Not Finding Nemo

Creating a random number generator
from a physical process

Authors:
Filip Gustafsson pt00fgu@student.bth.se
David Virdefors pt00dvi@student.bth.se

Supervisor:
Rune Gustavsson

Blekinge Institute of Technology

Index

1 Abstract

The need for information security is today an every growing subject and problem. Much effort and money are spent to improve and increase the security in information systems to ensure privacy and confidentiality. A large part in this area regards cryptography. There exists a vast number of different ways to transform plain text into a cipher, thus making it harder to read by a possible third part. Some of the crypto algorithms, e.g. the widely know RSA algorithm, are dependent on some sort of randomness. They are built so they need one or many random values. These values function as a parameter for the algorithm and without some sort of random number generator(RNG) they would not function at all.

One can divide RNGs into two large groups, pseudo random number generators (PRNGs) and generators which are based upon some sort of physical process as input. Pseudo random number generators are the ones that are most widely used today. The PRNGs are based on an algorithm which is feed by a start value, also known as seed. The seed could be e.g. current time. The algorithm in PRNGs may vary and are often robust and hard to predict, however the use of a seed creates many problems, thus creating predictable and not random numbers.

The alternative to PRNGs is to study some sort of physical environment and then transform the input to random numbers. This could be sound waves or to generate random numbers by looking at the radioactive decay.

Using a physical process to generate random numbers more often gives higher quality random numbers when compared to PRNGs.

We created our own two RNGs called Clean and Dirty Nemo based on a physical process. We set up a web cam in front of an aquarium containing a number of fishes and took a digital image every second. Then each image where analyzed based upon the movement of the fishes. This data was then run through an algorithm which creates bits, ones and zeros, which then was transformed into random numbers.

Our random number generator was then put through various tests together with Java’s and c#’s PRNGs. The different tests used were two and three dimensional visual test, chi-2 and n-block tests. Clean and Dirty Nemo proved very competitive against Java and c# in the chi-2 test, the same goes for the visual tests. However, none of the candidates passed the n-block test, all failed. This could mean a tendency towards some sort of cyclic behaviour.

Our intentions to create a random number generator based upon a physical process and to test its randomness were fulfilled. The generators also proved quite competitive compared to other PRNGs.

2

Guide to the reader

There are a couple of different ways you can read this thesis. To get a general overview, we recommend you read the parts named, Abstract, Why random numbers?, Problem statement, Evaluation and Conclusion. If your interest mainly lies in the RNG we created and its test results read the parts named Nemo and Future improvement. If your interest concern software you can read the parts named Processing software, Masking and Algorithms. These parts explain how we implemented our algorithms in the software application.

The final options is to read the whole thesis, this is of course what we recommend.

We assume that the reader has moderate knowledge in computer science and mathematics since we use some technical terms in the thesis.

3 Acknowledgments

We like to thank Claes Jogreus for his advice regarding statistical testing methods. We also thank our advisor Rune Gustavsson who helped us during the writing of this thesis. Further more we would like to thank Marcus Sjölin and Magnus Karlsson for valuable help with MatStat.

We would like to say that no fishes were harmed during the making of this master thesis, however this is not the case. Sadly one of the guppies lost its life in a tragic accident. This happened during the second day of the experiment.

4

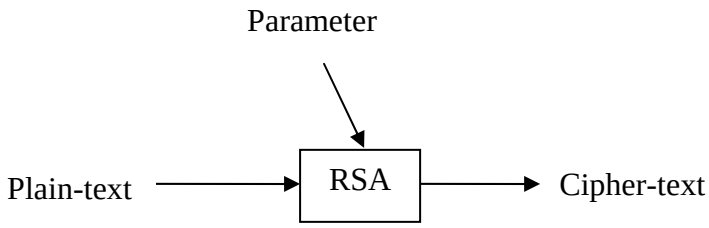
Background

4.1 Why random numbers?

Information security has long been viewed by information system professionals as being vital. Computing facilities and the information systems they support have become increasingly accessible as a result of the explosion of the open, public Internet since about 1993. A great deal of public attention is now being focussed on the topic.

A large part of information security concerns the field of cryptography. Even the ancient Greeks had methods to keep the contents of a message safe. They shaved the hair of the messenger, wrote the message on the head and let the hair grow back again. The drawback of this approach was that the message could not be too urgent. Since then much have happened and cryptography is now a widespread science. Coded messages are often used in war time by the military, hence why a good or flawed RNG could mean the difference between life and death. This might explain why e.g. US National Security Agency (NSA) has such in interest in this area.

The most known crypto algorithm might be RSA. The image below shows how the method works. A plain-text is transformed by an algorithm to cipher-text. RSA uses the same algorithm every time but reuses the key. For this to be safe the key or parameter as you also can say has to be random. To create random keys you must have a RNG. If the key that the random number generator produces is guessable in any way the whole concepts with a static algorithm fails. But if the RNG generates good quality random numbers it does not matter that the algorithm is well known and public.



There exist numerous different implementations of crypto algorithms where RSA is only one of them. One can divide the algorithms into two different types, the use of symmetric keys and the use of asymmetric keys. “A symmetric-key algorithm is an algorithm for cryptography that uses the same cryptographic key to encrypt and decrypt the message.”[7]. To ensure safety, the key should be randomly created. “Other terms for symmetric-key encryption are single-key and private-key encryption. Use of the latter term is discouraged because of conflict with the term private key in public key cryptography.“[7].

“In cryptography, an asymmetric key algorithm uses a pair of different, though related, cryptographic keys to encrypt and decrypt. The two keys are related mathematically; a message encrypted by the algorithm using one key can be decrypted by the same algorithm using the other. In a sense, one key "locks" a lock (encrypts); but a different key is required to unlock it (decrypt).

An analogy which can be used to understand the advantages of an asymmetric system is to imagine two people, Alice and Bob, sending a secret message through the public mail.

With a symmetric key system, Alice first puts the secret message in a box, and then padlocks the box using a lock to which she has a key. She then sends the box to Bob through regular mail. When Bob receives the box, he uses an identical copy of Alice's key (which he has somehow obtained previously) to open the box, and reads the message.

In an asymmetric key system, instead of opening the box when he receives it, he simply adds his own personal lock to the box, then returns the box to Alice. Alice uses her key to remove her lock, and returns the box to Bob, with Bob's lock still in place. Finally, Bob then uses his key to remove his lock, and reads the message from Alice.“ [6].

