3.8.4. If we say that the probability of getting a 1 is p we can get the probabilities of the above codes through

$$P(X_i = k) = \begin{cases} (1-p)^k p & \text{if } k < M, \\ (1-p)^k & \text{if } k = M. \end{cases} \quad (3.119)$$

The entropy for this coding is calculated from equation 3.1 and some simple algebra yields

$$H(X) = \log_2(1-p)\left(1 - \frac{1-(1-p)^{M+1}}{p}\right) - \log_2(p)(1 - (1-p)^M). \quad (3.120)$$

This function has some very interesting properties. Already for $M \approx 5$ the entropy function as a function of the probability p will give an almost linear curve, which is shown in figure 3.1, and we set the entropy function

$$H^*(p, M) = \begin{cases} \log_2(1-p)\left(1 - \frac{1-(1-p)^{M+1}}{p}\right) - \log_2(p)(1 - (1-p)^M) & \text{if } p \leq 1/2, \\ \log_2(p)\left(1 - \frac{1-p^{M+1}}{1-p}\right) - \log_2(1-p)(1 - p^M) & \text{if } p \geq 1/2. \end{cases} \quad (3.121)$$

This corresponds to coding the lengths of 1's and 0's and splitting them up on the intervals $(0, 0.5)$ and $(0.5, 1)$ respectively. The reason for doing so is because the entropy of the code is approximately linear from 0 to 0.5, so $H^*(p, M)$ is a triangular curve with maximum $H^*(0.5, M) = 2$ and minimum at $H^*(0, M) = 0$

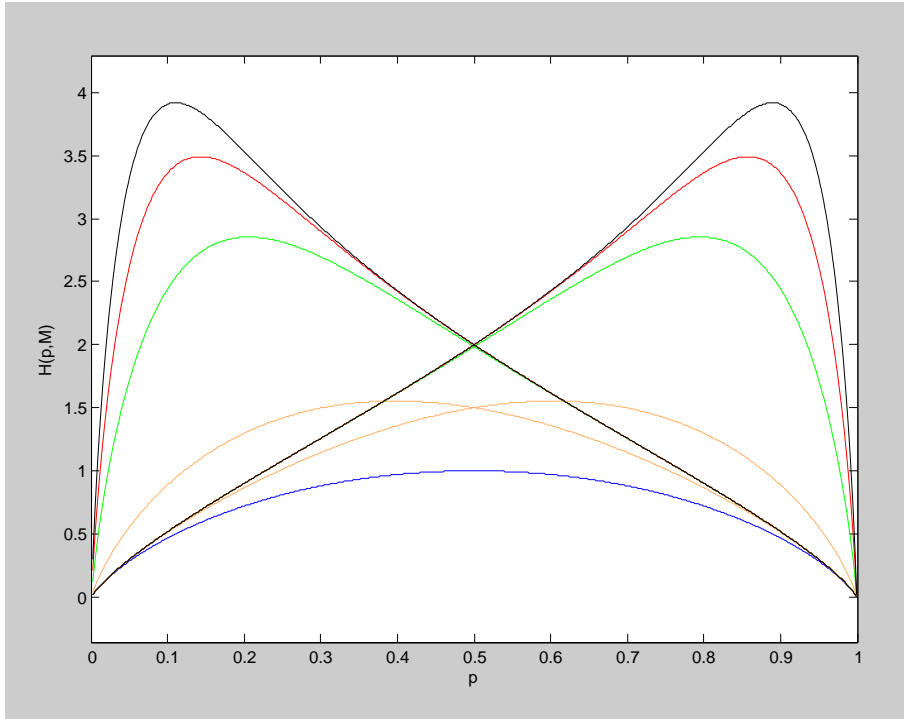


Figure 3.1: The entropy function for the run-length code. The graph is mirrored since it shows both the code for runs of 1's and 0's. In order of increasing value of maximum we have $M = 1$ (blue), $M = 2$ (orange), $M = 7$ (green), $M = 12$ (red) and $M = 17$ (black).

and $H^*(1, M) = 0$. This function will be very sensitive to changes in p , which is the point of this test. The theoretical entropy under H_0 is calculated as

$$H^*\left(\frac{1}{2}, M\right) = 2 - \left(\frac{1}{2}\right)^{M-1}, \quad (3.122)$$

and calculate the difference $\Delta H = |H^*(1/2, M) - \hat{H}(\hat{X})|$, which should be zero if the sequence is random.

3.9 Simulation Based Tests

In this section we outline some tests based on simulations. Since a major application for random number generators are Monte Carlo simulations, these tests should give some insight into how good an generator is for use with the Monte Carlo method. Another use is gambling, so we would also like to simulate such games to see if they perform as expected.

3.9.1 Craps Test

Craps is a casino dice game popular all over the world. This test shows how well a random number generator simulates the game by comparing the theoretical output with the actual ones produced by the simulations. [50]

We simulate N games of craps and count the number of wins and the number of throws needed to end each games. The game is played with two dices, if the sum of the eyes total 7 or 11 the thrower wins. When the total is 2,3 or 12, the thrower loses. For any other number, referred to as the throwers point i , the player must continue throwing and roll again until either he gets the point value i , resulting in a win, or a 7, resulting in a loss. Theoretically this could go on forever, but generally some upper limit for number of throws is presented.

The mathematics of this can easily be derived. We start by looking at table 3.4. In the table, $P(i = k)$ is the probability of roll a point i and $P(\text{win}|i = k)$ is the probability that the point $i = k$ will be rolled before 7 given that the first throw did not end the game. The probability of winning is thus

$$\sum_{i=2}^{12} P(i = k)P(\text{win}|i = k) = \frac{244}{495}. \quad (3.123)$$

We see that the probability of winning a game is $p = 244/495 \approx 0.49$. For N games the Bernoulli trial $B(N, p)$ is approximately normally distributed with mean $\mu = Np$ and variance $\sigma^2 = Np(1 - p)$, according to the de Moivre-Laplace theorem. The number of throws needed to win a game could be infinite, but for ease of calculation all games of strictly more than 20 throws are grouped. We split each game on how many throws was performed on the game. Assume N is large enough, then each of these are normally distributed and we can conduct a

k	$P(i = k)$	W/L	$P(\text{win} i = k)$
2	$\frac{1}{36}$	L	0
3	$\frac{2}{36}$	L	0
4	$\frac{3}{36}$		$\frac{3}{9}$
5	$\frac{4}{36}$		$\frac{4}{10}$
6	$\frac{5}{36}$		$\frac{5}{11}$
7	$\frac{6}{36}$	W	1
8	$\frac{5}{36}$		$\frac{5}{11}$
9	$\frac{4}{36}$		$\frac{4}{10}$
10	$\frac{3}{36}$		$\frac{3}{9}$
11	$\frac{2}{36}$	W	1
12	$\frac{1}{36}$	L	0

Table 3.4: Table of outcomes for the game craps taken from [50].

χ^2 test on the number of throws comparing the theoretical distribution with the empirical one. [5]

3.9.2 Pi Estimation Test

An interesting example of the Monte Carlo method is estimating decimals of π . This tests shows in a very direct way how appropriate an generator is for Monte-Carlo simulations. We will begin with a theorem that lay the foundation for the Monte Carlo method.

Theorem 3.9.1 (Monte Carlo Integration). *If X_1, X_2, \dots, X_n are i.i.d. random variables with probability distribution function $f(x)$ then for a function $g(x)$ an estimator is*

$$G_N = \frac{1}{N} \sum_{i=1}^N g(X_i), \quad (3.124)$$

where

$$E(G_N) = \int_{-\infty}^{\infty} f(x)g(x)dx, \quad (3.125)$$

$$\text{Var}(G_N) = \frac{1}{N} \text{Var}(g) \quad (3.126)$$

As $N \rightarrow \infty$ the distribution of possible values of G_N narrows around the mean and the probability of finding G_N some fixed distance away from $E(G_N)$ becomes smaller. We

see that the error decreases proportionally to $1/\sqrt{N}$.

It is thus possible to choose f and g so that π is estimated, which is the basic idea behind this test.

Example 9 (One-dimensional estimator). Consider the known integral for $f(x) = I(x \in (0, 1))$ and $g(x) = \sqrt{1 - x^2}$,

$$\int_0^1 f(x)g(x)dx = \frac{\pi}{4} \quad (3.127)$$

If we apply theorem 3.9.1 and sample f from a uniform random variable u_i a total of N times we get

$$G = \frac{1}{N} \sum_{i=1}^N \sqrt{1 - u_i^2} \approx \frac{\pi}{4}. \quad (3.128)$$

From a computational point of view calculating $\sqrt{1 - u_i^2}$ can be a problem, in the two-dimensional case below we do not have to perform this calculation.

Example 10 (Two-dimensional estimator). Consider the unit circle in the first quadrant of the xy plane. Define the function $f(x, y) = I((x, y) \in (0, 1)^2)$ and $g(x, y) = I((x, y) \in \{(x, y) : x^2 + y^2 \leq 1\})$. The area of the quarter circle can easily be calculated as

$$A = \int_0^1 \int_0^1 f(x, y)g(x, y)dxdy = \frac{\pi}{4}, \quad (3.129)$$

If we assume that x and y are independent, we get $f(x, y) = f(x)f(y)$, and we see that $f(x, y)$ can be sampled by drawing two i.i.d. uniforms u_i and v_i and forming an estimator G as in theorem 3.9.1. We can thus estimate π as

$$4G = 4 \frac{1}{N} \sum_{i=1}^N g(u_i, v_i) \approx \pi. \quad (3.130)$$

From theorem 3.9.1 we have $E(4G) \rightarrow 0$ and $\text{Var}(G) \rightarrow \text{Var}(g)/N$ we can define the test statistic

$$\Pi = \pi - 4G, \quad (3.131)$$

where G and g are as in either example 9 or 10, $E(\Pi) = 0$ and $\text{Var}(\Pi) = \frac{16\text{Var}(g)}{N}$. [20]

3.9.3 Ising Model Tests

The Ising Model is an model of ferro-magnetism in statistical physics. The model allows one to identify phase transitions. It is often simulated in three dimensions using the Monte Carlo method with Wolfs or Metropolis algorithms, but the two-dimensional square lattice model is analytically solvable. This is the main reason for it being interesting in the context of random number testing. [16]

In the model the magnetic dipole moments of atomic spins are represented with one of two states. We consider a lattice consisting of many such spins able to interact only with its neighbors. We begin by showing the *Ising Model Cluster test*. In this test we take a binary random bit sequence and put the bits in a L^2 square lattice. We map all 1's in the lattice as "spin up" and all 0's as "spin down", and thus have a total of 2^{L^2} equally weighted configurations of the lattice. Define the quantity average magnetization for $S_i = 1$ if i represents "spin up" and $S_i = -1$ if i represents "spin down" by

$$m = \frac{1}{L^2} \sum_{i=1}^{L^2} S_i, E(m) = 0. \quad (3.132)$$

Now we look at the distribution of connected spins, or clusters of size s on the lattice. The distribution of cluster sizes are given through $E(C_s) = sp^s D_s(p)$ in which $D_s(p)$ are polynomials in $p = 1/2$ and the normalization condition is

$$\sum_{s=1}^{\infty} E(C_s) = 1. \quad (3.133)$$

Enumeration of the polynomials $D_s(p)$ is a difficult combinatorial problem, assume that the enumerations up to a known constant l are known. Then we can calculate the size of clusters within $s = 1, \dots, l$ and denote them by $S_l^{(k)}$. Repeat this procedure N times yielding

$$S_l = \frac{1}{N} \sum_{k=1}^N S_l^{(k)}. \quad (3.134)$$

The theoretical counterpart is

$$s_l = \sum_{s=1}^l s C_s, \quad (3.135)$$

and we can calculate the empirical standard deviation for $S_l^{(k)}$, denoted σ_l . This gives the final test statistic

$$g_l = \frac{S_l - s_l}{\sigma_l}, \quad (3.136)$$

which should be standard normally distributed. [47]

Now we take a look at the *Ising Model Autocorrelation Test*. The autocorrelation

function C_A for some physical quantity A is defined as

$$C_A(t) = \frac{E(A(t_0)A(t_0 + t)) - E(A(t_0))^2}{E(A(t_0)^2) - E(A(t_0))^2}, \quad (3.137)$$

in which t denotes time. Consider a two-dimensional Ising model with critical coupling point K_c . The correlation length associated with model diverges at K_c , so we expect the system to be particularly sensitive to additional spatial correlations due to our random numbers. [47]

The total energy of the system is given by the Hamiltonian

$$H(\sigma) = - \sum_{i < j} J_{ij} \sigma_i \sigma_j - h \sum_i \sigma_i, \quad (3.138)$$

where the strength of attraction between particles i and j has been denoted J_{ij} and h is the strength of an external magnetic field. In the Ising model we only consider interactions between adjacent points on the lattice and the strength of attraction is assumed equal for all adjacent pairs. Thus, J_{ij} for all nonadjacent pairs and $J_{ij} = J$ for adjacent ones. Furthermore we assume the external magnetic field to be 0.

The Wolff algorithm works as follows: [16, p. 260]

1. Select a lattice point x randomly uniform.
2. Form a cluster around the x :
 - (a) For each neighboring point with the same polarity as x , add them to the cluster with probability $1 - e^{-2\beta J}$, here β denotes Boltzmann's constant;
 - (b) For each added point, check each of its unchecked neighbors and add those with the same polarity as x with probability $1 - e^{-2\beta J}$;
 - (c) Repeat until no unchecked neighbors left.
3. Change the polarity of all lattice points in the cluster.

We use the Wolff algorithm on a square lattice, and measure energy E , magnetic susceptibility $\tilde{\chi}$ and the normalized size of the flipped clusters c . Then calculate the corresponding integrated autocorrelation times \tilde{A} from C_A , for $A = E, \tilde{\chi}, c$, following the procedure described in [47], and scale them to the time unit of one Monte Carlo step. Since the exact value of the energy E is known for the two-dimensional Ising model, this serves as our reference value. The quantities $\tilde{\chi}$ and c can be used for evaluating the relative performance of random number generators. [47]

3.10 Other Tests

Here tests that did not fit into the other subsections are found. Notably three of them are derived from concepts of algorithmic information theory.

3.10.1 Bit Persistence Test

This is a very simple test that exploits a very common weakness of some random number generators. In this test we simply calculate the XOR on succeeding integers and record the result. The result will be a bit-mask that quickly exploits generators that for some reason have fixed bits, be it because of a bad generator or construction error. Such an error will undoubtedly cause the generator to fail more complicated tests. Once such bits have been identified one can simply mask these bits out of the sequence and perform other tests on the remaining bits. [5]

3.10.2 GCD Test

The *greatest common divisor* (GCD) test uses Euclid's algorithm with the pair u, v to calculate the GCD. For u, v drawn uniformly random from the interval $[1, 2^s]$ Euclid's algorithm will give us three objects of interest: (i) number of iterations k needed to find $GCD(u, v)$, (ii) a variable length sequence of partial quotients and (iii) the value of $GCD(u, v)$. In this test we are interested in the distribution of (i) the number of iterations k and (iii) the values of $GCD(u, v)$. [31]

For s big enough, the probability function for the greatest common divisor can be derived to be approximately

$$P(GCD(u, v) = j) = \frac{6}{\pi^2 j^2}. \quad (3.139)$$

An approximation for the probability distribution of k is derived to be binomial with an empirically calculated factor. See the original work by Marsaglia et al. [31] for details. A standard goodness-of-fit test for χ^2 distributions is applied to both $GCD(u, v)$ and k . What is referred to as the GCD test usually includes both of these tests. [31]

3.10.3 Solovay-Strassen Probabilistic Primality Test

This test is based on algorithmic information theory, and uses the Solovay-Strassen algorithm for finding out if a number is prime or not.

When using the Solovay-Strassen algorithm to find out if a number n is prime, the algorithm will randomly select a number i from the range $[1, n-1]$, and reach either the verdict that n is composite, thus ending the algorithm, or undecided. When the verdict is undecided, the algorithm draws a new integer i from $[1, n-1]$ and repeats up to k times. If the verdict n composite was never reached, we conclude that the number is probably prime. [6, p. 259]

A Carmichael number is a composite positive integer n for which the congruence

$$b^n \equiv b \pmod{n}, \quad (3.140)$$

holds for all integers b relative prime to n . Carmichael numbers are known for passing the Fermat primality test despite being composite. In contrast, the

Solovay-Strassen algorithm will incorrectly say that a composite number is probably prime with at most probability 2^{-k} . [8]

It has been shown that selecting k natural numbers between 1 and $n - 1$ is equivalent to choosing a binary string s of length $n - 1$ with k 1's in a way that the i th bit is 1 if and only if i is selected. In this test, we use the Solovay-Strassen algorithm on Carmichael numbers with prefixes of the sample strings as the binary string s . We create an metric by seeing how large a value of k is needed to reach the correct verdict of non-primality for a set number of Carmichael numbers. [8]

The test, the way it is presented in [8, p.7], is used mainly to compare generators, and does not present any theoretical distribution. However, extensive work have been done in bettering the estimated error probability, see for example [11]. With a good estimate of the probability of error, deriving a theoretical distribution for the above mentioned metric would seem possible.

3.10.4 Book Stack Test

As the name implies, this is a test based on the sorting of a book stack, and it has been shown to be a very good test for random number generators. In each step a book is chosen at random and put at the top of the stack. The main idea is to count the occurring frequencies of numbers of the symbols, unlike the *frequency test* described in subsection 3.3.1 which counts the occurrence frequencies of symbols in a sample. This subsection is based on the original paper on the Book Stack test by B. Ya. Ryabko and A. I. Pestunov. [44]

We start by defining an ordered alphabet $\mathcal{X} = \{a_1, a_2, \dots, a_S\}$, with $S > 1$. We want to the test hypothesis $H_0 : p(a_1) = p(a_2) = \dots = p(a_S) = 1/S$ for some sample x_1, x_2, \dots, x_n , against $H_1 = \neg H_0$.

