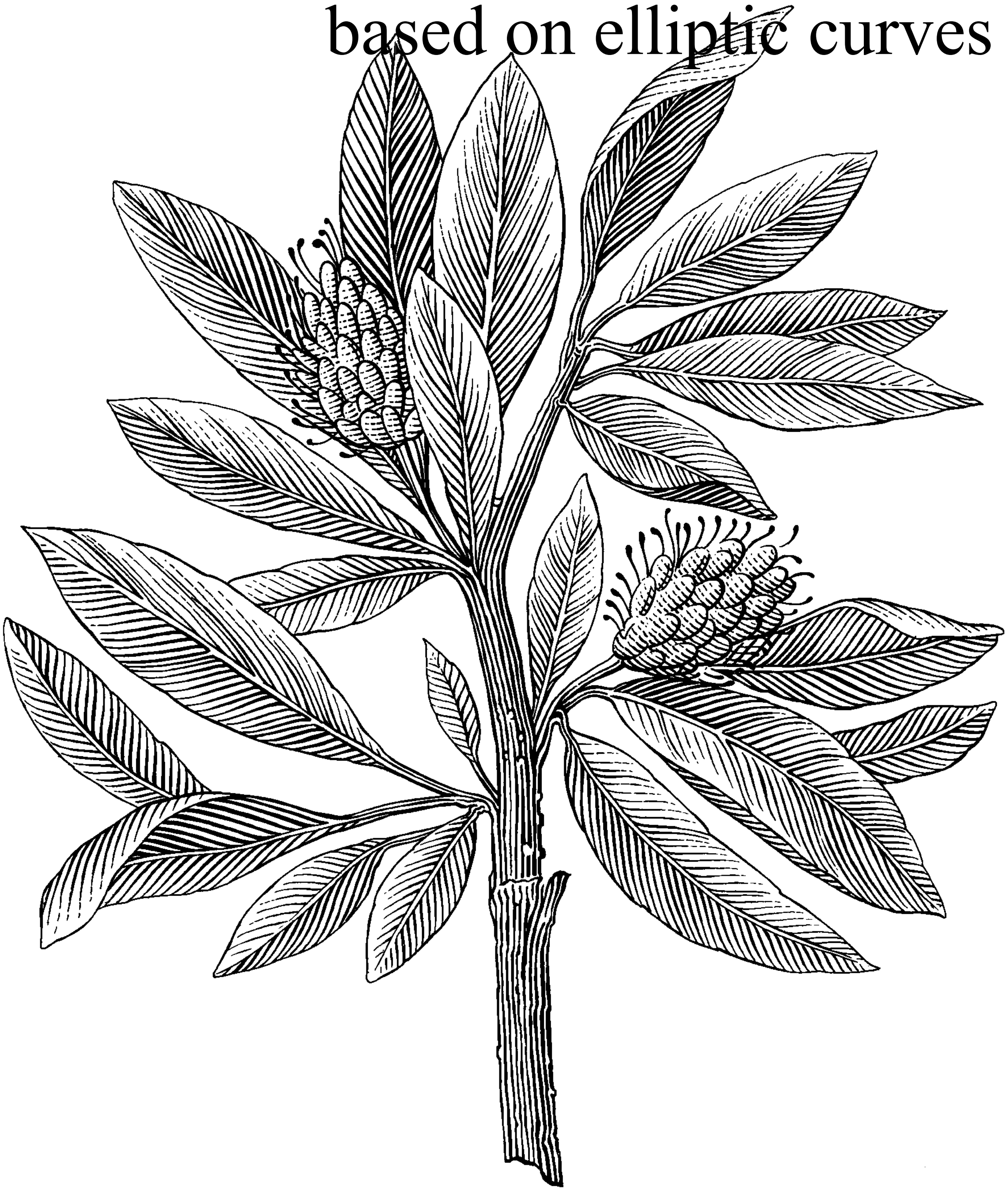
Linnæus University
Sweden

Degree project

Empirical testing of pseudo random number generators based on elliptic curves



Abstract

An introduction on random numbers, their history and applications is given, along with explanations of different methods currently used to generate them. Such generators can be of different kinds, and in particular they can be based on physical systems or algorithmic procedures. The latter type of procedures gives rise to pseudo-random number generators. Specifically, several such generators which are based on elliptic curves are examined. Therefore, in order to ease understanding, a basic primer on elliptic curves over fields and the operations arising from their group structure is also provided. Empirical tests to verify randomness of generated sequences are then considered. Afterwards, there are some statistical considerations and observations about theoretical properties of the generators at hand, useful in order to use them optimally. Finally, several randomly generated curves are created and used to produce pseudo-random sequences which are then tested by means of the previously described generators. In the end, an analysis of the results is attempted and some final considerations are made.

Keywords: elliptic curves, cryptography, pseudo random, number generation, PRNG, TRNG, linear congruential generator, power generator, Naor-Reingold generator, empirical testing, frequency test, serial test, run test, poker test, autocorrelation test

Acknowledgements

I would like to thank in particular Per-Anders Svensson for advice while choosing the topic for this thesis and the assistance throughout; Karl-Olof Lindahl for the useful lectures on the thesis process; and finally my family for supporting me during my studies.

Contents

Chapter 1

Introduction

DILBERT By Scott Adams
TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

© Dilbert, "Random Number Generator", October 25, 2001

The history of number generation is full of interesting anecdotes and attempts. Nowadays, random numbers are widely used, yet their true nature remains hard to define. Is randomness an intrinsic property that is naturally occurring, is it merely a mathematical concept tailored to suit our needs? The line between those possibilities is quite fine.

For a long time in history the random number generators of choices were simple tools such as—hopefully fair—dice, coins, and cards, used especially for betting games, dating back to several centuries ago. For gambling purposes, wheels were also used which eventually evolved in the game of roulette [2].

During the early twentieth century, it was common to use tables containing random numbers derived from data sources such as census or logarithmic tables [13]. Around 1945, the RAND Corporation produced a table with a large amount of random numbers picked by a machine and an operator together. In 1957, a machine called ERNIE 1 was first used to generate winning lottery numbers. It was the first machine capable of quick random number generation, and picked the curiosity of the public. Using noise generated by neon tubes, it was an example of a hardware random number generator—and quite a large machine. ERNIE 1 had several successors, each smaller and more efficient [3,6,14].

When Intel's Ivy Bridge processors came out in 2012, an instruction rdrand() was added in order to produce random numbers from a hardware entropic random source [4].

But such slower sources were not, and are not, practical in all situations. With the advent of computers, the matter of generating random numbers by means of algorithms became of interest. The reasons were obvious: fast generation with minimal time and resource expenditure would be of advantage. To be fair, algorithmic generation was suggested earlier, even if not particularly pursued. A monk known as Brother Edvin first proposed, around the thirteenth century, a method which would be revised and proposed by Von Neumann in 1951. It was called the middle square method. Starting from a four digit number, it would be squared, and the four middle digits would be subsequently squared. It was an obviously flawed method, but nonetheless it was a beginning [4].

Subsequently, more algorithms were invented. For instance, a very simple one is the linear congruential generator [2], a variation of which we consider in this work. Another example is the Blum Blum Shub generator [2], today well known as a cryptographically secure method. This kind of random number generators are quite interesting, because they can be studied both theoretically and empirically. In recent years, the necessity to have effective generators for several different uses had mathematicians and computer scientists try to find new methods to add to an already large pool of algorithms. In this work, we examine some of these in more detail. The generators we consider are all based on elliptic curves. These are algebraic curves with the property that it is possible to determine group operations that, along with the curve points, form an Abelian group. The security of elliptic curves relies on the difficulty of solving the discrete logarithm problem within them. So far, no method taking less that exponential time has been found [1,10]—it is thought to be a hard problem. There are a fair amount of generators of this kind, and some descriptions can be found for example in [16]. A particularly infamous generator is the *Dual EC-DRBG*, which is now known to have relevant security weaknesses [5]. In this paper, we look at a few of the simplest such generators, in order to study them more accurately, namely, the linear congruential, power, and Naor-Reingold generator, analysing and testing them.

1.1 Motivation and aim

Initially, the thesis topic was determined to be relating to cryptography, because of personal interest. By narrowing down the choice of topics, random generators based on elliptic curves were chosen as a basis because of being related to some of the most recent developments in the field of cryptography. It was decided that only a few of this kind of generators would be included in this work, which was so that the focus could be on thorough testing.

The initial conjecture was that it would have been possible to show that elliptic curves can be used to generate good random sequences. There were several limitations that are described in later sections, as of course it would not have been possible to test all curves and generators based on those, and apply all existing tests to them.

Although all the necessary notions—overviews of random number generation, testing algorithms, and elliptic curves—were researched beforehand, most of this work is original and centred on implementation and programming meaning not many references are given. However, a large part of the code is included in the appendix available for consultation and reuse.

1.2 Report outline

Chapter 2 deals with several notions that are essential to understand the remainder of this text. These include an overview on what random numbers are and how they are used, as well as several methods to produce them, in particular, those based on the use of elliptic curves. Such curves are also explained briefly. We then go over a few statistical randomness tests which are needed later on. Chapter 3 contains detailed explanations of the methods and implementation details for all the features we require, namely, the random number generation and testing procedures. Theoretical considerations are also included. The contents of Chapter 4 are mainly tables containing raw data obtained from running the implemented procedures. Finally, we investigate our results in Chapter 5, and make a few final considerations before concluding the report.

Chapter 2

Preliminaries

We present in the following sections the key concepts that form the basis of this report. Random numbers, generators and testing are covered somewhat broadly, before providing more focus. For context, several properties of randomness are examined, not only the specific ones we aim to test later. Our subsequent investigation only concerns several select random number generators to which we apply just a handful of carefully chosen tests, as described. Some of the mentioned notions are further investigated and developed in later sections, as needed. Elliptic curves are also covered briefly, as they form the basis for the generators we consider.