To ensure safety the key should be created randomly. The key has to be uniqe and hard to guess. If another key is generated that is identical the whole method looses its strenght. RSA which was mentioned before uses asymmetric keys. The algorithm was patented by MIT in 1983 in the United States of America and revolutionised the world of cryptography. It uses the concept of public and private keys. The private key is kept secret while the public key is distributed public as the names implie. The keys are created by choosing two large prime numbers at random. To break the cipher one must find out the two prime numbers. To ensure the security of the RSA algorithm one need a good RNG. Even if RSA is well known and the algorithm is released to the public it is still highly secure to use this algorithm.

There are more usages for RNGs than in cryptography, e.g. gambling. Man has always been thrilled by the game of chance. The oldest dice we know of were found in Pakistan and is estimated to be about 5000 years old. Even though we still play crap game using ordinary dice many gambling games have been substituted for digital random number generators. We can find RNGs both in slot machines and in an ever growing amount of internet poker- and gambling sites. If the RNG is somewhat skewed towards certain numbers or if it is producing predictable patterns the economical losses for e.g. a casino could be devastating.

Another area also using RNGs is when producing passwords. Since humans tend to come up with passwords that can easily be broken with a brute force attack, it can be better to let a computer using a RNG produce it. In software systems that require a one use only password, it is almost essential to have good RNG that creates a list of random passwords.

A more joyful use of RNGs is in video games and such. Everything from games as Pacman to Quake, the game play experience is enhanced by adding a bit of random factor to it. It would not be as fun to play if the monsters and other adversaries acted the exact same way all the time.

As you can se RNGs plays an important role in information security, especially when creating keys in cryptographic algorithms. This is one of the reasons that we decided to create and test our own random number generator approach.

4.2 Problems with Pseudo random generators

Pseudo random number generators are based on an algorithm. They take a start value as a parameter; the start value is called a seed. If you start with the same seed, you will get the

same sequence of values from the formula. Consider the following Java code (somewhat stripped for increased readability).

```
int mySeed = 42;
Random myRandom = new Random(mySeed);

for(int i = 0; i < 3; i++)
{
        System.out.println(myRandom.nextFloat());
}
```

The `Random` instance, `myRandom` is given a static seed value, in this example 42. The `nextFloat` method returns the next floating point value, this is done three times in the `for` loop. When running this application in two separate executions the following print out is displayed:

```
C:\Nemo>java SeedExample
0.7275637
0.054665208
0.6832234

C:\Nemo>java SeedExample
0.7275637
0.054665208
0.6832234
```

As seen here, the three generated floating point values are identical although executed at two different executions. This is not an error or bug in the compilator/run-time environment, this is how seeds work. Given the same seed the PRNG generates the same sequence of numbers. In the case of the Global Positioning System (GPS), this reproducibility is used as a way to give each satellite a predictable but different pattern of values that the GPS receiver can track. Nevertheless in most cases different techniques are used to come up with new seeds to avoid reproduction a common solution often used is to let the seed be based on time. Either current local time or the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. However, as described in more in Appendix 1, this technique should be used with care since certain problems might arise.

A chain is not stronger than its weakest link. This is the case even in pseudo random generators. Let us say that we have a random sequence with 52 bits and a seed with 8 bits. The testing set becomes 8 bit sequences instead of 52 bit sequences. Therefore we reduced the security from 52 bits to 8 bits just by knowing the environment.

Another problem that could occur is if we create seeds in a loop, then we have a seed sequence with some kind of pattern. For example see Appendix 1. This is because a loop is a cycle that runs in a predictable way.

As we just have shown many problems arise when using PRNGs that are feed by a seed. They have many unwanted behaviours and creates possibilities for predicting the generated numbers. Predictable numbers are the last thing you want when working with RNGs. Bearing in mind the numerous problems with PRNGs and the vast area of use in cryptography, we wanted to contribute with our own random number generator.

One alternative to PRNGs is to study a physical process such as sound waves or radio active decay. Since these two approaches require expensive equipment we wanted to create our own RNG with much simpler means but with the same effectiveness. A web cam taking pictures of an aquarium seemed to fulfil these requirements.

4.3 Definition of random

Random: Being or relating to a set whose numbers have an equal probability of occurring. In physics, substitute situation for set, values for numbers. In a random physical situation, all possible values of the pertinent parameters are equally probable. By being harsh one could argue that there is no such thing as quasi or pseudo random. A situation is random or it is not. If the atoms in a closed system (A system with no external influence) have random velocities, they will always have random velocities. Of course, there actually is no such thing as a true "closed system". Many physicists play fast and loose with "random", which can lead to dubious conclusions. It may be that there actually is no such thing as a true random situation.

4.4 Cryptographically strong random number generators

A cryptographically strong random number generator passes any statistical test for randomness that does not require an exponentially increasing to infinite amount of time to run. All statistical tests will be unable to distinguish the random number generator from a true random source.

The values produced by the generator are random in an absolutely precise way. To predict the sequence you might as well toss a coin. Furthermore, having a large chunk of output from the random number generator does not help in predicting past or future values.

4.5 Problem statement

We want to research whether it is possible to create a competitive RNG using a physical process. The physical input shall consist of an aquarium containing fishes.

To determine whether the RNG is competitive or not, we must test the numbers generated. To do this we have to choose appropriate and accepted testing methods. We want to use Java’s and c#’s PRNGs as test references.

5 Existing random number generators

This chapter will cover two types of RNGs, pseudo random number generators and random number generators based on a physical process.

PRNGs are based on an algorithm and use some type of seed to generate random numbers. PRNGs are robust in the meaning that they always behave in the same way. However the random number sequences might contain patterns. A good example is RANDU which has obvious flaws. See appendix 4.

The second type is RNGs based on a physical process. These RNGs often generates random numbers with higher quality. These types of RNGs are often easier to implement than coming up with a completely new PRNG algorithm. However they often require more equipment and they are dependent on the physical process at hand. In the case where a microphone records sound the RNG is dependent on that the microphone is functional.

5.1 Pseudo random number generators

Pseudo random number generators are the most common type of number generator. They are based on an algorithm and take a start value as a parameter. The start value is called a seed. These algorithms described below are only examples. For further explanations about hyper plane, period and the algorithms we refer to the section named References.

5.1.1 Linear congruential generator

The linear congruential generators (LCGs) are one of the oldest and best known pseudorandom number generator algorithms. It was back in 1948 that D.H.Lehmer came up with a new base for pseudorandom number generators. The theory behind these generators is not that complex, easy to implement and quite fast. However LCGs are far from ideal if higher quality random numbers are needed because of them being based upon an algorithm and seed.