The transformation in this test is best described by a function $v^t(a)$ the number of symbol $a \in \mathcal{X}$ after analyzing x_1, x_2, \dots, x_{t-1} with the arbitrary initial order $v^1(\cdot)$.

$$v^{t+1}(a) = \begin{cases} 1 & \text{if } v^t(a) = v^t(x_t) \\ v^t(a) + 1 & \text{if } v^t(a) < v^t(x_t) \\ v^t(a) & \text{if } v^t(a) > v^t(x_t) \end{cases} \quad (3.141)$$

For H_0 true, the probability for a symbol with any number to occur in the sample is $1/S$, and if H_1 is true the probability of some of the symbols is greater than $1/S$ and their numbers will on average be less those of symbols with less probabilities. For the purpose of applying the test we split the set of all numbers $\{1, \dots, S\}$ into $r > 1$ disjoint subsets $A_1 = \{1, 2, \dots, k_1\}$, $A_2 = \{k_1 + 1, \dots, k_2\}$, ..., $A_r = \{k_{r-1} + 1, \dots, k_r\}$. For $j = 1, \dots, r$ we denote the quantity n_j for a sample x_1, x_2, \dots, x_n to be how many numbers $v^t(x_i)$ belong to the subset A_j . In the case that H_0 holds true, the probability of $v^t(x_i)$ belonging to A_j is proportional to $|A_j|/S$. The χ^2 test is thus applied to n_1, \dots, n_r in order to verify the hypothesis H_0^* as described below against its inverse $H_1^* = \neg H_0^*$.

$$H_0^* : P\{v^t(x_t) \in A_j\} = |A_j|/S \quad (3.142)$$

Since H_0 true implies that H_0^* is true and H_1^* true implies that H_1 is true application of the described test is correct. The cardinality of A_j is determined experimentally. This method has been experimentally shown to be better (in computational time and/or accuracy) than most other statistical tests.

3.11 Live Tests

Here we outline ideas for how some of the mentioned tests can be modified for live testing.

3.11.1 Sliding Window Testing

The concept of looking at data through overlapping sliding windows is in no way anything new, but we have not found any signs of it being used the way we will use it here before. We would like to propose a model for live-testing of random number generators using sliding windows.

Start by defining windows (L_1, L_2, \dots) , each of length l , that overlap with the previous and next window over an interval of length L . The test will start by using data in the first window W_1 to initiate the process. In each read cycle the program will collect L new points of random data from the random number generator and discard the L oldest, generating the next window. We will call this process updating the window. At some point during the update, the random number test will be performed using primarily the test statistics from the previous test as well as the L new and the L discarded bits of data.

The tests implemented should be selected so that minimal calculations will be needed after each update of the window, since the window will be updated very frequently. At best, these tests should be so computationally simple that they are realizable with simple electronics, and could thus be included as a status indicator on the product. As reference, the Quantis random number generator has a bandwidth of 4 MB/s, leaving very little time for computation.

All tests are of course applicable to the above scheme, but some are more appropriate than others. To evaluate if a test is appropriate we should focus on to what extent the calculations in each update can be re-used and how computational the test will be. Calculating p -values for tests will include the error function for real values. We can often tabulate the value of the error function for common distributions and perform goodness-of-fit tests very computationally cheap.

Example 11 (Run-length Entropy test). *This test was designed with the intention of being used as a sliding window test. Let the run counts be X_0, \dots, X_{M-1} . Whenever a window is updated the leaving runs are subtracted from the counts and the incoming ones are added, thus we only need to look at the whole window when we initiate the*

test. The entropy is calculated as in subsection 3.8.5. Since M rarely is above 15, even for long windows, the entropy calculation is calculation-wise very cheap. It should also be very sensitive to strays from the probability $p = 0.5$, which was already argued.

One test among those covered in this paper stands out in that it has been shown to be an exceptionally good test for quantum random number generators, the Borel normality test (see [8]). This test is another great example of a test that can be applied in the above manner since it is probable that an hard-to-see hardware error in an quantum random number generator would have algorithmic characteristics. The application would much resemble the scheme of the above example, so it will not be covered in detail.

3.11.2 Convergence Tests

One problem with the sliding window approach described in subsection 3.11.1 is that it does not consider anything beyond the scope of the window. Convergence tests look at larger windows of data without increasing computational complexity. This scheme is best described with a good example.

Example 12. *Recall the π estimation test described in subsection 3.9.2. The test converges to the exact value of π with a known rate of convergence. Furthermore π is known to high precision. Thus we could easily construct a test that calculates π to a set accuracy. If the rate of convergence strays this indicates that the generator is not generating random enough numbers. Not that we do not need to know the exact rate, we just need to see that $\text{Var}(G) \propto 1/N$ as we increase N . Recall that there were two methods for calculating π , the two-dimensional estimator is to be preferred in order to avoid having to calculate the square root of a decimal number. Of course, storing π to infinite accuracy in the generator is not realistic, so running out of comparing digits could be a problem. To solve this, we simply define a sufficiently high accuracy of π and run two overlapping simulations with a fair bit of lag between them. If we do not do so there will be a window of uncertainty when we restart the test.*

This methodology could of course be applied to any simulation with known rate of convergence, for example the random walk tests described in subsection 3.5.3 or, assuming we have a known distribution, the Solovay-Strassen Probabilistic Primality test from subsection 3.10.3. These tests will mainly tell us if the sequence is statistically random, they say very little about unpredictability.

4

Random Number Test Suites

Depending on for what application a RNG will be used, it is advisable to focus on different aspects of it. For cryptography the unpredictability presented by a QRNG is very appealing, whereas a Monte Carlo simulation will benefit more from an even distribution and high bit-rate. The test suites outlined in this work are intended to test the RNGs for both of these purposes, although a more thorough cryptanalysis is recommended as a complement to statistical testing for cryptographic applications.

Each suite is presented with a short summarizing table. Note that the field "number of tests" presented here is an approximation. Depending on how they are implemented, each test could present an infinite amount of variability and as such, the number of named tests is somewhat trivial. It is, however, presented here as an indication of how big the test suite is, rather than of how good it is.

Diehard and NIST STS are suites developed as test batteries, and are used for various certifications (for example in gambling). The test suites DieHarder and TestU01 are bigger and intended for academical use, and as such will have a more thorough evaluation. ENT and SPRNG are developed for other reasons, and are only briefly mentioned.

4.1 Diehard

The *Diehard Battery of Test of Randomness*, often only referred to as the Diehard test suite, is in many ways the pioneer of random number test suites. The primary developer, George Marsaglia, was a prominent researcher with a focus on pseudo-random number generation and randomness tests. Most of the tests are taken from his own research.

Name	Diehard
Primary developer	George Marsaglia
Associated with	Florida State University, United States
License	Copyright George Marsaglia, source code publicly released
Last updated	1995
Latest version	-
Platforms	Windows, Linux, Unix, Dos
Number of tests	15
Programming language	C and Fortran
Ease of use	Easy to use from command line.
Documentation	Available in source code.
Distributed through	CD-ROM, http://www.stat.fsu.edu/pub/diehard/

Table 4.1: A summarizing table for the Diehard test suite. [29]

In a keynote speech from 1985 [30] Marsaglia briefly explains his dissatisfaction with random number testing at the time. He notes that the book by Knuth [21], which outlines several tests, is too good in the sense that it discourages people from developing new tests. The above, along with a growing market for true random number generators, prompted him to develop the Diehard Battery. With the battery he also presents data from three so called true random number generators, as well as several PRNGs. [29, cdmake.ps]

The suite is bundled with 4.8 billion random bits, generated using three sources of white noise, music and pseudo-random numbers. The white noise was acquired from three physical sources of randomness and the music is taken from digital recording of rap music. He also includes and compares 5 PRNGs in the suite.

Several weaknesses of Diehard have been observed since its release. From a theoretical point of view there is a noticeable lack of bit-level randomness tests and tests that focus on cryptographic strength. Furthermore, the way the tests are implemented they offer the user little to no parametric variability. [5]

As recently as 2010 METAS used Diehard as a tool for certifying that the Quantis QRNG produces satisfactory random numbers. [37] Finding similar certificates recently issued by other organizations is not hard. The Australian company iTech Labs issued certificates for gambling websites using Diehard as recently as 2013. [19]

4.1.1 Tests

Diehard consists of 15 district tests based on Marsaglias research: [29]

1. Birthday Spacings Test (see subsection 3.4.4)

2. Overlapping 5-Permutation Test
3. Binary Rank Test (see subsection 3.5.2)
4. Bitstream Test (see subsection 3.4.2)
5. Monkey Tests (OPSO, OQSO and DNA) (see subsection 3.4.3)
6. Count-the-Ones Tests
7. Parking Lot Test
8. Minimum Distance Tests (see section 3.7)
9. Squeeze Test
10. Overlapping Sums Test
11. Runs Test (see subsection 3.3.10)
12. Craps Test (see subsection 3.9.1)

The tests that are not referenced in the list have not been covered in this work, but information about them can be found through the Diehard homepage [29]. Generally speaking, these tests are good for evaluating statistical randomness and cover many of the common flaws of simpler PRNG.

4.2 NIST STS

Name	STS
Primary developer	National Institute of Standards and Technology
Associated with	U.S. Department of Commerce, United States
License	Pursuant to title 17 Section 105 of the United States Code this software is not subject to copyright protection and is in the public domain.
Last updated	2010-08-11
Latest version	2.1.1
Platforms	Linux, Mac OS X
Number of tests	15
Programming language	ANSI C
Ease of use	Easy to use from command line.
Documentation	http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf
Distributed through	http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

Table 4.2: A summarizing table for the NIST STS test suite. [42]

The *National Institute of Standard and Technology* developed their statistical test suite STS with the flaws of Diehard in mind. The main documentation for the test suite is a paper entitled *A Statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications*, from the title it is clear that this test suite is intended for assuring that random number generators are suitable for cryptographic applications. [42]

In the documentation to STS the developers point out that no set of statistical tests can absolutely certify that a certain generator is appropriate for use in cryptographic applications, however they emphasize that such tests can be used as a first step in evaluating a certain generator for use in cryptographic applications. [43]

The test suite offers an very thorough documentation, outlining both the theoretical background of all its tests, their significance as well as how they were implemented. As was illustrated by the frequency test in subsection 3.3.1, the same test can be implemented in many ways, which shows the importance of good documentation. It is bundled with several PRNGs. [43]

4.2.1 Tests

STS covers a wide variety of tests, for detailed explanation of the tests as implemented in STS, please see [43]. The tests listed are named as in the STS test suite, notably the Runs test here is not the same as the one in subsection 3.3.10.

1. Frequency Test (see subsection 3.3.1)
2. Frequency Test within a Block (see subsection 3.3.2)
3. Runs Test (different from subsection 3.3.10, similar to subsection 3.3.9)
4. Longest-Run-of-Ones in a Block Test
5. Binary Matrix Rank Test (see subsection 3.5.2)
6. Discrete Fourier Transform (Spectral) Test (see subsection 3.6.1)
7. Non-overlapping Template Matching Test
8. Maurer's "Universal Statistical" Test (See subsection 3.8.2)
9. Linear Complexity Test (See subsection 3.8.3)
10. Serial Test (See subsection 3.3.3)
11. Approximate Entropy Test (see subsection 3.8.1)
12. Cumulative Sums Test (similar to subsection 3.5.3)
13. Random Excursion Test (similar to subsection 3.5.3)
14. Random Excursion Variant Test (similar to subsection 3.5.3)

Despite the fact that the test suite in general holds a high quality, the developers NIST have recently been exposed of intentionally trying to encourage use of a

flawed deterministic random bit generator for data encoding, as shown through documents leaked in 2013. Even though there is no proof to suggest that this test suite has been tampered with in any way, many organizations might not be keen on using a product developed by an organization with such an background. [17]

4.3 ENT

Name	ENT
Primary developer	John Walker
Associated with	Fourmilab, Switzerland
License	GNU GPL
Last updated	2008-01-28
Latest version	-
Platforms	Windows, Linux
Number of tests	5
Programming language	C++
Ease of use	Easy to use from command line.
Documentation	http://www.fourmilab.ch/random/
Distributed through	http://www.fourmilab.ch/random/

Table 4.3: A summarizing table for the ENT test suite. [49]

ENT is a simple test suite for testing random number sequences. It primarily has one test that stand out, the Monte Carlo estimation for pi. The suite offer very limited documentation, but serves as an simple command line tool for a quick and simple evaluation of a random number sequence. It seems to have been developed more as a teaching tool for understanding random numbers and goodness-of-fit testing than as a tool intended for use within research. It is not bundled with any PRNG or other source of randomness. [49]

4.3.1 Tests

The suite offers the following tests, inspired from various sources:

1. Entropy Test (see subsection 3.8.1)
2. Chi-square Test (see subsection 3.2.1)
3. Arithmetic Mean Test
4. Monte Carlo Value for Pi (see subsection 3.9.2)
5. Serial Correlation Coefficient (see subsection 3.5.1)

Name	SPRNG
Primary developer	Michael Mascagni
Associated with	Florida State University, United States
License	Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License
Last updated	2012-03-16
Latest version	4.4
Platforms	Linux, various parallel machines
Number of tests	12
Programming language	C++ and Fortran
Ease of use	Requires basic knowledge of C++ and Fortran.
Documentation	http://www.sprng.org/Version4.0/users-guide.html
Distributed through	http://www.sprng.org/

Table 4.4: A summarizing table for the SPRNG test suite. [32]

4.4 SPRNG

SPRNG has a very different focus from the other test suites described in this work. The name stands for *The Scalable Parallel Random Number Generators Library*, and that is just what it is intended to be. The expressed goal for this project is to *"develop, implement and test a scalable package for parallel pseudo-random number generation which will be easy to use on a variety of architectures, especially in large-scale parallel Monte Carlo applications."* The library has a PRNG section, offering a wide variety of generators, and this is where the focus of the library lies. [32]

4.4.1 Tests

The test suite included in the library is split into a group of statistical tests, taken from [21] and physical tests taken from [47], the full list of tests follows: [32]

1. Collisions Test (see subsection 3.4.1)
2. Coupon Collector's Test (see subsection 3.3.8)
3. Equidistribution Test (see subsection 3.3.1)
4. Maximum-of- t Test (see subsection 3.3.5)
5. Permutations Test (see subsection 3.3.4)
6. Poker Test (see subsection 3.3.7)
7. Runs Up Test (see subsection 3.3.10)
8. Serial Test (see subsection 3.3.3)
9. Sum of Distributions Test

10. Gap Test (see subsection 3.3.9)
11. Ising Model Test (see subsection 3.9.3)
12. Random Walk Test (2D)

4.5 TestU01

Name	TestU01
Primary developer	Pierre L'Ecuyer
Associated with	Université de Montréal, Canada
License	Copyright Pierre L'Ecuyer, distributed under GNU GPL
Last updated	2009-08-18
Latest version	1.2.3
Platforms	Windows, Linux
Number of tests	63
Programming language	ANSI C
Ease of use	Requires basic knowledge of ANSI C.
Documentation	http://simul.iro.umontreal.ca/ testu01/guideshorttestu01.pdf
Distributed through	http://simul.iro.umontreal.ca/ testu01/tu01.html

Table 4.5: A summarizing table for the TestU01 test suite. [24]

TestU01 is a library developed over a very long period by French-Canadian Pierre L'Ecuyer. What started as implementations of the tests described in Knuth's book [21] has grown into a grand scale library, implementing years of L'Ecuyer's and other's research. The library has either the exact tests or similar (often improved) tests to those listed under Diehard and STS, with a few exceptions, primarily of Diehard tests that the author has deemed of lesser importance. The documentation offers an very thorough explanation of the tests, including a fair amount of theoretical explanation, and how to use the library.

The library includes a wide variety of PRNGs. The full library covers dozens of PRNGs, both very simple ones, such as the multiply-with-carry generator, state of the art generators used in cryptography, such as the AES OFB, and quick generators with good distribution properties commonly used in Monte Carlo simulations, such as the Mersenne Twister. [25]

4.5.1 Tests

Instead of listing specific tests, we will list and explain what the specific modules focus on. Tables 4.6 and 4.7 lists some of the many tests available. For a detailed explanation of the modules and tests as implemented in TestU01, please see [25].

1. **smultin**
Tests using the power divergence test statistic.
2. **sentrop**
Various entropy tests.
3. **snpair**
Tests based on the distance between closest points.
4. **sknuth**
Tests covered in [21] by Donald E. Knuth.
5. **smarsa**
Various tests developed by George Marsaglia.
6. **svaria**
Various tests based on relatively simple statistics.
7. **swalk**
Tests based on the discrete random walk.
8. **scomp**
Tests based on compression theory.
9. **sspectral**
Tests based on transformation theory.
10. **sstring**
Various tests that are applied to strings of random bits.
11. **sspacings**
Tests based on the sum of the spacings between the sorted observations of independent samples.

4.5.2 Batteries

TestU01 includes a number of batteries intended for various use. Notably it also includes one battery created with the intent of evaluating hardware random number generators, named Alphabit.

SmallCrush Battery

A light collection of tests intended to evaluate a generator and run quickly.

Crush Battery

A collection of tests intended to determine the state of a generator with high certainty. See table 4.7 for a list of test.