2.1 Random numbers

Randomness, on the surface, seems an easy enough concept to grasp, but when one digs deeper, it is exceptionally hard to define. As an experiment, try to ask anyone to come up with a random number, or even an entire sequence of numbers. Chances are, what you get won't be very random. People tend to think of certain numbers more frequently, for instance. If asked to come up with a random sequence, most people will think long and hard in order to make the result appear random. Humans have a hard time thinking without patterns, whether they're trying to avoid or find them—see for instance [12].

For this reason, determining the nature of randomness has been an important as well as interesting problem, mathematically and philosophically, and because of the vast amount of possible applications. In a general way, we can define random numbers as lacking predictability and intelligible patterns, especially in the case that what we are seeking for is a sequence, not just a single number. In the next section, we examine the concept in more detail, concentrating on the properties of random sequences, as we are going to investigate those in this work.

2.1.1 Criteria for randomness

Before we go on to define several useful criteria for measuring randomness, let us consider a few sequences as an example.

1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
0 0 0 0 0 0 0 0 0 0 0 0 0
0.15 0.23 0.09 0.49 0.21 0.57 0.11

For sure, upon a quick examination, these sequences look quite non-random because of the presence of obvious patterns. The first sequence is a repeating cycle of a short monotonous subsequence; the second is all zeros; and the last one, the values go up and down in an alternating way. But not all patterns are easy to identify quickly. Also, what if each of those sequences were part of a longer sequence? There is a difference between *local* and *global* non-randomness. We expect to have low non-randomness in general, but if a sequence is long enough, even the most seemingly non-random subsequences will begin to show up—after all, they are as likely as any other subsequence, and therefore are to be expected.

Another characteristic that a sequence should possess is that the occurrences of each different number in it should be uniformly distributed. It would be suspect if a number in a sequence occurred many more times than any other number. In a similar way, we would also like each possible subsequence of a certain length to appear roughly the same amount of times. We also want a sequence not to contain discernible repeating patterns.

Albeit these are the properties that are most easily applied to binary sequences, which interest us the most, there are other kinds of sequence as well, which may be made up of a larger range of integers, may allow repetition or not, or may be sequences of real numbers, most commonly normalised in an interval from zero to one. For the sake of exhaustiveness, we mention a few properties that are relevant in those cases as well.

When there are many possible numbers in a sequence, we may be interested in the spacings between the occurrences of certain numbers or numbers lying in a certain range, as well as the occurrence patterns of monotone subsequences. We may also examine the subsequences to see how many different integers are in them, or how many consecutive numbers in a sequence it takes to find a complete set of integers in a given range. The ordering of numbers in subsequences of a certain length may also be of interest. There are more formal measurements of randomness that we do not examine here, but the interested reader can see [1] for a starting point.

As we have seen, there are many desirable properties when it comes to generating a sequence. These give rise to several randomness tests, some of which we include in Section 2.4.

2.1.2 Applications

There are many variegate applications of random numbers. In cryptography, they are widely used for generating encryption keys, or parameters for certain cryptosystems. This gives rise to the distinction between sources of random numbers that are *cryptographically secure* versus those who aren't.

Random numbers are also useful in conjunction with different kinds of mathematical models, for instance when running simulations and physical research, when such numbers are needed in order to keep the model as realistic as possible. Random data sampling from given sets is also an important application which is closely related to simulations. Samples generated in this way can provide more insight about "typical" behaviours of certain systems. Another mathematical field which uses random numbers is numerical analysis, where they have been successfully used to solve a variety of problems.

Computer programming is also an area where the use of random numbers is profitable, as they are useful for testing the functionality of programs where many kinds of input could be provided [7]. Other applications are games and graphics, think for example of dice rolling, particle systems. Other fields that use random numbers are for example decision making, and generative art and music. For sure, many different fields can benefit from randomness in some way and the kind of problem one is attempting to solve impacts the nature of the random numbers we seek.

2.2 Elliptic curves

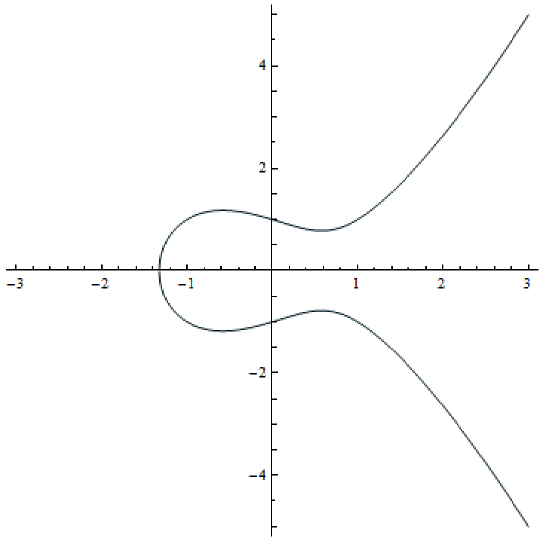
We now switch our focus for a moment from random numbers to elliptic curves. We do so as knowledge of these is necessary in order to fully understand the random number generators that are considered in this work.

Elliptic curves are curves which have the structure of an Abelian group when taken over a field K. They are of the form

$$E(K) : y^2 = x^3 + ax + b,$$

where all coordinates are pairs belonging to the field K. This is called the *Weierstrass equation.* It should be noted that this is a particular form of the curve equation the use of which should be avoided if the characteristic of the field K over which the curve is defined is equal to 2 or 3. However, we do not have to worry about this due to the fields we use. It is required that a and b are chosen in such a way that the curve is non singular, that is, $4a^3 + 27b^2 \neq 0$, computed in the field K. This ensures that the curve has no self intersections, cusps, or isolated points.

Elliptic curves can be over various fields, but for our purposes only the finite field $\mathbb{F}_p$ for different values of p is used.



The elliptic curve $y^2 = x^3 - x + 1$

An elliptic curve is always symmetric along an horizontal line—the x-axis, if $K = \mathbb{R}$—, as it is easy to see from the structure of the equation. Because of this, the reflection of any point $P \in E(K)$ also belongs to the curve, and we denote it with P'.

We include an additional point in the curve, the *point at infinity* O, which is defined to be infinitely far on every vertical line. We can now discuss several properties of the curve, when $K = \mathbb{R}$.

Addition. To add two points P and Q, we trace the line which connects them. Such a line intersects the curve at another point of the curve, R'. We define then $P \oplus Q = R$, such that R is the reflection of R' along the line of symmetry of the curve. Note that if $P = Q$, the line we look for is simply the tangent through the point; then we proceed normally. Because the line between P and Q is the same as the line between Q and P, addition is commutative. If $P = Q'$, the connecting line is vertical and then $R = R' = O$, the point at infinity.

We now develop explicit formulas for adding points on an elliptic curve, based on these geometrical considerations. We add $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ to obtain $R = (x_3, -y_3)$. The first step is to find the line through P and Q, which we assume here to be different points, so we find its slope which is defined as . Once we have found the slope, we can find an explicit expression for the line: $y - y_1 = m(x - x_1)$. Because of the definition given earlier, we know that this line passes through the reflection of the sum of the points. We can manipulate the expression as follows

$$y = m(x - x_1) + y_1 \iff y^2 = m^2(x - x_1)^2 + y_1^2 + 2my_1(x - x_1),$$

and then we can equate it to the elliptic curve formula, such that

$$\begin{aligned} x^3 + ax + b &= m^2(x - x_1)^2 + y_1^2 + 2my_1(x - x_1) \\ \iff x^3 + ax + b - m^2(x - x_1)^2 - y_1^2 - 2my_1(x - x_1) &= 0, \end{aligned} \tag{2.1}$$

and since it is of third degree, this polynomial has three roots, and can be rewritten as

$$\begin{aligned} (x - x_1)(x - x_2)(x - x_3) &= 0 \\ \iff x^3 - (x_1 + x_2 + x_3)x^2 + (x_1x_2 + x_2x_3 + x_1x_3)x - x_1x_2x_3 &= 0. \end{aligned} \tag{2.2}$$

Now we can equate the coefficients in (2.1) and (2.2) to obtain $m^2 = -(x_1 + x_2 + x_3)$, which gives

$$x_3 = m^2 - x_1 - x_2.$$

Moreover, we know that $y_3 - y_1 = m(x_3 - x_1)$ so that

$$y_3 = m(x_3 - x_1) + y_1.$$

In the case where $P = Q$, the formulas are derived similarly and then we have that

$$\begin{aligned} m &= \frac{3x_1^2 + a}{2y_1} \\ x_3 &= m^2 - 2x_1 \\ y_3 &= m(x_3 - x_1) + y_1, \end{aligned}$$

so that in the end we have

$$\begin{aligned} R' &= (x_3, y_3), \\ R &= (x_3, -y_3). \end{aligned}$$

Multiplication. Since for some applications it is necessary to perform addition of a point to itself several times, a fast multiplication algorithm can be helpful. Note on the other hand that points in general cannot be multiplied by each other. A known algorithm, which we use in this work, is the double and add algorithm. Say, for instance, that $Q = kP$. Then we need to find the binary expansion of k, that is express the number as a sum of powers of two, so that $k = \sum_{i=0}^{r} c_i \cdot 2^i$. Here, c_i is a coefficient which can be either one or zero. Afterwards, we produce a list of $Q_0, Q_1, \ldots, Q_r$ in an iterative way, and add them to obtain Q, as follows:

```
Q_0 := P
Q := 0
for i from 1 to r do
        Q_i := 2Q_(i-1)
        if c_i = 1 do
                Q := Q + Q_i
Return Q
```

Identity and inverse. When we attempt to compute $P \oplus P'$, we run into the problem of not having a third intersection point on the curve. This is when the point at infinity comes into play. We define $P \oplus P' = O$. Then, in group terms, P' is the inverse of P, and O is the identity element. We also have $P \oplus O = P$.