LCGs are defined by the recurrence relation:

$$V_{j+1} = A * V_j + B (\mathrm{mod}\, M) \qquad (1)$$

Where V_n is the sequence of values and A, B and M are generator-specific constants. A,B and M are choosen carefully to give the right characteristics: $a, b \geq 0$ and $M > V_0, a, b$

The period of a general LCG is at most M, and very often less than that. In addition, they tend to suffer from defects. For instance, if an LCG is used to choose points in an n-dimensional

space, triples of points will lie on, at most, $M^{1/n}$ hyper planes. This is due to serial correlation between successive values of the sequence $V_{n.}$

Another problem with LCGs is that the lower-order bits of the generated sequence may cycle with a far shorter period than the sequence as a whole, particularly if M is not a prime number.

5.1.2 Inversive congruential generator

In 1986 a new form of PRNGs where introduced by Eichenauer and Lehn. This was called Inversive congruential generators (ICG). It is only slightly different from the earlier LCG, and it is defined by:

$$y_{n+1} = (a \cdot \bar{y}_n + b) \bmod(M) \qquad (2)$$

This generator has been modified to exclude the problems regarding the old LCG. However, there are other problems with the ICG. One obvious drawback is that the ICG is slower than LCG in comparison. The reason for this is that inversion is not a simple matter when involving modulus, hence why the calculations are very time consuming. Further adjustments and refinements where made resulting in the EICG short for Explicit inversive congruential generators.

5.1.3 Blum Blum Shub

Another generator is the Blum Blum Shub generator (BBS), which is defined as follows:

$$y_{n+1} = (a \cdot y_n^{\,2} + b) \bmod(M) \qquad (3)$$

Here *M* should be the product of two primes. The big benefit is that the quadratic form gives safety to the system, and it will be as hard to predict the output of this generator as to crack the RSA encryption system. Though there is one major catch, the period is far shorter than *M*. This makes the BBS unsuitable for stochastic simulations and many other applications.

5.1.4 Shift-Register generator

A very important property concerning simulation applications is to have as little correlation effects as possible (if any at all), i.e. successive numbers or numbers some distance apart should not affect each other. If they do that, they will for instance often result in clusters of numbers occurring with some interval that show some common feature. This is heavily reduced using a Shift register generator (SRG). The algorithm works as follows. The random number is a string of r bits. Each bit in the next number is determined by the corresponding bits of the previous n numbers. If we denote the k-th bit of the i-th number in the sequence by $b_i^{(k)}$, we can write down a general formula for the generation:

$$b_i^{(k)} = (c_1 b_{i-1}^{(k)} + c_2 b_{i-2}^{(k)} + \ldots + c_n b_{i-n}^{(k)}) \bmod (2) \qquad (4)$$

where the numbers c_i are equal to 0 or 1. The maximum period is $2_n - 1$ and is obtained for very special combinations of c_i. A simple and common form of this generator is

$$y_i^{(k)} = (y_{i-q}^{(k)} + y_{i-p}^{(k)}) \bmod (2) \qquad (5)$$

This corresponds exactly to the XOR operation. On a computer it turns out to be really fast. The (p, q)-pairs resulting in an optimal cycle length are often called 'Magical Pairs' and some of them are (98, 27), (521, 32) and (250, 103). Before this generator can be used it needs n random generated numbers to start with. This can easily be done by e.g. LCG.

5.2 Random generators based on a physical process

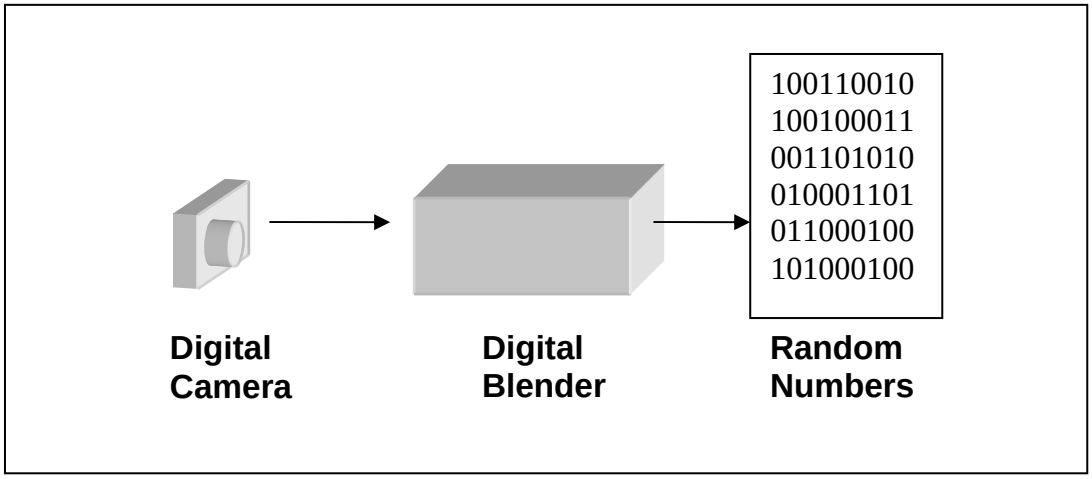
Random number generators based on a physical process typically generates their numbers by sampling and processing a source outside the computer. For example sound or radio active decay.

There is no truly random source in theory but by having a very complex nature as source, it becomes more or less impossible to recreate the same environment again.
So the expression truly random generator is a expression that is movable depending on current research. If someone comes up with an easy way to solve the decay of a radioactive source, this type of RNG will be consered a pseudo random generator.

5.2.1 Lava Lamp

Lava lamp RNG is a number generantor that is based on data from a physical source, in this case a lava lamp. Several pictures are taken. Every picture is then run through a digital blender algorithm. This Lavarnd digital blender algorithm consists of an n-way-turn; n

different SHA-1 cryptographic operations running in paralell, and n different xor-rotate operations. You could find info about this project on www.lavarnd.com.
Lavarnd is a cryptographically sound random number generator, this means that with no standard battery of statistical tests for randomness we could see differences between this type of RNG and a true random source.



The digital blender is the most complex part of lavarnd. It consists of several types of data transformation operators to get as good random numbers as possible out of the physical process.

5.2.2 Radio active decay

The rate at which radio active sources decay is a very complex and hard to predict. However there are some problems setting up the environment. Handling the material is also a somewhat troublesome task.

To create a random number generator you will need some radio active material, an isolated environment and a tool for collecting the decay, e.g. Geiger Müller tube. You have to have some measurements on the materials decomposition. How often decay occurs and in what amounts, this is to be able to configure the test environment in a correct way. If the material decomposes in a slower rate then expected the outcome will not be as good as if the environment is correctly configured.

5.2.3 Audio input

By using a microphone that picks up sounds from the surroundings we hopefully have a source with good chaotic structure.