BigCrush Battery

A heavy collection of tests intended to thoroughly determine the state of a generator with very high certainty.

Rabbit Battery

A collection of tests that are selected to determine a wide variety of non-randomness and at the same time run quickly.

Test	Description
Binary Overlapping Serial	The Overlapping serial test applied to binary sequences, see subsection 3.3.6.
Hamming Independency	Applies an independency test to the Hamming weights of L -bit blocks.
Hamming Correlation	Applies an correlation test to the Hamming weights of L -bit blocks.
Random Walk	See subsection 3.5.3.

Table 4.6: List of tests in the TestU01 Alphabit Battery. See [25] for details.

Alphabit Battery

A collection of tests designed to test hardware random bits generators. See table 4.6 for a list of tests. They have been selected to detect correlations between successive bits.

BlockAlphabit Battery

Applies Alphabit Battery of tests repeatedly after reordering the bits by blocks of different sizes.

pseudoDiehard Battery

A collection of tests included in the Diehard battery of the tests.

FIPS_140_2 Battery

Includes tests from the FIPS PUB 140-2 [35] standard from NIST.

All of these batteries can be run in repeat mode, meaning that if a test has suspect p -value it will rerun that test a predetermined number of times. This is useful since we expect that even a good random number generator should fail a test, albeit rarely.

4.5.3 Tools

TestU01 offers various tools for handling in-data and the results of the tests. Notably the *ufile* module contains lots of functions for reading random data from files, the *swrite* module contains functions for writing the statistical test results and *scatter* makes scatter plots of the results.

4.6 DieHarder

DieHarder is a random number generator testing suite, designed to permit one to push a weak generator to unambiguous failure. The name is a movie sequel pun to Diehard. It incorporate generalized versions of Diehard as well as some other tests, listed below. The suite also has an added goal of being usable from the command line in Linux, which makes it very user friendly, and to be easy to implement in other programs. It also includes a wide variety of PRNGs.

DieHarder is notably different from TestU01 in that it is made to be an easy to

Test	Description
Overlapping Serial	See subsection 3.3.6.
Overlapping Collision	Collision test applied with overlapping samples.
Birthday Spacings	See subsection 3.4.4.
Close-point Spatial (L_p -norm)	See subsection 3.7.
Close-point Spatial (Binary Metric)	Applied with alternative binary distance measure.
Poker	See subsection 3.3.7.
Coupon Collector	See subsection 3.3.8.
Gap	See subsection 3.3.9.
Run	See subsection 3.3.10.
Permutation	See subsection 3.3.4.
Collision Permutation	The Collision test applied with samples as in the Permutation test.
Max-of-t	See subsection 3.3.5.
Sample Product	Empirical distribution of products of samples.
Sample Mean	Empirical distribution of the mean of samples.
Sample Correlation	Empirical distribution of the correlation of samples.
Maurer's Universal	See subsection 3.8.2.
Weight Distribution	Like Gap test, but counts how many samples falls into a gap.
Sum Collector	Similar to the Coupon Collectors test but instead checks if a sum is surpassed.
Matrix Rank	See subsection 3.5.2.
Modified Savir	Generates $S_1 \in \{1, \dots, m\}$, $S_2 \in \{1, \dots, S_1\}$ etc. Compares with the theoretical distribution.
GCD	See subsection 3.10.2.
Random Walk	See subsection 3.5.3.
Linear Complexity	See subsection 3.8.3.
Lempel-Ziv	See subsection 3.8.4.
DFT CLT	See subsection 3.6.3.
Longest Head Run	Counts longest run of 1's in blocks.
Periods in Strings	Test based on the distribution of the correlations in bit strings.
Frequency within a Block	See subsection 3.3.2.
Hamming Independency	See table 4.6.
Hamming Correlation	See table 4.6.
STS Runs	Looks at runs of binary string, similar to the Gap test.
Autocorrelation	Applies an Autocorrelation test on XOR of strings.

Table 4.7: List of tests in the TestU01 Crush Battery. See [25] for details.

Name	DieHarder
Primary developer	Robert G. Brown
Associated with	Duke University, United States
License	Copyright Copyright Robert G. Brown. distributed under GNU GPL
Last updated	2013-11-13
Latest version	3.31.1
Platforms	Linux
Number of tests	26
Programming language	ANSI C
Ease of use	Easy to use from command line, advanced functions require knowledge of ANSI C.
Documentation	Available in source code.
Distributed through	http://www.phy.duke.edu/~rgb/General/ dieharder.php

Table 4.8: A summarizing table for the DieHarder test suite. [4]

use tool for certification, similar to STS or Diehard. Very little prior knowledge of computers is needed to use the suite.

4.6.1 Tests

DieHarder implements generalized versions of all the original Diehard tests as well as a few STS tests, some tests implemented by Robert G. Brown. (RGB) and some by David Bauer (DAB). [5]

1. All Diehard Tests
See section 4.1.
2. Marsaglia and Tsang GCD Test
See subsection 3.10.2.
3. STS Monobit Test
See subsection 3.3.1.
4. STS Runs Test
Looks at runs of binary string, similar to the Gap test.
5. STS Serial Test (Generalized)
See subsection 3.3.3.
6. RGB Bit Distribution Test
See subsection 3.3.11.
7. RGB Generalized Minimum Distance Test
Diehard Minimum distance tests generalized to higher dimensions, see [14].

8. RGB Permutations Test
See subsection 3.3.4.
9. RGB Lagged Sum Test
Calculates the mean of the sequence with n lags, then compares it with the theoretical value as in the frequency test.
10. RGB Kolmogorov-Smirnov Test
See subsection 3.2.4.
11. DAB Byte Distribution Test
Extracts n bytes from each of k consecutive words, then applies the χ^2 test.
12. DAB Discrete Cosine Transform (DCT) Test
Computes the discrete cosine transform of over independent blocks and then tested with a χ^2 test.
13. DAB Fill Tree Test
Fills binary trees of fixed depth with words, when a word can not be inserted into the tree the depth is recorded and a new tree started. Then a χ^2 test is applied.
14. DAB Fill Tree 2 Test
Similar to the normal Fill tree test, but fills with bits instead.
15. DAB Monobit 2 Test
Applies the Monobit test to blocks of data.

5

Testing

In this chapter we outline the tests performed and present the results.

5.1 Methods for Testing

The testing has been done using three predefined batteries from TestU01 and all tests in DieHarder on three PRNGs and one QRNG.

The tested random number generators were:

1. RANDU
2. Mersenne Twister
3. AES OFB
4. Quantis

The data from Quantis was acquired directly from their USB Quantum random number generator, using the original software. We generated a file of 200 GB and used this with the test suites. For the other generators we used the built in functions in each library respectively. The version of Mersenne Twister used is the original one from 1998, except for in Matlab in which the standard Matlab one was used.

These were chosen because they represent four distinctly different sources of random numbers. Quantis is a QRNG, being marketed as a "true random number generator based on quantum physics." Mersenne Twister, or rather various versions of it, is the pseudo random number algorithm that is currently most widely

used for simulations, and should have a very good distribution properties. Notably Matlab implements a version of Mersenne Twister as its standard random number generator. AES OFB is a state of the art algorithm for cryptography, and should produce unpredictable random sequences. Lastly, RANDU is a notoriously bad random number generator, and should serve as a test of the tests. Any test passing RANDU should be used with caution.

The applied batteries and tests were as listed below:

Plot Test

Generate a scatter plot using random data. The plots here have been generated using TestU01 library with the scatter module. The test generates N uniforms on a hypercube $[0, 1]^t$, for each dimension t we specify upper and lower bounds \bar{U} and \bar{L} respectively. All values falling outside of the bounds are discarded. The plot then shows the projection of all points in all dimension on two specified dimensions. Lastly the plot is cut as specified by the user. We used $N = 100000000$, $t = 3$, $\bar{L} = (0, 0, 0)$ and $\bar{U} = (0.1, 0.5, 1)$.

Bit Distribution Test

We test the generators using the Bit Distribution test from DieHarder, see subsection 3.3.11. All tests are performed with the high accuracy setting -k 1 and the resolve ambiguity mode. The test is set to run 200 times for each window of size n , with n increasing until the test fails. We then conclude that the generator is random up to n bits.

DieHarder Battery

We ran DieHarder in the standard "all" mode, meaning that it will run all tests with the default options to create a report. All tests were performed with the high accuracy setting -k 1 and the tests were performed twice, once with standard settings and once in the resolve ambiguity mode. The resolve ambiguity mode will rerun tests with weak values until they are resolved, either with a fail or a pass. All generators were run with a random seed, but the same seed were used for all tests in the same run.

Crush Battery

The main battery of TestU01. First we applied the SmallCrush battery, and then the normal Crush battery for those tests that pass SmallCrush. The batteries were run with standard settings. See table 4.7 for a list of tests.

Alphabit Battery

This battery from TestU01 is focused on finding weaknesses often found in HRNG. The battery was run with standard settings on RANDU, Mersenne Twister and two different Quantis USB devices. See table 4.6 for a list of tests.

Sliding Window Run-length Entropy Test

This test was implemented in Matlab as described in subsections 3.8.5 and 3.11.1. It was run with data from Quantis and the built-in Matlab Mersenne Twister, notably not the same version that was used for the other tests. We

also included a third erroneous stream. The test ran on windows of size 5000 bits and slid over 1000000 bits of data. Two errors were introduced into the erroneous stream, the first one having a probability of generating 1 equal to 45% between bit 1500000 and 3000000, the second one having a probability of generating 1 equal to 55% between bit 5500000 and 6000000.

5.2 Test Results

This section outlines the results of performed tests. Raw result data for the tests as taken from DieHarder and TestU01 can be found in appendix C.

5.2.1 Plot Test

This test is relatively self-explanatory. The human eye is sensitive to pattern recognition, and from observing figure 5.1 it is clear that RANDU is not a very good PRNG. The other three, however, perform very well and show no obviously visible patterns.

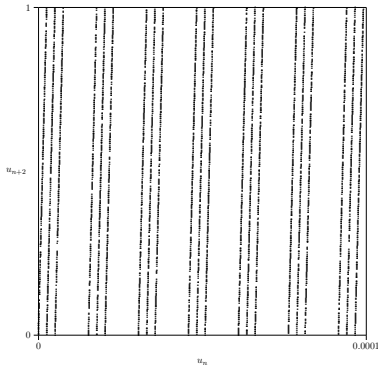
5.2.2 Bit Distribution Test

RANDU failed the test at the lowest level. The other three generators were tested to be random up to 18 bits.

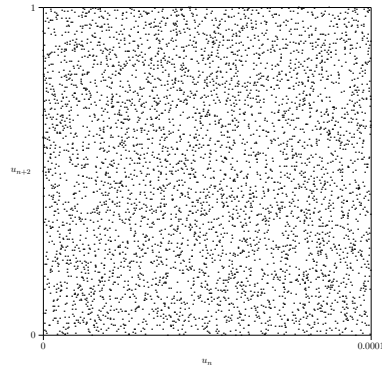
5.2.3 DieHarder Tests

The test results are summarized in table 5.1. We see that RANDU performed poor, as expected. What is interesting is however that RANDU passes the Diehard Overlapping Sums and STS Runs tests, which only Quantis fails. Part of an explanation for this can be found in the DieHarder documentation, where the comment "Do Not Use" is found next to the Diehard Overlapping Sums test, and because of this we will not analyze this result further. The STS Runs test is however more interesting. This test detects oscillations, and if there are too many or too few oscillation, it will fail a sequence. The p -value is high, indicating that there were more oscillations than should be expected.

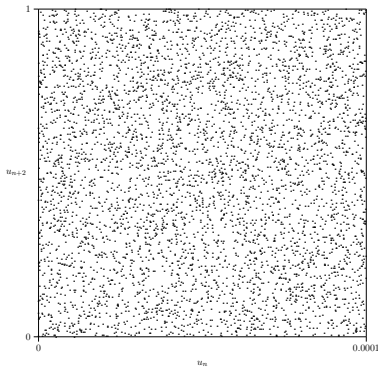
Another interesting result that is not represented in the table is that of the Lagged Sums test. With $N = 300$ tests it failed decisively for both lags of size $n = 16$ and $n = 27$, but passed for all other values of $n \in \{1, \dots, 32\}$. The collected p value for the 300 tests were close to 0, implying that they were not as uniform as expected. This might be expected for a true random sequence. The sequence is allowed to stray, in contrast to pseudo RNGs that are engineered to always produce perfectly uniform values. Of course, when used with for example Monte Carlo simulations this would could be a very bad characteristic, as certain outcomes would become more likely.



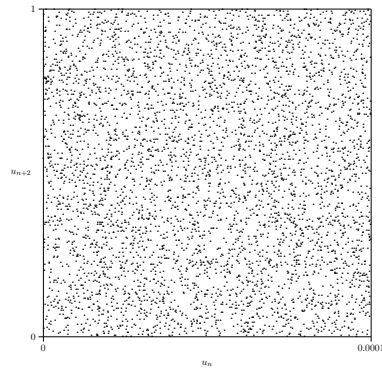
(a) RANDU



(b) AES OFB



(c) Mersenne Twister



(d) Quantis

Figure 5.1: Scatter plot using TestU01. Each dot represents a pair of uniforms distributed on $[0,1)$, the horizontal axis has scale 0 to 0.0001 and the vertical has scale 0 to 1.

Test	RANDU	Mersenne Twister	AES OFB	Quantis
Diehard Birthday Spacings	W/O	O/O	O/O	O/O
Diehard Overlapping 5- Permutation	X/X	O/O	W/O	O/O
Diehard 32x32 Matrix Rank	O/O	O/O	O/O	O/O
Diehard 6x8 Matrix Rank	X/X	O/O	W/O	O/O
Diehard Bitstream	X/X	O/O	O/O	O/O
Diehard OPSO	X/X	O/O	O/O	O/O
Diehard OQSO	X/X	O/O	O/O	O/O
Diehard DNA	X/X	O/O	O/O	O/O
Diehard Count-the-Ones Bit	X/X	O/O	O/O	O/O
Diehard Count-the-Ones Byte	X/X	O/O	O/O	O/O
Diehard Parking Lot	O/O	O/O	O/O	O/O
Diehard Minimum 2D Distance	X/X	O/O	O/O	O/O
Diehard Minimum 3D Distance	X/X	O/O	O/O	O/O
Diehard Squeeze	X/X	O/O	O/O	O/O
Diehard Overlapping Sums	O/O	W/O	O/O	W/X
Diehard Runs	W/W	O/O	O/O	O/O
Diehard Craps	X/X	O/O	O/O	O/O
Marsaglia and Tsang GCD	X/X	O/O	W/O	O/O
STS Monobit	W/O	O/O	O/O	O/O
STS Runs	O/O	O/O	O/O	W/X
STS Serial	X/X	O/O	O/O	O/O
RGB Bit Distribution	X/X	O/O	O/O	O/O
RGB Generalized Minimum Dis- tance	X/X	O/O	W/O	O/O
RGB Permutations	W/X	O/O	O/O	O/O
RGB Lagged Sum	O/O	O/O	O/O	O/O
RGB Kolmogorov-Smirnov	O/O	O/O	O/O	O/O
DAB Byte Distribution	X/X	O/O	O/O	O/O
DAB Discrete Cosine Transform	X/X	O/O	O/O	O/O
DAB Fill Tree	X/X	O/O	O/O	O/O
DAB Fill Tree 2	X/X	O/O	O/O	O/O
DAB Monobit 2	X/X	O/O	O/O	O/O

Table 5.1: This table shows the summarized results of the DieHarder tests, the left value is from the basic test and the right from the resolve ambiguously test. Here O means passed, W means weak and X means failed. Any result with a p -values within 0.000001 from 0 or 1 is considered failed and within 0.005 is considered weak. Some tests are run several times with increasing difficulty, in those cases the most stringent tests result is used to represent the test in this table. The raw test result data is found in appendix C. Quantis here refers to Quantis 1 (200 GB).

Test	Mersenne Twister	AES OFB	Quantis
Linear Complexity	X	O	O
Sample Product	O	O	X
Weight Distribution	O	O	X
Random Walk H	O	O	X
Random Walk J	O	O	X
Random Walk M	O	O	X
Hamming Independency	O	O	X
STS Runs	O	O	X
Autocorrelation	O	O	X

Table 5.2: Here O means passed, W means weak and X means failed. Any result with a p -values within 0.000001 from 0 or 1 is considered failed and within 0.001 is considered weak. The raw result data is found in appendix C. Quantis here refers to Quantis 1 (200 GB).

5.2.4 Crush Battery

First we applied SmallCrush, notably RANDU failed all tests and was excluded from subsequent testing, the other three generators passed SmallCrush. We then applied Crush to Quantis, AES OFB and Mersenne Twister. Table 5.2 lists the tests that were failed by one of the three generators, it does not include tests passed by all three. AES OFB passed all tests.

Notably Mersenne Twister fails Linear complexity tests. This might not be so surprising if one considers the construction of Mersenne Twister. The generator is constructed through linear recursions over \mathbb{F}_2 , so naturally tests looking at linear dependence in the generated binary data should fail.

When looking at the results for Quantis it notably nearly passes the Random walk tests. In [8] it is noted that the random walks for their tested QRNGs tend to stray further from the mean compared with PRNGs, which somewhat corresponds to our results. It is hard to argue that a bad random walk result over an finite walk is a sign of a bad random number generator, but we can at least conclude that the distribution might not be the best.