As already mentioned, these geometrical explanations only apply when the field over which we take the curve is $\mathbb{R}$; on the other hand, since we from here on use a prime field $\mathbb{F}_p$, it is worth giving a geometrical explanation for this case as well, which is, anyway, analogous to the real case. When taken over a prime field, the elliptic curve does not look like a curve any more. This is because we take only points with integer coordinates, modulo p. Addition follows the same rules explained for the real field, adapted to this case, keeping in mind to perform operations modulo p.

Finding and counting points. When we consider a curve over a finite field, there are only a finite amount of points within it. It may be useful to determine all points on an elliptic curve. To do this, we simply input all possible values of x, from 0 to $p-1$, in the curve equation $y^2 = x^3 + ax + b$. If $x^3 + ax + b$ is a quadratic residue modulo p for a certain x, it means it is possible to take its square root modulo p, and therefore we obtain two solutions to the equation, and two points of the curve. On the other hand, if it is not a quadratic residue, we obtain no new points for that value of x. The formula $\left(\frac{x}{p}\right) \equiv x^{(p-1)/2} \mod p$, called Legendre symbol, has value 1 if x is a quadratic residue, -1 if it is not, and 0 in the case when $x^3 + ax + b \mod p \equiv 0$. Therefore, we know that the expression $1 + \left(\frac{x^3+ax+b}{p}\right)$ has value 2 if there are two solutions to the curve equation, 1 is there is only one solution and the line through x is vertical, and 0 otherwise. Therefore, if we sum this value as calculated for each $x \in \mathbb{F}_p$, and add the point at infinity, we obtain the total number of points on the curve:

$$\begin{aligned} S &= 1 + \sum_{x \in \mathbb{F}_p} 1 + \left(\frac{x^3 + ax + b}{p}\right) \\ &= 1 + p + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3 + ax + b}{p}\right). \end{aligned}$$

Having defined group properties and with the additional consideration that addition is associative—although we omit the proof here— we see that the set of points on the elliptic curve together with the binary operation give rise to an

Abelian group. The examples that follow illustrate the properties discussed so far.

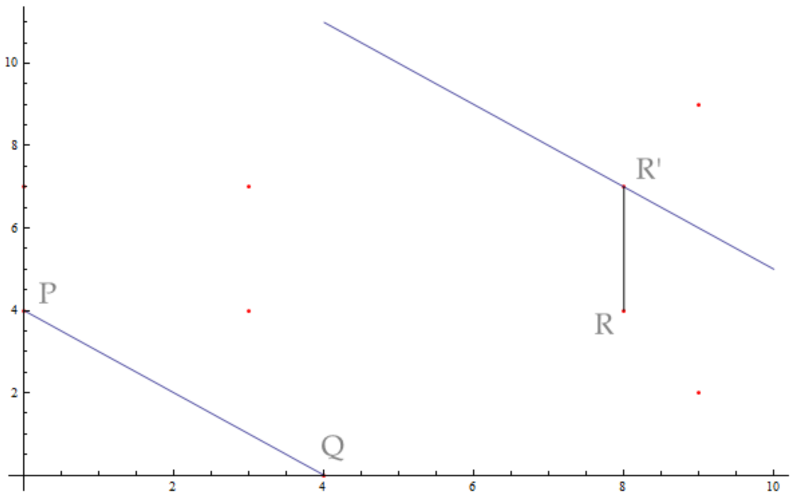
Example 1. Let us take the curve $E(\mathbb{F}_{11}) : y^2 = x^3 + 2x + 5$ which includes as an example the points $P = (x_1, y_1) = (0, 4)$ and $Q = (x_2, y_2) = (4, 0)$ as is readily verified by inputting the values into the equation. We try to find $R = P \oplus Q = (x_3, -y_3)$. To begin with, we have that $m = \frac{-4}{4} = 1$. We can then find the coordinates of R as follows

$$x_3 = (-1)^2 - 0 - 4 = -3 \equiv 8 \mod 11$$
$$y_3 = -1(8 - 0) + 4 = -4 \equiv 7 \mod 11,$$

which gives then

$$R' = (8, 7),$$
$$R = P \oplus Q = (8, 4).$$

The graph below shows graphically the steps to compute the sum. The red dots represent all points in the curve.



Example 2. Say we have the curve $E(\mathbb{F}_{13}) : y^3 = x^3 + 3x + 7$ and we know it contains the point $P = (3, 2)$. We try to find $Q = 11P$.

Expanding 11, we have that $11 = 2^0 + 2^1 + 2^3$. So, $c_0 = c_1 = c_3 = 1$, and $c_2 = 0$. We compute Q_0 through Q_3—addition is left to the reader this time around—as follows:

$$Q_0 = (3, 2)$$
$$Q_1 = (3, 2) + (3, 2) = (8, 6)$$
$$Q_2 = (8, 6) + (8, 6) = (10, 7)$$
$$Q_3 = (10, 7) + (10, 7) = (9, 10),$$

and then we have that

$$11P = (3,2) + (8,6) + (9,10) = (12,9) + (9,10) = (8,7).$$

Example 3. We wish to look at the curve $E(\mathbb{F}_5) : y^2 = x^3 + 2x + 4$. We check that if it is non singular first and see that $4(2^3) + 27(4^2) = 464 \not\equiv 0 \mod 5$. Then we input the values to calculate the points and obtain the following points:

x	y^2	y
0	4	2 and 3
1	2	no roots
2	1	1 and 4
3	2	no roots
4	1	1 and 4

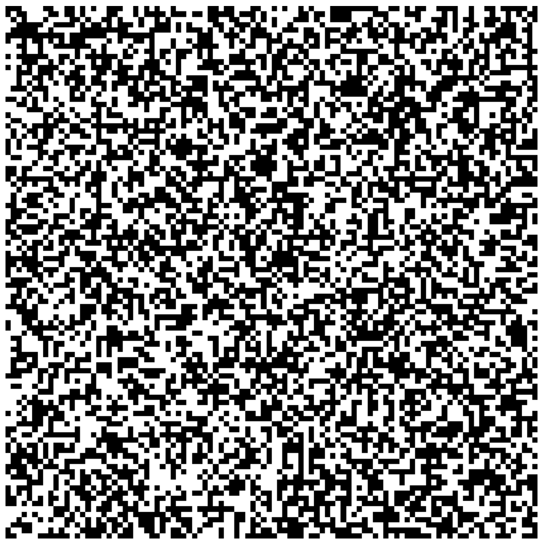
The curve contains the points $\{O, (0,2), (0,3), (2,1), (2,4), (4,1), (4,4)\}$.

2.3 Random number generation

As we have seen, finding a way to generate random numbers effectively has been a very relevant problem in the past century. One could say that there are as many ways of generating random numbers as there are applications. Some are slower, some are faster; some generate longer sequences, some generate individual numbers. It is important to choose the optimal method in each case. There are two classes of generators, that differ when it comes to predictability.

2.3.1 Non-deterministic

One way of generating random numbers is by using *non-deterministic* generators. In this case, the sequence is not predictable, and no assumptions can be made on its structure until it is generated. The resulting numbers are said to be *truly random*. Ways of generating numbers in a non-deterministic way are, for instance, making use of atmospheric noise, electromagnetic and quantum phenomena, and so forth. These methods have the disadvantage of being very slow, and therefore suitable only for applications where a limited amount of numbers is needed, such as lottery drawings, for instance [3,15]. We may be tempted to include items such as coins and fair dice as true random number generators, however, such macroscopic phenomena are only unpredictable because we can't determine the physical starting conditions and forces acting on them with sufficient accuracy. Moreover, the possible outcomes are very limited, which differs from the other true generators proposed above. If one were able to measure all physical data with high accuracy, then these tools would produce fully predictable outcome. In contrast, quantum phenomena are completely unpredictable and therefore make for the "truest" possible generators.



This is what a true random sequence looks like. Zeros are white; ones are black. It is a sequence of length 10000, created with a dedicated sequence generator available on RANDOM.org. No evident patterns are visible.

2.3.2 Deterministic

When on the other hand a generator is *deterministic*, it means that the numbers are generated in an algorithmic, iterative way. We define a sequence generated in this way to be *pseudo random*. When the same generator is run more than once on the same parameters, the same sequence results. What this implies is that it is always possible to predict what the following number will be, provided one knows the workings of the generator. In other words, each number of the sequence is uniquely determined, given a seed—that we should find by means of a truly random generator—and the other possible parameters. Such methods have the advantage of being somewhat fast, allowing to generate long sequences, and it is possible to optimize the output by making several considerations beforehand. A downside is that these generators inevitably produce cycles. Usually though, the numbers do not repeat within a single cycle, which may be useful for some applications. The generators that follow are all deterministic and based on elliptic curves.

The following table summarises the differences between pseudo random and true random generators.

Non-deterministic (TRNG)	**Deterministic (PRNG)**
Generally slow	Generally fast
Allows repetition	Disallows repetition*
Physical source	Algorithmic source
Hard implementation	Easy implementation
Completely unpredictable	Unpredictable to uninitiated

*This is meant in the sense that since in a PRNG each number depends on the previous one, then if you see a number repeat you know the whole sequence is going to repeat. In many applications, only the first iteration of a sequence is of interest.

We will now have a look at the specific pseudo-random generators which were tested.

Linear congruential generator

The linear congruential generator (EC-LCG) is a generator which we define as the following sequence

$$U_n = G \oplus U_{n-1} = nG \oplus U_0, \qquad n = 1, 2, \dots,$$

where $G \in E(\mathbb{F}_p)$ is some given point, and $U_0 \in E(\mathbb{F}_p)$ is an initial seed.

Power generator

We define the power generator (EC-PG) as the sequence

$$W_n = eW_{n-1} = e^n G, \qquad n = 1, 2, \dots,$$

where $e \geq 2$ is an integer and $G \in E(\mathbb{F}_p)$ a point on the curve.