The analogue sound is converted to voltage by the microphone and then into data bits by a audio converter. For example, on a SPARCstation, one can read from the `/dev/audio` device with nothing plugged into the microphone jack. Such data is essentially random thermal noise although it should not be trusted without some checking in case of hardware failure. Under appropriate circumstances, such input can provide reasonably high quality random bits. The "input" from a sound digitizer with no source plugged in or a camera with

the lens cap on, if the system has enough gain to detect anything, is essentially thermal white noise.

White noise is a type of noise that is produced by combining sounds of all different frequencies together. If you took all of the imaginable tones that a human can hear and combined them together, you would have white noise.

The adjective "white" is used to describe this type of noise because of the way white light works. White light is light that is made up of all of the different colors (frequencies) of light combined together (a prism or a rainbow separates white light back into its component colors). In the same way, white noise is a combination of all of the different frequencies of sound. You can think of white noise as 20,000 tones all playing at the same time.

While white noise has the same distribution of power for all frequencies there is the same amount of power between 0 and 500 Hz, 500 and 1,000 Hz or 20,000 and 20,500 Hz.

Pink noise has the same distribution of power for each octave, so the power between 0.5 Hz and 1 Hz is the same as between 5,000 Hz and 10,000 Hz.

5.2.4 Mouse movement

Using mouse or keyboard movement as a source for creating random numbers is an alternative. The most known application currently using this is PGP, short for Pretty Good Privacy. PGP uses the user’s mouse movement when creating private and public keys for the RSA algorithm.

The advantage of mouse movement is the fact that it is hard to reconstruct the actual event and it is hard to guess how the user was moving the mouse. But this approach has its disadvantages; the human user is not always available, the server could be in a basement somewhere and do not even have a mouse interface, the server could have more than one process and therefore the human could not keep up and give the computer all its desired movement data. But the most common problem with this approach is if the user is forced to do this random movement a number of times after each other. Then the movement has a tendency to be less unique then first believed because the human lack of unlimited stamina.

5.2.5 Disk Drives

Disk drives have small random fluctuations in their rotational speed due to air turbulence which is very hard to predict. By adding low level disk seek time instrumentation to a system, a series of measurements can be obtained that include this randomness. Such data is usually highly correlated so that significant processing is needed. Nevertheless experimentation has shown that, with such processing, disk drives easily produce 100 bits a minute or more of excellent random data.

5.3 Mixing functions

5.3.1 Basic functions

A mixing function is one which combines two or more inputs and produces an output where each output bit is a different complex non linear function of all the input bits. Mixing functions can be used to enhance both PRNGs as well as RNGs based on external physical input. On average, changing any input bit will change about half the output bits. But because the relationship is complex and non-linear, no particular output bit is guaranteed to change when any particular input bit is changed.

Consider the problem of converting a set of bits to a shorter set that is more random than the input bits. In this case the set of bits are skewed towards 0 or 1. This is a typical example when a mixing function is desired.

One simple approach to illustrate the use of mixing functions is to apply exclusive or (XOR) to the set of bits. Divide the input set in half, creating two equally large sets. Then XOR the two sets of input with each other. This will create our new output set, as seen in the table below.

input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

If inputs 1 and 2 are uncorrelated and combined in this fashion then the output will be an even better (less skewed) random bit sequence than the input.

5.3.2 Stronger mixing functions

The US Government Data Encryption Standard [DES] is an example of a strong mixing function for multiple bit quantities. It takes up to 120 bits of input (64 bits of "data" and 56 bits of "key") and produces 64 bits of output each of which is dependent on a complex non linear function of all input bits. Other strong encryption functions with this characteristic can also be used by considering them to mix their entire key and data input bits.

Another good family of mixing functions are the "message digest" or hashing functions such as The US Government Secure Hash Standard [SHS] and the [MD2, MD4, MD5] series. These functions all take an arbitrary amount of input and produce an output mixing all the input bits. The MD* series produce 128 bits of output and SHS produces 160 bits.

5.3.3 Factors when choosing mixing functions

For local use, DES has the advantages that it has been widely tested for flaws, is widely documented, and is widely implemented with hardware and software implementations available all over the world. The SHS and MD* family are younger algorithms which have not been tested as much. However there is no particular reason to believe they are flawed, but time will tell. Both MD5 and SHS were derived from the earlier MD4 algorithm. They are all public and the source code is available as well.

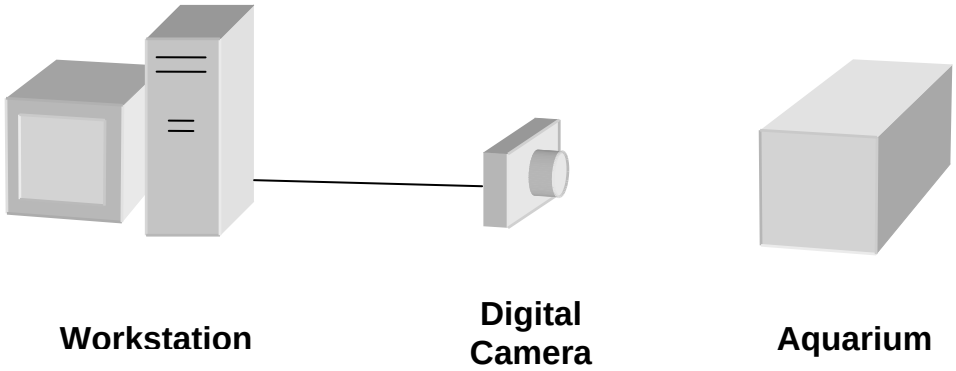
DES and SHS have been vouched for the US National Security Agency (NSA) on the basis of criteria that primarily remain secret. While this is the cause of much speculation and doubt, investigation of DES over the years has indicated that NSA involvement in modifications to its design, which originated with IBM, was primarily to strengthen it. No concealed or special weakness has been found in DES. It is almost certain that the NSA modification to MD4 to produce the SHS similarly strengthened the algorithm, possibly against threats not yet known in the public cryptographic community.

DES, SHS, MD4, and MD5 are royalty free for all purposes. MD2 has been freely licensed only for non-profit use in connection with Privacy Enhanced Mail [PEM]. Between the MD* algorithms, some people believe that MD2 is strong but too slow, MD4 is fast but too weak, and MD5 is just right. What it all comes down to is what environment you want to use your mixing function in.

6 Nemo

6.1 Experiment setup and environment

For this experiment we put up an aquarium containing a number of standard guppy aquarium fishes. We provided sufficient light directed at the aquarium to enhance the quality of the images. Furthermore a dark purple paper ark was attached to the back of the aquarium to get a better contrast to the fishes. Stones and water plants were removed so the fishes were not able to “hide” behind these objects.