The Sample product test looks at the probability distribution of the product of uniforms. STS Runs, Hamming independency and Autocorrelation tests focus on bit-level oscillations, and should guarantee that bits are well mixed. [43], [25]

5.2.5 Alphabit Battery

The Alphabit battery is notably very slow when running on PRNGs, AES OFB was thus not included in this test. Mersenne Twister passed all tests, and RANDU failed all. Quantis failed all but one test, the failed tests are listed in table 5.3. Notably Alphabit applies random walks with longer step lengths than Crush, which resulted in more certain failures for all the random walk test statistics compared to the Crush Battery.

Test	Quantis 1 (8 GB)	Quantis 1 (200 GB)	Quantis 2 (8 GB)
Binary Overlapping Serial (L = 2)	X	X	X
Binary Overlapping Serial (L = 4)	X	X	X
Binary Overlapping Serial (L = 8)	X	X	X
Binary Overlapping Serial (L = 16)	O	X	X
Hamming Independency (L = 16)	X	X	X
Hamming Independency (L = 32)	X	X	X
Random Walk H (L = 64)	X	X	X
Random Walk M (L = 64)	X	X	X
Random Walk J (L = 64)	X	X	X
Random Walk R (L = 64)	X	X	X
Random Walk H (L = 320)	X	X	X
Random Walk M (L = 320)	X	X	X
Random Walk J (L = 320)	X	X	X
Random Walk R (L = 320)	O	X	X

Table 5.3: Results from the Alphabit battery. Here O means passed, W means weak and X means failed. Any result with a p-values within 0.000001 from 0 or 1 is considered failed and within 0.001 is considered weak. The raw result data is found in appendix C. The L parameter reflects how long bit-strings are used in each test respectively. Here Quantis 1 and Quantis 2 represent two different Quantis USB devices.

We already noted that failing STS Runs, Autocorrelation and Hamming independence tests imply a bad mix of 1's and 0's, thus failing the binary version of the Overlapping serial test is not surprising. The reason for these failed results could be many, but since Alphabit consist of tests chosen by one of the leading researchers in random number testing for testing HRNGs, these results are particularly interesting.

5.2.6 Sliding Window Run-length Coding Test

The test clearly identifies errors on the 5% level, as shown in figure 5.2. By calculating the mean of the vector used in the plotting we can also see that Quantis had an on average higher entropy for the run-length code, though marginally so. This would indicate that the oscillations in the sample generated by Quantis are more irregular than the sample due to Mersenne Twister. From the graph we can conclude that a 5% error corresponds to the test statistics entropy growing to around 0.2, an reasonable warning level to detect such an error would thus be

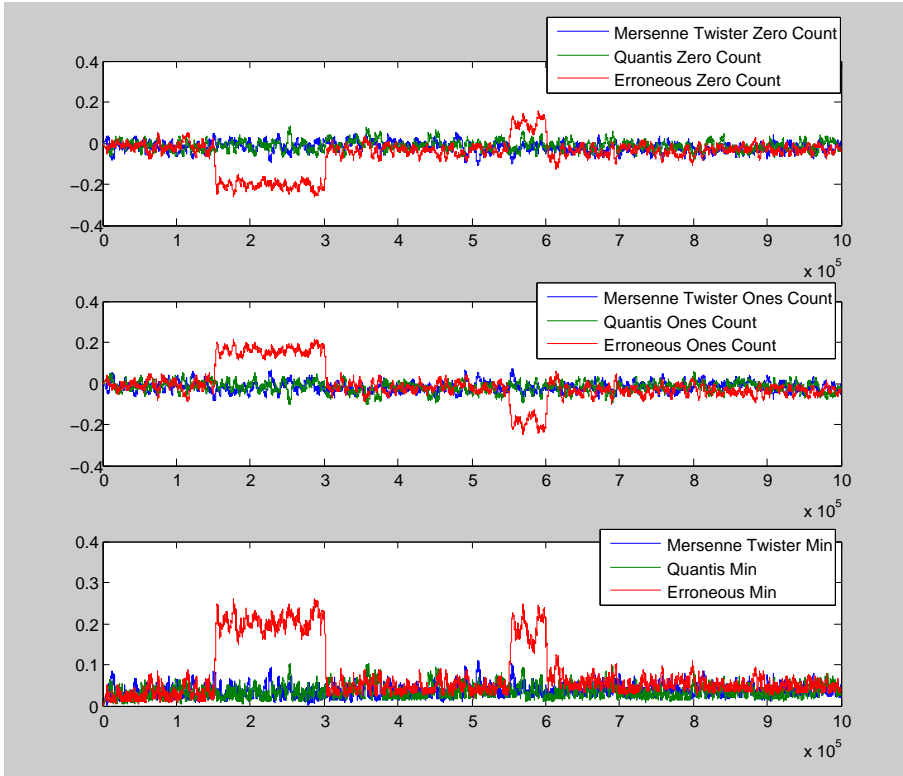


Figure 5.2: Simulation results for the Sliding Window Run-length Coding test. The top graph shows the runs of 0's, the middle graph the runs of 1's and the bottom one the minimum of the absolute value of the top two graphs. The sliding window here is of size 5000 bits.

around 0.15.

6

Conclusion and Future Work

This chapter presents our conclusion, give suggestions for future work and lastly present conclusions.

6.1 Conclusions

Here we present our conclusions on the various topics covered in this paper. The conclusion is split up into the major topics covered in the paper.

6.1.1 Random Number Testing

The idea behind statistically testing if a sequence is random is a somewhat complex topic. On one hand, we know that there is no definite test that will tell us with certainty if a finite sequence is random. Any finite sequence could be a part of an infinite random sequence. Still, we seek methods for seeing how probable it is that a finite sample is taken from an infinite random sequence, and that is at the core of random number testing.

Many characteristics of randomness can be observed, and as such many of the tests focus on distribution properties, compressibility or unpredictability. Effort is put into designing tests that draw out as much of what we consider to be random behavior from a sequence with as little effort as possible. This is then compared to what an ideal random sequence would produce. Some such tests are based on known flaws of PRNGs and a few even on known flaws of HRNGs. Example of the first are the linear dependency tests, which are the only tests we have seen that actually fail Mersenne Twister, and example of the latter is the Borel Normality test.

We have seen that certain tests are in particular interesting when analyzing HRNGs,

both through looking at others work and through direct testing. In the paper [8] the Book Stack and Borel Normality tests are applied to QRNGs and PRNGs, with very promising results. Our testing has shown that in particular tests focusing on bit-level randomness fail our tested QRNG. These include the Random Walk tests and the STS Runs test, among others.

When looking at PRNGs we have seen surprisingly few tests that actually fails them. Notably Mersenne Twister fails the Linear dependency tests, which is not surprising considering how Mersenne Twister is constructed. When presented with a notably bad PRNG, RANDU, we saw that it would actually pass the STS Runs test. This is in particular peculiar as the same test fails our QRNG, and we conclude that this test should be used with some caution.

We used the Bit Distribution test to get an idea to what level the generators were distributed. AES OFB, Mersenne Twister and Quantis all showed to be distributed to 18 bits with our set parameters, which shows that they should have similar distribution properties. The fact that all three of these pass most distribution tests further strengthen this claim. We can conclude that all three are thus overall good RNGs. The bit-level correlations found in our QRNG is thus extra peculiar.

6.1.2 Quantis

We have argued that Quantis is random in the same sense that algorithmically random sequences are random to computers. The fact that we can not explain where the unpredictability in some quantum processes come from is the very motivation for constructing QRNGs. The tested QRNG Quantis provide a good source of randomness overall, but fails most tests focusing on bit-level correlation. The possible reasons for this could be many, but since our testing holds the same for two different Quantis USB devices it would seem unlikely that the result is due to an hardware malfunction.

One explanation could be due to the chosen post-processing used in Quantis. Recall from subsection 2.3.3 that we assume $P(X = 1) = p$, but what if the probability of getting a 1 changes with time? This model assumes bias, but constant bias at that, and if the generated sequence in Quantis has some erroneous noise that changes with time this could result in bit-level correlation. Another more extreme explanation could be that our understanding of Quantum indeterminacy is flawed in some way, though an construction error seems more probable.

We can not conclude that the randomness produced by Quantis is true, since it seems to show some bit-correlation not consistent with what we consider to be random. In fact, AES OFB seem to produce "better" random sequences than Quantis based on the fact that it passes all tests we ran. We can however conclude that we have no known way of properly predicting Quantis, though given the bit-level correlation it might be interesting to look further into this.

6.1.3 Pseudo Random Number Generators

We can conclude that Mersenne Twister is an exceptional generator when it comes to simulations. It passes all randomness tests we subjected it to, with the exception of two linear dependency tests, and is very efficient. It is clear that it is not without reason it is being widely implied in various fields of research. Similarly AES OFB also provided exceptionally good results, though it is worth noting that AES OFB take a cryptographic key in addition to a seed to generate the random number, meaning that its initial entropy is much higher.

6.1.4 Test Suites

Through our testing we have covered all tests in NIST STS, Diehard and DieHarder. We have also covered many of the tests in TestU01. It is worth noting that even though we present the test results from Diehard through DieHarder, the tests implemented in DieHarder are all generalizations with higher level of accuracy compared to the original ones. Thus, passing Diehard does not guarantee passing the Diehard tests in DieHarder, but the converse holds, which is shown in [5].

Our testing shows that the tests in Diehard are not able to clearly distinguish between the QRNG and PRNGs tested. This is hardly surprising, since Diehard is known for lacking tests that focus on bit-level distribution properties, which is the main weakness in or tested QRNG. Furthermore, the tests in Diehard, as they were originally implemented, will incorrectly pass many PRNGs that are today considered to be bad. As such, we conclude that finding alternatives to Diehard should be wise.

The three major alternatives we present are NIST STS, DieHarder and TestU01. NIST STS has a collection of good tests, focusing very much on bit-level randomness, but due to recent scandals using a product of NIST might be less appealing to some. NIST is also comparatively quite small, therefore we will instead focus on the other two. DieHarder showed to produce good results, but was noticeably not able to properly catch any of the bit-level correlation that our QRNG produced. Of course, it is still a promising tool and its ease of use is a definitive strong point. In particularly running the collected "all tests" with resolve ambiguously is powerful when wanting to apply a strong battery of tests without having to actually concern oneself with using a compiler and setting up search paths, which TestU01 requires.

The test suite that stand out the most at the time of conducting this research is TestU01. It offers any user with some knowledge of basic C the ability to test almost any aspect of an RNG. It is a library that is both easy to use and furthermore provides several batteries for testing both RNGs in general but also one specifically designed for testing HRNGs. It covers most kinds of tests, though notably not the Book Stack test or the Borel Normality test, which we propose could make interesting additions to the suite. Still, it is an state of the art library with much variability, and it is the tool we recommend organizations certifying RNGs should use.

6.1.5 Live Testing

We propose the following setup for a live testing module that could easily be implemented into a HRNG.

Bit Persistence Test

This test could easily be used as an live test with almost no modification from how it was explained in 3.3.11. If a bit remains unchanged over too long of a sample size then this test should indicate that something is wrong, and potentially even filter out the not working bit if the other tests indicate that the remaining sequence is random enough.

Bit Distribution Test

As described in subsection 3.3.11, we begin by testing to what level the HRNG is bit distributed, and then apply the test over non-overlapping sequences to see if it stays bit distributed to the same number of bits.

Sliding Window Run-length Coding Test

As proposed in subsection 3.11.1, this test seems promising based on the testing done in Matlab. It should be an computationally good test with high level of accuracy over relatively small windows. It should detect any irregularities in probability distribution as well as oscillations.

Pi Convergence Test

We theorize that the proposed pi test from subsection 3.11.2 could be a good candidate, and it should indicate if the distribution over large sample periods is good.

Other tests of interest includes the Random Walk tests and the Borel Normality test. The former are not included since they showed bad results for our QRNG when testing. If this is because the QRNG is somehow flawed, adding these tests would be recommended. The Borel Normality test is not included since it would be computationally quite demanding, as it would have to be evaluated for every $1 \leq i \leq 2^k, k = 1, \dots, 5$. Of course, a lighter version of this test could be interesting. A more thorough analysis of what parameters are needed for a good result might open up for this test as well.

6.2 Future work

Given the peculiar results from the Alphabit battery, a more thorough study into the reason behind this is definitely needed. The Alphabit battery is noted to be developed for testing HRNGs, and for it to fail Quantis is interesting to say the least. One reason could of course be that the tests are flawed in some way. We leave this question for future research to answer.

One noticeable weakness when working with random number testing is the fact that the research is somewhat limited to work due to people with a background in pseudo-random number generation. As such, many tests are clearly developed

with evaluating PRNGs in mind, and often to exploit some specific characteristic of a PRNG.

An noticeable exception to this is Cristian S. Calude et al. in their work [8]. They introduce several tests that are not implemented in any of the test suites mentioned in this paper, and that clearly focus on exploiting aspects that could differentiate between a QRNGs and PRNGs. Implementation of the Book stack test or the Borel normality test into DieHarder or TestU01 would, based on the results from their work, seem to make more diverse testing.

We found no tests utilizing stochastic programming or more advanced filter theory. There is much research that focus on trying to predict future change in something that is perceived to be unpredictable based on recent patterns, for example in Finance and Control Engineering. Bridging research from those majors to random number testing would present opportunities for interesting new tests.

The new live tests presented by us, notably the Run-length Coding test, could be further analyzed and used with more generators, and potentially be constructed as a hardware chip for use with a HRNG. A theoretical distribution of the test statistic would further enhance the strength of the test. Other tests could of course also be applied to live testing with either of the methods we proposed. We also note that the Run-length test might be good for evaluating QRNGs, as we have shown that they prohibit some quite unpredictable bit-level change in oscillations, that might be reflected in the entropy of the runs.

6.3 Summary

Random number theory dictates that no finite test can prove randomness. The tests described in this work can only give a probabilistic answer to if a sequence is what we perceive to be random. We have used various tests that seek to exploit properties that we classify as not random, such as bad distribution, oscillations and predictability, and in some ways to great success. It is hard to argue that such testing is not relevant, even though it can never give an definite answer.

We have looked at randomness tests from various fields of research, and noted a few of particular interest. As QRNGs differ from PRNGs in many ways, the applicable tests also differ. Our simulations suggest that Quantis will fail certain tests due to it having bit-level oscillations that are not consistent to what we perceive to be normal for a random sequence. Though a more thorough analysis is needed to give an definitive answer to this. The test suites Diehard, DieHarder and STS had a few tests capable of detecting this, but the battery Alphabit in TestU01 stood out in that all its tests failed Quantis.

If such flaws are of interest to the user is difficult to say, but our testing seems to suggest that the sequences generated by Quantis are in some sense less random than those from Mersenne Twister or AES OFB. It would however seem like Quantis pass the tests intended to look for predictable patterns, which is important in gambling and cryptography. This does not necessarily mean that the product is

safe for use in these fields, but as far as statistical testing can determine it would not seem like Quantis is in any way unsafe.

We also present a new test, called the Run-length coding test, and present ideas for possible integrated live tests that could be used with HRNGs. Our simulations suggest that the Run-length coding test applied to live testing could be a useful tool in live testing of HRNGs, as it can clearly distinguish errors in the distribution on an 5% level with minimal computational needs. A collection of tests for live testing is also proposed.

Appendix

A

Distribution functions

The following definitions are taken from [15] and [52].

A.1 Discrete distributions

Definition A.1.1 (Bernoulli Distribution). For $0 \leq p \leq 1$,

$$P_X(x) = \begin{cases} 1-p & \text{if } x = 0 \\ p & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases} \quad \phi_X(s) = 1 - p + pe^s$$
$$E(X) = p \quad \text{Var}(X) = p(1-p)$$

Definition A.1.2 (Binomial Distribution). For $0 \leq p \leq 1$ and positive integer n ,

$$P_X(x) = \binom{n}{x} p^x (1-p)^{n-x} \quad \phi_X(s) = (1 - p + pe^s)^n$$
$$E(X) = np \quad \text{Var}(X) = np(1-p)$$

Definition A.1.3 (Multinomial Distribution). For integers $n > 0$, $p_i \geq 0$ for $i = 1, \dots, n$ and $p_1 + \dots + p_n = 1$,

$$P_{X_1, \dots, X_r}(x_1, \dots, x_r) = \binom{n}{x_1, \dots, x_r} p_1^{x_1} \dots p_r^{x_r}$$
$$E(X_i) = np_i \quad \text{Var}(X_i) = np_i(1-p_i)$$

Definition A.1.4 (Poisson Distribution). For $\alpha > 0$,

$$P_X(x) = \begin{cases} \frac{\alpha^x e^{-\alpha}}{x!} & \text{for } x = 1, 2, \dots, n \\ 0 & \text{otherwise} \end{cases} \quad \phi_X(s) = e^{\alpha(e^s - 1)}$$

$$E(X) = \alpha \quad \text{Var}(X) = \alpha$$

A.2 Continuous Distributions

Definition A.2.1 (Exponential Distribution). For $\lambda > 0$,

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad \phi_X(s) = \frac{\lambda}{\lambda + s}$$

$$E(X) = 1/\lambda \quad \text{Var}(X) = 1/\lambda^2$$

Definition A.2.2 (Normal Distribution). For $\mu \in \mathbb{R}$ and $\sigma^2 > 0$,

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \phi_X(s) = e^{it\mu - \frac{1}{2}\sigma^2 s^2}$$

$$E(X) = \mu \quad \text{Var}(X) = \sigma^2$$

Definition A.2.3 (Half-Normal Distribution). For standard normally distributed $Z \sim N(0, \sigma^2)$ we have that $X = |Z|$ is half-normally distributed. Let $x > 0$,

$$f_X(x) = \frac{\sqrt{2}}{\sigma\sqrt{\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

$$E(X) = \frac{\sigma\sqrt{2}}{\sqrt{\pi}} \quad \text{Var}(X) = \sigma^2(1 - \frac{2}{\pi})$$

Definition A.2.4 (χ^2 Distribution). For standard normally distributed $Z_i \sim N(0, \sigma^2)$, $0 \leq i < k$, we have that $X = Z_0^2 + \dots + Z_{k-1}^2$ is χ^2 distributed with k degrees of freedom. Let $k \in \mathbb{N}_+$ and $x \geq 0$,

$$f_X(x) = \frac{1}{2^{k/2}\Gamma(\frac{k}{2})} x^{k/2-1} e^{-x/2} \quad \phi_X(s) = (1 - 2is)^{-k/2}$$

$$E(X) = k \quad \text{Var}(X) = 2k$$

Definition A.2.5 (Uniform Distribution). For constants $a < b$,

$$f_X(x) = \begin{cases} \frac{1}{b-a} & \text{for } a < x < b \\ 0 & \text{otherwise} \end{cases} \quad \phi_X(s) = \frac{e^{bs} - e^{as}}{s(b-a)}$$

$$E(X) = \frac{a+b}{2} \quad \text{Var}(X) = \frac{(b-a)^2}{12}$$

B

Mathematical Definitions

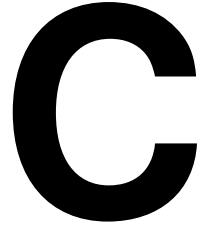
Here we gather mathematical definitions that are used in this work.