Naor-Reingold generator

Finally, we go over the Naor-Reingold generator (EC-NRG). The sequence is defined as

$$F_a(n) = a_1^{\nu_1} \dots a_k^{\nu_k} G, \qquad n = 1, 2, \dots, \tag{2.3}$$

where $a = (a_1, a_2, \dots, a_k)$ is a vector such that each $a_k \in \mathbb{Z} \backslash t\mathbb{Z}$, where t is the order of the point $G \in E(\mathbb{F}_p)$ and $v = (\nu_1, \nu_2, \dots, \nu_k)$ is the bit representation of n, with $0 \leq n \leq 2^k - 1$.

Note that in conjunction with these generators we use an output function in order to produce a binary sequence. The function we use takes as input the x coordinate of a point $P = (x, y)$ and if $x > \frac{p-1}{2}$ then the output value is one; otherwise it is zero. This method is appropriate for use on an entire sequence, if we assume that the points on the curve are evenly distributed on the x-axis.

Example. We demonstrate how each generator functions by taking for instance the curve $E(\mathbb{F}_{17}) : y^2 = x^3 + x + 1$. Then, suppose that $G = (6, 11)$ and $U_0 = (10, 5)$. The sequence generated by the linear congruential generator looks like this:

$$\begin{gathered} U_1 = G \oplus U_0 = (6, 11) + (10, 5) = (16, 4) \\ U_2 = G \oplus U_1 = (6, 11) + (16, 4) = (13, 16) \\ U_3 = G \oplus U_2 = (6, 11) + (13, 16) = (11, 0) \\ U_4 = G \oplus U_3 = (6, 11) + (11, 0) = (13, 1) \ldots \end{gathered}$$

continuing this way until all the sequence is generated. If we apply the power generator, and we choose $e = 4$ for instance, then the sequence is as follows:

$$\begin{gathered} W_1 = 4G = 4(6, 11) = (15, 12) \\ W_2 = 4(15, 12) = (9, 5) \\ W_3 = 4(9, 5) = (13, 1), \end{gathered}$$

and that's the end of the sequence, since $W_4 = W_0$ as easily verifiable. Finally, to try the Naor-Reingold generator, we take for example the vector $a = (2, 5)$. Then we know from the size of the vector that we should end up with $2^2 = 4$ terms in the sequence:

$$\begin{gathered} F_a(1) = 2^0 5^0 G = G = (6, 11) \\ F_a(2) = 2^0 5^1 G = 5G = (0, 1) \\ F_a(3) = 2^1 5^0 G = 2G = (9, 12) \\ F_a(4) = 2^1 5^1 G = 10G = (13, 1). \end{gathered}$$

By applying the output function defined in this section to each of the sequences above, we obtain binary sequences that look like this

$$\begin{gathered} U_{binary} = \{1, 1, 1, 1, \ldots\} \\ W_{binary} = \{1, 1, 1\} \\ F_{a_{binary}} = \{0, 0, 1, 1\}. \end{gathered}$$

For more examples and to see other generators based on elliptic curves, see [8,16].

2.4 Testing for randomness

Given that, as we have seen, randomness is a difficult concept to define, it is hard to give an absolute answer as to whether a random number generator produces good enough output. What we can do is reject the possibility of randomness, or fail to reject it—but we can never be completely sure. Essentially, there are two general ways of testing for randomness. A possible approach is to use several theoretical tests on the generators to estimate their quality, without having to generate an actual sequence first. Another possibility is to examine just the output sequences, which is known as empirical testing. Although we do make a few theoretical considerations later on in the text, the focus in this work is on empirical tests. There are many kinds of such tests, but we only consider a few ones. Five or six tests are generally considered enough, and the following are the ones we choose to implement for binary sequences. We follow the versions of the tests as described in [9], except for the frequency test which slightly differs.

2.4.1 Frequency test

This test involves verifying that zeros and ones appear in the sequence roughly the same amount of times. The statistic we use follows directly from applying the chi-square test to determine if a sample distribution follows an uniform distribution.

$$X_1 = \frac{2(n_0 - \frac{1}{2}n)^2}{n} + \frac{2(n_1 - \frac{1}{2}n)^2}{n},$$

where n_0 and n_1 represent the amount of zeros and ones respectively, and n is the length of the sequence. This statistic follows directly from applying the chi-square test to determine if a sample distribution follows an uniform distribution. Therefore it follows a χ^2 distribution with one degree of freedom.

2.4.2 Serial test

The serial test is similar to the frequency test and uses an analogous statistic. In this case, instead of just considering the amount on zeros and ones, we consider how many times each the subsequences 00, 01, 10 and 11 appear. There are versions of this test which don't let the subsequences overlap, however the test we use lets them overlap and uses a modified χ^2 statistic that takes into account the lack of complete independence,

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1,$$

which follows a χ^2 distribution with 2 degrees of freedom, when n_{00}, n_{01}, n_{10}, and n_{11} are defined in an analogous way as for the frequency test.

2.4.3 Poker test

An even more generalised version of the frequency and serial test, the poker test counts the number of occurrences of each possible subsequence of length m. In our version of this test we take the subsequences to be non overlapping, and it follows that m must divide the length n of the sequence. We take m such that

$$\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot (2^m),$$

and we define $k = \lfloor \frac{n}{m} \rfloor$. In our case, since we have $n = 1000$, good values are $m = 5$ and $k = 200$. We can say that m is the size of our "poker hand". The statistic we compute is also given by a modified chi-square test and is given by

$$X_3 = \frac{2^m}{k} \left(\sum_{i=1}^{2^m} n_i^2 \right) - k,$$

following a χ^2 distribution with $2^m - 1$ degrees of freedom.

2.4.4 Run test

Run tests also come in several varieties, for some consider the lengths of increasing or decreasing subsequences, while others consider the amount of subsequences consisting of only one number repeating. For binary sequences, we use this latter type, where the subsequences made up of zeros are called *gaps* while those made up of ones are known as *blocks*. We define $e_i = \frac{(n-i+3)}{2^{i+2}}$ as the expected number of blocks, or gaps, of length i—see [9] for details. Then, we have that k is the largest integer i for which it holds that $e_1 \geq 5$. This is so we don't consider blocks and gaps that are too long and would probably appear a negligible amount of times. We need the following statistic

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

where B_i and G_i are the number of blocks and gaps of length i respectively. We use a χ^2 distribution with $2k - 2$ degrees of freedom.

2.4.5 Autocorrelation test

This last test that we look at measures the correlation of the sequence with itself shifted by an offset d, non cyclically, which means that some terms at the beginning and end of the sequence are disregarded. Then, we hope that not too many or too few bits are equal to their d-shifts. We define $A(d) = \sum_{i=0}^{n-d-1} s_i + s_{i+d} \mod 2$ as the number of bits in the sequence that are equal to their d-shifts. The statistic is

$$X_5 = \frac{2(A(d) - \frac{n-d}{2})}{\sqrt{n-d}},$$

Note that the larger the offset, the less pairs of bits are examined. The statistic follows a $N(0, 1)$ distribution and a two-tailed test shall be used for the reason stated above.

Example. Let us consider a sequence of length $n = 180$, for instance

110010 100001 101110 000101 011111 100000 000001 001011 000011 101110
011011 001100 100100 111100 100001 111011 001011 111101 101110 100001
001100 111100 100111 101000 001100 101110 001011 101101 011010 000010.

We try to run all our tests on it.

1. *Frequency test.* We have that $n_0 = 81$ and $n_1 = 79$. When we apply the statistic formula, we get $X_1 = 0.025$.

2. *Serial test.* It can be seen that $n_{00} = 44$, $n_{01} = 36$, $n_{10} = 37$, and $n_{11} = 42$. Therefore, the statistic is $X_2 = 1.10079$.

3. *Poker test.* The values for m and k we gave in the test description do not apply for this sequence, because of its length. We try instead $m = 3$, and we see that $\lfloor \frac{180}{3} \rfloor \geq 5 \cdot 2^3$ holds. Then, we also have $k = 60$. The subsequences 000 through 111—in binary numbering order—appear respectively 6, 10, 3, 9, 12, 9, 5, and 6 times. The statistic is $X_3 = 8.26667$.

4. *Run test.* Here we have that $k = 3$, and $e_1 = 22.75$, $e_2 = 11.3125$, $e_3 = 5.625$. The blocks of length 1, 2, and 3 are respectively 19, 12, and 6, whereas the gaps of the same lengths are 20, 14, and 1. The statistic value is $X_4 = 5.45858$.

5. *Autocorrelation test.* By using an offset $d = 8$, we have $A(d) = 92$ and a statistic $X_5 = 0.914991$.

The statistic values don't let us accept or reject a sequence by themselves. This depends instead on these values combined with the statistical significance level [17], which allows to find upper bounds for the values of the statistics—or both upper and lower bounds for the autocorrelation test.

In order to select such bounds, which we call *critical values*, we first need to choose an α value, which determines the error rate that we are willing to accept. It is fairly standard to pick $\alpha = 0.05$, and it is the value we use throughout this thesis. Depending on the test used and its parameters, each α value has one or more critical values associated with it, which can be easily found in statistical tables. If the statistic values calculated are below the critical value—for one-sided tests— or between the upper and lower critical values—for two-sided tests—we consider the result significant. What this means in our case is that a specific randomness test is passed by the given sequence.

The following table gives the critical values for each randomness test, based also on the degrees of freedom, where the chi-square distribution is used.

Test	Critical value(s)
Frequency	3.841
Serial	5.991
Poker ($m = 5$ and $k = 200$)	44.985
Run	15.507
Autocorrelation ($d = 7$*)	± 1.96

*Randomly chosen, small prime offset.

Because of our choice of α, we know that a sequence that is sufficiently random has a 95% chance of passing the test. We keep in mind this consideration for when we interpret the general results in Chapter 5.

Chapter 3

Methods

Not considering preparatory work, e.g., gathering information about writing style, essentially two major steps were part of the creation of this report, namely, research of background information and subsequent implementation. The topic for this work was selected in consultation with several teachers. The general idea was to work on something related to cryptography, a particularly interesting field of applied mathematics. Working on elliptic curves was recommended. The choice of topics was then narrowed down until it was decided that pseudo random number generators based on elliptic curves was the most suitable topic.