A digital camera of the brand `MUSTEK-Gsmart mini 3` was then aligned properly in front of the aquarium so the shots taken covered the whole front of the aquarium. When the camera was in position it was fixed by tape so it would not move around. The camera can be set in both movie and picture mode. Since we only work with images in this experiment it was set to picture mode.

The images are then transmitted via an USB cable to a PC workstation where they were stored for later processing. A third party web camera software was used for this experiment. It makes the camera take a picture every second and saves the images as a jpeg files on the hard disk drive. The images were stored in the resolution 320 by 240 pixels and 24 bits colour depth.

6.2 Processing software

6.2.1 Masking

The images them self taken by the web camera did not provide us with any random numbers. In order to generate meaningful random numbers from the images, we wrote a software application. This application was written in the language c#.

The application traverses all the images taken by the web camera. For each image traversed the application have a very important task, namely to actually find the fishes. We tried different techniques but ended up using a colour range to pin point the guppies. This is done by specifying a lower `rgb` value followed by a higher `rgb` value. Each pixel in the image is then examined and everything that is in that specific color range is considered to be a fish.

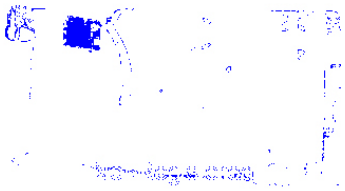
The two pictures below shows this procedure. The picture on the left is the original picture taken by the web camera. Picture B on the right has gone through the masking process previously described. For this image the application was given a lower `rgb` value of `RGB(70, 120 ,65)` and a higher `rgb` value of `RGB(255, 255 ,255).`

picture A

picture B

To illustrate the pixels that are considered to be a fish we have substituted them for a blue pixel and pixels considered to be anything else with a white pixel.

The method to find the appropriate lower and higher `rgb` bound was nothing more than a trial and error process. It was a matter of just testing different values until an acceptable result showed up. Picture C on the next page have a slightly different rgb span and therefore considers fewer pixels to be fish. However picture D have a broader and more accurate rgb span and therefore shows more blue pixels.



picture C
lower RGB(90, 140, 85)
higher RGB(240, 240, 240)

picture D
lower RGB(70, 120, 65)
higher RGB(255, 255, 255)

Once again consider the original and masked images. Most of the fishes in this image are in the top right area. But as you can see, it is not only the guppies that are masked blue and considered to be a fish. The white tubing providing fresh new water as well as the sand and metal frame are also in the same colour range and therefore displayed with a blue colour. Also some light reflexes are “mistaken” for being a fish. However our two algorithms that we created do not care about this. As long as the fishes are being successfully masked in every picture all is good.

6.3 Algorithms

We came up with two different algorithms to produce random numbers. Based upon Pixars animated movie Finding Nemo we called our two algorithms Dirty Nemo and Clean Nemo. The source for our random generators is depending on the fact that fishes have a constant movement. Hence the name Nemo is short for **N**ever **e**nding **m**otion **o**bjects. (Note: The title of the thesis “Not finding Nemo” is based on the fact that we do not want to be able to predict the movement of the fishes).

6.3.1 Clean Nemo

The pixel masking technique described in the “Processing software” part is used to initiate Clean Nemo. It starts of by counting the number of pixels considered to be a fish i.e. the blue pixels in the masked image. Each picture taken of the aquarium generates one bit each. This bit can of course only have the value one or zero. By calculating the number of masked pixels modulus 2 each picture produces either a one or a zero. I.e. if the number of pixels considered to be fish is even the image generates a zero.

For some of the experiments we wanted a greater span than just zeros and ones. To be able to do this we arranged the bits in blocks of six bits. Pretty much like a byte with the exception that this consists of six bits instead of eight. This way we created 6 bit numbers ranging from 0 to 63.

The reason that we did not create 8 bit numbers or even 32 bit numbers was due to the lack of samples. We took approximately 131 000 pictures which gave us the same amount of bits. Dividing this value by six was the best way to balance the amount of numbers to the span of the numbers. By creating 6 bit numbers we generated about 22000 numbers ranging from 0 to

63. If we would have created 4 byte integers consisting of 32 bits we would only have gotten approximately 4100 numbers ranging from 0 to about 4,3 billion. This range would have been unnecessary large.

6.3.2 Dirty Nemo

Clean Nemo got its name from the fact that it is clean from any mixing function or other post processing other than the 6 bit conversion. Dirty Nemo on the other hand uses exactly that, a mixing function.

The procedure to come up with bits from images, using pixel masking, is identical from Clean to Dirty Nemo. The difference is that Dirty Nemo is using a MD5 Hashing algorithm.
As described earlier MD5 takes an arbitrary amount of input and produces 128 bits of output. Dirty Nemo takes the first 256 bits from the 256 first images as input to the MD5 mixing function. The 128 bit output is then used to create 6 bit numbers the same way Clean Nemo does. This procedure is then repeated with all bits.

Since the amount of input bits are twice as big as the number of output bits this method only produces half the amount of generated numbers compared to Clean Nemo.

6.4 How to test randomness

When testing randomness we could talk about how hard it is to guess the sequence. For example an ordinary pseudo-random data generator is often very good at usual statistic tests but it could instead have some other weaknesses such as a time based seed or similar. This is however angles of the problem we will discard at this point, we will only discuss how to test the number sequences and not the weaknesses of how they where created.
First of all when testing random number sequences, one could test the numbers with an ordinary frequency test, such as chi-2 test or standard deviation. This is to be sure that all possible possibilities occur at the same range.

After the frequency test it is important to test if there are any patterns in the sequence. Patterns are the weakest link of random number sequences. Patterns in sequences are hard to test, there are some different tests that try to solve this problem but they are specialised at different types of patterns. There is no magic test that could test all types of sequences. To be sure that you have good numbers you have to run several tests on your sequence. Numbers may pass one test and fail another, this could be somewhat confusing. How many numbers you are putting in to the test is very important. If you have a small chump of numbers the test will probably get false output and will not give you fair results.