Definition B.0.6 (Stirling number of the second kind). *The Stirling number of the second kind calculates the number of ways to partition a set of k elements into exactly n parts. It is defined as*

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n.$$

Definition B.0.7. *The L_p -norm on the k -dimensional hypercube $[0, 1]^t$ defined as*

$$\|X\|_p = \begin{cases} (|x_1|^p + \dots + |x_k|^p)^{1/p} & \text{if } 1 \leq p < \infty, \\ \max(|x_1|, \dots, |x_k|) & \text{if } p = \infty. \end{cases}$$



Raw Result Data from Tests

Here is the result data as it is delivered from DieHarder and TestU01.

C.1 DieHarder Tests

DieHarder Mersenne Twister

```
#=====
# dieharder version 3.31.1 Copyright 2003 Robert G. Brown #
#=====
rng_name |rands/second| Seed |
mt19937_1998| 1.17e+08 |4071569564|
#=====
test_name |ntup| tsamples |psamples| p-value |Assessment
#=====
diehard_birthdays| 0| 100| 100|0.38975184| PASSED
diehard_operm5| 0| 1000000| 100|0.71771599| PASSED
diehard_rank_32x32| 0| 40000| 100|0.57324721| PASSED
diehard_rank_6x8| 0| 100000| 100|0.03177293| PASSED
diehard_bitstream| 0| 2097152| 100|0.58673746| PASSED
diehard_opso| 0| 2097152| 100|0.48747875| PASSED
diehard_oqso| 0| 2097152| 100|0.96500898| PASSED
diehard_dna| 0| 2097152| 100|0.07622554| PASSED
diehard_count_1s_str| 0| 256000| 100|0.38362808| PASSED
diehard_count_1s_byt| 0| 256000| 100|0.87078777| PASSED
diehard_parking_lot| 0| 12000| 100|0.77743129| PASSED
diehard_2dsphere| 2| 8000| 100|0.01622306| PASSED
diehard_3dsphere| 3| 4000| 100|0.77433012| PASSED
```

```
diehard_squeeze| 0| 100000| 100|0.99880322| WEAK
diehard_squeeze| 0| 100000| 200|0.53995686| PASSED
diehard_sums| 0| 100| 100|0.14165110| PASSED
diehard_runs| 0| 100000| 100|0.12419058| PASSED
diehard_runs| 0| 100000| 100|0.40855602| PASSED
diehard_craps| 0| 200000| 100|0.26445353| PASSED
diehard_craps| 0| 200000| 100|0.77584803| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.96012610| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.96897121| PASSED
sts_monobit| 1| 100000| 100|0.81743765| PASSED
sts_runs| 2| 100000| 100|0.49258147| PASSED
sts_serial| 1| 100000| 100|0.91375803| PASSED
sts_serial| 2| 100000| 100|0.65570495| PASSED
sts_serial| 3| 100000| 100|0.92582844| PASSED
sts_serial| 3| 100000| 100|0.13643494| PASSED
sts_serial| 4| 100000| 100|0.56718986| PASSED
sts_serial| 4| 100000| 100|0.34982109| PASSED
sts_serial| 5| 100000| 100|0.28101046| PASSED
sts_serial| 5| 100000| 100|0.63311352| PASSED
sts_serial| 6| 100000| 100|0.23933668| PASSED
sts_serial| 6| 100000| 100|0.40263167| PASSED
sts_serial| 7| 100000| 100|0.23568127| PASSED
sts_serial| 7| 100000| 100|0.17297617| PASSED
sts_serial| 8| 100000| 100|0.17135106| PASSED
sts_serial| 8| 100000| 100|0.23775455| PASSED
sts_serial| 9| 100000| 100|0.94800011| PASSED
sts_serial| 9| 100000| 100|0.44840669| PASSED
sts_serial| 10| 100000| 100|0.99442652| PASSED
sts_serial| 10| 100000| 100|0.41704433| PASSED
sts_serial| 11| 100000| 100|0.98677589| PASSED
sts_serial| 11| 100000| 100|0.44703511| PASSED
sts_serial| 12| 100000| 100|0.75648403| PASSED
sts_serial| 12| 100000| 100|0.30814612| PASSED
sts_serial| 13| 100000| 100|0.42872464| PASSED
sts_serial| 13| 100000| 100|0.41606709| PASSED
sts_serial| 14| 100000| 100|0.74658843| PASSED
sts_serial| 14| 100000| 100|0.94707481| PASSED
sts_serial| 15| 100000| 100|0.53195149| PASSED
sts_serial| 15| 100000| 100|0.99728230| WEAK
sts_serial| 16| 100000| 100|0.37208596| PASSED
sts_serial| 16| 100000| 100|0.90703113| PASSED
sts_serial| 1| 100000| 200|0.66720828| PASSED
sts_serial| 2| 100000| 200|0.83253186| PASSED
sts_serial| 3| 100000| 200|0.89973971| PASSED
sts_serial| 3| 100000| 200|0.16389899| PASSED
sts_serial| 4| 100000| 200|0.73729964| PASSED
```

sts_serial	4	100000	200 0.36470694	PASSED
sts_serial	5	100000	200 0.45669726	PASSED
sts_serial	5	100000	200 0.29481462	PASSED
sts_serial	6	100000	200 0.95408083	PASSED
sts_serial	6	100000	200 0.51634157	PASSED
sts_serial	7	100000	200 0.11962882	PASSED
sts_serial	7	100000	200 0.26755816	PASSED
sts_serial	8	100000	200 0.18226694	PASSED
sts_serial	8	100000	200 0.58030396	PASSED
sts_serial	9	100000	200 0.43306052	PASSED
sts_serial	9	100000	200 0.76959832	PASSED
sts_serial	10	100000	200 0.99567390	WEAK
sts_serial	10	100000	200 0.05266538	PASSED
sts_serial	11	100000	200 0.55810230	PASSED
sts_serial	11	100000	200 0.35125430	PASSED
sts_serial	12	100000	200 0.20909196	PASSED
sts_serial	12	100000	200 0.34636983	PASSED
sts_serial	13	100000	200 0.59570094	PASSED
sts_serial	13	100000	200 0.97396504	PASSED
sts_serial	14	100000	200 0.69866606	PASSED
sts_serial	14	100000	200 0.62741552	PASSED
sts_serial	15	100000	200 0.53260038	PASSED
sts_serial	15	100000	200 0.96989746	PASSED
sts_serial	16	100000	200 0.36444632	PASSED
sts_serial	16	100000	200 0.63732472	PASSED
sts_serial	1	100000	300 0.10629715	PASSED
sts_serial	2	100000	300 0.14638383	PASSED
sts_serial	3	100000	300 0.31418910	PASSED
sts_serial	3	100000	300 0.47969261	PASSED
sts_serial	4	100000	300 0.65331013	PASSED
sts_serial	4	100000	300 0.58822451	PASSED
sts_serial	5	100000	300 0.58669820	PASSED
sts_serial	5	100000	300 0.37353466	PASSED
sts_serial	6	100000	300 0.77168126	PASSED
sts_serial	6	100000	300 0.25703655	PASSED
sts_serial	7	100000	300 0.09286361	PASSED
sts_serial	7	100000	300 0.24869091	PASSED
sts_serial	8	100000	300 0.26430961	PASSED
sts_serial	8	100000	300 0.91808807	PASSED
sts_serial	9	100000	300 0.27537619	PASSED
sts_serial	9	100000	300 0.85833570	PASSED
sts_serial	10	100000	300 0.95430897	PASSED
sts_serial	10	100000	300 0.05317704	PASSED
sts_serial	11	100000	300 0.21120960	PASSED
sts_serial	11	100000	300 0.53337492	PASSED
sts_serial	12	100000	300 0.11865953	PASSED

```

sts_serial| 12| 100000| 300|0.39307850| PASSED
sts_serial| 13| 100000| 300|0.09880592| PASSED
sts_serial| 13| 100000| 300|0.51850907| PASSED
sts_serial| 14| 100000| 300|0.11431932| PASSED
sts_serial| 14| 100000| 300|0.64762215| PASSED
sts_serial| 15| 100000| 300|0.38394176| PASSED
sts_serial| 15| 100000| 300|0.87311750| PASSED
sts_serial| 16| 100000| 300|0.70248694| PASSED
sts_serial| 16| 100000| 300|0.84268411| PASSED
rgb_bitdist| 1| 100000| 100|0.46332690| PASSED
rgb_bitdist| 2| 100000| 100|0.92856307| PASSED
rgb_bitdist| 3| 100000| 100|0.35339242| PASSED
rgb_bitdist| 4| 100000| 100|0.44207487| PASSED
rgb_bitdist| 5| 100000| 100|0.70218284| PASSED
rgb_bitdist| 6| 100000| 100|0.12780573| PASSED
rgb_bitdist| 7| 100000| 100|0.53422465| PASSED
rgb_bitdist| 8| 100000| 100|0.98858910| PASSED
rgb_bitdist| 9| 100000| 100|0.92446099| PASSED
rgb_bitdist| 10| 100000| 100|0.26300841| PASSED
rgb_bitdist| 11| 100000| 100|0.81126139| PASSED
rgb_bitdist| 12| 100000| 100|0.79652370| PASSED
rgb_minimum_distance| 2| 10000| 1000|0.29052622| PASSED
rgb_minimum_distance| 3| 10000| 1000|0.00512162| PASSED
rgb_minimum_distance| 4| 10000| 1000|0.60195008| PASSED
rgb_minimum_distance| 5| 10000| 1000|0.43302898| PASSED
rgb_permutations| 2| 100000| 100|0.27168383| PASSED
rgb_permutations| 3| 100000| 100|0.30374294| PASSED
rgb_permutations| 4| 100000| 100|0.94995660| PASSED
rgb_permutations| 5| 100000| 100|0.12998123| PASSED
rgb_lagged_sum| 0| 1000000| 100|0.42863534| PASSED
rgb_lagged_sum| 1| 1000000| 100|0.97088952| PASSED
rgb_lagged_sum| 2| 1000000| 100|0.96599390| PASSED
rgb_lagged_sum| 3| 1000000| 100|0.31779373| PASSED
rgb_lagged_sum| 4| 1000000| 100|0.92363132| PASSED
rgb_lagged_sum| 5| 1000000| 100|0.35459940| PASSED
rgb_lagged_sum| 6| 1000000| 100|0.85147378| PASSED
rgb_lagged_sum| 7| 1000000| 100|0.88047565| PASSED
rgb_lagged_sum| 8| 1000000| 100|0.28000537| PASSED
rgb_lagged_sum| 9| 1000000| 100|0.31055528| PASSED
rgb_lagged_sum| 10| 1000000| 100|0.07083999| PASSED
rgb_lagged_sum| 11| 1000000| 100|0.23580863| PASSED
rgb_lagged_sum| 12| 1000000| 100|0.98958838| PASSED
rgb_lagged_sum| 13| 1000000| 100|0.48505384| PASSED
rgb_lagged_sum| 14| 1000000| 100|0.65446662| PASSED
rgb_lagged_sum| 15| 1000000| 100|0.88469731| PASSED
rgb_lagged_sum| 16| 1000000| 100|0.21509434| PASSED

```

```

rgb_lagged_sum| 17| 1000000| 100|0.62405478| PASSED
rgb_lagged_sum| 18| 1000000| 100|0.64959221| PASSED
rgb_lagged_sum| 19| 1000000| 100|0.05325570| PASSED
rgb_lagged_sum| 20| 1000000| 100|0.90112379| PASSED
rgb_lagged_sum| 21| 1000000| 100|0.78511300| PASSED
rgb_lagged_sum| 22| 1000000| 100|0.99801545| WEAK
rgb_lagged_sum| 22| 1000000| 200|0.44967352| PASSED
rgb_lagged_sum| 23| 1000000| 100|0.02925801| PASSED
rgb_lagged_sum| 24| 1000000| 100|0.25367023| PASSED
rgb_lagged_sum| 25| 1000000| 100|0.62563761| PASSED
rgb_lagged_sum| 26| 1000000| 100|0.27548944| PASSED
rgb_lagged_sum| 27| 1000000| 100|0.54234894| PASSED
rgb_lagged_sum| 28| 1000000| 100|0.72210168| PASSED
rgb_lagged_sum| 29| 1000000| 100|0.99411433| PASSED
rgb_lagged_sum| 30| 1000000| 100|0.89512069| PASSED
rgb_lagged_sum| 31| 1000000| 100|0.97311154| PASSED
rgb_lagged_sum| 32| 1000000| 100|0.90706748| PASSED
rgb_kstest_test| 0| 10000| 1000|0.20162443| PASSED
dab_bytedistrib| 0| 51200000| 1|0.11586907| PASSED
dab_dct| 256| 50000| 1|0.87887427| PASSED
Preparing to run test 207.  ntuple = 0
dab_filltree| 32| 15000000| 1|0.91032130| PASSED
dab_filltree| 32| 15000000| 1|0.23424036| PASSED
Preparing to run test 208.  ntuple = 0
dab_filltree2| 0| 5000000| 1|0.18601777| PASSED
dab_filltree2| 1| 5000000| 1|0.93403969| PASSED
Preparing to run test 209.  ntuple = 0
dab_monobit2| 12| 65000000| 1|0.05525865| PASSED

```

DieHarder Quantis (200 GB)

```

#=====
# dieharder version 3.31.1 Copyright 2003 Robert G. Brown #
#=====
rng_name | filename |rands/second|
file_input_raw| /data/krija267/random.bit| 1.41e+07 |
#=====
test_name |ntup| tsamples |psamples| p-value |Assessment
#=====
diehard_birthdays| 0| 100| 100|0.20024032| PASSED
diehard_operm5| 0| 1000000| 100|0.70492705| PASSED
diehard_rank_32x32| 0| 40000| 100|0.42131611| PASSED
diehard_rank_6x8| 0| 100000| 100|0.44404264| PASSED
diehard_bitstream| 0| 2097152| 100|0.51725458| PASSED
diehard_opso| 0| 2097152| 100|0.16780978| PASSED
diehard_oqso| 0| 2097152| 100|0.61168801| PASSED