3.1 Research of information

The research began by seeking information about elliptic curves, as they are the basic structure upon which the pseudo random number generators under consideration rely. By the time this process started, the topic hadn't been covered by any university lecture yet, so quite some time was spent on studying it. The starting point was the document [8] which in particular contains a quick overview of some generators. From there, the investigation branched out with additional documentation and web searches. Overall, though, most of the work went into the implementation. The research was not extremely in-depth, as the aim of this thesis is oriented on analysing existing structures by means of original work, rather than exposition of known concepts.

3.2 Statistical procedures

We use two statistical distributions, namely the chi-square distribution—which we already encountered in Section 2.4 when describing randomness tests—and the normal distribution. The chi-square distribution enables us to evaluate how close the probability distribution of a sample of random variables is to a given distribution, given that these variables are independent from one another. We take k possible categories which a random variable can belong to. The degrees of freedom are denoted by $\nu = k - 1$. If we take n as sequence length, p_k as the probability of an element of falling in category i, and n_i the number of elements in category i, then the statistic is

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - np_i)^2}{np_i},$$

which we apply unaltered to perform the frequency test. For other tests we use variations of it which take into account different circumstances. Whenever a chi-square test is involved, we perform a one tailed test, and expect to get small, positive values.

The only test which makes use of the normal distribution is the autocorrelation test, where we expect to obtain negative or positive values close to zero.

3.2.1 Remarks on interpreting results

When we run each of our tests a number of times, we need to define exactly in what cases we will reject our generators and in what cases we will fail to do so. We take the following hypotheses

H_0: the generators produce non-random sequences;
H_1: the generators produce random sequences.

We are only able to prove non-randomness of a sequence; it is not possible to prove in a definitive way that a sequence is random enough. The best we can do and the result we aim for is to fail to reject the hypothesis of each generator being non-random. Each single test measures a different property of a sequence which is important for randomness, and we therefore want all of these properties to be fulfilled. In other words, in order to fail to reject H_0, we expect all of the tests to be passed.

Note on the other hand that each generator-test combination is tested 1000 times in total as that is the number of elliptic curves we consider. Therefore it may happen that a certain test on a specific generator fails. Unless that happens the majority of the times, it is unreasonable to reject a whole generator just because of one or a couple failed tests. Instead, we consider the probability of a test failing and we compare it to how many times a fail actually occurred. As was stated in Section 2.4, the significance level at which we perform the tests is 95%. What this means in our case is that approximately 5% of sequences produced by a certain generator might fail a given test even if they are valid. In other words, if more than 50 failures happen when we are testing all our curves with a specific test on a given generator, we interpret is as the generator having failed that test. This has to be kept in mind when dealing with the results.

3.3 Implementation

The implementation was fully done on Wolfram Mathematica. Apart from a short code snippet implementing basic elliptic curve functions, the rest of the code is completely original. The main tasks that needed implementation were, to begin with, the three generators at hand, and secondly, the testing methods. The rest of the code consists of helper methods, which I used to simplify the programming.

3.3.1 Theoretical observations

Even if the testing is to be done on the output binary sequences only, it is reasonable to at least have a quick look at the theoretical properties of the generators. With this information at hand, it is possible to provide the generators with very good parameters, allowing for better sequences. If we used somewhat arbitrary parameters—generators and seeds, coefficients of the elliptic curves, fields and other factors related to each particular generator—the likelihood to produce short cycles and/or sequences with predictable patterns would be far greater.

The sequence length we take into consideration is $n = 1000$. This is a value high enough to ensure accuracy on all tests considered, but low enough to be able to manipulate several sequences at once. Because n is quite large, it is important to pay particular attention to the choice of suitable prime fields.

Simplifications

A useful—but not strictly necessary—step is to pick primes p such that the order of the elliptic curves over $\mathbb{F}_p$ would also be prime. Is is then guaranteed that, except for the point at infinity, each element is a generator, i.e., of the maximum possible order. This is because in a finite group, the order of an element divides the order of the group. Then, if a group is of prime order q, the only possible element orders are 1 and q. Of course, this also depends on the parameters a and b of the curve. To generate suitable curves, we randomly select some parameters a and points (x, y) within a predefined range and compute the b and p based on these. We then check the order of these curves as a final step, and if it is prime we accept them and in this way, obtain several *ad hoc* elliptic curves.

Since the primes with which we construct fields are quite large, it is very infeasible to compute the entire elliptic curves. Fortunately, the complete sets of points are not particularly useful to us. Because of our previous assumptions, there is not much of a difference among all the points on each curve. Therefore, we may pick any point we like as generator—and another for a seed, if necessary. We simply calculate the first few points on the curve and use these.

EC-LCG

An advantage of the linear congruential generator is that is has a very simple structure. Therefore, it is easy to evaluate. Recall, from Section 2.3.2, that

$$U_n = nG \oplus U_0, \qquad n = 1, 2, \ldots.$$

We now wish to maximise the period of this generator. Then, we determine that we want nG to take on as many different values as possible. This is equivalent to saying that $\mathrm{ord}(G)$ needs to be as high as possible. This is easy, because of the limitations outlined in the previous paragraph. Any element of the group will then be a good candidate for the generator G.

EC-PG

When it comes to the power generator, we have one more factor to take into account, that is the integer e. Recall the formula for the generator,

$$W_n = e^n G, \qquad n = 1, 2, \ldots.$$

Similarly as with the linear congruential generator, we want $e^n G$ to assume as many values as possible. Consider that e is an integer mod t, with $t = \mathrm{ord}(G)$. It is then possible for e to range through all values from 1 to $t - 1$ in some order, if and only if it is a primitive root modulo t. For the purposes of our implementation, we simply use the smallest primitive root we can find.

EC-NRG

The Naor-Reingold generator is a slightly more complex construction. We recall the definition

$$F_a(n) = a_1^{\nu_1} \ldots a_k^{\nu_k} G, \qquad n = 1, 2, \ldots.$$

Here the most important variable is the input vector $(a_1, a_2, \ldots, a_k)$. We would like all the possible products $a_1^{\nu_1} \ldots a_k^{\nu_k}$ to be different. Because of how the generator works, if wanting an output sequence of length $n = 1000$, we need a minimum vector length $k = \lceil \log_2 n \rceil = 10$. Using the first ten primes to construct the vector is, then, a suitable choice. In this way, we are sure that all possible products, which belong to $\mathbb{Z}$, are different, because the vector entries share no factors. We still have to reduce each of the products mod t, and we might then find out that a few of the products are congruent. However, most of the products will be smaller than p and thus distinct.

We may also rearrange the vector entries, if another sequence is needed. Then we will obtain a permutation of the original sequence.

Constraints and limitations

Because of the limited amount of time at our disposal, we only investigate the aforementioned generators—other random number generators based on elliptic curves exist, see for example [16]. For the same reason, we only make our investigation for a few selected primes. There are a number of randomness tests other than these used in this report, but we need not apply all of them, as just a few are sufficient. Note that apart from the frequency test, all the tests used can be expanded and/or have several variations, as described in for instance [7]. However, each test is only performed in one way for this report. These limitations make the theoretical considerations more relevant, as we are interested in obtaining the best outcomes possible.

Chapter 4

Results

This section gathers the results presented in various ways. First, some example curves are shown. Then, a table is given which gathers several of the testing results. Interpretation and discussion are left to the following chapter.

4.1 Some curves in detail

We will now take a look at some curves selected from the tests, and display extended results. The curves are chosen because of particularly representative behaviours—a curve that passed all tests, curves that failed on only one or more generators, curves that failed similar tests on the same generator. We discuss these properties in Section 5.1. From here onwards, s_0 is the seed and g the generating point used.

Example 1.

$$E(\mathbb{F}_{162947}) : y^2 = x^3 + 36806x + 69342$$
$$s_0 = \{59179, 152510\} \quad g = \{109032, 159310\}$$

Linear congruential
Frequency test: 0.004 (passed)
Serial test: 2.21722 (passed)
Poker test: 24.96 (passed)
Run test: 13.9025 (passed)
Autocorrelation test: 0.533333 (passed)

Power
Frequency test: 0.9 (passed)
Serial test: 2.58649 (passed)
Poker test: 36.16 (passed)
Run test: 10.7527 (passed)
Autocorrelation test: -0.733333 (passed)

Naor-Reingold
Frequency test: 0.064 (passed)
Serial test: 0.907972 (passed)
Poker test: 44.16 (passed)
Run test: 11.4798 (passed)
Autocorrelation test: 1. (passed)

Example 2.

$$E(\mathbb{F}_{127601}) : y^2 = x^3 + 29143x + 65073$$
$$s_0 = \{95416, 33660\} \quad g = \{101909, 82366\}$$

Linear congruential

Frequency test: 0.9 (passed)
Serial test: 1.07297 (passed)
Poker test: 24. (passed)
Run test: 3.73409 (passed)
Autocorrelation test: 0.8 (passed)

Power

Frequency test: 0.256 (passed)
Serial test: 0.275532 (passed)
Poker test: 26.24 (passed)
Run test: 2.73576 (passed)
Autocorrelation test: -1.73333 (passed)

Naor-Reingold

Frequency test: 4.356 (NOT passed)
Serial test: 9.99735 (NOT passed)
Poker test: 34.88 (passed)
Run test: 10.9456 (passed)
Autocorrelation test: 1.26667 (passed)

Example 3.

$$E(\mathbb{F}_{136693}) : y^2 = x^3 + 89534x + 58653$$
$$s_0 = \{120271, 17779\} \quad g = \{80413, 31297\}$$

Linear congruential

Frequency test: 4.356 (NOT passed)
Serial test: 4.11147 (passed)
Poker test: 28.16 (passed)
Run test: 17.4151 (NOT passed)
Autocorrelation test: -0.6 (passed)