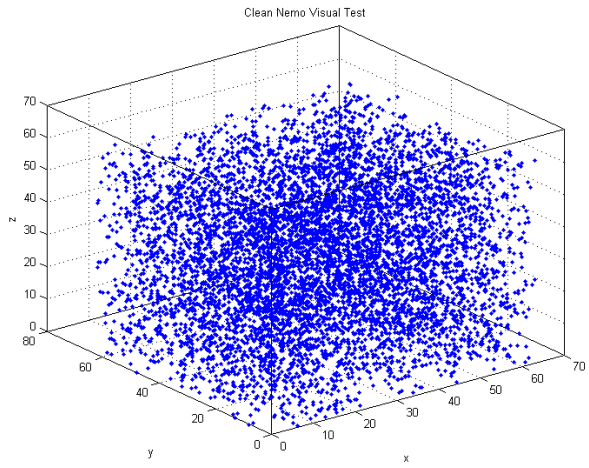
6.5 Experiment results

6.5.1 Visual Test

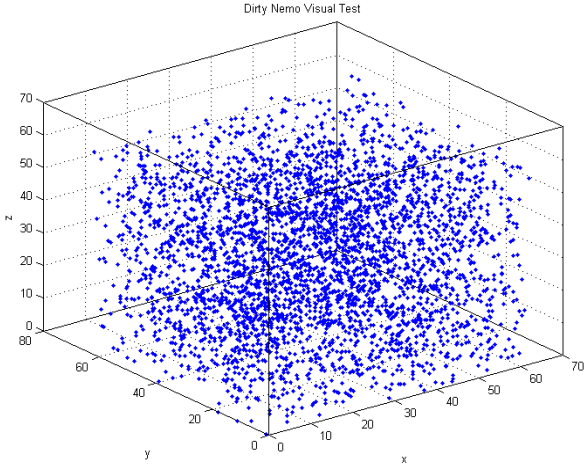
A visual test can be used to find irregularities in the distribution. This is visualized as clusters as some PRNGs seem to put points together in clusters. Periodical irregularities can also be spotted such as 3 dimensional planes. This is e.g. known to be produced by the RANDU generator. This effect is known as the Marsaglia effect, see Appendix 4. By examining two graphs close together they can be compared by looking for darker areas.

Because Dirty Nemo only produces half the input amount, the graphs corresponding to Dirty Nemo have half the amount of samples compared to the other tests.

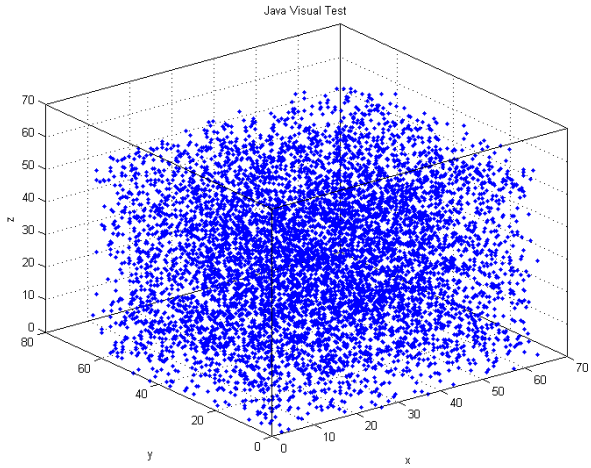
The 3 dimensional graphs are created by simply dividing an array of numbers ranging from 0-63 into three equally large arrays. The three smaller arrays then represent x-, y- and z coordinates.



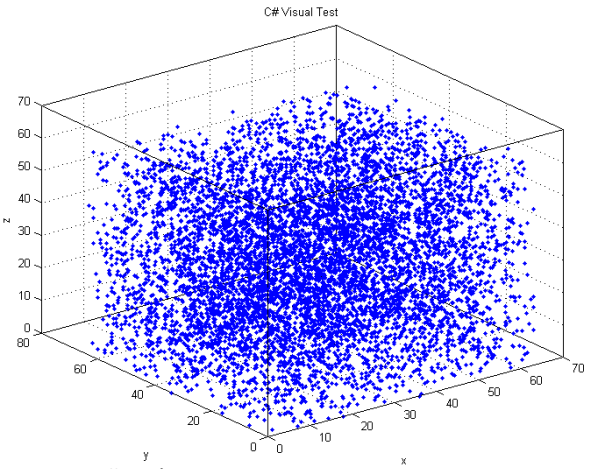
Clean Nemo Visual Test



Dirty Nemo Visual Test



Java Visual Test

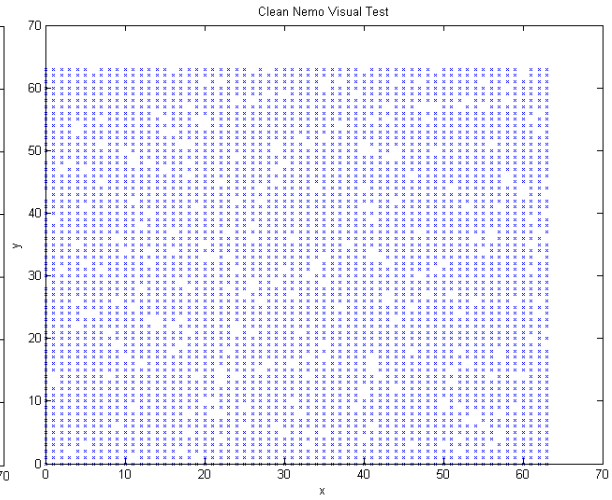


C# Visual Test

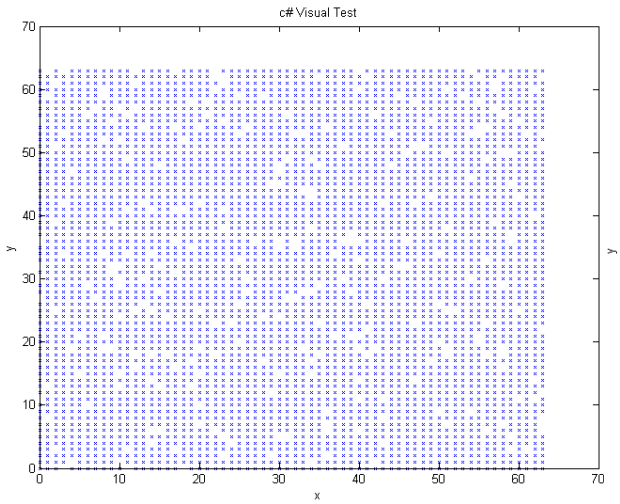
All of the four candidates passed the visual 3 dimensional test. The graphs where searched by rotating the graphs around the different axes. We tried to find patterns by searching for planes, clusters and other irregularities, but no obvious flaws where found.

Two-dimensional graphs where created as well to be able to do a two-dimensional visual test. This was done by dividing an array of numbers ranging from 0-63 into two equally large arrays. These arrays then represent x- and y coordinates in the graph. Again Dirty Nemo has half the amount of samples compared to the other RNGs.