```

```
diehard_dna| 0| 2097152| 100|0.48948288| PASSED
diehard_count_1s_str| 0| 256000| 100|0.77637943| PASSED
diehard_count_1s_byt| 0| 256000| 100|0.94024518| PASSED
diehard_parking_lot| 0| 12000| 100|0.43824831| PASSED
diehard_2dsphere| 2| 8000| 100|0.57179798| PASSED
diehard_3dsphere| 3| 4000| 100|0.54522602| PASSED
diehard_squeeze| 0| 100000| 100|0.20488768| PASSED
diehard_sums| 0| 100| 100|0.00028394| WEAK
diehard_sums| 0| 100| 200|0.00020527| WEAK
diehard_sums| 0| 100| 300|0.00089749| WEAK
diehard_sums| 0| 100| 400|0.00465149| WEAK
diehard_sums| 0| 100| 500|0.00067678| WEAK
diehard_sums| 0| 100| 600|0.00063877| WEAK
diehard_sums| 0| 100| 700|0.00001444| WEAK
diehard_sums| 0| 100| 800|0.00001552| WEAK
diehard_sums| 0| 100| 900|0.00000588| WEAK
diehard_sums| 0| 100| 1000|0.00000077| FAILED
diehard_runs| 0| 100000| 100|0.80082793| PASSED
diehard_runs| 0| 100000| 100|0.90324065| PASSED
diehard_craps| 0| 200000| 100|0.09459424| PASSED
diehard_craps| 0| 200000| 100|0.01405001| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.51231301| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.31176517| PASSED
sts_monobit| 1| 100000| 100|0.03448921| PASSED
sts_runs| 2| 100000| 100|0.00000613| WEAK
sts_runs| 2| 100000| 200|0.00000734| WEAK
sts_runs| 2| 100000| 300|0.00000004| FAILED
sts_serial| 1| 100000| 100|0.04594939| PASSED
sts_serial| 2| 100000| 100|0.11483011| PASSED
sts_serial| 3| 100000| 100|0.10542886| PASSED
sts_serial| 3| 100000| 100|0.66217333| PASSED
sts_serial| 4| 100000| 100|0.50302288| PASSED
sts_serial| 4| 100000| 100|0.47688934| PASSED
sts_serial| 5| 100000| 100|0.56532845| PASSED
sts_serial| 5| 100000| 100|0.54835471| PASSED
sts_serial| 6| 100000| 100|0.82232008| PASSED
sts_serial| 6| 100000| 100|0.73718791| PASSED
sts_serial| 7| 100000| 100|0.81751666| PASSED
sts_serial| 7| 100000| 100|0.63337945| PASSED
sts_serial| 8| 100000| 100|0.75610904| PASSED
sts_serial| 8| 100000| 100|0.99987772| WEAK
sts_serial| 9| 100000| 100|0.69597119| PASSED
sts_serial| 9| 100000| 100|0.88045515| PASSED
sts_serial| 10| 100000| 100|0.85463952| PASSED
sts_serial| 10| 100000| 100|0.92871816| PASSED
sts_serial| 11| 100000| 100|0.30010585| PASSED
```

sts_serial		11		100000		100		0.24624141		PASSED
sts_serial		12		100000		100		0.12061449		PASSED
sts_serial		12		100000		100		0.62071638		PASSED
sts_serial		13		100000		100		0.58440622		PASSED
sts_serial		13		100000		100		0.85009757		PASSED
sts_serial		14		100000		100		0.82922436		PASSED
sts_serial		14		100000		100		0.85734470		PASSED
sts_serial		15		100000		100		0.34502288		PASSED
sts_serial		15		100000		100		0.50006208		PASSED
sts_serial		16		100000		100		0.83435688		PASSED
sts_serial		16		100000		100		0.74665811		PASSED
sts_serial		1		100000		200		0.06369777		PASSED
sts_serial		2		100000		200		0.21898036		PASSED
sts_serial		3		100000		200		0.25980861		PASSED
sts_serial		3		100000		200		0.78806130		PASSED
sts_serial		4		100000		200		0.06194520		PASSED
sts_serial		4		100000		200		0.99901485		WEAK
sts_serial		5		100000		200		0.02941060		PASSED
sts_serial		5		100000		200		0.17222191		PASSED
sts_serial		6		100000		200		0.16154263		PASSED
sts_serial		6		100000		200		0.96886443		PASSED
sts_serial		7		100000		200		0.07953438		PASSED
sts_serial		7		100000		200		0.18945613		PASSED
sts_serial		8		100000		200		0.79522360		PASSED
sts_serial		8		100000		200		0.91586923		PASSED
sts_serial		9		100000		200		0.34305976		PASSED
sts_serial		9		100000		200		0.86527338		PASSED
sts_serial		10		100000		200		0.19056174		PASSED
sts_serial		10		100000		200		0.85552762		PASSED
sts_serial		11		100000		200		0.91495040		PASSED
sts_serial		11		100000		200		0.48778086		PASSED
sts_serial		12		100000		200		0.62808973		PASSED
sts_serial		12		100000		200		0.99113605		PASSED
sts_serial		13		100000		200		0.75271482		PASSED
sts_serial		13		100000		200		0.90505360		PASSED
sts_serial		14		100000		200		0.56354094		PASSED
sts_serial		14		100000		200		0.74704638		PASSED
sts_serial		15		100000		200		0.58563786		PASSED
sts_serial		15		100000		200		0.42641544		PASSED
sts_serial		16		100000		200		0.61177937		PASSED
sts_serial		16		100000		200		0.58333748		PASSED
sts_serial		1		100000		300		0.03052267		PASSED
sts_serial		2		100000		300		0.18457528		PASSED
sts_serial		3		100000		300		0.22003015		PASSED
sts_serial		3		100000		300		0.27214553		PASSED
sts_serial		4		100000		300		0.05811784		PASSED

```

sts_serial| 4| 100000| 300|0.32581955| PASSED
sts_serial| 5| 100000| 300|0.01856640| PASSED
sts_serial| 5| 100000| 300|0.42922845| PASSED
sts_serial| 6| 100000| 300|0.61440683| PASSED
sts_serial| 6| 100000| 300|0.19583723| PASSED
sts_serial| 7| 100000| 300|0.23672164| PASSED
sts_serial| 7| 100000| 300|0.08322065| PASSED
sts_serial| 8| 100000| 300|0.43354135| PASSED
sts_serial| 8| 100000| 300|0.98709772| PASSED
sts_serial| 9| 100000| 300|0.14699070| PASSED
sts_serial| 9| 100000| 300|0.82664585| PASSED
sts_serial| 10| 100000| 300|0.02069139| PASSED
sts_serial| 10| 100000| 300|0.43060926| PASSED
sts_serial| 11| 100000| 300|0.65940026| PASSED
sts_serial| 11| 100000| 300|0.46161079| PASSED
sts_serial| 12| 100000| 300|0.47745405| PASSED
sts_serial| 12| 100000| 300|0.61917577| PASSED
sts_serial| 13| 100000| 300|0.88767254| PASSED
sts_serial| 13| 100000| 300|0.96945254| PASSED
sts_serial| 14| 100000| 300|0.35485617| PASSED
sts_serial| 14| 100000| 300|0.75075465| PASSED
sts_serial| 15| 100000| 300|0.88318116| PASSED
sts_serial| 15| 100000| 300|0.86809046| PASSED
sts_serial| 16| 100000| 300|0.95923455| PASSED
sts_serial| 16| 100000| 300|0.88122832| PASSED
rgb_bitdist| 1| 100000| 100|0.78508802| PASSED
rgb_bitdist| 2| 100000| 100|0.84991462| PASSED
rgb_bitdist| 3| 100000| 100|0.17825539| PASSED
rgb_bitdist| 4| 100000| 100|0.98292062| PASSED
rgb_bitdist| 5| 100000| 100|0.27508784| PASSED
rgb_bitdist| 6| 100000| 100|0.31137249| PASSED
rgb_bitdist| 7| 100000| 100|0.74534734| PASSED
rgb_bitdist| 8| 100000| 100|0.30765689| PASSED
rgb_bitdist| 9| 100000| 100|0.94651313| PASSED
rgb_bitdist| 10| 100000| 100|0.62260424| PASSED
rgb_bitdist| 11| 100000| 100|0.46454454| PASSED
rgb_bitdist| 12| 100000| 100|0.50470065| PASSED
rgb_minimum_distance| 2| 10000| 1000|0.97272554| PASSED
rgb_minimum_distance| 3| 10000| 1000|0.11407453| PASSED
rgb_minimum_distance| 4| 10000| 1000|0.24258662| PASSED
rgb_minimum_distance| 5| 10000| 1000|0.73726727| PASSED
rgb_permutations| 2| 100000| 100|0.20409831| PASSED
rgb_permutations| 3| 100000| 100|0.75663365| PASSED
rgb_permutations| 4| 100000| 100|0.10102010| PASSED
rgb_permutations| 5| 100000| 100|0.22390625| PASSED
rgb_lagged_sum| 0| 1000000| 100|0.98873436| PASSED

```

rgb_lagged_sum	1	1000000	100	0.06527286	PASSED
rgb_lagged_sum	2	1000000	100	0.36225546	PASSED
rgb_lagged_sum	3	1000000	100	0.07540369	PASSED
rgb_lagged_sum	4	1000000	100	0.98991541	PASSED
rgb_lagged_sum	5	1000000	100	0.30245706	PASSED
rgb_lagged_sum	6	1000000	100	0.38047884	PASSED
rgb_lagged_sum	7	1000000	100	0.59789063	PASSED
rgb_lagged_sum	8	1000000	100	0.99738785	WEAK
rgb_lagged_sum	8	1000000	200	0.45781632	PASSED
rgb_lagged_sum	9	1000000	100	0.01353834	PASSED
rgb_lagged_sum	10	1000000	100	0.05572060	PASSED
rgb_lagged_sum	11	1000000	100	0.50267590	PASSED
rgb_lagged_sum	12	1000000	100	0.93050517	PASSED
rgb_lagged_sum	13	1000000	100	0.46886121	PASSED
rgb_lagged_sum	14	1000000	100	0.33222280	PASSED
rgb_lagged_sum	15	1000000	100	0.13175410	PASSED
rgb_lagged_sum	16	1000000	100	0.00458108	WEAK
rgb_lagged_sum	16	1000000	200	0.00004082	WEAK
rgb_lagged_sum	16	1000000	300	0.00000112	WEAK
rgb_lagged_sum	16	1000000	400	0.00009106	WEAK
rgb_lagged_sum	16	1000000	500	0.00002368	WEAK
rgb_lagged_sum	16	1000000	600	0.00007886	WEAK
rgb_lagged_sum	16	1000000	700	0.00001614	WEAK
rgb_lagged_sum	16	1000000	800	0.00001548	WEAK
rgb_lagged_sum	16	1000000	900	0.00000133	WEAK
rgb_lagged_sum	16	1000000	1000	0.00000004	FAILED
rgb_lagged_sum	17	1000000	100	0.00761064	PASSED
rgb_lagged_sum	18	1000000	100	0.09099857	PASSED
rgb_lagged_sum	19	1000000	100	0.05079528	PASSED
rgb_lagged_sum	20	1000000	100	0.18616133	PASSED
rgb_lagged_sum	21	1000000	100	0.02094260	PASSED
rgb_lagged_sum	22	1000000	100	0.07392761	PASSED
rgb_lagged_sum	23	1000000	100	0.10000068	PASSED
rgb_lagged_sum	24	1000000	100	0.49674737	PASSED
rgb_lagged_sum	25	1000000	100	0.27245130	PASSED
rgb_lagged_sum	26	1000000	100	0.15493895	PASSED
rgb_lagged_sum	27	1000000	100	0.00014746	WEAK
rgb_lagged_sum	27	1000000	200	0.00013763	WEAK
rgb_lagged_sum	27	1000000	300	0.00007666	WEAK
rgb_lagged_sum	27	1000000	400	0.00006285	WEAK
rgb_lagged_sum	27	1000000	500	0.00001700	WEAK
rgb_lagged_sum	27	1000000	600	0.00004772	WEAK
rgb_lagged_sum	27	1000000	700	0.00009628	WEAK
rgb_lagged_sum	27	1000000	800	0.00000646	WEAK
rgb_lagged_sum	27	1000000	900	0.00002806	WEAK
rgb_lagged_sum	27	1000000	1000	0.00000092	FAILED

```

rgb_lagged_sum| 28| 1000000| 100|0.02179577| PASSED
rgb_lagged_sum| 29| 1000000| 100|0.13090360| PASSED
rgb_lagged_sum| 30| 1000000| 100|0.00534522| PASSED
rgb_lagged_sum| 31| 1000000| 100|0.70822885| PASSED
rgb_lagged_sum| 32| 1000000| 100|0.18014285| PASSED
rgb_kstest_test| 0| 10000| 1000|0.53673096| PASSED
dab_bytedistrib| 0| 51200000| 1|0.05445111| PASSED
dab_dct| 256| 50000| 1|0.81814110| PASSED
Preparing to run test 207.  ntuple = 0
dab_filltree| 32| 15000000| 1|0.62001003| PASSED
dab_filltree| 32| 15000000| 1|0.01181793| PASSED
Preparing to run test 208.  ntuple = 0
dab_filltree2| 0| 5000000| 1|0.42173767| PASSED
dab_filltree2| 1| 5000000| 1|0.05568427| PASSED
Preparing to run test 209.  ntuple = 0
dab_monobit2| 12| 65000000| 1|0.91761533| PASSED

```

DieHarder RANDU

```

#=====
# dieharder version 3.31.1 Copyright 2003 Robert G. Brown #
#=====
rng_name |rands/second| Seed |
randu| 2.95e+08 |2886143523|
#=====
test_name |ntup| tsamples |psamples| p-value |Assessment
#=====
diehard_birthdays| 0| 100| 100|0.17991653| PASSED
diehard_operm5| 0| 1000000| 100|0.00000000| FAILED
diehard_rank_32x32| 0| 40000| 100|0.66305699| PASSED
diehard_rank_6x8| 0| 100000| 100|0.00000000| FAILED
diehard_bitstream| 0| 2097152| 100|0.00000000| FAILED
diehard_opso| 0| 2097152| 100|0.00000000| FAILED
diehard_oqso| 0| 2097152| 100|0.00000000| FAILED
diehard_dna| 0| 2097152| 100|0.00000000| FAILED
diehard_count_1s_str| 0| 256000| 100|0.00000000| FAILED
diehard_count_1s_byt| 0| 256000| 100|0.00000000| FAILED
diehard_parking_lot| 0| 12000| 100|0.71266869| PASSED
diehard_2dsphere| 2| 8000| 100|0.00000001| FAILED
diehard_3dsphere| 3| 4000| 100|0.00000000| FAILED
diehard_squeeze| 0| 100000| 100|0.00000041| FAILED
diehard_sums| 0| 100| 100|0.01078321| PASSED
diehard_runs| 0| 100000| 100|0.00000456| WEAK
diehard_runs| 0| 100000| 100|0.00896383| PASSED
diehard_runs| 0| 100000| 200|0.00001279| WEAK
diehard_runs| 0| 100000| 200|0.02540677| PASSED

```

```
diehard_runs| 0| 100000| 300|0.00000270| WEAK
diehard_runs| 0| 100000| 300|0.00179226| WEAK
diehard_runs| 0| 100000| 400|0.00000024| FAILED
diehard_runs| 0| 100000| 400|0.00045372| WEAK
diehard_craps| 0| 200000| 100|0.00000000| FAILED
diehard_craps| 0| 200000| 100|0.00000000| FAILED
marsaglia_tsang_gcd| 0| 1000000| 100|0.00000000| FAILED
marsaglia_tsang_gcd| 0| 1000000| 100|0.00000000| FAILED
sts_monobit| 1| 100000| 100|0.01661228| PASSED
sts_runs| 2| 100000| 100|0.03206736| PASSED
sts_serial| 1| 100000| 100|0.13303758| PASSED
sts_serial| 2| 100000| 100|0.00000001| FAILED
sts_serial| 3| 100000| 100|0.00000000| FAILED
sts_serial| 3| 100000| 100|0.00000000| FAILED
sts_serial| 4| 100000| 100|0.00000000| FAILED
sts_serial| 4| 100000| 100|0.00000000| FAILED
sts_serial| 5| 100000| 100|0.00000000| FAILED
sts_serial| 5| 100000| 100|0.00000000| FAILED
sts_serial| 6| 100000| 100|0.00000000| FAILED
sts_serial| 6| 100000| 100|0.00000000| FAILED
sts_serial| 7| 100000| 100|0.00000000| FAILED
sts_serial| 7| 100000| 100|0.00000000| FAILED
sts_serial| 8| 100000| 100|0.00000000| FAILED
sts_serial| 8| 100000| 100|0.00000000| FAILED
sts_serial| 9| 100000| 100|0.00000000| FAILED
sts_serial| 9| 100000| 100|0.00000000| FAILED
sts_serial| 10| 100000| 100|0.00000000| FAILED
sts_serial| 10| 100000| 100|0.00000000| FAILED
sts_serial| 11| 100000| 100|0.00000000| FAILED
sts_serial| 11| 100000| 100|0.00000000| FAILED
sts_serial| 12| 100000| 100|0.00000000| FAILED
sts_serial| 12| 100000| 100|0.00000000| FAILED
sts_serial| 13| 100000| 100|0.00000000| FAILED
sts_serial| 13| 100000| 100|0.00000000| FAILED
sts_serial| 14| 100000| 100|0.00000000| FAILED
sts_serial| 14| 100000| 100|0.00000000| FAILED
sts_serial| 15| 100000| 100|0.00000000| FAILED
sts_serial| 15| 100000| 100|0.00000000| FAILED
sts_serial| 16| 100000| 100|0.00000000| FAILED
sts_serial| 16| 100000| 100|0.00000000| FAILED
rgb_bitdist| 1| 100000| 100|0.00000000| FAILED
rgb_bitdist| 2| 100000| 100|0.00000000| FAILED
rgb_bitdist| 3| 100000| 100|0.00000000| FAILED
rgb_bitdist| 4| 100000| 100|0.00000000| FAILED
rgb_bitdist| 5| 100000| 100|0.00000000| FAILED
rgb_bitdist| 6| 100000| 100|0.00000000| FAILED
```

```

rgb_bitdist| 7| 100000| 100|0.00000000| FAILED
rgb_bitdist| 8| 100000| 100|0.00000000| FAILED
rgb_bitdist| 9| 100000| 100|0.00000000| FAILED
rgb_bitdist| 10| 100000| 100|0.00000000| FAILED
rgb_bitdist| 11| 100000| 100|0.00000000| FAILED
rgb_bitdist| 12| 100000| 100|0.00000000| FAILED
rgb_minimum_distance| 2| 10000| 1000|0.00000000| FAILED
rgb_minimum_distance| 3| 10000| 1000|0.00000000| FAILED
rgb_minimum_distance| 4| 10000| 1000|0.00000000| FAILED
rgb_minimum_distance| 5| 10000| 1000|0.00000000| FAILED
rgb_permutations| 2| 100000| 100|0.47988056| PASSED
rgb_permutations| 3| 100000| 100|0.06162656| PASSED
rgb_permutations| 4| 100000| 100|0.62660480| PASSED
rgb_permutations| 5| 100000| 100|0.00008135| WEAK
rgb_permutations| 5| 100000| 200|0.00000001| FAILED
rgb_lagged_sum| 0| 1000000| 100|0.02839992| PASSED
rgb_lagged_sum| 1| 1000000| 100|0.82939418| PASSED
rgb_lagged_sum| 2| 1000000| 100|0.73086635| PASSED
rgb_lagged_sum| 3| 1000000| 100|0.99734929| WEAK
rgb_lagged_sum| 3| 1000000| 200|0.78046753| PASSED
rgb_lagged_sum| 4| 1000000| 100|0.42639215| PASSED
rgb_lagged_sum| 5| 1000000| 100|0.54139725| PASSED
rgb_lagged_sum| 6| 1000000| 100|0.63968798| PASSED
rgb_lagged_sum| 7| 1000000| 100|0.87842088| PASSED
rgb_lagged_sum| 8| 1000000| 100|0.74531750| PASSED
rgb_lagged_sum| 9| 1000000| 100|0.48658924| PASSED
rgb_lagged_sum| 10| 1000000| 100|0.98161242| PASSED
rgb_lagged_sum| 11| 1000000| 100|0.99680711| WEAK
rgb_lagged_sum| 11| 1000000| 200|0.94765664| PASSED
rgb_lagged_sum| 12| 1000000| 100|0.12096407| PASSED
rgb_lagged_sum| 13| 1000000| 100|0.13011395| PASSED
rgb_lagged_sum| 14| 1000000| 100|0.90319052| PASSED
rgb_lagged_sum| 15| 1000000| 100|0.06082912| PASSED
rgb_lagged_sum| 16| 1000000| 100|0.88116576| PASSED
rgb_lagged_sum| 17| 1000000| 100|0.89149104| PASSED
rgb_lagged_sum| 18| 1000000| 100|0.84293646| PASSED
rgb_lagged_sum| 19| 1000000| 100|0.94692574| PASSED
rgb_lagged_sum| 20| 1000000| 100|0.59427907| PASSED
rgb_lagged_sum| 21| 1000000| 100|0.14460664| PASSED
rgb_lagged_sum| 22| 1000000| 100|0.96985102| PASSED
rgb_lagged_sum| 23| 1000000| 100|0.74817348| PASSED
rgb_lagged_sum| 24| 1000000| 100|0.85229164| PASSED
rgb_lagged_sum| 25| 1000000| 100|0.99617848| WEAK
rgb_lagged_sum| 25| 1000000| 200|0.98338496| PASSED
rgb_lagged_sum| 26| 1000000| 100|0.52637781| PASSED
rgb_lagged_sum| 27| 1000000| 100|0.87319805| PASSED