Power

Frequency test: 1.296 (passed)
Serial test: 1.23753 (passed)
Poker test: 50.88 (NOT passed)
Run test: 14.5481 (passed)
Autocorrelation test: 0.133333 (passed)

Naor-Reingold

Frequency test: 0.144 (passed)
Serial test: 0.227371 (passed)
Poker test: 33.28 (passed)
Run test: 4.29812 (passed)
Autocorrelation test: 0.133333 (passed)

Example 4.

$$E(\mathbb{F}_{124297}) : y^2 = x^3 + 3686x + 28995$$
$$s_0 = \{65915, 57599\} \quad g = \{117832, 43407\}$$

Linear congruential
Frequency test: 1.024 (passed)
Serial test: 1.55758 (passed)
Poker test: 14.4 (passed)
Run test: 5.38103 (passed)
Autocorrelation test: 0.133333 (passed)

Power
Frequency test: 1.764 (passed)
Serial test: 2.56333 (passed)
Poker test: 53.12 (NOT passed)
Run test: 12.9025 (passed)
Autocorrelation test: -1.13333 (passed)

Naor-Reingold
Frequency test: 0.144 (passed)
Serial test: 1.51666 (passed)
Poker test: 27.2 (passed)
Run test: 11.6783 (passed)
Autocorrelation test: 0.466667 (passed)

4.2 An extract of the results

The table that follows gathers some randomly selected results in a more compact form, which allows us to show several more data in comparison to last section. Because of the larger amount of curves here included, this data is more representative of the complete output for all curves, and can be used to quickly compare different results. The information contained is essentially the same as in the most extensive examples in the previous section. In particular, the matrices in the results column are to be read as follows:

1. Each matrix column shows outcomes for all five tests on a single generator, given in the same order as in the previous section.
2. Each matrix row shows outcomes for a specific test on all generators, respectively the linear congruential, the power and the Naor-Reingold generator.
3. An “1” represents a failed test, while a “0” represents a pass.

In reading the table, it is advised to pay particular attention to any correlations between outputs of similar kinds of tests, as well as correlations among the performance of different generators based on the same curve. In Chapter 5 we elaborate on these observations.

Curve			**Seed**	**Generator**	**Results**
a	b	p	s_0	g	
32886	599	138389	$\{118711, 13012\}$	$\{105666, 86305\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
35467	114475	145043	$\{41009, 80717\}$	$\{14533, 115803\}$	$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
127676	70251	162209	$\{93798, 97137\}$	$\{143361, 73605\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$
2001	80929	137443	$\{56452, 9841\}$	$\{121017, 7676\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
91345	111855	120811	$\{57141, 70744\}$	$\{116084, 22962\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$
4917	47427	113657	$\{98385, 26736\}$	$\{85261, 112514\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
119621	63788	149971	$\{23242, 81650\}$	$\{82130, 115637\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
80508	32532	106243	$\{75313, 99156\}$	$\{12927, 55379\}$	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$
129588	42387	139753	$\{13517, 49169\}$	$\{21942, 122202\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$
95019	63505	138139	$\{45896, 73892\}$	$\{61779, 18926\}$	$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
15895	63080	137713	$\{1508, 38079\}$	$\{79196, 22751\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
46052	74821	115321	$\{48561, 6348\}$	$\{8998, 50513\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
112313	81853	113717	$\{6077, 9775\}$	$\{25014, 22673\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
22591	113837	116131	$\{65462, 14963\}$	$\{74709, 36388\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

	Curve		**Seed**	**Generator**	**Results**
a	b	p	s_0	g	
34130	29071	129769	$\{19710, 25326\}$	$\{7199, 98947\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$
112752	66488	125201	$\{112341, 6505\}$	$\{169, 92155\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
57750	2315	112759	$\{97443, 26919\}$	$\{33886, 36059\}$	$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
188	34592	125659	$\{81933, 7277\}$	$\{47522, 86938\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
874	47810	105673	$\{21288, 94339\}$	$\{17279, 8134\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$
9222	597	125863	$\{87100, 22734\}$	$\{25073, 54561\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$
100731	47	110813	$\{30833, 95937\}$	$\{50472, 64596\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
27160	44476	118457	$\{98644, 2408\}$	$\{66177, 79732\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$
51477	11802	107183	$\{52414, 43675\}$	$\{14477, 3479\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
5176	78989	126653	$\{75626, 74614\}$	$\{61489, 38881\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
42267	43784	143831	$\{74472, 121315\}$	$\{123356, 9686\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$
30254	21506	140867	$\{127442, 78256\}$	$\{3061, 139809\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$
56953	93006	129901	$\{77939, 82347\}$	$\{105168, 118296\}$	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

Chapter 5

Discussion

In this final chapter we deal with and interpret the results we have obtained. Statistical considerations are made, and a description of the running times of the algorithms is given. Some final remarks are made.

5.1 Single curves

When looking at the results for each curve, there are radically different results. Several curves give rise to perfectly pseudo-random sequences, according to the tests we run, on all generators. On the other hand, there are also many curves that fail one or more tests for at least one generator, therefore leading us to reject the random sequence.

Taking a look at the tables and logs of the results, testing failures for a given generator on a specific curve do not correlate strongly with failures for the same curve and a different generator. In other words, no elliptic curve appears to be bad by itself, by only in relation to a specific generator.

Another interesting observation is that a failure of passing some specific tests can correlate to the likelihood of failing other test as well. For instance, often times a curve which does not pass a frequency test on a generator does not pass a serial test either. This is because those tests are very similar and test similar properties.

There are a number of possible solutions to this issue. We may decide for instance to apply a larger number of tests, and investigate the result not in the absolute sense, each on its own, but by comparing the outputs of the various tests. We can also try to simply use a different combination of tests, taken for instance from [7], which contains many more tests other than those proposed here. Another way is to apply tests on the raw generated sequence as well, that is, before the output function creating the binary sequence is applied. After all, if in the raw sequence there are no patterns, there is no reason to believe the binary sequence contains patterns—but of course this also might depend on the choice of output function.

5.2 In general

We run into problems when we look at the results in general, looking to determine how good a generator is in general, by considering how it performs on quite a large amount of different curves. As mentioned in Section 3.2.1, if we are to accept a failure rate of no more than 5%, that means that out of 1000 no more than 50 tests can fail. This is where we see that quite a few generators fail on given tests, although the values are not very much higher than 50. Then, the reasons for this may be either simple bad luck on the choice of curves and/or the fact that even when there are many good curves, all other curves failed quite a few tests, which outweighed the influence of the suitable curves when computing the total percentages.

On the other hand, this might not be the best way of testing a generator by itself, which could very well have affected the results. Such empirical tests are good if we are looking for data on the sequence, but they prove here to be unsuitable for testing a generator. The previous observations were made for the purpose of a general overview, but it is advised to use theoretical tests if needing to test the general performance of a generator.

5.3 Running times

Practically, the time taken to run the whole generating and testing algorithm was about a few hours. However we do not worry about the actual time taken by a specific machine, because it is not an accurate measurement of the efficiency of the implementation. Instead, we try to briefly define the running times in more general terms, by using the *big-O* notation. In order to do this, we take a closer look at the algorithms for both generating and testing. The algorithms for generators and helpful routines can be found in Appendix A, while the code for the tests in Appendix B. For what concerns the time taken by simple operations integrated in Mathematica, we take it to be constant for lack of implementation details, and also because, as long as they don't take more than our routines—very unlikely—, they don't influence the result. The following analysis may be useful to anyone wanting to use or improve the code.

We start by taking a look at EC-LCG. The first operations are assignments, which take constant time and therefore we disregard. We then have a loop, which iterates n times, where n is the length of the sequence we want to generate. Inside the loop, the interesting command is *addpoints*. This command takes constant time, in that it does not depend on the choice of curve. Therefore the loop as a whole takes $O(n)$. We then have the output function for a binary sequence. This command also takes constant time. That means the generator as a whole takes $O(n)$.

For the EC-PG, we proceed similarly. We have a loop of length n, and inside the operation *multpoint*. The way it is implemented, by the double and add algorithm, it is easy to see that the time taken is proportional to $O(\log_2 e)$, where e is the factor by which we multiply a point. Therefore the total time for the loop is $n \cdot O(\log_2 e) = O(n \log_2 k)$. Note that since e is later on selected as a smallest primitive root, it is very small. In this case we may approximate the total duration to be $O(n)$ instead, but this does not hold in the general case.

The last generator, EC-NRG, is a little more complex. We consider the routine *Order of Point*. There is a loop which repeats as many times as there are divisors of the order of the curve, the number of which we denote by k. Multpoint takes as much time as it takes to run for the largest divisor, which is the order itself and we denote by G, and thus we have $k \cdot O(\log_2 G) = O(k \log_2 G)$ total time. Note that, in our special case, the order of the curve is a prime p. This means that the loop only runs once and the time taken then is proportional to the logarithm of that prime, which is quite a long time, since the primes used in this investigation are somewhat large. Then, the time for running the routine reduces to $O(\log_2 p)$. Then, we have a loop that iterates $k = \log_2 n$ times. Another loop with constant time operations to find factors by which to multiply the generating point repeats n times. Afterwards, we compute all these $O(n)$ factors $\mod t$, which takes constant time, and finally the main loop of the algorithm computes the points by using multpoint, taking in total $O(n \log_2 t)$. We just sum the most prominent terms and disregard the others, to get a total time of $O(\log_2 p + n \log_2 t)$. Since in our case, with additional considerations, p and t are similarly large, we may then write $O(\log_2 p + n \log_2 p) = O(n \log_2 p)$. Thus this is the generator that takes the longest time.

Lastly, we briefly look at the test implementations. It is straightforward from the code for the chi and serial tests that they both take $O(n)$ time. The poker test is more interesting, as there are two loops one of which iterates 2^m times and the other $k = \frac{n}{m}$. By the definition of how to derive the values m and k, we know that $2^m < k$, and therefore total time is $O(k)$. The run test is also interesting because it is made up with a helper routine called *BlocksGaps*. This takes $O(n)$. In the testing method itself we have a loop of n steps, and a second loop of at most n steps, which executes the previously mentioned $O(n)$ routine. Therefore we can see that in total, the time taken is $O(n^2)$ for this loop and for the whole method. Finally, the autocorrelation algorithm takes $O(n)$ as well, as it is easy to verify.