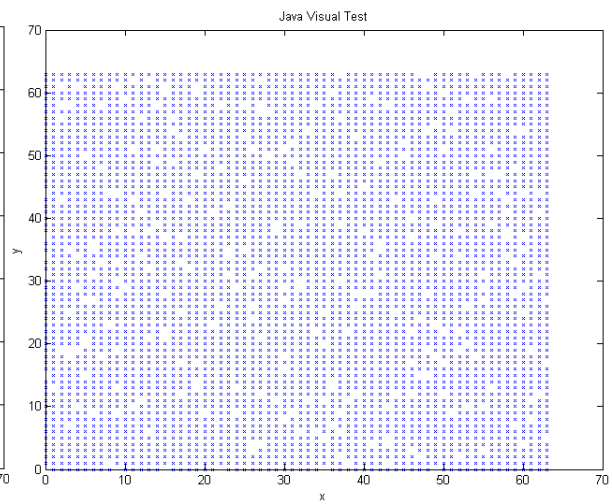
Clean Nemo Visual Test



Dirty Nemo Visual Test



c# Visual Test



Java Visual Test

All though all graphs have some cluster formations there is no obvious pattern. Thus all candidates passed the visual two-dimensional test.

6.5.2 Chi-2 test

This type of test is a method for analyse of frequency tables, the method means testing the actual values against the expected values. At first we formulate a hypothesis, this hypothesis is then the point we will compare the result against.

In our experiment we have 4 types of different test data. They are Java, c#, dirty and clean Nemo. Clean Nemo is random numbers based on an aquarium as physical source. Dirty Nemo is the same source but the numbers is manipulated afterward by a mixing function.

Chi-2 values for binary frequency table

Random Generator	Number of samples	Chi-2 value
Clean Nemo	131903	≈0.1223
Dirty Nemo	65920	≈ 0.9481
Java	131903	≈0.2165
c#	131903	≈14.71

This test is based upon a binary frequency table. This table holds the number of ones and zeros in the bit sequence. A chi-2 value is then calculated as seen in the table above.

Because of the Dirty Nemo algorithm the number of samples is more than halved.

We have one degree of freedom and to pass this test the chi-2 value has to be equal to or less than 3.84, as shown in the chi-2 table in appendix 3.

As shown in the table of chi-2 values above Clean Nemo, Dirty Nemo and Java passed this test. c# however with a chi-2 value of 14.71 did not.
The lowest and best value had Clean Nemo. This was rather a surprise since we thought that the MD5 mixing function in Dirty Nemo should make it better than Clean Nemo. The fact that Dirty Nemo has approximately half the amount of samples might have something to do with the higher value.

One can always argue that more samples should have been used in all tests, and this might be the case why c# had such a poor result.

A chi-2 calculation example can be found in Appendix 2 and chi-2 values in Appendix 3.

Chi-2 values for 6 bits frequency table

Random Generator	Number of samples	Chi-2 value
Clean Nemo	21983	≈0.3944
Dirty Nemo	10986	≈ 0.0303
Java	21983	≈0.5210
c#	21983	≈0.7617

We could not find a chi-2 table with 63 degrees of freedom however all values seems to be fine. Dirty Nemo is the one that stands out the most with a chi-2 value of 0.0303. The reason

for Dirty Nemo's outstanding performance is probably because it uses the MD5 mixing function.

A chi-2 calculation example can be found in Appendix 2.

6.5.3 n-block test

In this test a sequence of bits are divided into equally large blocks. The number of bits in each block is referred to as block size. Block size is denoted as n $\{r_1, r_2, r_n\}$.
In each block the majority of bit values are then calculated, if there are more ones than zeros we choose $y_1 = 1$ and vice versa, $y_1 = 0$. This procedure is repeated N times and then a chi square test with one degree of freedom are made on the variables y_i.

To past the chi-2 test the value has to be below 3.84.

Chi-2 values for the n-block test.

Random Generator	Block size(N)	Number of Blocks(n)	Chi-2 value
Clean Nemo	**30**	**4396**	**≈99.69**
Dirty Nemo	**30**	**2197**	**≈61.3**
Java	**30**	**4396**	**≈105.19**
c#	**30**	**4396**	**≈69.31**

As with the normal chi-2 test above once again a chi-2 value of 3.84 or less with one degree of freedom is the required limit to pass the test.

All the four candidates' results are way over the accepted value to pass the test and of course none passed it.

The random generator used by Java had the worst result in this test. This is however quite meaningless since all the values are so extremely poor. This shows that there could be some sort of pattern in all of the four generators.

The answer could also be that the block size is too small and therefore giving incorrect results.

Chi-2 values for the n-block test.

Random Generator	Block size(N)	Number of Blocks(n)	Chi-2 value
Clean Nemo	**60**	**2198**	**≈28.44**
Dirty Nemo	**60**	**1098**	**≈15.39**
Java	**60**	**2198**	**≈32.19**
c#	**60**	**2198**	**≈23.24**

Once again the requirement to pass the test is a chi-2 value of 3.84 given one degree of freedom.

Neither this n-block test was passed by any generator. The values were not quite as bad as with block size 30. However they are all way above 3.84.

The best result had Dirty Nemo with a chi-2 value of 15.39 showing that this had fewer tendencies towards a pattern in the sequence.

The size of the blocks in the n-block test is depending on the size of the patterns we want to find. A bigger block is not that sensitive to small patterns and a bigger block could not always detect smaller patterns.

6.6 Evaluation

Our objective was to create a competitive RNG using a physical process as input. This goal was achieved by taking pictures of an aquarium and its fishes. We have thereby proven that it is possible to create a RNG by using some what simple means.

We also showed that the random number sequences that were created was competitive when compared to Java's and c#'s PRNGs. This was done by using three different testing methods. The three methods chosen was chi-2, n-block and visual test which all are generally accepted as testing methods.

7 Future improvement

Even tough we where satisfied with the experiment and our algorithms, some improvement is still possible. Mixing functions other than MD5 can be used e.g. SHS. One could also elaborate with a higher number of input bits to the MD5 algortihm.

Since our method produces quite few bits per time cycle we came up with a new idea for an algorithm. Because of the lack of time it was never implemented. The idea was to let each pixel in an image represent a value. An example is to use each pixel's hue value. All hue values from an image are then used as input to the MD5 algorithm which in turn produces 128 bits of output. This way we could produce 128 bits per image and second in contrast to our current rate which is 1 bit per image and second.

Another improvement could be that the experiment environment would be made cleaner. E.g. removing light reflections in the aquarium glass or by only have fishes of a single colour thus making them easier to trace. However, it might be that these "improvements" could have generated poorer results. Since we did not get rid of light reflections and such we actually widen our physical environment beyond the aquarium. We can not say which of these two approaches that would have been better. The same goes for using a higher resolution camera or adding more fishes.

8 Conclusion

Our goals with this thesis were to create a random number generator with a physical process as input. The reason for this was because of the large amount of problems related to PRNGs and the great need for RNGs in the area of cryptography and information security.