```

```

rgb_lagged_sum| 28| 1000000| 100|0.95604421| PASSED
rgb_lagged_sum| 29| 1000000| 100|0.59173426| PASSED
rgb_lagged_sum| 30| 1000000| 100|0.61980408| PASSED
rgb_lagged_sum| 31| 1000000| 100|0.06105720| PASSED
rgb_lagged_sum| 32| 1000000| 100|0.42277545| PASSED
rgb_kstest_test| 0| 10000| 1000|0.25969202| PASSED
dab_bytedistrib| 0| 51200000| 1|0.00000000| FAILED
dab_dct| 256| 50000| 1|0.00000000| FAILED
Preparing to run test 207.  ntuple = 0
dab_filltree| 32| 15000000| 1|0.00000000| FAILED
dab_filltree| 32| 15000000| 1|0.00000000| FAILED
Preparing to run test 208.  ntuple = 0
dab_filltree2| 0| 5000000| 1|0.00000000| FAILED
dab_filltree2| 1| 5000000| 1|0.00000000| FAILED
Preparing to run test 209.  ntuple = 0
dab_monobit2| 12| 65000000| 1|1.00000000| FAILED

```

DieHarder AES OFB

```

#=====
# dieharder version 3.31.1 Copyright 2003 Robert G. Brown #
#=====
rng_name |rands/second| Seed |
AES_OFB| 3.73e+07 |1527154789|
#=====
test_name |ntup| tsamples |psamples| p-value |Assessment
#=====
diehard_birthdays| 0| 100| 100|0.42658989| PASSED
diehard_operm5| 0| 1000000| 100|0.24651155| PASSED
diehard_rank_32x32| 0| 40000| 100|0.99705369| WEAK
diehard_rank_32x32| 0| 40000| 200|0.78572011| PASSED
diehard_rank_6x8| 0| 100000| 100|0.96282430| PASSED
diehard_bitstream| 0| 2097152| 100|0.93073810| PASSED
diehard_opso| 0| 2097152| 100|0.08969315| PASSED
diehard_oqso| 0| 2097152| 100|0.13813094| PASSED
diehard_dna| 0| 2097152| 100|0.47558623| PASSED
diehard_count_1s_str| 0| 256000| 100|0.52988043| PASSED
diehard_count_1s_byt| 0| 256000| 100|0.21833348| PASSED
diehard_parking_lot| 0| 12000| 100|0.85149486| PASSED
diehard_2dsphere| 2| 8000| 100|0.11854867| PASSED
diehard_3dsphere| 3| 4000| 100|0.19826442| PASSED
diehard_squeeze| 0| 100000| 100|0.97308988| PASSED
diehard_sums| 0| 100| 100|0.03064500| PASSED
diehard_runs| 0| 100000| 100|0.30912708| PASSED
diehard_runs| 0| 100000| 100|0.25953987| PASSED
diehard_craps| 0| 200000| 100|0.89692425| PASSED

```

```
diehard_craps| 0| 200000| 100|0.40391990| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.43779081| PASSED
marsaglia_tsang_gcd| 0| 1000000| 100|0.09932021| PASSED
sts_monobit| 1| 100000| 100|0.38449936| PASSED
sts_runs| 2| 100000| 100|0.69714040| PASSED
sts_serial| 1| 100000| 100|0.00937747| PASSED
sts_serial| 2| 100000| 100|0.15560178| PASSED
sts_serial| 3| 100000| 100|0.36103148| PASSED
sts_serial| 3| 100000| 100|0.50682305| PASSED
sts_serial| 4| 100000| 100|0.53850648| PASSED
sts_serial| 4| 100000| 100|0.69184177| PASSED
sts_serial| 5| 100000| 100|0.50853239| PASSED
sts_serial| 5| 100000| 100|0.19963515| PASSED
sts_serial| 6| 100000| 100|0.93495912| PASSED
sts_serial| 6| 100000| 100|0.87010246| PASSED
sts_serial| 7| 100000| 100|0.95618567| PASSED
sts_serial| 7| 100000| 100|0.54526112| PASSED
sts_serial| 8| 100000| 100|0.96492056| PASSED
sts_serial| 8| 100000| 100|0.90509386| PASSED
sts_serial| 9| 100000| 100|0.28130158| PASSED
sts_serial| 9| 100000| 100|0.10007987| PASSED
sts_serial| 10| 100000| 100|0.92733646| PASSED
sts_serial| 10| 100000| 100|0.99858080| WEAK
sts_serial| 11| 100000| 100|0.76989544| PASSED
sts_serial| 11| 100000| 100|0.98609718| PASSED
sts_serial| 12| 100000| 100|0.77165061| PASSED
sts_serial| 12| 100000| 100|0.10160289| PASSED
sts_serial| 13| 100000| 100|0.28852572| PASSED
sts_serial| 13| 100000| 100|0.51878297| PASSED
sts_serial| 14| 100000| 100|0.55934048| PASSED
sts_serial| 14| 100000| 100|0.56673726| PASSED
sts_serial| 15| 100000| 100|0.85460615| PASSED
sts_serial| 15| 100000| 100|0.72157355| PASSED
sts_serial| 16| 100000| 100|0.86718598| PASSED
sts_serial| 16| 100000| 100|0.92782145| PASSED
sts_serial| 1| 100000| 200|0.39774150| PASSED
sts_serial| 2| 100000| 200|0.69476460| PASSED
sts_serial| 3| 100000| 200|0.12769416| PASSED
sts_serial| 3| 100000| 200|0.00591826| PASSED
sts_serial| 4| 100000| 200|0.18802220| PASSED
sts_serial| 4| 100000| 200|0.99115314| PASSED
sts_serial| 5| 100000| 200|0.72395411| PASSED
sts_serial| 5| 100000| 200|0.03173497| PASSED
sts_serial| 6| 100000| 200|0.91897933| PASSED
sts_serial| 6| 100000| 200|0.86891270| PASSED
sts_serial| 7| 100000| 200|0.68233579| PASSED
```



```
sts_serial| 7| 100000| 200|0.34468092| PASSED
sts_serial| 8| 100000| 200|0.14104668| PASSED
sts_serial| 8| 100000| 200|0.09283330| PASSED
sts_serial| 9| 100000| 200|0.15757341| PASSED
sts_serial| 9| 100000| 200|0.03477417| PASSED
sts_serial| 10| 100000| 200|0.74404993| PASSED
sts_serial| 10| 100000| 200|0.70858041| PASSED
sts_serial| 11| 100000| 200|0.64961345| PASSED
sts_serial| 11| 100000| 200|0.37182854| PASSED
sts_serial| 12| 100000| 200|0.98203247| PASSED
sts_serial| 12| 100000| 200|0.38205090| PASSED
sts_serial| 13| 100000| 200|0.79564591| PASSED
sts_serial| 13| 100000| 200|0.78922002| PASSED
sts_serial| 14| 100000| 200|0.96163996| PASSED
sts_serial| 14| 100000| 200|0.69371246| PASSED
sts_serial| 15| 100000| 200|0.64214440| PASSED
sts_serial| 15| 100000| 200|0.35931406| PASSED
sts_serial| 16| 100000| 200|0.94947403| PASSED
sts_serial| 16| 100000| 200|0.92512003| PASSED
rgb_bitdist| 1| 100000| 100|0.57788401| PASSED
rgb_bitdist| 2| 100000| 100|0.92838955| PASSED
rgb_bitdist| 3| 100000| 100|0.58164659| PASSED
rgb_bitdist| 4| 100000| 100|0.69635417| PASSED
rgb_bitdist| 5| 100000| 100|0.50870586| PASSED
rgb_bitdist| 6| 100000| 100|0.85406661| PASSED
rgb_bitdist| 7| 100000| 100|0.90063872| PASSED
rgb_bitdist| 8| 100000| 100|0.99996645| WEAK
rgb_bitdist| 8| 100000| 200|0.98690598| PASSED
rgb_bitdist| 9| 100000| 100|0.85143992| PASSED
rgb_bitdist| 10| 100000| 100|0.54014887| PASSED
rgb_bitdist| 11| 100000| 100|0.22265853| PASSED
rgb_bitdist| 12| 100000| 100|0.81043583| PASSED
rgb_minimum_distance| 2| 10000| 1000|0.63957157| PASSED
rgb_minimum_distance| 3| 10000| 1000|0.06924414| PASSED
rgb_minimum_distance| 4| 10000| 1000|0.36309567| PASSED
rgb_minimum_distance| 5| 10000| 1000|0.11595436| PASSED
rgb_permutations| 2| 100000| 100|0.85348523| PASSED
rgb_permutations| 3| 100000| 100|0.88829521| PASSED
rgb_permutations| 4| 100000| 100|0.36040004| PASSED
rgb_permutations| 5| 100000| 100|0.92455318| PASSED
rgb_lagged_sum| 0| 1000000| 100|0.98177938| PASSED
rgb_lagged_sum| 1| 1000000| 100|0.94285568| PASSED
rgb_lagged_sum| 2| 1000000| 100|0.29555324| PASSED
rgb_lagged_sum| 3| 1000000| 100|0.32192227| PASSED
rgb_lagged_sum| 4| 1000000| 100|0.71824458| PASSED
rgb_lagged_sum| 5| 1000000| 100|0.21154122| PASSED
```

```

rgb_lagged_sum| 6| 1000000| 100|0.31157654| PASSED
rgb_lagged_sum| 7| 1000000| 100|0.79860683| PASSED
rgb_lagged_sum| 8| 1000000| 100|0.38278532| PASSED
rgb_lagged_sum| 9| 1000000| 100|0.20434465| PASSED
rgb_lagged_sum| 10| 1000000| 100|0.27960600| PASSED
rgb_lagged_sum| 11| 1000000| 100|0.15483988| PASSED
rgb_lagged_sum| 12| 1000000| 100|0.97181558| PASSED
rgb_lagged_sum| 13| 1000000| 100|0.31592281| PASSED
rgb_lagged_sum| 14| 1000000| 100|0.70609570| PASSED
rgb_lagged_sum| 15| 1000000| 100|0.59483142| PASSED
rgb_lagged_sum| 16| 1000000| 100|0.99982595| WEAK
rgb_lagged_sum| 16| 1000000| 200|0.72223945| PASSED
rgb_lagged_sum| 17| 1000000| 100|0.61870923| PASSED
rgb_lagged_sum| 18| 1000000| 100|0.33600738| PASSED
rgb_lagged_sum| 19| 1000000| 100|0.93479185| PASSED
rgb_lagged_sum| 20| 1000000| 100|0.78873025| PASSED
rgb_lagged_sum| 21| 1000000| 100|0.50582964| PASSED
rgb_lagged_sum| 22| 1000000| 100|0.76019119| PASSED
rgb_lagged_sum| 23| 1000000| 100|0.25193820| PASSED
rgb_lagged_sum| 24| 1000000| 100|0.82926627| PASSED
rgb_lagged_sum| 25| 1000000| 100|0.75565478| PASSED
rgb_lagged_sum| 26| 1000000| 100|0.92588754| PASSED
rgb_lagged_sum| 27| 1000000| 100|0.79898110| PASSED
rgb_lagged_sum| 28| 1000000| 100|0.98221038| PASSED
rgb_lagged_sum| 29| 1000000| 100|0.22694518| PASSED
rgb_lagged_sum| 30| 1000000| 100|0.55949256| PASSED
rgb_lagged_sum| 31| 1000000| 100|0.47722235| PASSED
rgb_lagged_sum| 32| 1000000| 100|0.55772316| PASSED
rgb_kstest_test| 0| 10000| 1000|0.87235485| PASSED
dab_bytedistrib| 0| 51200000| 1|0.85820332| PASSED
dab_dct| 256| 50000| 1|0.92079963| PASSED
Preparing to run test 207.  ntuple = 0
dab_filltree| 32| 15000000| 1|0.62354304| PASSED
dab_filltree| 32| 15000000| 1|0.30172198| PASSED
Preparing to run test 208.  ntuple = 0
dab_filltree2| 0| 5000000| 1|0.71386915| PASSED
dab_filltree2| 1| 5000000| 1|0.54783379| PASSED
Preparing to run test 209.  ntuple = 0
dab_monobit2| 12| 65000000| 1|0.47056609| PASSED

```

C.2 Crush Tests

Crush Battery on RANDU

===== Summary results of Crush =====

Version: TestU01 1.2.3
Generator: ulcg_CreateLCG
Number of statistics: 142
Total CPU time: 00:30:35.32
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value $< 1.0e-300$):
(eps1 means a value $< 1.0e-15$):

Test p-value

1 SerialOver, t = 2 eps
2 SerialOver, t = 4 eps
3 CollisionOver, t = 2 1 - eps1
4 CollisionOver, t = 2 eps
5 CollisionOver, t = 4 eps
6 CollisionOver, t = 4 eps
7 CollisionOver, t = 8 1 - eps1
8 CollisionOver, t = 8 eps
9 CollisionOver, t = 20 1 - eps1
10 CollisionOver, t = 20 eps
11 BirthdaySpacings, t = 2 eps
12 BirthdaySpacings, t = 3 eps
13 BirthdaySpacings, t = 4 eps
14 BirthdaySpacings, t = 7 eps
15 BirthdaySpacings, t = 7 eps
16 BirthdaySpacings, t = 8 eps
17 BirthdaySpacings, t = 8 eps
18 ClosePairs NP, t = 2 $3.2e-157$
18 ClosePairs mNP, t = 2 $3.2e-157$
18 ClosePairs mNP1, t = 2 eps
18 ClosePairs mNP2, t = 2 eps
18 ClosePairs NJumps, t = 2 1 - eps1
19 ClosePairs NP, t = 3 $3.2e-157$
19 ClosePairs mNP, t = 3 $3.2e-157$
19 ClosePairs mNP1, t = 3 eps
19 ClosePairs NJumps, t = 3 1 - eps1
20 ClosePairs NP, t = 7 $5.8e-11$
20 ClosePairs mNP, t = 7 $1.8e-79$
20 ClosePairs mNP1, t = 7 eps
20 ClosePairs mNP2, t = 7 eps
20 ClosePairs NJumps, t = 7 eps
20 ClosePairs mNP2S, t = 7 eps
21 ClosePairsBitMatch, t = 2 1 - eps1
23 SimpPoker, d = 16 eps
24 SimpPoker, d = 16 eps