5.4 Possible developments

Apart from trying the solutions anticipated in the previous sections, it is also worth investigating the curves themselves. It is possible that a different choice of elliptic curve and/or prime field yields better sequences. However, this was not in the scope of this work, as shows the fact that we used randomly picked curves. Perhaps a bigger curve would work better, but going that direction would require much more computational power. It also needs to be considered that the tests suggested may be somewhat updated. Using a newer suite of tests may be more useful.

Chapter 6

Conclusion

Several different aspects of randomness were examined in this work. Not only did we attempt to define randomness, but we also took a look at it from an historical context. It is remarkable what efforts have been made by scientists and mathematicians to come up with sources of random numbers. Some of such these are now extremely outdated, despite having been popular in the past, and a modern day mathematician wouldn't probably even think of using them if not for study purposes. It is fascinating how the concept of randomness subtly changes to fit different applications, and the way it is applicable to a large variety of fields of human knowledge.

Without introducing randomness in such a broad way, it would not have been possible to present the different concepts that were needed for this work in an understandable, interesting manner. We went over different aspects and challenges of random number generation, and saw that there are essentially two ways of generating such random sequences. That is, deterministic and non-deterministic, both enclosing tens of different generators, each used for specific purposes, according to how much time and resources we are willing to spend, and the final application. We went over testing methods, mentioning in particular those which would be used during our investigation.

Another essential concept that has been looked at is elliptic curves over finite fields. The explanations given here were not intended as a complete lecture on the topic, but rather as a primer including only the necessary information for the reader to be able to follow the theory of pseudo-random number generators based on elliptic curves.

Apart from these introductory notions, this work is largely original. The result of independent research is what ended up in the preliminary section, and the rest was programming work. Although it could surely be improved, the implementation was good enough and functional for the intended purposes. Before going on to the testing, we made some statistical remarks which were essential to interpret our results. Moreover, we delved deeper into the matter of optimising the inner workings of the generators in order to eliminate as many sources of non-randomness as possible. Simplifications had to be made because of the limited time and resources for this work, and they were carefully described.

There were expectations for our results, and not all of these were met, which is not a bad thing by itself. When doing scientific work any result is a good result if it provides new insight, and if we are able to give possible explanations for it in order to improve further investigation. Finally, to end the report, we went over the results and interpreted them from different points of view.

Bibliography

[1] Cohen, Henry and Frey, Gerhard. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Boca Raton: Chapman & Hall/CRC, 2006.

[2] De Vito, Bob. *History of Random Number Generators*. Probability and Statistics, 2005. Retrieved from `http://people.brandeis.edu/~moshep/_0Lect8/Presentation/RandomNumberGenerators_BobDeVito.ppt`, May 26, 2015.

[3] Fairhead, Harry. *ERNIE - A Random Number Generator*. I Programmer, September 2013. Retrieved from `http://www.i-programmer.info/history/machines/6317-ernie-a-random-number-generator.html`, May 26, 2015.

[4] Garrett, Matthew. *A Short History of Random Numbers (And Why You Need to Care)*. Portland, 2013. Retrieved from `http://www.codon.org.uk/~mjg59/oscon_random_2013.odp`, May 26, 2015.

[5] Green, Matthew. *The Many Flaws of Dual_EC_DRBG*. A Few Thoughts on Cryptographic Engineering, September 2013. Retrieved from `http://blog.cryptographyengineering.com/2013/09/the-many-flaws-of-dualecdrbg.html`, May 26,2015.

[6] Kamath, John-Paul. *Photo story: ERNIE goes on display at the Science Museum*. Computer Weekly, June 2008. Retrieved from `http://www.computerweekly.com/feature/Photo-story-ERNIE-goes-on-display-at-the-Science-Museum`, May 26, 2015.

[7] Knuth, Donald E. *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms. 3rd ed. Boston: Addison-Wesley, 1997.

[8] Lange, Tanja. *Analysis of pseudo-random number generators based on elliptic curves*. 31st Australasian Conference on Combinatorial Mathematics & Combinatorial Computing (ACCMCC), Alice Springs, 2006. Retrieved from `http://www.hyperelliptic.org/tanja/#conf`, May 26, 2015.

[9] Menezes, Alfred J., Van Oorschot, Paul C. and Vanstone, Scott A. *Handbook of Applied Cryptography*. Boca Raton: CRC Press, Inc., 1997.

[10] Miller, Victor S. *Elliptic Curves and their use in Cryptography.* Princeton, 1997. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.1785&rep=rep1&type=pdf, May 26, 2015.

[11] Silverman, Joseph H. *An Introduction to the Theory of Elliptic Curves.* Summer school on Computational Number Theory and Applications to Cryptography, University of Wyoming, 2006. Retrieved from http://www.math.brown.edu/~jhs/Presentations/WyomingEllipticCurve.pdf, May 26, 2015.

[12] Williams, Joseph J. and Griffiths, Thomas L. *Why are People Bad at Detecting Randomness? Because it is Hard.* 30th Annual Conference of the Cognitive Science Society, Washington DC, 2008. Retrieved from https://cocosci.berkeley.edu/tom/papers/hard.pdf, May 26, 2015.

[13] *A Million Random Digits with 100000 Normal Deviates.* 2nd ed. RAND, 2001. Retrieved from http://www.rand.org/content/dam/rand/pubs/monograph_reports/MR1418/MR1418.introduction.pdf, May 26, 2015.

[14] *ERNIE 1.* Science Museum, London. Retrieved from http://objectwiki.sciencemuseum.org.uk/wiki/ERNIE_1.html, May 26, 2015.

[15] *RANDOM.org.* Retrieved from https://www.random.org/, May 26, 2015.

[16] *Recent Trends in Cryptography.* Universidad Internacional Menndez Pelayo Santander, Spain, 2005. E-book. Retrieved from https://books.google.se/, May 26, 2015.

[17] *Statistical Significance.* StatPac. Retrieved from https://www.statpac.com/surveys/statistical-significance.htm, May 26, 2015.

Appendices

Appendix A

Source code for random number generators

The following is the complete source code for this project, written in Mathematica language.

```
EllipticCurve[a_, b_, p_] :=
Module[{e, lsn},
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is
    singular"];
Abort[]];
e = {{\[Infinity], \[Infinity]}};
Do[
lsn = y /. Solve[y^2 == x^3 + a x + b, y, Modulus -> p];
If[Length[lsn] > 0, e = Union[e, {{x, lsn[[1]]}, {x, lsn
    [[2]]}}]],
{x, 0, p - 1}];
RotateRight[e, 1]]

OrderOfElliptic[a_, b_, p_] :=
If[Mod[4 a^3 + 27 b^2 + b, p] == 0, Print["The curve is
    singular!"],
p + 1 + Sum[JacobiSymbol[x^3 + a x + b, p], {x, 0, p -
    1}]]

addpoints[{x1_, y1_}, {x2_, y2_}, a_, b_, p_] :=
Module[{\[Lambda], x3, y3},
Quiet[
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is
    singular"];
Abort[]];
If[Mod[y1^2 - x1^3 - a x1 - b, p] != 0 ||
Mod[y2^2 - x2^3 - a x2 - b, p] != 0,
Print["Error: Point(s) not on curve!"]; Abort[]];
If[{x1, y1} == {\[Infinity], \[Infinity]}, Return[{x2, y2
    }];
Break[]];
If[{x2, y2} == {\[Infinity], \[Infinity]}, Return[{x1, y1
    }];
Break[]];
If[x1 == x2 && y1 == Mod[-y2, p],
Return[{\[Infinity], \[Infinity]}]; Break[]];
If[{x1, y1} == {x2, y2}, \[Lambda] =
Mod[(3 x1^2 + a) PowerMod[2 y1, -1, p], p], \[Lambda] =
Mod[(y2 - y1) PowerMod[x2 - x1, -1, p], p]];
x3 = Mod[\[Lambda]^2 - x1 - x2, p];
y3 = Mod[\[Lambda] (x1 - x3) - y1, p];
{x3, y3}]
]

multpoint[n_, {x_, y_}, a_, b_, p_] :=
Module[{double, base2, rounds, x2, y2},
{x2, y2} = {x, y};
double = {};
base2 = Reverse[IntegerDigits[n, 2]];
Do[
AppendTo[double, {x2, y2}];
{x2, y2} = addpoints[{x2, y2}, {x2, y2}, a, b, p],
{Length[base2]}];
{x2, y2} = {\[Infinity], \[Infinity]};
Do[
If[base2[[i]] == 1, {x2, y2} =
addpoints[{x2, y2}, double[[i]], a, b, p]],
{i, Length[base2]}];
{x2, y2}
]

OrderOfPoint[{x_, y_}, a_, b_, p_] :=
Module[{candidates},
If[Mod[4 a^3 + 27 b^2, p] == 0, Print["The curve is
    singular!"];
Abort[]];
If[Mod[y^2 - x^3 - a x - b, p] != 0,
Print["Error: Point not on curve!"]; Abort[]];
candidates = Divisors[OrderOfElliptic[a, b, p]];
Do[
If[multpoint[candidates[[i]], {x, y}, a, b,
p] == {\[Infinity], \[Infinity]}, Return[candidates[[i
    ]]];
Break[],
{i, Length[candidates]}]
]

MakeBinarySequence[seq_, p_] := Module[{output, threshold
    , curr, i},
threshold = (p - 1)/2;

output = {};
For[i = 1, i <= Length[seq], i++,
If [seq[[i]][[1]] <= threshold, curr = 0, curr = 1];
output = Append[output, curr];
];

Return[output];
];

ECLCG[{x0_, y0_}, {x1_, y1_}, a_, b_, p_] :=
Module[{current, list, g, i, seed, binary},
seed = {x0, y0};
current = seed;
g = {x1, y1};
list = {{x0, y0}};
For[i = 0, i <= 1000, i++,
If[Length[list] == 1 || current != seed,
list = Append[list, current];
current = addpoints[current, g, a, b, p];
];
];
list = Delete[list, 1];
list = Delete[list, 1];
binary = MakeBinarySequence[list, p];
Return[binary];
]

ECPG[{x0_, y0_}, e_, a_, b_, p_] :=
Module[{i, seed, list, current, length, binary},
seed = {x0, y0};
current = seed;
list = {};
length = 1;
For[i = 0, i <= 1000, i++,
If[current == seed && length != 1, Break[],];
list = Append[list, current];
```

```
length++;
current = multpoint[e, current, a, b, p];
];
list = Delete[list, 1];
binary = MakeBinarySequence[list, p];
Return[binary];
]

ECNRG[{g0_, g1_}, n_, a_, b_, p_] :=
Module[{k, vect, t, i, r, sequence, j, expvec, zeros, num
    , l, e, c,
def, m, pt, x},
k = Log[2, n];
vect = {};

t = OrderOfPoint[{g0, g1}, a, b, p];
For[i = 0, i < k, i++,
r = RandomInteger[{2, t - 1}];
vect = Append[vect, Prime[i + 1]];
];

sequence = {};
For[j = 0, j <= 2^k - 1, j++,
expvec = IntegerDigits[j, 2];
zeros = Length[vect] - Length[expvec];
num = 1;
If[zeros > 0,
For[e = 1, e <= zeros, e++,
expvec = Prepend[expvec, 0];
],];

For[l = 1, l <= Length[vect], l++,
c = num*(vect[[l]]^expvec[[l]]);
num = c;
];
sequence = Append[sequence, num];
num = 1;
];
For[x = 1, x <= Length[sequence], x++,
sequence[[x]] = Mod[sequence[[x]], t];
];

def = {};
For[m = 1, m <= Length[sequence], m++,
pt = multpoint[sequence[[m]], {g0, g1}, a, b, p];
def = Append[def, pt];
```

```
];
def = MakeBinarySequence[def, p];
Return[def];
];
```