We succeeded with the task of creating the RNG with non expensive equipment such as an aquarium and a web cam. We where hoping to generate high quality random numbers at an efficient rate. However we had some problems with the efficiency because we could only generate one bit per second. Never the less, the quality of the generated values was over our expectations and very satisfying. During our different test methods we compared our values against Java's and c#'s pseudo random number generators with good results. Both Clean and Dirty Nemo gave overall better results than both Java and c# in our specific tests.

However testing random values is a complex matter. It is easy to verify that a sequence is not random, but it is hard to prove that the sequence is random. The only thing one can do by testing a sequence is to show that it could be random. But if patterns are found in any test this verifies that the sequence is not random.

By working with this thesis we have increased our interest as well as knowledge in many different fields. We learned that producing good random numbers is harder than we first thought. Testing the quality of the numbers is even more challenging. The main reason for this is that there exist at least as many tests as there are RNGs.

Another interesting and purely philosophical idea we came across says that nothing in the universe is random. This is based on the idea that nothing happens by chance. It is rather that some environments such as atoms movement and behaviour are probably far too complex to predict. This means that if we can map all the quantum particles in the whole universe and simulate their movement one could actually predict the future. Of course this is today far beyond both our knowledge of quantum mechanics and the strength of our computers and will probably never happen.

9 Glossary

RNG – Random number generator
Pseudo random generators (PRNG) – Random generators that have a seed as a start value for number generation. They are built upon a mathematical formula.
Seed – A value that is used to start the pseudo random generator.
A physical process – A delimited physical environment which can have the complexity needed for random number generation.
RSA - RSA is an asymmetric algorithm for public key cryptography, widely used in electronic commerce. The algorithm was described in 1977 by Ron Rivest, Adi Shamir and Len Adleman; the letters RSA are the initials of their surnames. ‘
Cipher Text - Data that has been encrypted. Cipher text is unreadable until it has been converted into plain text (decrypted) with a key.
RBG Value - The RGB color model is an additive color model in which red, green, and blue light are combined in various ways to create other colors. The very idea for the model itself and the abbreviation "RGB" come from the three primary colors in additive light models.

10

References

[1] J.M. Thijssen, *Computaional Physics*, University Press, Cambridge (2001).

[2] I. Vattulainen, T. Ala-Nissala and K. Kankaala *Phys. Rev. Lett. 69*, 3382 (1992).

[3] G. Marsaglia, *A Current View of Random Number Generators* (08/06/2004)
http://stat.fsu.edu/~geo/diehard.html

[4]M. Matsumoto, T. Nishimura, *Mersenne Twister Homepage* (08/06/2004)
http://www.math.keio.ac.jp/~matumoto/emt.html

[5] K. Entacher, *Classical LCGs*, (08/06/2004)
http://crypto.mat.sbg.ac.at/results/karl/server/node4.html

[6] http://en.wikipedia.org/wiki/Symmetric_key_algorithm

[7] http://en.wikipedia.org/wiki/Asymmetric_key_algorithm

11

Appendix

11.1 Appendix 1 – Internal clock lagging

When the following Java code is executed the random numbers is repeated. There could be some simple explanations fore this; either that the computer is faster then the sensitivity that the random generator could recognize time variables; another explanation is that the time class in Java is lagging.

Java Code

```
import java.util.Random;

public class Seed
{
    public static void main(String args[])
    {
        for(int i = 0; i < 100; i++)
        {
            Random myRandom = new Random();
            System.out.println(myRandom.nextFloat());
        }
    }
}
```

Result

0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.90553856	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719
0.90553856	0.9069719	0.9069719

11.2 Appendix 2 – chi-2 example

Observed Values (O)

0	n
1	m

Formula

$$x^2 = \sum \frac{(O-E)^2}{E}$$

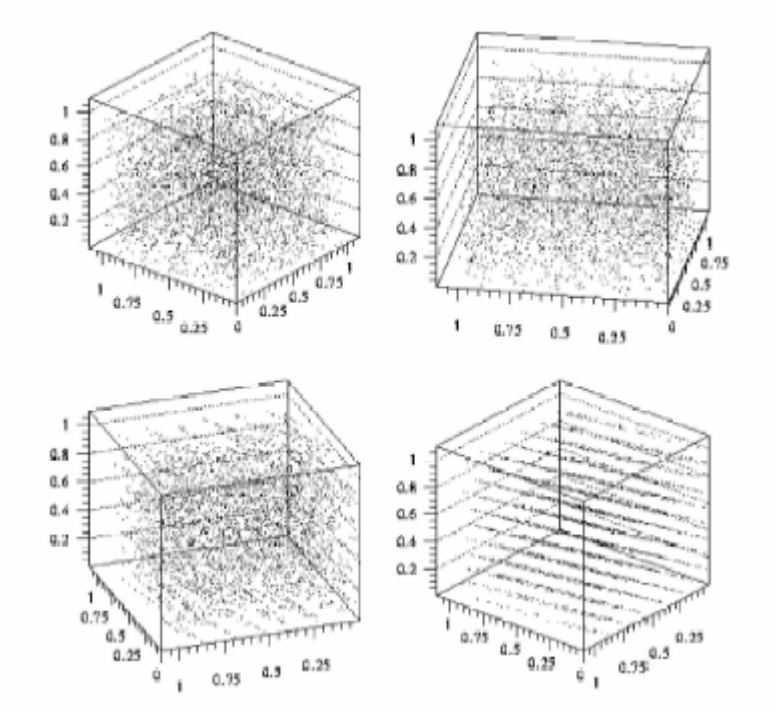
Expected Value (E), E = (n+m) / 2 = E

$$x^2 = \left(\frac{(N-E)^2}{E}\right) + \left(\frac{(M-E)^2}{E}\right) = x^2$$

11.3 Appendix 3 – chi-2 table

Degrees of Freedom	99%	95%	90%	70%	50%	30%	10%	5%	1%
1	0.00016	0.0039	0.016	0.15	0.46	1.07	2.71	3.84	6.64
2	0.020	0.10	0.21	0.71	1.39	2.41	4.60	5.99	9.21
3	0.12	0.35	0.58	1.42	2.37	3.67	6.25	7.82	11.34
4	0.30	0.71	1.06	2.20	3.36	4.88	7.78	9.49	15.09

11.4 Appendix 4 – RANDU Marsaglia effect



The 3D-plots on the left are created by RANDU's pseudo random generator. They all show the same random numbers, seen from four different angles. As seen in the bottom right graph RANDU creates several planes in 3D, thus creating highly predictable numbers. "This is called the Marsaglia effect, after M. Marsaglia who published the paper Random Numbers Fall Mainly in the Planes in 1968."[3]

Pictures taken from http://www.andrew.cmu.edu/course/21-420/lect/Section1_6.pdf