25 SimpPoker, d = 64 eps
26 SimpPoker, d = 64 eps
27 CouponCollector, d = 4 eps
28 CouponCollector, d = 4 eps
29 CouponCollector, d = 16 eps
30 CouponCollector, d = 16 eps
31 Gap, r = 0 eps
32 Gap, r = 27 eps
33 Gap, r = 0 eps
34 Gap, r = 22 eps
35 Run of U01, r = 0 eps
36 Run of U01, r = 15 eps
37 Permutation, r = 0 eps
38 Permutation, r = 15 eps
39 CollisionPermut, r = 0 3.8e-211
40 CollisionPermut, r = 15 eps
41 MaxOft, t = 5 eps
41 MaxOft AD, t = 5 6.2e-43
42 MaxOft, t = 10 eps
42 MaxOft AD, t = 10 1.9e-41
43 MaxOft, t = 20 eps
43 MaxOft AD, t = 20 1 - 6.4e-14
44 MaxOft, t = 30 eps
44 MaxOft AD, t = 30 1 - eps1
45 SampleProd, t = 10 0.9998
46 SampleProd, t = 30 0.9997
49 AppearanceSpacings, r = 0 1 - eps1
50 AppearanceSpacings, r = 20 1 - eps1
51 WeightDistrib, r = 0 1.1e-16
52 WeightDistrib, r = 8 eps
53 WeightDistrib, r = 16 eps
54 WeightDistrib, r = 24 eps
55 SumCollector eps
56 MatrixRank, 60 x 60 eps
57 MatrixRank, 60 x 60 eps
58 MatrixRank, 300 x 300 eps
59 MatrixRank, 300 x 300 eps
60 MatrixRank, 1200 x 1200 eps
61 MatrixRank, 1200 x 1200 eps
63 GCD, r = 0 eps
64 GCD, r = 10 eps
65 RandomWalk1 H (L = 90) eps
65 RandomWalk1 M (L = 90) eps
65 RandomWalk1 J (L = 90) eps
65 RandomWalk1 R (L = 90) eps
65 RandomWalk1 C (L = 90) eps

```
66 RandomWalk1 H (L = 90) eps
66 RandomWalk1 M (L = 90) eps
66 RandomWalk1 J (L = 90) eps
66 RandomWalk1 R (L = 90) eps
66 RandomWalk1 C (L = 90) eps
67 RandomWalk1 H (L = 1000) eps
67 RandomWalk1 M (L = 1000) eps
67 RandomWalk1 J (L = 1000) eps
67 RandomWalk1 R (L = 1000) eps
67 RandomWalk1 C (L = 1000) eps
68 RandomWalk1 H (L = 1000) eps
68 RandomWalk1 M (L = 1000) eps
68 RandomWalk1 J (L = 1000) eps
68 RandomWalk1 R (L = 1000) eps
68 RandomWalk1 C (L = 1000) eps
69 RandomWalk1 H (L = 10000) eps
69 RandomWalk1 M (L = 10000) eps
69 RandomWalk1 J (L = 10000) eps
69 RandomWalk1 R (L = 10000) eps
69 RandomWalk1 C (L = 10000) eps
70 RandomWalk1 H (L = 10000) eps
70 RandomWalk1 M (L = 10000) eps
70 RandomWalk1 J (L = 10000) eps
70 RandomWalk1 R (L = 10000) eps
70 RandomWalk1 C (L = 10000) eps
72 LinearComp, r = 29 1 - eps1
72 LinearComp, r = 29 1 - eps1
73 LempelZiv 1 - eps1
74 Fourier3, r = 0 eps
75 Fourier3, r = 20 eps
76 LongestHeadRun, r = 0 eps
76 LongestHeadRun, r = 0 1 - eps1
77 LongestHeadRun, r = 20 eps
77 LongestHeadRun, r = 20 1 - eps1
78 PeriodsInStrings, r = 0 eps
79 PeriodsInStrings, r = 15 eps
80 HammingWeight2, r = 0 eps
81 HammingWeight2, r = 20 eps
82 HammingCorr, L = 30 eps
83 HammingCorr, L = 300 eps
84 HammingCorr, L = 1200 eps
85 HammingIndep, L = 30 eps
86 HammingIndep, L = 30 eps
87 HammingIndep, L = 300 eps
88 HammingIndep, L = 300 eps
89 HammingIndep, L = 1200 eps
```

```

90 HammingIndep, L = 1200 eps
91 Run of bits, r = 0 eps
91 Run of bits, r = 0 1 - eps1
92 Run of bits, r = 20 eps
92 Run of bits, r = 20 1 - eps1
93 AutoCor, d = 1 eps
94 AutoCor, d = 1 eps
95 AutoCor, d = 30 1 - eps1
96 AutoCor, d = 10 1 - eps1
-----

```

All other tests were passed

Crush Battery on Mersenne Twister

===== Summary results of Crush =====

```

Version:  TestU01 1.2.3
Generator:  ugfsr_CreateMT19937_98
Number of statistics:  144
Total CPU time:  00:32:18.40
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

```

Test p-value

```

-----
71 LinearComp, r = 0 1 - eps1
72 LinearComp, r = 29 1 - eps1
-----

```

All other tests were passed

Crush Battery on Quantis 1 (200 GB)

===== Summary results of Crush =====

```

Version:  TestU01 1.2.3
Generator:  ufile_CreateReadBin
Number of statistics:  144
Total CPU time:  00:30:20.01
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

```

Test p-value

```

-----
46 SampleProd, t = 30 1 - 4.3e-8
51 WeightDistrib, r = 0 3.2e-5

```

```

65 RandomWalk1 H (L = 90) 2.3e-5
65 RandomWalk1 J (L = 90) 7.3e-4
67 RandomWalk1 H (L = 1000) 2.1e-4
69 RandomWalk1 M (L = 10000) 4.9e-4
85 HammingIndep, L = 30 2.3e-11
91 Run of bits, r = 0 eps
91 Run of bits, r = 0 1 - 2.9e-14
92 Run of bits, r = 20 eps
92 Run of bits, r = 20 1 - 1.1e-12
93 AutoCor, d = 1 2.9e-54
94 AutoCor, d = 1 3.9e-22

```

All other tests were passed

Crush Battery on AES OFB

===== Summary results of Crush =====

```

Version: TestU01 1.2.3
Generator: ucrypto_CreateAES
Number of statistics: 144
Total CPU time: 01:01:45.25

```

All tests were passed

C.3 Alphabit Tests

Alphabit Battery on RANDU

===== Summary results of Alphabit =====

```

Version: TestU01 1.2.3
Generator: ulcg_CreateLCG
Number of bits: 137438953472
Number of statistics: 17
Total CPU time: 01:13:33.89
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

```

Test p-value

1 MultinomialBitsOver, L = 2 eps
2 MultinomialBitsOver, L = 4 eps
3 MultinomialBitsOver, L = 8 eps
4 MultinomialBitsOver, L = 16 eps

```

5 HammingIndep, L = 16 eps
6 HammingIndep, L = 32 eps
7 HammingCorr, L = 32 eps
8 RandomWalk1 H (L = 64) eps
8 RandomWalk1 M (L = 64) eps
8 RandomWalk1 J (L = 64) eps
8 RandomWalk1 R (L = 64) eps
8 RandomWalk1 C (L = 64) eps
9 RandomWalk1 H (L = 320) eps
9 RandomWalk1 M (L = 320) eps
9 RandomWalk1 J (L = 320) eps
9 RandomWalk1 R (L = 320) eps
9 RandomWalk1 C (L = 320) eps
-----

```

Alphabit Battery on Mersenne Twister

===== Summary results of Alphabit =====

```

Version:  TestU01 1.2.3
Generator:  ugfsr_CreateMT19937_98
Number of bits:  1717986918400
Number of statistics:  17
Total CPU time:  22:09:20.45

```

All tests were passed

Alphabit Battery on Quantis 1 (200 GB file)

===== Summary results of Alphabit =====

```

Version:  TestU01 1.2.3
File:  random.bit
Number of bits:  1717986918400
Number of statistics:  17
Total CPU time:  14:09:11.98
The following tests gave p-values outside [0.001, 0.9990]:
(eqs means a value < 1.0e-300):
(eqs1 means a value < 1.0e-15):

```

Test p-value

```

-----
1 MultinomialBitsOver, L = 2 eps
2 MultinomialBitsOver, L = 4 eps
3 MultinomialBitsOver, L = 8 eps
4 MultinomialBitsOver, L = 16 4.7e-8
5 HammingIndep, L = 16 eps

```



```

6 HammingIndep, L = 32 eps
8 RandomWalk1 H (L = 64) eps
8 RandomWalk1 M (L = 64) eps
8 RandomWalk1 J (L = 64) eps
8 RandomWalk1 R (L = 64) 9.6e-193
9 RandomWalk1 H (L = 320) eps
9 RandomWalk1 M (L = 320) eps
9 RandomWalk1 J (L = 320) eps
9 RandomWalk1 R (L = 320) 3.7e-21

```

Alphabit Battery on Quantis 1 (8 GB file)

===== Summary results of Alphabit =====

```

Version: TestU01 1.2.3
File: /data/krija267/random8gb.bit
Number of bits: 64000000000
Number of statistics: 17
Total CPU time: 00:31:38.90
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

```

Test p-value

```

-----
1 MultinomialBitsOver, L = 2 eps
2 MultinomialBitsOver, L = 4 6.9e-243
3 MultinomialBitsOver, L = 8 1.5e-42
5 HammingIndep, L = 16 7.6e-39
6 HammingIndep, L = 32 6.5e-24
8 RandomWalk1 H (L = 64) 4.3e-93
8 RandomWalk1 M (L = 64) 1.7e-25
8 RandomWalk1 J (L = 64) 1.1e-48
8 RandomWalk1 R (L = 64) 2.3e-4
9 RandomWalk1 H (L = 320) eps
9 RandomWalk1 M (L = 320) eps
9 RandomWalk1 J (L = 320) eps
-----

```

All other tests were passed

Alphabit Battery on Quantis 2 (8 GB file)

===== Summary results of Alphabit =====

```

Version: TestU01 1.2.3
File: /data/krija267/quantis2binary.bit
Number of bits: 64000000000

```

Number of statistics: 17
Total CPU time: 00:31:28.95
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value $< 1.0e-300$):
(eps1 means a value $< 1.0e-15$):

Test p-value

1 MultinomialBitsOver, L = 2 eps
2 MultinomialBitsOver, L = 4 eps
3 MultinomialBitsOver, L = 8 eps
4 MultinomialBitsOver, L = 16 6.3e-99
5 HammingIndep, L = 16 eps
6 HammingIndep, L = 32 4.3e-232
8 RandomWalk1 H (L = 64) eps
8 RandomWalk1 M (L = 64) 3.3e-58
8 RandomWalk1 J (L = 64) 1.6e-43
8 RandomWalk1 R (L = 64) 7.9e-210
9 RandomWalk1 H (L = 320) eps
9 RandomWalk1 M (L = 320) 1.2e-9
9 RandomWalk1 J (L = 320) eps
9 RandomWalk1 R (L = 320) eps

All other tests were passed

Bibliography

- [1] Deborah J. Bennett. *Randomness*. Harvard University Press, 1998.
- [2] E.R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill series in systems science. Aegean Park Press, 1984.
- [3] V.B. Braginsky, V.B. Braginskiĭ, F.Y. Khalili, and K.S. Thorne. *Quantum Measurement*. Cambridge University Press, 1995.
- [4] Robert G. Brown. dieharder, . URL <http://www.phy.duke.edu/~rgb/General/dieharder.php>.
- [5] Robert G. Brown. DieHarder: A Gnu Public License Random Number Tester. . URL <http://www.phy.duke.edu/~rgb/General/dieharder.php>.
- [6] Schneier Bruce. *Applied Cryptography*. John Wiley & Sons, 2 edition, 1996.
- [7] Cristian Calude. Borel Normality and Algorithmic Randomness. In *Developments in Language Theory*, pages 113–129, 1993.
- [8] Cristian S. Calude, Michael J. Dinneen, Monica Dumitrescu, and Karl Svozil. Experimental evidence of quantum randomness incomputability. *Phys. Rev. A*, 82:022102, Aug 2010.
- [9] Jean-Sébastien Coron and David Naccache. An Accurate Evaluation of Maurer’s Universal Test. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 1998.
- [10] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2 edition, 2006.
- [11] Ivan Damgård, Peter Landrock, and Carl Pomerance. Average Case Error Estimates for the Strong Probable Prime Test. *Mathematics of Computation*, 61(203):pp. 177–194, 1993. ISSN 00255718.
- [12] E. J. Dudewicz, E. C. van der Meulen, M. G. Sriam, and N. K. W. Teoh. Entropy-Based Random Number Evaluation. *American Journal of Mathematical and Management Sciences*, 15:115–153, 1995.

- [13] Fida El Haje, Yuri Golubev, P-Y Liardet, and Yannick Teglia. On statistical testing of random numbers generators. In *Security and Cryptography for Networks*, pages 271–287. Springer, 2006.
- [14] M. Fischler. Distribution of minimum distance among n random points in d dimensions. 2002.
- [15] C. S. Forbes. *Statistical distributions*. Wiley-Blackwell, Oxford, 4th ed. edition, 2010.
- [16] J.E. Gentle. *Random Number Generation and Monte Carlo Methods*. Statistics and Computing. Springer, 2003.
- [17] Larry Greenemeier. NSA Efforts to Evade Encryption Technology Damaged U.S. Cryptography Standard. URL <http://www.scientificamerican.com/article/nsa-nist-encryption-scandal/>.
- [18] Brian C. Hall. *Quantum Theory for Mathematicians*. Springer, 2013.
- [19] iTech Labs. RNG Testing & Certification. URL <http://www.itechlabs.com.au/sidebar-eng/rng-testing-certification/>.
- [20] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo methods*. Wiley-Blackwell, Weinheim, 2. rev. and enl. ed. edition, 2008.
- [21] Donald E Knuth. *The Art of Computer Programming*. Addison Wesley Longman, Inc., 3 edition, 1997.
- [22] Compliance Testing Laboratory. Certificate of Compliance, 2011. URL <http://www.idquantique.com/images/stories/PDF/quantis-random-generator/ctl-certificate.pdf>.
- [23] Pierre L'Écuyer, Jean-François Cordeau, and Richard Simard. Close-Point Spatial Tests and Their Application to Random Number Generators. *Operations Research*, 48(2):pp. 308–317, 2000. ISSN 0030364X.
- [24] Pierre L'Ecuyer. TestU01. URL <http://simul.iro.umontreal.ca/testu01/tu01.html>.
- [25] Pierre L'Ecuyer and Richard Simard. TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators. URL <http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf>.
- [26] Pierre L'Ecuyer and Richard Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Softw.*, 33(4), August 2007. ISSN 0098-3500.
- [27] Pierre L'Ecuyer, Richard Simard, and Stefan Wegenkittl. Sparse Serial Tests of Uniformity for Random Number Generators, journal = .
- [28] G. Marsaglia and A. Zaman. Monkey tests for random number generators. *Computers & Mathematics with Applications*, 26(9):1 – 10, 1993. ISSN 0898-1221.

- [29] George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. URL <http://www.stat.fsu.edu/pub/diehard/>.
- [30] George Marsaglia. A Current View of Random Number Generators. In Lynne Billard, editor, *Computer science and statistics : proceedings of the Sixteenth Symposium on the Interface, Atlanta, Georgia, March 1984*, Computer science and statistics, pages 3–10. North-Holland, 1985.
- [31] George Marsaglia and Wai Wan Tsang. Some Difficult-to-pass Tests of Randomness. *Journal of Statistical Software*, 7(3):1–9, 1 2002. ISSN 1548-7660.
- [32] Michael Mascagni. The Scalable Parallel Random Number Generators Library (SPRNG). URL <http://www.sprng.org/>.
- [33] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. ISSN 1049-3301.
- [34] Ueli Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of cryptology*, 5:89–105, 1992.
- [35] U.S.Department of Commerce, National Institute of Standards, and Technology. FIPS PUB 140-2, Security Requirements for Cryptographic Modules. 2002.
- [36] Federal Department of Justice and Police. Quantis True Random Number Generator Certified by Compliance Testing Laboratory, 2011. URL <http://www.idquantique.com/press-releases/ctl-certification.html>.
- [37] Swiss Federal Office of Metrology. Annex to METAS Certificate Nr. 151-04687, 2010. URL www.idquantique.com/images/stories/PDF/quantis-random-generator/annex-metas-certificate.pdf.
- [38] Swiss Federal Office of Metrology. Certificate of Conformity No 151-04687, 2010. URL <http://www.idquantique.com/images/stories/PDF/quantis-random-generator/metas-certificate.pdf>.
- [39] ID Quantique. ID Quantique White Paper, Apr 2010. URL <http://www.idquantique.com/images/stories/PDF/quantis-random-generator/quantis-whitepaper.pdf>.
- [40] Nornadiyah Razali and Yap B. Wah. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2, June 2011.
- [41] Timothy R C Read and Noel Cressie. *Goodness-of-fit statistics for discrete multivariate data*. Springer series in statistics. Springer, New York, NY, 1988.

- [42] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, Andrew Rukhin, Juan Soto, Miles Smid, Stefan Leigh, Mark Vangel, Alan Heckert, James Dray, and Lawrence E Bassham Iii. Download Documentation and Software. URL http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.
- [43] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, Andrew Rukhin, Juan Soto, Miles Smid, Stefan Leigh, Mark Vangel, Alan Heckert, James Dray, and Lawrence E Bassham Iii. A statistical test suite for random and pseudorandom number generators for cryptographic applications. 2001. URL <http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf>.
- [44] B.Ya. Ryabko and A.I. Pestunov. "Book Stack" as a New Statistical Test for Random Numbers. *Problems of Information Transmission*, 40(1):66–71, 2004.
- [45] M. A. Stephens. EDF Statistics for Goodness of Fit and Some Comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974.
- [46] A. Stevenson. *Oxford Dictionary of English*. Oxford Dictionary of English. OUP Oxford, 2010.
- [47] I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical models as tests of randomness. *Phys. Rev. E*, 52:3205–3214, Sep 1995.
- [48] John von Neumann and A. H. Taub. *John von Neumann Collected Works*. Pergamon Printing and Art Services, 1963.
- [49] John Walker. ENT. URL <http://www.fourmilab.ch/random/>.
- [50] Eric W. Weisstein. Craps. URL <http://mathworld.wolfram.com/Craps.html>.
- [51] Jacob Wolfowitz. Asymptotic distribution of runs up and down. *The Annals of Mathematical Statistics*, 15:163–165, 1944.
- [52] R.D. Yates and D.J. Goodman. *Probability and stochastic processes: a friendly introduction for electrical and computer engineers*. John Wiley & Sons, 2005.

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>