Appendix B

Source code for testing

```
ChiTest[seq_] := Module[{ones, zeros, i, p, expectedOnes,
     statistic},
ones = 0;
For[i = 1, i <= Length[seq], i++,
If[seq[[i]] == 1, ones++,];
];
zeros = Length[seq] - ones;
p = 0.5;
expectedOnes = Length[seq]*p;
statistic = ((ones - expectedOnes)^2/
expectedOnes) + ((zeros - expectedOnes)^2/expectedOnes);
If[statistic < 3.841, Print["The frequency test is passed
    ." ];
Return[True],
Print["The frequency test is NOT passed." ]; Return[False
    ]];
]

SerialTest[seq_ ] := Module[{ones, zeros, n00, n01, n10,
    n11, i, stat},
ones = 0;
For[i = 1, i <= Length[seq], i++,
If[seq[[i]] == 1, ones++,];
];
zeros = Length[seq] - ones;
n00 = 0;
n01 = 0;
n10 = 0;
n11 = 0;
For[i = 1, i < Length[seq], i++,
If[seq[[i]] == 0,
If[seq[[i + 1]] == 0, n00++, n01++];,
If[seq[[i + 1]] == 0, n10++, n11++];
];
];
stat = ((4/(Length[seq] - 1))*(n00^2 + n01^2 + n10^2 +
n11^2)) - ((2/Length[seq])*(zeros^2 + ones^2)) + 1;
If [stat > 5.991, Print["The serial test is NOT passed
    ."];
Return[False], Print["The serial test is passed."];
    Return[True]];
]

PokerTest[seq_, m_, k_] :=
Module[{i, numList, elements, sum, statistic},
list = {};
For[i = 1, i <= 2^m, i++,
list = Append[list, 0];
];

For[i = 1, i <= Length[seq], i = i + 5,
elements = {};
For[j = 1, j <= m, j++,
elements = Append[elements, seq[[i + j - 1]]];
];

list[[FromDigits[elements, 2] + 1]]++;
];

statistic = (2^m/k)*Sum[list[[i]]^2, {i, 2^m}] - k;

If[statistic <= 44.985, Print["The poker test is passed
    ."];
Return[True], Print["The poker test is NOT passed."];
Return[False]];
]

BlocksGaps[seq_, length_] :=
Module[{i, blocks, gaps, currLength, n, list},
blocks = 0;
gaps = 0;
currLength = 1;

n = Length[seq];
list = {};
For[i = 2, i <= n, i++,
If [seq[[i]] != seq[[i - 1]], list = Append[list,
   currLength];
currLength = 1, currLength++];

];
list = Append[list, currLength];

For[i = 1, i <= Length[list], i++,
If[list[[i]] == length,
If[seq[[1]] == 1,
If[Mod[i, 2] == 1,
blocks++;
, gaps++];
, If[Mod[i, 2] == 1,
gaps++;
, blocks++];];
,];
];

Return[{blocks, gaps}];
]

RunTest[seq_] := Module[{n, i, ei, k, statistic, B, G, j,
    e},
n = Length[seq];
i = n - 995;
For[i = n, i >= 0, i--,
ei = (n - i + 3)/(2^(i + 2));
If[ei >= 5, k = i; Break[]; , ,];
];
statistic = 0;
For[j = 1, j <= k, j++,
{B, G} = BlocksGaps[seq, j];
e = (n - j + 3)/(2^(j + 2));
statistic = statistic + (((B - e)^2)/e) + (((G - e))^2/e)
    ;
];
If[statistic > 15.507, Print["The run test is NOT passed
    ."];
Return[False], Print["The run test is passed."]; Return[
   True]];
]

AutoCorrelation[seq_] := Module[{d, n, A, statistic},
d = 100;
n = Length[seq];
```

```
A = Sum[BitXor[seq[[i]], seq[[i + d]]], {i, n - d}];
statistic = 2*(A - ((n - d)/2))/Sqrt[n - d];
If[statistic >= -1.96 && statistic <= 1.96,
Print["The autocorrelation test is passed."]; Return[True
    ],
Print["The autocorrelation test is NOT passed."]; Return[
    False]];
];

DoAllTests[seq_, list_, d_] := Module[{fails, passed, m,
    k},
fails = {0, 0, 0, 0, 0};
m = 5;
k = 200;
passed = ChiTest[seq];
If [passed == False, fails[[1]]++,];
passed = SerialTest[seq];
If [passed == False, fails[[2]]++,];
passed = PokerTest[seq, m, k];
If [passed == False, fails[[3]]++,];
passed = RunTest[seq];
If [passed == False, fails[[4]]++,];
passed = AutoCorrelation[seq];
If [passed == False, fails[[5]]++,];
Return[fails];
];

GenerateCurve := Module[{i, p, x, y, A, B, q},
i = 0;
While[True,
p = Prime[RandomInteger[{10000, 15000}]];
{x, y, A} = RandomChoice[Range[0, p - 1], 3];
B = Mod[y^2 - x^3 - A x, p];
q = p + 1 + Sum[JacobiSymbol[x^3 + A x + B, p], {x, 0, p
    - 1}];
If[PrimeQ[q], Break[], i++]];
{A, B, p, q, i}
];

GetTwoPoints[a_, b_, p_] := Module[{x, Y2, P, k, Q, R},
x = 1;
While[True,
Y2 = x^3 + a*x + b;
If[JacobiSymbol[Y2, p] == 1, P = {x, PowerMod[Y2, 1/2, p
    ]};
Break[],];
```

```
x++;
];
k = RandomInteger[{1, p}];
Q = multpoint[k, P, a, b, p];
k = RandomInteger[{1, p}];
R = multpoint[k, P, a, b, p];
Return[{Q, R}];
]

Test[n_] :=
Module[{curveData, allCurves, failures, k, points, A, B,
    p, q, i,
eclcg, ecpg, ecnrg, totalFails, fails1, fails2, fails3,
    fails,
idx}, curveData = {};
totalFails = {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0,
     0, 0}};
allCurves = {};
For[k = 1, k <= n, k++, fails = {};
{A, B, p, q, i} = GenerateCurve;
curveData = Append[curveData, A];
curveData = Append[curveData, B];
curveData = Append[curveData, p];
Print["------------------- For the curve ", Superscript["Y
    ", 2],
" = ", Superscript["X", 3], " + ", A, "X + ", B, " over
    ",
Subscript["F", p], " -------------------"];
Print["-------------------------"\
];
points = GetTwoPoints[A, B, p];
curveData = Append[curveData, points[[1]]];
curveData = Append[curveData, points[[2]]];
Print[points];
eclcg = ECLCG[points[[1]], points[[2]], A, B, p];
Print["Linear congruential:"];
fails1 = DoAllTests[eclcg, failures, 0];
ecpg = ECPG[points[[1]], PrimitiveRoot[q], A, B, p];
Print["Power generator:"];
fails2 = DoAllTests[ecpg, failures, 1];
ecnrg = ECNRG[points[[1]], 1000, A, B, p];
Print["Naor Reingold:"];
fails3 = DoAllTests[ecnrg, failures, 2];
fails = Append[Append[Append[fails, fails1], fails2],
    fails3];
curveData = Append[curveData, fails];
```

```
totalFails = totalFails + fails;
Print["Total fails so far: ", totalFails];
allCurves = Append[allCurves, curveData];
curveData = {};
];
TableForm[allCurves,
TableHeadings -> {None, {"A", "B", "P", Subscript["s",
    0], "g",
"Results"}}]
]
```

Author: Alice Reinaudo
Supervisor: Per-Anders Svensson
Examiner: Marcus Nilsson
Date: 2015-06-20
Course Code: 2MA11E
Subject: Applied Mathematics
Level: Undergraduate

Department Of Mathematics